MongoDB Reference Manual

Release 2.8.0-rc2

MongoDB Documentation Project

December 18, 2014

Contents

This document contains all of the reference material from the `MongoDB Manual`, reflecting the 2.8.0-rc2 release. See the full manual, for complete documentation of MongoDB, it's operation, and use.

CHAPTER 1

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].

1.1 License

This manual is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 3.0 Unported[4]" (i.e. "CC-BY-NC-SA") license.

1.2 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] (without reference.)
- HTML tar.gz[9]

You also can access PDF files that contain subsets of the MongoDB Manual:

- MongoDB Reference Manual[10]
- MongoDB CRUD Operations[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/master/MongoDB-manual.epub
[7] http://docs.mongodb.org/master/single/
[8] http://docs.mongodb.org/master/MongoDB-manual.pdf
[9] http://docs.mongodb.org/master/manual.tar.gz
[10] http://docs.mongodb.org/master/MongoDB-reference-manual.pdf
[11] http://docs.mongodb.org/master/MongoDB-crud-guide.pdf

- Data Models for MongoDB[12]
- MongoDB Data Aggregation[13]
- Replication and MongoDB[14]
- Sharding and MongoDB[15]
- MongoDB Administration[16]
- MongoDB Security[17]

MongoDB Reference documentation is also available as part of dash[18]. You can also access the MongoDB Man Pages[19] which are also distributed with the official MongoDB Packages.

1.3 Version and Revisions

This version of the manual reflects version 2.8 of MongoDB.

See the MongoDB Documentation Project Page[20] 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[21].

This edition reflects "`master`" branch of the documentation as of the "`585dc8768a736a5b055ab2d13f1bee75aac83b96`" revision. This branch is explicitly accessible via "http://docs.mongodb.org/master" and you can always reference the commit of the current manual in the release.txt[22] file.

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

1.4 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[23].

1.5 Contribute to the Documentation

1.5.1 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.

[12] http://docs.mongodb.org/master/MongoDB-data-models-guide.pdf
[13] http://docs.mongodb.org/master/MongoDB-aggregation-guide.pdf
[14] http://docs.mongodb.org/master/MongoDB-replication-guide.pdf
[15] http://docs.mongodb.org/master/MongoDB-sharding-guide.pdf
[16] http://docs.mongodb.org/master/MongoDB-administration-guide.pdf
[17] http://docs.mongodb.org/master/MongoDB-security-guide.pdf
[18] http://kapeli.com/dash
[19] http://docs.mongodb.org/master/manpages.tar.gz
[20] http://docs.mongodb.org
[21] https://github.com/mongodb/docs
[22] http://docs.mongodb.org/master/release.txt
[23] https://jira.mongodb.org/browse/DOCS

To this end, the MongoDB Documentation Project is preparing to launch a translation effort to allow the community to help bring the documentation to speakers of other languages.

If you would like to express interest in helping to translate the MongoDB documentation once this project is opened to the public, please:

- complete the MongoDB Contributor Agreement[24], and
- join the mongodb-translators[25] user group.

The mongodb-translators[26] 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 4)
- *Style Guide and Documentation Conventions* (page 6)
- *MongoDB Manual Organization* (page 15)
- *MongoDB Documentation Practices and Processes* (page 12)
- *MongoDB Documentation Build System* (page 16)

The entire documentation source for this manual is available in the mongodb/docs repository[27], which is one of the MongoDB project repositories on GitHub[28].

To contribute to the documentation, you can open a GitHub account[29], fork the mongodb/docs repository[30], 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[31].

You can clone the repository by issuing the following command at your system shell:

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

1.5.2 About the Documentation Process

The MongoDB Manual uses Sphinx[32], a sophisticated documentation engine built upon Python Docutils[33]. The original reStructured Text[34] 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:

[24] http://www.mongodb.com/legal/contributor-agreement
[25] http://groups.google.com/group/mongodb-translators
[26] http://groups.google.com/group/mongodb-translators
[27] https://github.com/mongodb/docs
[28] http://github.com/mongodb
[29] https://github.com/
[30] https://github.com/mongodb/docs
[31] http://www.mongodb.com/contributor
[32] http://sphinx-doc.org//
[33] http://docutils.sourceforge.net/
[34] http://docutils.sourceforge.net/rst.html

Style Guide and Documentation Conventions

This document provides an overview of the style for the MongoDB documentation stored in this repository. The overarching goal of this style guide is to provide an accessible base style to ensure that our documentation is easy to read, simple to use, and straightforward to maintain.

For information regarding the MongoDB Manual organization, see *MongoDB Manual Organization* (page 15).

Document History

2011-09-27: Document created with a (very) rough list of style guidelines, conventions, and questions.

2012-01-12: Document revised based on slight shifts in practice, and as part of an effort of making it easier for people outside of the documentation team to contribute to documentation.

2012-03-21: Merged in content from the Jargon, and cleaned up style in light of recent experiences.

2012-08-10: Addition to the "Referencing" section.

2013-02-07: Migrated this document to the manual. Added "map-reduce" terminology convention. Other edits.

2013-11-15: Added new table of preferred terms.

Naming Conventions

This section contains guidelines on naming files, sections, documents and other document elements.

- File naming Convention:
  - For Sphinx, all files should have a `.txt` extension.
  - Separate words in file names with hyphens (i.e. `-`.)
  - For most documents, file names should have a terse one or two word name that describes the material covered in the document. Allow the path of the file within the document tree to add some of the required context/categorization. For example it's acceptable to have `http://docs.mongodb.org/manual/core/sharding.rst` and `http://docs.mongodb.org/manual/administration/sharding.rst`.
  - For tutorials, the full title of the document should be in the file name. For example, `http://docs.mongodb.org/manual/tutorial/replace-one-configuration-server-in-a-sharc`
- Phrase headlines and titles so users can determine what questions the text will answer, and material that will be addressed, without needing them to read the content. This shortens the amount of time that people spend looking for answers, and improvise search/scanning, and possibly "SEO."
- Prefer titles and headers in the form of "Using foo" over "How to Foo."
- When using target references (i.e. `:ref:` references in documents), use names that include enough context to be intelligible through all documentation. For example, use "`replica-set-secondary-only-node`" as opposed to "`secondary-only-node`". This makes the source more usable and easier to maintain.

Style Guide

This includes the local typesetting, English, grammatical, conventions and preferences that all documents in the manual should use. The goal here is to choose good standards, that are clear, and have a stylistic minimalism that does not interfere with or distract from the content. A uniform style will improve user experience and minimize the effect of a multi-authored document.

Punctuation

- Use the Oxford comma.

  Oxford commas are the commas in a list of things (e.g. "something, something else, and another thing") before the conjunction (e.g. "and" or "or.").

- Do not add two spaces after terminal punctuation, such as periods.
- Place commas and periods inside quotation marks.

Headings Use title case for headings and document titles. Title case capitalizes the first letter of the first, last, and all significant words.

Verbs Verb tense and mood preferences, with examples:

- **Avoid** the first person. For example do not say, "We will begin the backup process by locking the database," or "I begin the backup process by locking my database instance."
- **Use** the second person. "If you need to back up your database, start by locking the database first." In practice, however, it's more concise to imply second person using the imperative, as in "Before initiating a backup, lock the database."
- When indicated, use the imperative mood. For example: "Backup your databases often" and "To prevent data loss, back up your databases."
- The future perfect is also useful in some cases. For example, "Creating disk snapshots without locking the database will lead to an invalid state."
- Avoid helper verbs, as possible, to increase clarity and concision. For example, attempt to avoid "this does foo" and "this will do foo" when possible. Use "does foo" over "will do foo" in situations where "this foos" is unacceptable.

Referencing

- To refer to future or planned functionality in MongoDB or a driver, *always* link to the Jira case. The Manual's `conf.py` provides an `:issue:` role that links directly to a Jira case (e.g. `:issue:\`SERVER-9001\``).
- For non-object references (i.e. functions, operators, methods, database commands, settings) always reference only the first occurrence of the reference in a section. You should *always* reference objects, except in section headings.
- Structure references with the *why* first; the link second.

  For example, instead of this:

  Use the `http://docs.mongodb.org/manual/tutorial/convert-replica-set-to-replicated-shard-c` procedure if you have an existing replica set.

  Type this:

  To deploy a sharded cluster for an existing replica set, see `http://docs.mongodb.org/manual/tutorial/convert-r`

General Formulations

- Contractions are acceptable insofar as they are necessary to increase readability and flow. Avoid otherwise.
- Make lists grammatically correct.
  - Do not use a period after every item unless the list item completes the unfinished sentence before the list.

  - Use appropriate commas and conjunctions in the list items.
  - Typically begin a bulleted list with an introductory sentence or clause, with a colon or comma.
- The following terms are one word:
  - standalone
  - workflow
- Use "unavailable," "offline," or "unreachable" to refer to a `mongod` instance that cannot be accessed. Do not use the colloquialism "down."
- Always write out units (e.g. "megabytes") rather than using abbreviations (e.g. "MB".)

Structural Formulations

- There should be at least two headings at every nesting level. Within an "h2" block, there should be either: no "h3" blocks, 2 "h3" blocks, or more than 2 "h3" blocks.
- Section headers are in title case (capitalize first, last, and all important words) and should effectively describe the contents of the section. In a single document you should strive to have section titles that are not redundant and grammatically consistent with each other.
- Use paragraphs and paragraph breaks to increase clarity and flow. Avoid burying critical information in the middle of long paragraphs. Err on the side of shorter paragraphs.
- Prefer shorter sentences to longer sentences. Use complex formations only as a last resort, if at all (e.g. compound complex structures that require semi-colons).
- Avoid paragraphs that consist of single sentences as they often represent a sentence that has unintentionally become too complex or incomplete. However, sometimes such paragraphs are useful for emphasis, summary, or introductions.

  As a corollary, most sections should have multiple paragraphs.
- For longer lists and more complex lists, use bulleted items rather than integrating them inline into a sentence.
- Do not expect that the content of any example (inline or blocked) will be self explanatory. Even when it feels redundant, make sure that the function and use of every example is clearly described.

ReStructured Text and Typesetting

- Place spaces between nested parentheticals and elements in JavaScript examples. For example, prefer `{ [ a, a, a ] }` over `{[a,a,a]}`.
- For underlines associated with headers in RST, use:
  - = for heading level 1 or h1s. Use underlines and overlines for document titles.
  - − for heading level 2 or h2s.
  - ~ for heading level 3 or h3s.
  - ` for heading level 4 or h4s.
- Use hyphens (-) to indicate items of an ordered list.
- Place footnotes and other references, if you use them, at the end of a section rather than the end of a file.

  Use the footnote format that includes automatic numbering and a target name for ease of use. For instance a footnote tag may look like: `[#note]_` with the corresponding directive holding the body of the footnote that resembles the following: `.. [#note]`.

  Do **not** include `.. code-block:: [language]` in footnotes.

- As it makes sense, use the `.. code-block:: [language]` form to insert literal blocks into the text. While the double colon, `::`, is functional, the `.. code-block:: [language]` form makes the source easier to read and understand.
- For all mentions of referenced types (i.e. commands, operators, expressions, functions, statuses, etc.) use the reference types to ensure uniform formatting and cross-referencing.

Jargon and Common Terms

Preferred Term	Concept	Dispreferred Alternatives	Notes
document	A single, top-level object/record in a MongoDB collection.	record, object, row	Prefer document over object because of concerns about cross-driver language handling of objects. Reserve record for "allocation" of storage. Avoid "row," as possible.
database	A group of collections. Refers to a group of data files. This is the "logical" sense of the term "database."		Avoid genericizing "database." Avoid using database to refer to a server process or a data set. This applies both to the datastoring contexts as well as other (related) operational contexts (command context, authentication/authorization context.)
instance	A daemon process. (e.g. **mongos** or **mongod**)	process (acceptable sometimes), node (never acceptable), server.	Avoid using instance, unless it modifies something specifically. Having a descriptor for a process/instance makes it possible to avoid needing to make mongod or mongos plural. Server and node are both vague and contextually difficult to disambiguate with regards to application servers, and underlying hardware.
field name	The identifier of a value in a document.	key, column	Avoid introducing unrelated terms for a single field. In the documentation we've rarely had to discuss the identifier of a field, so the extra word here isn't burdensome.
field/value	The name/value pair that describes a unit of data in MongoDB.	key, slot, attribute	Use to emphasize the difference between the name of a field and its value For example, "_id" is the field and the default value is an ObjectId.
value	The data content of a field.	data	
MongoDB	A group of processes, or deployment that implement the MongoDB interface.	mongo, mongodb, cluster	Stylistic preference, mostly. In some cases it's useful to be able to refer generically to instances (that may be either **mongod** or **mongos**.)
sub-document	An embedded or nested document within a document or an array.	embedded document, nested document	
map-reduce	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 physical 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[35].)

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[36].

Err on the side of creating a larger number of discrete commits rather than bundling large set of changes into one commit.

[35] https://jira.mongodb.org/browse/DOCS-41
[36] 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[37] 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[38], fork the mongodb/docs repository[39], 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[40] 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.

[37] https://jira.mongodb.org/browse/DOCS
[38] https://github.com/
[39] https://github.com/mongodb/docs
[40] http://sphinx.pocoo.org/

At the present time, future work transpires in the `master`, with the main publication being `current`. As the documentation stabilizes, the documentation team will begin to maintain branches of the documentation for specific MongoDB releases.

Migration from Legacy Documentation

The MongoDB.org Wiki contains a wealth of information. As the transition to the Manual (i.e. this project and resource) continues, it's *critical* that no information disappears or goes missing. The following process outlines *how* to migrate a wiki page to the manual:

1. Read the relevant sections of the Manual, and see what the new documentation has to offer on a specific topic.

   In this process you should follow cross references and gain an understanding of both the underlying information and how the parts of the new content relates its constituent parts.

2. Read the wiki page you wish to redirect, and take note of all of the factual assertions, examples presented by the wiki page.

3. Test the factual assertions of the wiki page to the greatest extent possible. Ensure that example output is accurate. In the case of commands and reference material, make sure that documented options are accurate.

4. Make corrections to the manual page or pages to reflect any missing pieces of information.

   The target of the redirect need *not* contain every piece of information on the wiki page, **if** the manual as a whole does, and relevant section(s) with the information from the wiki page are accessible from the target of the redirection.

5. As necessary, get these changes reviewed by another writer and/or someone familiar with the area of the information in question.

   At this point, update the relevant Jira case with the target that you've chosen for the redirect, and make the ticket unassigned.

6. When someone has reviewed the changes and published those changes to Manual, you, or preferably someone else on the team, should make a final pass at both pages with fresh eyes and then make the redirect.

   Steps 1-5 should ensure that no information is lost in the migration, and that the final review in step 6 should be trivial to complete.

Review Process

Types of Review The content in the Manual undergoes many types of review, including the following:

Initial Technical Review Review by an engineer familiar with MongoDB and the topic area of the documentation. This review focuses on technical content, and correctness of the procedures and facts presented, but can improve any aspect of the documentation that may still be lacking. When both the initial technical review and the content review are complete, the piece may be "published."

Content Review Textual review by another writer to ensure stylistic consistency with the rest of the manual. Depending on the content, this may precede or follow the initial technical review. When both the initial technical review and the content review are complete, the piece may be "published."

Consistency Review This occurs post-publication and is content focused. The goals of consistency reviews are to increase the internal consistency of the documentation as a whole. Insert relevant cross-references, update the style as needed, and provide background fact-checking.

When possible, consistency reviews should be as systematic as possible and we should avoid encouraging stylistic and information drift by editing only small sections at a time.

Subsequent Technical Review If the documentation needs to be updated following a change in functionality of the server or following the resolution of a user issue, changes may be significant enough to warrant additional technical review. These reviews follow the same form as the "initial technical review," but is often less involved and covers a smaller area.

Review Methods If you're not a usual contributor to the documentation and would like to review something, you can submit reviews in any of the following methods:

- If you're reviewing an open pull request in GitHub, the best way to comment is on the "overview diff," which you can find by clicking on the "diff" button in the upper left portion of the screen. You can also use the following URL to reach this interface:

  ```
  https://github.com/mongodb/docs/pull/[pull-request-id]/files
  ```

  Replace `[pull-request-id]` with the identifier of the pull request. Make all comments inline, using GitHub's comment system.

  You may also provide comments directly on commits, or on the pull request itself but these commit-comments are archived in less coherent ways and generate less useful emails, while comments on the pull request lead to less specific changes to the document.

- Leave feedback on Jira cases in the DOCS[41] project. These are better for more general changes that aren't necessarily tied to a specific line, or affect multiple files.

- Create a fork of the repository in your GitHub account, make any required changes and then create a pull request with your changes.

  If you insert lines that begin with any of the following annotations:

  ```
  .. TODO:
  TODO:
  .. TODO
  TODO
  ```

  followed by your comments, it will be easier for the original writer to locate your comments. The two dots `..` format is a comment in reStructured Text, which will hide your comments from Sphinx and publication if you're worried about that.

  This format is often easier for reviewers with larger portions of content to review.

MongoDB Manual Organization

This document provides an overview of the global organization of the documentation resource. Refer to the notes below if you are having trouble understanding the reasoning behind a file's current location, or if you want to add new documentation but aren't sure how to integrate it into the existing resource.

If you have questions, don't hesitate to open a ticket in the Documentation Jira Project[42] or contact the documentation team[43].

[41] http://jira.mongodb.org/browse/DOCS
[42] https://jira.mongodb.org/browse/DOCS
[43] docs@mongodb.com

Global Organization

Indexes and Experience The documentation project has two "index files": `http://docs.mongodb.org/manual/contents.txt` and `http://docs.mongodb.org/manual/index.txt`. The "contents" file provides the documentation's tree structure, which Sphinx uses to create the left-pane navigational structure, to power the "Next" and "Previous" page functionality, and to provide all overarching outlines of the resource. The "index" file is not included in the "contents" file (and thus builds will produce a warning here) and is the page that users first land on when visiting the resource.

Having separate "contents" and "index" files provides a bit more flexibility with the organization of the resource while also making it possible to customize the primary user experience.

Topical Organization The placement of files in the repository depends on the *type* of documentation rather than the *topic* of the content. Like the difference between `contents.txt` and `index.txt`, by decoupling the organization of the files from the organization of the information the documentation can be more flexible and can more adequately address changes in the product and in users' needs.

Files in the `source/` directory represent the tip of a logical tree of documents, while *directories* are containers of types of content. The `administration` and `applications` directories, however, are legacy artifacts and with a few exceptions contain sub-navigation pages.

With several exceptions in the `reference/` directory, there is only one level of sub-directories in the `source/` directory.

Tools

The organization of the site, like all Sphinx sites derives from the `toctree`[44] structure. However, in order to annotate the table of contents and provide additional flexibility, the MongoDB documentation generates `toctree`[45] structures using data from YAML files stored in the `source/includes/` directory. These files start with `ref-toc` or `toc` and generate output in the `source/includes/toc/` directory. Briefly this system has the following behavior:

- files that start with `ref-toc` refer to the documentation of API objects (i.e. commands, operators and methods), and the build system generates files that hold `toctree`[46] directives as well as files that hold *tables* that list objects and a brief description.
- files that start with `toc` refer to all other documentation and the build system generates files that hold `toctree`[47] directives as well as files that hold *definition lists* that contain links to the documents and short descriptions the content.
- file names that have `spec` following `toc` or `ref-toc` will generate aggregated tables or definition lists and allow ad-hoc combinations of documents for landing pages and quick reference guides.

MongoDB Documentation Build System

This document contains more direct instructions for building the MongoDB documentation.

Getting Started

Install Dependencies The MongoDB Documentation project depends on the following tools:

[44] http://sphinx-doc.org/markup/toctree.html#directive-toctree
[45] http://sphinx-doc.org/markup/toctree.html#directive-toctree
[46] http://sphinx-doc.org/markup/toctree.html#directive-toctree
[47] http://sphinx-doc.org/markup/toctree.html#directive-toctree

- GNU Make
- GNU Tar
- Python
- Git
- Sphinx (documentation management toolchain)
- Pygments (syntax highlighting)
- PyYAML (for the generated tables)
- Droopy (Python package for static text analysis)
- Fabric (Python package for scripting and orchestration)
- Inkscape (Image generation.)
- python-argparse (For Python 2.6.)
- LaTeX/PDF LaTeX (typically texlive; for building PDFs)
- Common Utilities (rsync, tar, gzip, sed)

OS X Install Sphinx, Docutils, and their dependencies with `easy_install` the following command:

```
easy_install Sphinx Jinja2 Pygments docutils  PyYAML droopy fabric
```

Feel free to use `pip` rather than `easy_install` to install python packages.

To generate the images used in the documentation, download and install Inkscape[48].

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[49]. This is **only** required to build PDFs.

Arch Linux Install packages from the system repositories with the following command:

```
pacman -S python2-sphinx python2-yaml inkscape python2-pip
```

Then install the following Python packages:

```
pip install droopy fabric
```

Optional

To generate PDFs for the full production build, install the following packages from the system repository:

```
pacman -S texlive-bin texlive-core texlive-latexextra
```

Debian/Ubuntu Install the required system packages with the following command:

```
apt-get install python-sphinx python-yaml python-argparse inkscape python-pip
```

Then install the following Python packages:

[48] http://inkscape.org/download/
[49] http://www.tug.org/mactex/2011/

```
pip install droopy fabric
```

Optional

To generate PDFs for the full production build, install the following packages from the system repository:

```
apt-get install texlive-latex-recommended texlive-latex-recommended
```

Setup and Configuration Clone the repository:

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

Then run the `bootstrap.py` script in the `docs/` repository, to configure the build dependencies:

```
python bootstrap.py
```

This downloads and configures the mongodb/docs-tools[50] repository, which contains the authoritative build system shared between branches of the MongoDB Manual and other MongoDB documentation projects.

You can run `bootstrap.py` regularly to update build system.

Building the Documentation

The MongoDB documentation build system is entirely accessible via `make` targets. For example, to build an HTML version of the documentation issue the following command:

```
make html
```

You can find the build output in `build/<branch>/html`, where `<branch>` is the name of the current branch.

In addition to the `html` target, the build system provides the following targets:

`publish` Builds and integrates all output for the production build. Build output is in `build/public/<branch>/`. When you run `publish` in the `master`, the build will generate some output in `build/public/`.

`push`; `stage` Uploads the production build to the production or staging web servers. Depends on `publish`. Requires access production or staging environment.

`push-all`; `stage-all` Uploads the entire content of `build/public/` to the web servers. Depends on `publish`. Not used in common practice.

`push-with-delete`; `stage-with-delete` Modifies the action of `push` and `stage` to remove remote file that don't exist in the local build. Use with caution.

`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.

[50] http://github.com/mongodb/docs-tools/

Build Mechanics and Tools

Internally the build system has a number of components and processes. See the docs-tools README[51] for more information on the internals. This section documents a few of these components from a very high level and lists useful operations for contributors to the documentation.

Fabric Fabric is an orchestration and scripting package for Python. The documentation uses Fabric to handle the deployment of the build products to the web servers and also unifies a number of independent build operations. Fabric commands have the following form:

```
fab <module>.<task>[:<argument>]
```

The `<argument>` is optional in most cases. Additionally some tasks are available at the root level, without a module. To see a full list of fabric tasks, use the following command:

```
fab -l
```

You can chain fabric tasks on a single command line, although this doesn't always make sense.

Important fabric tasks include:

`tools.bootstrap` Runs the `bootstrap.py` script. Useful for re-initializing the repository without needing to be in root of the repository.

`tools.dev`; `tools.reset` `tools.dev` switches the `origin` remote of the `docs-tools` checkout in `build` directory, to `../docs-tools` to facilitate build system testing and development. `tools.reset` resets the `origin` remote for normal operation.

`tools.conf` `tools.conf` returns the content of the configuration object for the current project. These data are useful during development.

`stats.report:<filename>` Returns, a collection of readability statistics. Specify file names relative to `source/` tree.

`make` Provides a thin wrapper around Make calls. Allows you to start make builds from different locations in the project repository.

`process.refresh_dependencies` Updates the time stamp of `.txt` source files with changed include files, to facilitate Sphinx's incremental rebuild process. This task runs internally as part of the build process.

Buildcloth Buildcloth[52] is a meta-build tool, used to generate Makefiles programmatically. This makes the build system easier to maintain, and makes it easier to use the same fundamental code to generate various branches of the Manual as well as related documentation projects. See makecloth/ in the docs-tools repository[53] for the relevant code.

Running `make` with no arguments will regenerate these parts of the build system automatically.

Rstcloth Rstcloth[54] is a library for generating reStructuredText programmatically. This makes it possible to generate content for the documentation, such as tables, tables of contents, and API reference material programmatically and transparently. See rstcloth/ in the docs-tools repository[55] for the relevant code.

If you have any questions, please feel free to open a Jira Case[56].

[51] https://github.com/mongodb/docs-tools/blob/master/README.rst
[52] https://pypi.python.org/pypi/buildcloth/
[53] https://github.com/mongodb/docs-tools/tree/master/makecloth
[54] https://pypi.python.org/pypi/rstcloth
[55] https://github.com/mongodb/docs-tools/tree/master/rstcloth
[56] https://jira.mongodb.org/browse/DOCS

CHAPTER 2

Interfaces Reference

2.1 mongo Shell Methods

JavaScript in MongoDB

Although these methods use JavaScript, most interactions with MongoDB do not use JavaScript but use an idiomatic driver in the language of the interacting application.

2.1.1 Collection

Collection Methods

Name	Description
db.collection.aggregate() (page 22)	Provides access to the aggregation pipeline.
db.collection.copyTo() (page 25)	Wraps eval (page 237) to copy data between collections in a sing
db.collection.count() (page 26)	Wraps count (page 213) to return a count of the number of docu
db.collection.createIndex() (page 27)	Builds an index on a collection. Use db.collection.ensure
db.collection.dataSize() (page 28)	Returns the size of the collection. Wraps the size (page 349) fiel
db.collection.distinct() (page 28)	Returns an array of documents that have distinct values for the spe
db.collection.drop() (page 28)	Removes the specified collection from the database.
db.collection.dropIndex() (page 29)	Removes a specified index on a collection.
db.collection.dropIndexes() (page 30)	Removes all indexes on a collection.
db.collection.ensureIndex() (page 30)	Creates an index if it does not currently exist. If the index exists e
db.collection.explain() (page 33)	Returns information on the query execution of various methods.
db.collection.find() (page 36)	Performs a query on a collection and returns a cursor object.
db.collection.findAndModify() (page 42)	Atomically modifies and returns a single document.
db.collection.findOne() (page 46)	Performs a query and returns a single document.
db.collection.getIndexStats() (page 47)	Renders a human-readable view of the data collected by indexSt
db.collection.getIndexes() (page 48)	Returns an array of documents that describe the existing indexes o
db.collection.getShardDistribution() (page 49)	For collections in sharded clusters, db.collection.getShar
db.collection.getShardVersion() (page 51)	Internal diagnostic method for shard cluster.
db.collection.group() (page 51)	Provides simple data aggregation function. Groups documents in a
db.collection.indexStats() (page 54)	Renders a human-readable view of the data collected by indexSt
db.collection.insert() (page 55)	Creates a new document in a collection.
db.collection.isCapped() (page 58)	Reports if a collection is a *capped collection*.
db.collection.mapReduce() (page 58)	Performs map-reduce style data aggregation.

Table 2.1 – continued from pre

Name	Description
db.collection.reIndex() (page 65)	Rebuilds all existing indexes on a collection.
db.collection.remove() (page 66)	Deletes documents from a collection.
db.collection.renameCollection() (page 69)	Changes the name of a collection.
db.collection.save() (page 70)	Provides a wrapper around an insert() (page 55) and update
db.collection.stats() (page 71)	Reports on the state of a collection. Provides a wrapper around the
db.collection.storageSize() (page 72)	Reports the total size used by the collection in bytes. Provides a wr
db.collection.totalIndexSize() (page 72)	Reports the total size used by the indexes on a collection. Provides
db.collection.totalSize() (page 72)	Reports the total size of a collection, including the size of all docu
db.collection.update() (page 72)	Modifies a document in a collection.
db.collection.validate() (page 79)	Performs diagnostic operations on a collection.

db.collection.aggregate()

New in version 2.2.

Definition

db.collection.**aggregate**(*pipeline*, *options*)

Calculates aggregate values for the data in a collection.

param array pipeline A sequence of data aggregation operations or stages. See the *aggregation pipeline operators* (page 484) for details.

Changed in version 2.6: The method can still accept the pipeline stages as separate arguments instead of as elements in an array; however, if you do not specify the pipeline as an array, you cannot specify the options parameter.

param document options Additional options that aggregate() (page 22) passes to the aggregate (page 210) command.

New in version 2.6: Available only if you specify the pipeline as an array.

The options document can contain the following fields and values:

field boolean explain Specifies to return the information on the processing of the pipeline. See *Return Information on Aggregation Pipeline Operation* (page 24) for an example.

New in version 2.6.

field boolean allowDiskUse Enables writing to temporary files. When set to true, aggregation operations can write data to the _tmp subdirectory in the dbPath directory. See *Perform Large Sort Operation with External Sort* (page 24) for an example.

New in version 2.6.

field document cursor Specifies the *initial* batch size for the cursor. The value of the cursor field is a document with the field batchSize. See *Specify an Initial Batch Size* (page 24) for syntax and example.

New in version 2.6.

Returns

A *cursor* to the documents produced by the final stage of the aggregation pipeline operation, or if you include the explain option, the document that provides details on the processing of the aggregation operation.

If the pipeline includes the $out (page 491) operator, aggregate() (page 22) returns an empty cursor. See $out (page 491) for more information.

Changed in version 2.6: The db.collection.aggregate() (page 22) method returns a cursor and can return result sets of any size. Previous versions returned all results in a single document, and the result set was subject to a size limit of 16 megabytes.

Changed in version 2.4: If an error occurs, the aggregate() (page 22) helper throws an exception. In previous versions, the helper returned a document with the error message and code, and ok status field not equal to 1, same as the aggregate (page 210) command.

See also:

For more information, see http://docs.mongodb.org/manual/core/aggregation-pipeline, *Aggregation Reference* (page 564), http://docs.mongodb.org/manual/core/aggregation-pipeline-limits, and aggregate (page 210).

Cursor Behavior In the mongo (page 610) shell, if the cursor returned from the db.collection.aggregate() (page 22) is not assigned to a variable using the var keyword, then the mongo (page 610) shell automatically iterates the cursor up to 20 times. See http://docs.mongodb.org/manual/core/cursors for cursor behavior in the mongo (page 610) shell and http://docs.mongodb.org/manual/tutorial/iterate-a-cursor for handling cursors in the mongo (page 610) shell.

Cursors returned from aggregation only supports cursor methods that operate on evaluated cursors (i.e. cursors whose first batch has been retrieved), such as the following methods:

- cursor.hasNext() (page 87)
- cursor.next() (page 93)
- cursor.toArray() (page 99)
- cursor.forEach() (page 86)
- cursor.map() (page 88)
- cursor.objsLeftInBatch() (page 93)
- cursor.itcount()
- cursor.pretty()

Examples The examples in this section use the db.collection.aggregate() (page 22) helper provided in the 2.6 version of the mongo (page 610) shell.

The following examples use the collection orders that contains the following documents:

```
{ _id: 1, cust_id: "abc1", ord_date: ISODate("2012-11-02T17:04:11.102Z"), status: "A", amount: 50 }
{ _id: 2, cust_id: "xyz1", ord_date: ISODate("2013-10-01T17:04:11.102Z"), status: "A", amount: 100 }
{ _id: 3, cust_id: "xyz1", ord_date: ISODate("2013-10-12T17:04:11.102Z"), status: "D", amount: 25 }
{ _id: 4, cust_id: "xyz1", ord_date: ISODate("2013-10-11T17:04:11.102Z"), status: "D", amount: 125 }
{ _id: 5, cust_id: "abc1", ord_date: ISODate("2013-11-12T17:04:11.102Z"), status: "A", amount: 25 }
```

Group by and Calculate a Sum The following aggregation operation selects documents with status equal to "A", groups the matching documents by the cust_id field and calculates the total for each cust_id field from the sum of the amount field, and sorts the results by the total field in descending order:

```
db.orders.aggregate([
                      { $match: { status: "A" } },
                      { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
                      { $sort: { total: -1 } }
                    ])
```

The operation returns a cursor with the following documents:

```
{ "_id" : "xyz1", "total" : 100 }
{ "_id" : "abc1", "total" : 75 }
```

The mongo (page 610) shell iterates the returned cursor automatically to print the results. See http://docs.mongodb.org/manual/tutorial/iterate-a-cursor for handling cursors manually in the mongo (page 610) shell.

Return Information on Aggregation Pipeline Operation The following aggregation operation sets the option explain to true to return information about the aggregation operation.

```
db.orders.aggregate(
                      [
                        { $match: { status: "A" } },
                        { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
                        { $sort: { total: -1 } }
                      ],
                      {
                        explain: true
                      }
                    )
```

The operation returns a cursor with the document that contains detailed information regarding the processing of the aggregation pipeline. For example, the document may show, among other details, which index, if any, the operation used. [1] If the orders collection is a sharded collection, the document would also show the division of labor between the shards and the merge operation, and for targeted queries, the targeted shards.

Note: The intended readers of the explain output document are humans, and not machines, and the output format is subject to change between releases.

The mongo (page 610) shell iterates the returned cursor automatically to print the results. See http://docs.mongodb.org/manual/tutorial/iterate-a-cursor for handling cursors manually in the mongo (page 610) shell.

Perform Large Sort Operation with External Sort Aggregation pipeline stages have *maximum memory use limit*. To handle large datasets, set allowDiskUse option to true to enable writing data to temporary files, as in the following example:

```
var results = db.stocks.aggregate(
                                    [
                                      { $project : { cusip: 1, date: 1, price: 1, _id: 0 } },
                                      { $sort : { cusip : 1, date: 1 } }
                                    ],
                                    {
                                      allowDiskUse: true
                                    }
                                  )
```

Specify an Initial Batch Size To specify an initial batch size for the cursor, use the following syntax for the cursor option:

[1] *index-filters* can affect the choice of index used. See *index-filters* for details.

```
cursor: { batchSize: <int> }
```

For example, the following aggregation operation specifies the *initial* batch size of `0` for the cursor:

```
db.orders.aggregate(
                      [
                        { $match: { status: "A" } },
                        { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
                        { $sort: { total: -1 } },
                        { $limit: 2 }
                      ],
                      {
                        cursor: { batchSize: 0 }
                      }
                    )
```

A `batchSize` of `0` means an empty first batch and is useful for quickly returning a cursor or failure message without doing significant server-side work. Specify subsequent batch sizes to *OP_GET_MORE*[2] operations as with other MongoDB cursors.

The `mongo` (page 610) shell iterates the returned cursor automatically to print the results. See `http://docs.mongodb.org/manual/tutorial/iterate-a-cursor` for handling cursors manually in the `mongo` (page 610) shell.

db.collection.copyTo()

Definition

`db.collection.`**`copyTo`**(*newCollection*)

Copies all documents from `collection` into `newCollection` using server-side JavaScript. If `newCollection` does not exist, MongoDB creates it.

If authorization is enabled, you must have access to all actions on all resources in order to run `db.collection.copyTo()` (page 25). Providing such access is not recommended, but if your organization requires a user to run `db.collection.copyTo()` (page 25), create a role that grants `anyAction` on *resource-anyresource*. Do not assign this role to any other user.

param string newCollection The name of the collection to write data to.

> **Warning:** When using `db.collection.copyTo()` (page 25) 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()` (page 101) to maintain type fidelity.
> The `db.collection.copyTo()` (page 25) method uses the `eval` (page 237) command internally. As a result, the `db.collection.copyTo()` (page 25) operation takes a global lock that blocks all other read and write operations until the `db.collection.copyTo()` (page 25) completes.

`copyTo()` (page 25) returns the number of documents copied. If the copy fails, it throws an exception.

Behavior Because `copyTo()` (page 25) uses `eval` (page 237) internally, the copy operations will block all other operations on the `mongod` (page 583) instance.

Example The following operation copies all documents from the `source` collection into the `target` collection.

[2] http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol/#wire-op-get-more

```
db.source.copyTo(target)
```

db.collection.count()

Definition

`db.collection.count(<query>)`

Returns the count of documents that would match a `find()` (page 36) query. The `db.collection.count()` (page 26) method does not perform the `find()` (page 36) operation but instead counts and returns the number of results that match a query.

The `db.collection.count()` (page 26) method has the following parameter:

param document query The query selection criteria.

The `db.collection.count()` (page 26) method is equivalent to the `db.collection.find(<query>).count()` construct.

See also:

`cursor.count()` (page 83)

Behavior

Sharded Clusters On a sharded cluster, `db.collection.count()` (page 26) can result in an *inaccurate* count if *orphaned documents* exist or if a `chunk migration` is in progress.

To avoid these situations, on a sharded cluster, use the `$group` (page 486) stage of the `db.collection.aggregate()` (page 22) method to `$sum` (page 554) the documents. For example, the following operation counts the documents in a collection:

```
db.collection.aggregate(
   [
      { $group: { _id: null, count: { $sum: 1 } } }
   ]
)
```

To get a count of documents that match a query condition, include the `$match` (page 490) stage as well:

```
db.collection.aggregate(
   [
      { $match: <query condition> },
      { $group: { _id: null, count: { $sum: 1 } } }
   ]
)
```

See *Perform a Count* (page 491) for an example.

Index Use Consider a collection with the following index:

```
{ a: 1, b: 1 }
```

When performing a count, MongoDB can return the count using only the index if:

- the query can use an index,
- the query only contains conditions on the keys of the index, *and*
- the query predicates access a single contiguous range of index keys.

For example, the following operations can return the count using only the index:

```
db.collection.find( { a: 5, b: 5 } ).count()
db.collection.find( { a: { $gt: 5 } } ).count()
db.collection.find( { a: 5, b: { $gt: 10 } } ).count()
```

If, however, the query can use an index but the query predicates do not access a single contiguous range of index keys or the query also contains conditions on fields outside the index, then in addition to using the index, MongoDB must also read the documents to return the count.

```
db.collection.find( { a: 5, b: { $in: [ 1, 2, 3 ] } } ).count()
db.collection.find( { a: { $gt: 5 }, b: 5 } ).count()
db.collection.find( { a: 5, b: 5, c: 5 } ).count()
```

In such cases, during the initial read of the documents, MongoDB pages the documents into memory such that subsequent calls of the same count operation will have better performance.

Examples

Count all Documents in a Collection To count the number of all documents in the `orders` collection, use the following operation:

```
db.orders.count()
```

This operation is equivalent to the following:

```
db.orders.find().count()
```

Count all Documents that Match a Query Count the number of the documents in the `orders` collection with the field `ord_dt` greater than `new Date('01/01/2012')`:

```
db.orders.count( { ord_dt: { $gt: new Date('01/01/2012') } } )
```

The query is equivalent to the following:

```
db.orders.find( { ord_dt: { $gt: new Date('01/01/2012') } } ).count()
```

db.collection.createIndex()

Definition

db.collection.**createIndex**(*keys*, *options*)

Deprecated since version 1.8.

Creates indexes on collections.

param document keys For each field to index, a key-value pair with the field and the index order: `1` for ascending or `-1` for descending.

param document options One or more key-value pairs that specify index options. For a list of options, see `db.collection.ensureIndex()` (page 30).

See also:

`http://docs.mongodb.org/manual/indexes`, `db.collection.createIndex()` (page 27), `db.collection.dropIndex()` (page 29), `db.collection.dropIndexes()` (page 30), `db.collection.getIndexes()` (page 48), `db.collection.reIndex()` (page 65), and `db.collection.totalIndexSize()` (page 72)

db.collection.dataSize()

db.collection.**dataSize**()

Returns The size of the collection. This method provides a wrapper around the `size` (page 349) output of the `collStats` (page 348) (i.e. `db.collection.stats()` (page 71)) command.

db.collection.distinct()

Definition

db.collection.**distinct**(*field*, *query*)

Finds the distinct values for a specified field across a single collection and returns the results in an array.

param string field The field for which to return distinct values.

param document query A query that specifies the documents from which to retrieve the distinct values.

The `db.collection.distinct()` (page 28) method provides a wrapper around the `distinct` (page 215) command. Results must not be larger than the maximum *BSON size* (page 692).

When possible to use covered indexes, the `db.collection.distinct()` (page 28) method will use an index to find the documents in the query as well as to return the data.

Examples The following are examples of the `db.collection.distinct()` (page 28) method:

- Return an array of the distinct values of the field `ord_dt` from all documents in the `orders` collection:

```
db.orders.distinct( 'ord_dt' )
```

- Return an array of the distinct values of the field `sku` in the subdocument `item` from all documents in the `orders` collection:

```
db.orders.distinct( 'item.sku' )
```

- Return an array of the distinct values of the field `ord_dt` from the documents in the `orders` collection where the `price` is greater than `10`:

```
db.orders.distinct( 'ord_dt', { price: { $gt: 10 } } )
```

db.collection.drop()

Definition

db.collection.**drop**()

Removes a collection from the database. The method also removes any indexes associated with the dropped collection. The method provides a wrapper around the `drop` (page 335) command.

`db.collection.drop()` (page 28) has the form:

```
db.collection.drop()
```

`db.collection.drop()` (page 28) takes no arguments and will produce an error if called with any arguments.

Returns

- `true` when successfully drops a collection.

- `false` when collection to drop does not exist.

Behavior This method obtains a write lock on the affected database and will block other operations until it has completed.

Example The following operation drops the `students` collection in the current database.

```
db.students.drop()
```

db.collection.dropIndex()

Definition

`db.collection.dropIndex(`*index*`)`

Drops or removes the specified index from a collection. The `db.collection.dropIndex()` (page 29) method provides a wrapper around the `dropIndexes` (page 335) command.

Note: You cannot drop the default index on the `_id` field.

The `db.collection.dropIndex()` (page 29) method takes the following parameter:

param string,document index Specifies the index to drop. You can specify the index either by the index name or by the index specification document. [3]

To drop a `text` index, specify the index name.

To get the index name or the index specification document for the `db.collection.dropIndex()` (page 29) method, use the `db.collection.getIndexes()` (page 48) method.

Example Consider a `pets` collection. Calling the `getIndexes()` (page 48) method on the `pets` collection returns the following indexes:

```
[
   {  "v" : 1,
      "key" : { "_id" : 1 },
      "ns" : "test.pets",
      "name" : "_id_"
   },
   {
      "v" : 1,
      "key" : { "cat" : -1 },
      "ns" : "test.pets",
      "name" : "catIdx"
   },
   {
      "v" : 1,
      "key" : { "cat" : 1, "dog" : -1 },
      "ns" : "test.pets",
      "name" : "cat_1_dog_-1"
   }
]
```

[3] When using a `mongo` (page 610) shell version earlier than 2.2.2, if you specified a name during the index creation, you must use the name to drop the index.

The single field index on the field `cat` has the user-specified name of `catIdx` [4] and the index specification document of `{ "cat" : -1 }`.

To drop the index `catIdx`, you can use either the index name:

```
db.pets.dropIndex( "catIdx" )
```

Or you can use the index specification document `{ "cat" : -1 }`:

```
db.pets.dropIndex( { "cat" : -1 } )
```

db.collection.dropIndexes()

`db.collection.`**`dropIndexes`**`()`

Drops all indexes other than the required index on the `_id` field. Only call `dropIndexes()` (page 30) as a method on a collection object.

db.collection.ensureIndex()

Definition

`db.collection.`**`ensureIndex`**`(`*keys*, *options*`)`

Creates an index on the specified field if the index does not already exist.

The `ensureIndex()` (page 30) method has the following fields:

param document keys A document that contains the field and value pairs where the field is the index key and the value describes the type of index for that field. For an ascending index on a field, specify a value of `1`; for descending index, specify a value of `-1`.

MongoDB supports several different index types including *text*, `geospatial`, and *hashed* indexes. See *index-type-list* for more information.

param document options A document that contains a set of options that controls the creation of the index. See *Options* (page 30) for details.

Options The `options` document contains a set of options that controls the creation of the index. Different index types can have additional options specific for that type.

Options for All Index Types The following options are available for all index types unless otherwise specified:

param Boolean background Builds the index in the background so that building an index does *not* block other database activities. Specify `true` to build in the background. The default value is `false`.

param Boolean unique Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify `true` to create a unique index. The default value is `false`.

The option is *unavailable* for `hashed` indexes.

param string name The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.

Whether user specified or MongoDB generated, index names including their full namespace (i.e. `database.collection`) cannot be longer than the `Index Name Limit` (page 693).

[4] During index creation, if the user does **not** specify an index name, the system generates the name by concatenating the index key field and value with an underscore, e.g. `cat_1`.

param Boolean dropDups Creates a unique index on a field that *may* have duplicates. MongoDB indexes only the first occurrence of a key and **removes** all documents from the collection that contain subsequent occurrences of that key. Specify `true` to create unique index. The default value is `false`.

The option is *unavailable* for `hashed` indexes.

Deprecated since version 2.6.

> **Warning:** `dropDups` will delete data from your collection when building the index.

param Boolean sparse If `true`, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is `false`. See `http://docs.mongodb.org/manual/core/index-sparse` for more information.

Changed in version 2.6: `2dsphere` indexes are sparse by default and ignore this option. For a compound index that includes `2dsphere` index key(s) along with keys of other types, only the `2dsphere` index fields determine whether the index references a document.

`2d`, `geoHaystack`, and `text` indexes behave similarly to the `2dsphere` indexes.

param integer expireAfterSeconds Specifies a value, in seconds, as a *TTL* to control how long MongoDB retains documents in this collection. See `http://docs.mongodb.org/manual/tutorial/expire-data` for more information on this functionality. This applies only to *TTL* indexes.

param index version v The index version number. The default index version depends on the version of `mongod` (page 583) running when creating the index. Before version 2.0, the this value was 0; versions 2.0 and later use version 1, which provides a smaller and faster index format. Specify a different index version *only* in unusual situations.

param document storageEngine New in version 2.8.

Allows users to specify configuration to the storage engine on a per-index basis when creating an index. The value of the `storageEngine` option should take the following form:

```
{ <storage-engine-name>: <options> }
```

Storage engine configuration specified when creating indexes are validated and logged to the *oplog* during replication to support replica sets with members that use different storage engines.

Options for `text` Indexes The following options are available for `text` indexes only:

param document weights For `text` indexes, a document that contains field and weight pairs. The weight is an integer ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score. You can specify weights for some or all the indexed fields. See `http://docs.mongodb.org/manual/tutorial/control-results-of-text-search` to adjust the scores. The default value is `1`.

param string default_language For `text` indexes, the language that determines the list of stop words and the rules for the stemmer and tokenizer. See *text-search-languages* for the available languages and `http://docs.mongodb.org/manual/tutorial/specify-language-for-text-index` for more information and examples. The default value is `english`.

param string language_override For `text` indexes, the name of the field, in the collection's documents, that contains the override language for the document. The default value is `language`. See *specify-language-field-text-index-example* for an example.

param integer textIndexVersion For `text` indexes, the `text` index version number. Version can be either `1` or `2`.

In MongoDB 2.6, the default version is `2`. MongoDB 2.4 can only support version `1`.

New in version 2.6.

Options for `2dsphere` Indexes The following option is available for `2dsphere` indexes only:

param integer 2dsphereIndexVersion For `2dsphere` indexes, the `2dsphere` index version number. Version can be either `1` or `2`.

In MongoDB 2.6, the default version is `2`. MongoDB 2.4 can only support version `1`.

New in version 2.6.

Options for `2d` Indexes The following options are available for `2d` indexes only:

param integer bits For `2d` indexes, the number of precision of the stored *geohash* value of the location data.

The `bits` value ranges from 1 to 32 inclusive. The default value is `26`.

param number min For `2d` indexes, the lower inclusive boundary for the longitude and latitude values. The default value is `-180.0`.

param number max For `2d` indexes, the upper inclusive boundary for the longitude and latitude values. The default value is `180.0`.

Options for `geoHaystack` Indexes The following option is available for `geoHaystack` indexes only:

param number bucketSize For `geoHaystack` indexes, specify the number of units within which to group the location values; i.e. group in the same bucket those location values that are within the specified number of units to each other.

The value must be greater than 0.

Behaviors The `ensureIndex()` (page 30) method has the behaviors described here.

- To add or change index options you must drop the index using the `dropIndex()` (page 29) method and issue another `ensureIndex()` (page 30) operation with the new options.

  If you create an index with one set of options, and then issue the `ensureIndex()` (page 30) method with the same index fields and different options without first dropping the index, `ensureIndex()` (page 30) will *not* rebuild the existing index with the new options.

- If you call multiple `ensureIndex()` (page 30) methods with the same index specification at the same time, only the first operation will succeed, all other operations will have no effect.

- Non-background indexing operations will block all other operations on a database.

- MongoDB will **not** `create an index` (page 30) on a collection if the index entry for an existing document exceeds the `Maximum Index Key Length`. Previous versions of MongoDB would create the index but not index such documents.

  Changed in version 2.6.

Examples

Create an Ascending Index on a Single Field The following example creates an ascending index on the field `orderDate`.

```
db.collection.ensureIndex( { orderDate: 1 } )
```

If the `keys` document specifies more than one field, then `ensureIndex()` (page 30) creates a *compound index*.

Create an Index on a Multiple Fields The following example creates a compound index on the `orderDate` field (in ascending order) and the `zipcode` field (in descending order.)

```
db.collection.ensureIndex( { orderDate: 1, zipcode: -1 } )
```

A compound index cannot include a *hashed index* component.

Note: The order of an index is important for supporting `sort()` (page 95) operations using the index.

See also:

- The `http://docs.mongodb.org/manual/indexes` section of this manual for full documentation of indexes and indexing in MongoDB.
- `http://docs.mongodb.org/manual/core/index-text` for details on creating `text` indexes.
- *index-feature-geospatial* and *index-geohaystack-index* for geospatial queries.
- *index-feature-ttl* for expiration of data.
- `db.collection.getIndexes()` (page 48) to view the specifications of existing indexes for a collection.

db.collection.explain()

Description

`db.collection.explain()`

New in version 2.8.

Returns information on the query plan for the following operations: `aggregate()` (page 22); `count()` (page 26); `find()` (page 36); `group()` (page 51); `remove()` (page 66); and `update()` (page 72) methods.

To use `db.collection.explain()` (page 33), append to `db.collection.explain()` (page 33) the method(s) available to explain:

```
db.collection.explain().<method(...)>
```

For example,

```
db.products.explain().remove( { category: "apparel" }, { justOne: true } )
```

For more examples, see *Examples* (page 35). For a list of methods available for use with `db.collection.explain()` (page 33), see *db.collection.explain().help()* (page 35).

The `db.collection.explain()` (page 33) method has the following parameter:

param string verbosity Specifies the verbosity mode for the explain output. The mode affects the behavior of `explain()` and determines the amount of information to return. The possible modes are: `"queryPlanner"`, `"executionStats"`, and `"allPlansExecution"`.

Default mode is `"queryPlanner"`.

For backwards compatibility with earlier versions of `cursor.explain()` (page 85), MongoDB interprets `true` as `"allPlansExecution"` and `false` as `"queryPlanner"`.

For more information on the modes, see *Verbosity Modes* (page 34).

Behavior

Verbosity Modes The behavior of `db.collection.explain()` (page 33) and the amount of information returned depend on the `verbosity` mode.

`queryPlanner` Mode By default, `db.collection.explain()` (page 33) runs in `queryPlanner` verbosity mode.

MongoDB runs the `query optimizer` to choose the winning plan for the operation under evaluation. `db.collection.explain()` (page 33) returns the `queryPlanner` information for the evaluated method.

`executionStats` Mode MongoDB runs the `query optimizer` to choose the winning plan, executes the winning plan to completion, and returns statistics describing the execution of the winning plan.

For write operations, `db.collection.explain()` (page 33) returns information about the update or delete operations that *would* be performed, but does *not* apply the modifications to the database.

`db.collection.explain()` (page 33) returns the `queryPlanner` and `executionStats` information for the evaluated method. However, `executionStats` does not provide query execution information for the rejected plans.

`allPlansExecution` Mode MongoDB runs the `query optimizer` to choose the winning plan and executes the winning plan to completion. In `"allPlansExecution"` mode, MongoDB returns statistics describing the execution of the winning plan as well as statistics for the other candidate plans captured during *plan selection*.

For write operations, `db.collection.explain()` (page 33) returns information about the update or delete operations that *would* be performed, but does *not* apply the modifications to the database.

`db.collection.explain()` (page 33) returns the `queryPlanner` and `executionStats` information for the evaluated method. The `executionStats` includes the *completed* query execution information for the *winning plan*.

If the query optimizer considered more than one plan, `executionStats` information also includes the *partial* execution information captured during the *plan selection phase* for both the winning and rejected candidate plans.

`explain()` Mechanics The `db.collection.explain()` (page 33) method wraps the `explain` (page 354) command and is the preferred way to run `explain` (page 354).

`db.collection.explain().find()` is similar to `db.collection.find().explain()` (page 85) with the following key differences:

- The `db.collection.explain().find()` construct allows for the additional chaining of query modifiers. For list of query modifiers, see *db.collection.explain().find().help()* (page 35).
- The `db.collection.explain().find()` returns a cursor, which requires a call to `.next()`, or its alias `.finish()`, to return the `explain()` results. If run interactively in the `mongo` (page 610) shell, the `mongo` (page 610) shell automatically calls `.finish()` to return the results. For scripts, however, you must explicitly call `.next()`, or `.finish()`, to return the results. For list of cursor-related methods, see *db.collection.explain().find().help()* (page 35).

`db.collection.explain().aggregate()` is equivalent to passing the *explain* (page 24) option to the `db.collection.aggregate()` (page 22) method.

`help()` To see the list of operations supported by `db.collection.explain()` (page 33), run:

```
db.collection.explain().help()
```

`db.collection.explain().find()` returns a cursor, which allows for the chaining of query modifiers. To see the list of query modifiers supported by `db.collection.explain().find()` (page 33) as well as cursor-related methods, run:

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

You can chain multiple modifiers to `db.collection.explain().find()`. For an example, see *Explain find() with Modifiers* (page 35).

Examples

`queryPlanner` Mode By default, `db.collection.explain()` (page 33) runs in `"queryPlanner"` verbosity mode.

The following example runs `db.collection.explain()` (page 33) in *"queryPlanner"* (page 34) verbosity mode to return the query planning information for the specified `count()` (page 26) operation:

```
db.products.explain().count( { quantity: { $gt: 50 } } )
```

`executionStats` Mode The following example runs `db.collection.explain()` (page 33) in *"executionStats"* (page 34) verbosity mode to return the query planning and execution information for the specified `find()` (page 36) operation:

```
db.products.explain("executionStats").find(
   { quantity: { $gt: 50 }, category: "apparel" }
)
```

`allPlansExecution` Mode The following example runs `db.collection.explain()` (page 33) in *"allPlansExecution"* (page 34) verbosity mode. The `db.collection.explain()` (page 33) returns the `queryPlanner` and `executionStats` for all considered plans for the specified `update()` (page 72) operation:

Note: The execution of this explain will *not* modify data but runs the query predicate of the update operation. For candidate plans, MongoDB returns the execution information captured during the *plan selection phase*.

```
db.products.explain("allPlansExecution").update(
   { quantity: { $lt: 1000}, category: "apparel" },
   { $set: { reorder: true } }
)
```

Explain `find()` with Modifiers `db.collection.explain().find()` construct allows for the chaining of query modifiers. For example, the following operation provides information on the `find()` (page 36) method with `sort()` (page 95) and `hint()` (page 87) query modifiers.

```
db.products.explain("executionStats").find(
   { quantity: { $gt: 50 }, category: "apparel" }
).sort( { quantity: -1 } ).hint( { category: 1, quantity: -1 } )
```

For a list of query modifiers available, run in the mongo (page 610) shell:

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

Iterate the `explain().find()` Return Cursor `db.collection.explain().find()` returns a cursor to the explain results. If run interactively in the mongo (page 610) shell, the mongo (page 610) shell automatically iterates the cursor using the `.next()` method. For scripts, however, you must explicitly call `.next()` (or its alias `.finish()`) to return the results:

```
var explainResult = db.products.explain().find( { category: "apparel" } ).next();
```

Output db.collection.explain() (page 33) operations can return information regarding:

- *queryPlanner*, which details the plan selected by the `query optimizer` and lists the rejected plans;
- *executionStats*, which details the execution of the winning plan and the rejected plans; and
- *serverInfo*, which provides information on the MongoDB instance.

The verbosity mode (i.e. `queryPlanner`, `executionStats`, `allPlansExecution`) determines whether the results include *executionStats* and whether *executionStats* includes data captured during *plan selection*.

For details on the output, see `http://docs.mongodb.org/manual/reference/explain-results`.

For a mixed version sharded cluster with version 2.8 mongos (page 601) and at least one 2.6 mongod (page 583) shard, when you run db.collection.explain() (page 33) in a version 2.8 mongo (page 610) shell, db.collection.explain() (page 33) will retry with the $explain (page 556) operator to return results in the 2.6 format.

db.collection.find()

Definition

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

Selects documents in a collection and returns a *cursor* to the selected documents. [5]

param document criteria Specifies selection criteria using *query operators* (page 400). To return all documents in a collection, omit this parameter or pass an empty document (`{}`).

param document projection Specifies the fields to return using *projection operators* (page 444). To return all fields in the matching document, omit this parameter.

Returns

A *cursor* to the documents that match the `query` criteria. When the find() (page 36) method "returns documents," the method is actually returning a cursor to the documents.

If the `projection` argument is specified, the matching documents contain only the `projection` fields and the `_id` field. You can optionally exclude the `_id` field.

Executing find() (page 36) directly in the mongo (page 610) shell automatically iterates the cursor to display up to the first 20 documents. Type `it` to continue iteration.

[5] db.collection.find() (page 36) is a wrapper for the more formal query structure that uses the $query (page 560) operator.

To access the returned documents with a driver, use the appropriate cursor handling mechanism for the driver language.

The projection parameter takes a document of the following form:

```
{ field1: <boolean>, field2: <boolean> ... }
```

The <boolean> value can be any of the following:

- 1 or true to include the field. The find() (page 36) method always includes the *_id* field even if the field is not explicitly stated to return in the *projection* parameter.
- 0 or false to exclude the field.

A projection *cannot* contain *both* include and exclude specifications, except for the exclusion of the _id field. In projections that *explicitly include* fields, the _id field is the only field that you can *explicitly exclude*.

Examples

Find All Documents in a Collection The find() (page 36) method with no parameters returns all documents from a collection and returns all fields for the documents. For example, the following operation returns all documents in the bios collection:

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

Find Documents that Match Query Criteria To find documents that match a set of selection criteria, call find() with the <criteria> parameter. The following operation returns all the documents from the collection products where qty is greater than 25:

```
db.products.find( { qty: { $gt: 25 } } )
```

Query for Equality The following operation returns documents in the bios collection where _id equals 5:

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

Query Using Operators The following operation returns documents in the bios collection where _id equals either 5 or ObjectId("507c35dd8fada716c89d0013"):

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

Query for Ranges Combine comparison operators to specify ranges. The following operation returns documents with field between value1 and value2:

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

Query a Field that Contains an Array If a field contains an array and your query has multiple conditional operators, the field as a whole will match if either a single array element meets the conditions or a combination of array elements meet the conditions.

Given a collection `students` that contains the following documents:

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

The following query:

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

Matches the following documents:

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

In the document with `_id` equal to `1`, the `score: [ -1, 3 ]` meets the conditions because the element `-1` meets the `$lt: 2` condition and the element `3` meets the `$gt: 0` condition.

In the document with `_id` equal to `2`, the `score: [ 1, 5 ]` meets the conditions because the element `1` meets both the `$lt: 2` condition and the `$gt: 0` condition.

Query Arrays

Query for an Array Element The following operation returns documents in the `bios collection` where the array field `contribs` contains the element `"UNIX"`:

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

Query an Array of Documents The following operation returns 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 }
                }
      }
   }
)
```

Query Subdocuments

Query Exact Matches on Subdocuments The following operation returns 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. The query does **not** match documents with the following `name` fields:

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

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

Query Fields of a Subdocument The following operation returns 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"
}
```

Projections The `projection` parameter specifies which fields to return. The parameter contains either include or exclude specifications, not both, unless the exclude is for the `_id` field.

Specify the Fields to Return The following operation returns all the documents from the `products` collection where `qty` is greater than `25` and returns only the `_id`, `item` and `qty` fields:

```
db.products.find( { qty: { $gt: 25 } }, { item: 1, qty: 1 } )
```

The operation returns the following:

```
{ "_id" : 11, "item" : "pencil", "qty" : 50 }
{ "_id" : ObjectId("50634d86be4617f17bb159cd"), "item" : "bottle", "qty" : 30 }
{ "_id" : ObjectId("50634dbcbe4617f17bb159d0"), "item" : "paper", "qty" : 100 }
```

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

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

Explicitly Excluded Fields The following operation queries the `bios collection` and returns all fields *except* the `first` field in the `name` subdocument and the `birth` field:

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

Explicitly Exclude the `_id` Field The following operation excludes the `_id` and `qty` fields from the result set:

```
db.products.find( { qty: { $gt: 25 } }, { _id: 0, qty: 0 } )
```

The documents in the result set contain all fields *except* the `_id` and `qty` fields:

```
{ "item" : "pencil", "type" : "no.2" }
{ "item" : "bottle", "type" : "blue" }
{ "item" : "paper" }
```

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

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

On Arrays and Subdocuments The following operation queries the `bios collection` and returns the `last` field in the `name` subdocument and the first two elements in the `contribs` array:

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

Iterate the Returned Cursor The `find()` (page 36) method returns a *cursor* to the results. In the `mongo` (page 610) shell, if the returned cursor is not assigned to a variable using the `var` keyword, the cursor is automatically iterated up to 20 times to access up to the first 20 documents that match the query. You can use the `DBQuery.shellBatchSize` to change the number of iterations. See *Flags* (page 82) and *cursor-behaviors*. To iterate manually, assign the returned cursor to a variable using the `var` keyword.

With Variable Name The following example uses the variable `myCursor` to iterate over the cursor and print the matching documents:

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

myCursor
```

With `next()` Method The following example uses the cursor method `next()` (page 93) to access the documents:

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

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

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

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

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

With `forEach()` Method The following example uses the cursor method `forEach()` (page 86) to iterate the cursor and access the documents:

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

myCursor.forEach(printjson);
```

Modify the Cursor Behavior The `mongo` (page 610) shell and the `drivers` provide several cursor methods that call on the *cursor* returned by the `find()` (page 36) method to modify its behavior.

Order Documents in the Result Set The `sort()` (page 95) method orders the documents in the result set. The following operation returns documents in the `bios collection` sorted in ascending order by the `name` field:

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

`sort()` (page 95) corresponds to the `ORDER BY` statement in SQL.

Limit the Number of Documents to Return The `limit()` (page 88) method limits the number of documents in the result set. The following operation returns at most `5` documents in the `bios collection`:

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

`limit()` (page 88) corresponds to the `LIMIT` statement in SQL.

Set the Starting Point of the Result Set The `skip()` (page 94) method controls the starting point of the results set. The following operation skips the first `5` documents in the `bios collection` and returns all remaining documents:

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

Combine Cursor Methods The following example chains cursor methods:

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

Regardless of the order you chain the `limit()` (page 88) and the `sort()` (page 95), the request to the server has the structure that treats the query and the `sort()` (page 95) modifier as a single object. Therefore, the `limit()` (page 88) operation method is always applied after the `sort()` (page 95) regardless of the specified order of the operations in the chain. See the *meta query operators* (page 555).

db.collection.findAndModify()

Definition

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

Modifies and returns a single document. By default, the returned document does not include the modifications made on the update. To return the document with the modifications made on the update, use the `new` option. The `findAndModify()` (page 42) method is a shell helper around the `findAndModify` (page 239) command.

The `findAndModify()` (page 42) method has the following form:

```
db.collection.findAndModify({
    query: <document>,
    sort: <document>,
    remove: <boolean>,
    update: <document>,
    new: <boolean>,
    fields: <document>,
    upsert: <boolean>
});
```

The `db.collection.findAndModify()` (page 42) method takes a document parameter with the following subdocument fields:

param document query The selection criteria for the modification. The `query` field employs the same *query selectors* (page 400) as used in the `db.collection.find()` (page 36) method. Although the query may match multiple documents, `findAndModify()` (page 42) **will only select one document to modify**.

param document sort Determines which document the operation modifies if the query selects multiple documents. `findAndModify()` (page 42) modifies the first document in the sort order specified by this argument.

param Boolean remove Must specify either the `remove` or the `update` field. Removes the document specified in the `query` field. Set this to `true` to remove the selected document . The default is `false`.

param document update Must specify either the `remove` or the `update` field. Performs an update of the selected document. The `update` field employs the same *update operators* (page 451) or `field: value` specifications to modify the selected document.

param Boolean new When `true`, returns the modified document rather than the original. The `findAndModify()` (page 42) method ignores the `new` option for `remove` operations. The default is `false`.

param document fields A subset of fields to return. The `fields` document specifies an inclusion of a field with `1`, as in: `fields: { <field1>: 1, <field2>: 1, ... }`. See *projection*.

param Boolean upsert Used in conjunction with the `update` field.

When `true`, `findAndModify()` (page 42) creates a new document if no document matches the `query`, or if documents match the `query`, `findAndModify()` (page 42) performs an update. To avoid multiple upserts, ensure that the `query` fields are *uniquely indexed*.

The default is `false`.

Return Data The `findAndModify()` (page 42) method returns either: the pre-modification document or, if `new: true` is set, the modified document.

Note:

- If the query finds no document for `update` or `remove` operations, `findAndModify()` (page 42) returns `null`.
- If the query finds no document for an `update` with an `upsert` operation, `findAndModify()` (page 42) creates a new document. If `new` is `false`, **and** the `sort` option is **NOT** specified, the method returns `null`.
- If the query finds no document for an `update` with an `upsert` operation, `findAndModify()` (page 42) creates a new document. If `new` is `false`, **and** a `sort` option, the method returns an empty document `{}`.

Behavior

Upsert and Unique Index When `findAndModify()` (page 42) includes the `upsert: true` option **and** the query field(s) is not uniquely indexed, the method could insert a document multiple times in certain circumstances.

In the following example, no document with the name `Andy` exists, and multiple clients issue the following command:

```
db.people.findAndModify({
    query: { name: "Andy" },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } },
    upsert: true
})
```

Then, if these clients' `findAndModify()` (page 42) methods finish the `query` phase before any command starts the `modify` phase, **and** there is no unique index on the `name` field, the commands may all perform an upsert, creating multiple duplicate documents.

To prevent the creation of multiple duplicate documents, create a *unique index* on the `name` field. With the unique index in place, the multiple methods will exhibit one of the following behaviors:

- Exactly one `findAndModify()` (page 42) successfully inserts a new document.
- Zero or more `findAndModify()` (page 42) methods update the newly inserted document.
- Zero or more `findAndModify()` (page 42) methods fail when they attempt to insert a duplicate. If the method fails due to a unique index constraint violation, you can retry the method. Absent a delete of the document, the retry should not fail.

Sharded Collections When using `findAndModify` (page 239) in a *sharded* environment, the `query` **must** contain the *shard key* for all operations against the shard cluster for the *sharded* collections.

`findAndModify` (page 239) operations issued against `mongos` (page 601) instances for *non-sharded* collections function normally.

Comparisons with the `update` Method When updating a document, `findAndModify()` (page 42) and the `update()` (page 72) method operate differently:

- By default, both operations modify a single document. However, the `update()` (page 72) method with its `multi` option can modify more than one document.
- If multiple documents match the update criteria, for `findAndModify()` (page 42), you can specify a `sort` to provide some measure of control on which document to update.

  With the default behavior of the `update()` (page 72) method, you cannot specify which single document to update when multiple documents match.
- By default, `findAndModify()` (page 42) method returns the pre-modified version of the document. To obtain the updated document, use the `new` option.

  The `update()` (page 72) method returns a `WriteResult` (page 201) object that contains the status of the operation. To return the updated document, use the `find()` (page 36) method. However, other updates may have modified the document between your update and the document retrieval. Also, if the update modified only a single document but multiple documents matched, you will need to use additional logic to identify the updated document.
- You cannot specify a `write concern` to `findAndModify()` (page 42) to override the default write concern whereas, starting in MongoDB 2.6, you can specify a write concern to the `update()` (page 72) method.

When modifying a *single* document, both `findAndModify()` (page 42) and the `update()` (page 72) method *atomically* update the document. See `http://docs.mongodb.org/manual/core/write-operations-atomicity` for more details about interactions and order of operations of these methods.

Examples

Update and Return The following method updates and returns an existing document in the people collection where the document matches the query criteria:

```
db.people.findAndModify({
    query: { name: "Tom", state: "active", rating: { $gt: 10 } },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } }
})
```

This method performs the following actions:

1. The `query` finds a document in the `people` collection where the `name` field has the value `Tom`, the `state` field has the value `active` and the `rating` field has a value `greater than` 10.
2. The `sort` orders the results of the query in ascending order. If multiple documents meet the `query` condition, the method will select for modification the first document as ordered by this `sort`.
3. The update `increments` the value of the `score` field by 1.
4. The method returns the original (i.e. pre-modification) document selected for this update:

```
{
  "_id" : ObjectId("50f1e2c99beb36a0f45c6453"),
  "name" : "Tom",
  "state" : "active",
  "rating" : 100,
  "score" : 5
}
```

To return the modified document, add the `new:true` option to the method.

If no document matched the `query` condition, the method returns `null`:

```
null
```

Upsert The following method includes the `upsert: true` option for the `update` operation to either update a matching document or, if no matching document exists, create a new document:

```
db.people.findAndModify({
    query: { name: "Gus", state: "active", rating: 100 },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } },
    upsert: true
})
```

If the method finds a matching document, the method performs an update.

If the method does **not** find a matching document, the method creates a new document. Because the method included the `sort` option, it returns an empty document `{ }` as the original (pre-modification) document:

```
{ }
```

If the method did **not** include a `sort` option, the method returns `null`.

```
null
```

Return New Document The following method includes both the `upsert: true` option and the `new:true` option. The method either updates a matching document and returns the updated document or, if no matching document exists, inserts a document and returns the newly inserted document in the `value` field.

In the following example, no document in the `people` collection matches the `query` condition:

```
db.people.findAndModify({
    query: { name: "Pascal", state: "active", rating: 25 },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } },
    upsert: true,
    new: true
})
```

The method returns the newly inserted document:

```
{
   "_id" : ObjectId("50f49ad6444c11ac2448a5d6"),
   "name" : "Pascal",
   "rating" : 25,
   "score" : 1,
   "state" : "active"
}
```

Sort and Remove By including a `sort` specification on the `rating` field, the following example removes from the `people` collection a single document with the `state` value of `active` and the lowest `rating` among the matching documents:

```
db.people.findAndModify(
   {
     query: { state: "active" },
     sort: { rating: 1 },
     remove: true
   }
)
```

The method returns the deleted document:

```
{
   "_id" : ObjectId("52fba867ab5fdca1299674ad"),
   "name" : "XYZ123",
   "score" : 1,
   "state" : "active",
   "rating" : 3
}
```

db.collection.findOne()

Definition

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

Returns one document that satisfies the specified query criteria. If multiple documents satisfy the query, this method returns the first document according to the *natural order* which reflects the order of documents on the disk. In *capped collections*, natural order is the same as insertion order.

param document criteria Specifies query selection criteria using *query operators* (page 400).

param document projection Specifies the fields to return using *projection operators* (page 444). Omit this parameter to return all fields in the matching document.

The `projection` parameter takes a document of the following form:

```
{ field1: <boolean>, field2: <boolean> ... }
```

The `<boolean>` can be one of the following include or exclude values:

- `1` or `true` to include. The `findOne()` (page 46) method always includes the *_id* field even if the field is not explicitly specified in the *projection* parameter.
- `0` or `false` to exclude.

The projection argument cannot mix include and exclude specifications, with the exception of excluding the `_id` field.

returns One document that satisfies the criteria specified as the first argument to this method. If you specify a `projection` parameter, `findOne()` (page 46) returns a document that only contains the `projection` fields. The `_id` field is always included unless you explicitly exclude it.

Although similar to the `find()` (page 36) method, the `findOne()` (page 46) method returns a document rather than a cursor.

Examples

With Empty Query Specification The following operation returns a single document from the `bios collection`:

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

With a Query Specification 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 The `projection` parameter specifies which fields to return. The parameter contains either include or exclude specifications, not both, unless the exclude is for the `_id` field.

Specify the Fields to Return The following operation finds a document in the `bios collection` and returns only the `name`, `contribs` and `_id` fields:

```
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:

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

The `findOne` Result Document You cannot apply cursor methods to the result of `findOne()` (page 46) because a single document is returned. You have access to the document directly:

```
var myDocument = db.bios.findOne();

if (myDocument) {
   var myName = myDocument.name;

   print (tojson(myName));
}
```

db.collection.getIndexStats()

Definition

db.collection.**getIndexStats**(*index*)

Displays a human-readable summary of aggregated statistics about an index's B-tree data structure. The information summarizes the output returned by the indexStats (page 360) command and indexStats() (page 54) method. The getIndexStats() (page 47) method displays the information on the screen and does not return an object.

The getIndexStats() (page 47) method has the following form:

```
db.<collection>.getIndexStats( { index : "<index name>" } )
```

param document index The *index name*.

The getIndexStats() (page 47) method is available only when connected to a mongod (page 583) instance that uses the *--enableExperimentalIndexStatsCmd* option.

To view *index names* for a collection, use the getIndexes() (page 48) method.

Warning: Do not use getIndexStats() (page 47) or indexStats (page 360) with production deployments.

Example The following command returns information for an index named type_1_traits_1:

```
db.animals.getIndexStats({index:"type_1_traits_1"})
```

The command returns the following summary. For more information on the B-tree statistics, see indexStats (page 360).

```
-- index "undefined" --
  version 1 | key pattern {  "type" : 1,  "traits" : 1 } | storage namespace "test.animals.$type_1_t
  2 deep, bucket body is 8154 bytes

  bucket count       45513   on average 99.401 % (±0.463 %) full     49.581 % (±4.135 %) bson keys,

  -- depth 0 --
    bucket count     1       on average 71.511 % (±0.000 %) full     36.191 % (±0.000 %) bson keys,

  -- depth 1 --
    bucket count     180     on average 98.954 % (±5.874 %) full     49.732 % (±5.072 %) bson keys,

  -- depth 2 --
    bucket count     45332   on average 99.403 % (±0.245 %) full     49.580 % (±4.130 %) bson keys,
```

db.collection.getIndexes()

Definition

db.collection.**getIndexes**()

Returns an array that holds a list of documents that identify and describe the existing indexes on the collection. You must call the db.collection.getIndexes() (page 48) on a collection. For example:

```
db.collection.getIndexes()
```

Change collection to the name of the collection whose indexes you want to learn.

Considerations Changed in version 2.8.0.

If you use `db.collection.getIndexes()` (page 48) from a version of the `mongo` (page 610) shell before 2.8.0, on a `mongod` (page 583) instance that uses the‘‘wiredTiger‘‘ storage engine, `db.collection.getIndexes()` (page 48) will return no data, even if there are existing indexes.

Output The `db.collection.getIndexes()` (page 48) items consist of the following fields:

`system.indexes.v`
Holds the version of the index.

The index version depends on the version of `mongod` (page 583) that created the index. Before version 2.0 of MongoDB, the this value was 0; versions 2.0 and later use version 1.

`system.indexes.key`
Contains a document holding the keys held in the index, and the order of the index. Indexes may be either descending or ascending order. A value of negative one (e.g. `-1`) indicates an index sorted in descending order while a positive value (e.g. `1`) indicates an index sorted in an ascending order.

`system.indexes.ns`
The namespace context for the index.

`system.indexes.name`
A unique name for the index comprised of the field names and orders of all keys.

db.collection.getShardDistribution()

Definition

`db.collection.getShardDistribution()`

Returns

Prints the data distribution statistics for a *sharded* collection. You must call the `getShardDistribution()` (page 49) method on a sharded collection, as in the following example:

```
db.myShardedCollection.getShardDistribution()
```

In the following example, the collection has two shards. The output displays both the individual shard distribution information as well the total shard distribution:

```
Shard <shard-a> at <host-a>
 data : <size-a> docs : <count-a> chunks : <number of chunks-a>
 estimated data per chunk : <size-a>/<number of chunks-a>
 estimated docs per chunk : <count-a>/<number of chunks-a>

Shard <shard-b> at <host-b>
 data : <size-b> docs : <count-b> chunks : <number of chunks-b>
 estimated data per chunk : <size-b>/<number of chunks-b>
 estimated docs per chunk : <count-b>/<number of chunks-b>

Totals
 data : <stats.size> docs : <stats.count> chunks : <calc total chunks>
 Shard <shard-a> contains  <estDataPercent-a>% data, <estDocPercent-a>% docs in cluster, avg obj
 Shard <shard-b> contains  <estDataPercent-b>% data, <estDocPercent-b>% docs in cluster, avg obj
```

See also:

`http://docs.mongodb.org/manual/sharding`

Output The output information displays:

- <shard-x> is a string that holds the shard name.
- <host-x> is a string that holds the host name(s).
- <size-x> is a number that includes the size of the data, including the unit of measure (e.g. b, Mb).
- <count-x> is a number that reports the number of documents in the shard.
- <number of chunks-x> is a number that reports the number of chunks in the shard.
- <size-x>/<number of chunks-x> is a calculated value that reflects the estimated data size per chunk for the shard, including the unit of measure (e.g. b, Mb).
- <count-x>/<number of chunks-x> is a calculated value that reflects the estimated number of documents per chunk for the shard.
- <stats.size> is a value that reports the total size of the data in the sharded collection, including the unit of measure.
- <stats.count> is a value that reports the total number of documents in the sharded collection.
- <calc total chunks> is a calculated number that reports the number of chunks from all shards, for example:

  ```
  <calc total chunks> = <number of chunks-a> + <number of chunks-b>
  ```

- <estDataPercent-x> is a calculated value that reflects, for each shard, the data size as the percentage of the collection's total data size, for example:

  ```
  <estDataPercent-x> = <size-x>/<stats.size>
  ```

- <estDocPercent-x> is a calculated value that reflects, for each shard, the number of documents as the percentage of the total number of documents for the collection, for example:

  ```
  <estDocPercent-x> = <count-x>/<stats.count>
  ```

- stats.shards[<shard-x>].avgObjSize is a number that reflects the average object size, including the unit of measure, for the shard.

Example Output For example, the following is a sample output for the distribution of a sharded collection:

```
Shard shard-a at shard-a/MyMachine.local:30000,MyMachine.local:30001,MyMachine.local:30002
data : 38.14Mb docs : 1000003 chunks : 2
estimated data per chunk : 19.07Mb
estimated docs per chunk : 500001

Shard shard-b at shard-b/MyMachine.local:30100,MyMachine.local:30101,MyMachine.local:30102
data : 38.14Mb docs : 999999 chunks : 3
estimated data per chunk : 12.71Mb
estimated docs per chunk : 333333

Totals
data : 76.29Mb docs : 2000002 chunks : 5
Shard shard-a contains 50% data, 50% docs in cluster, avg obj size on shard : 40b
Shard shard-b contains 49.99% data, 49.99% docs in cluster, avg obj size on shard : 40b
```

db.collection.getShardVersion()

`db.collection.getShardVersion()`

This method returns information regarding the state of data in a *sharded cluster* that is useful when diagnosing underlying issues with a sharded cluster.

For internal and diagnostic use only.

db.collection.group()

Recommended Alternatives

Because `db.collection.group()` (page 51) uses JavaScript, it is subject to a number of performance limitations. For most cases the `$group` (page 486) operator in the `aggregation pipeline` provides a suitable alternative with fewer restrictions.

Definition

`db.collection.group(` *{ key, reduce, initial [, keyf] [, cond] [, finalize] }* `)`

Groups documents in a collection by the specified keys and performs simple aggregation functions such as computing counts and sums. The method is analogous to a `SELECT <...> GROUP BY` statement in SQL. The `group()` (page 51) method returns an array.

The `db.collection.group()` (page 51) accepts a single *document* that contains the following:

field document key The field or fields to group. Returns a "key object" for use as the grouping key.

field function reduce An aggregation function that operates on the documents during the grouping operation. These functions may return a sum or a count. The function takes two arguments: the current document and an aggregation result document for that group.

field document initial Initializes the aggregation result document.

field function keyf Alternative to the `key` field. Specifies a function that creates a "key object" for use as the grouping key. Use `keyf` instead of `key` to group by calculated fields rather than existing document fields.

field document cond The selection criteria to determine which documents in the collection to process. If you omit the `cond` field, `db.collection.group()` (page 51) processes all the documents in the collection for the group operation.

field function finalize A function that runs each item in the result set before `db.collection.group()` (page 51) returns the final value. This function can either modify the result document or replace the result document as a whole.

The `db.collection.group()` (page 51) method is a shell wrapper for the `group` (page 216) command. However, the `db.collection.group()` (page 51) method takes the `keyf` field and the `reduce` field whereas the `group` (page 216) command takes the `$keyf` field and the `$reduce` field.

Behavior

Limits and Restrictions The `db.collection.group()` (page 51) method does not work with *sharded clusters*. Use the *aggregation framework* or *map-reduce* in *sharded environments*.

The result set must fit within the *maximum BSON document size* (page 692).

In version 2.2, the returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. For group by operations that results in more than 20,000 unique groupings, use `mapReduce` (page 220). Previous versions had a limit of 10,000 elements.

Prior to 2.4, the `db.collection.group()` (page 51) method took the `mongod` (page 583) instance's JavaScript lock, which blocked all other JavaScript execution.

`mongo` Shell JavaScript Functions/Properties Changed in version 2.4: In MongoDB 2.4, `map-reduce operations` (page 220), the `group` (page 216) command, and `$where` (page 421) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 610) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 220), `group` (page 216) commands, or `$where` (page 421) operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations` (page 220), the `group` (page 216) command, and `$where` (page 421) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	

`args`
`MaxKey`
`MinKey`
	`assert()`
	`BinData()`
	`DBPointer()`
	`DBRef()`
	`doassert()`
	`emit()`
	`gc()`
	`HexData()`
	`hex_md5()`
	`isNumber()`
	`isObject()`
	`ISODate()`
	`isString()`
		`Map()`
		`MD5()`
		`NumberInt()`
		`NumberLong()`
		`ObjectId()`
		`print()`
		`printjson()`
		`printjsononeline()`
		`sleep()`
		`Timestamp()`
		`tojson()`
		`tojsononeline()`
		`tojsonObject()`
		`UUID()`
		`version()`

Examples The following examples assume an `orders` collection with documents of the following prototype:

```
{
  _id: ObjectId("5085a95c8fada716c89d0021"),
  ord_dt: ISODate("2012-07-01T04:00:00Z"),
  ship_dt: ISODate("2012-07-02T04:00:00Z"),
  item: { sku: "abc123",
          price: 1.99,
          uom: "pcs",
          qty: 25 }
}
```

Group by Two Fields The following example groups by the `ord_dt` and `item.sku` fields those documents that have `ord_dt` greater than `01/01/2011`:

```
db.orders.group(
   {
     key: { ord_dt: 1, 'item.sku': 1 },
     cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
     reduce: function ( curr, result ) { },
     initial: { }
   }
)
```

The result is an array of documents that contain the group by fields:

```
[
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc123"},
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc456"},
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "bcd123"},
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "efg456"},
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "abc123"},
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "efg456"},
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "ijk123"},
  { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc123"},
  { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc456"},
  { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc123"},
  { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc456"}
]
```

The method call is analogous to the SQL statement:

```
SELECT ord_dt, item_sku
FROM orders
WHERE ord_dt > '01/01/2012'
GROUP BY ord_dt, item_sku
```

Calculate the Sum The following example groups by the `ord_dt` and `item.sku` fields, those documents that have `ord_dt` greater than `01/01/2011` and calculates the sum of the `qty` field for each grouping:

```
db.orders.group(
   {
     key: { ord_dt: 1, 'item.sku': 1 },
     cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
     reduce: function( curr, result ) {
                 result.total += curr.item.qty;
             },
     initial: { total : 0 }
   }
)
```

The result is an array of documents that contain the group by fields and the calculated aggregation field:

```
[ { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc456", "total" : 25 },
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "bcd123", "total" : 10 },
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "efg456", "total" : 10 },
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "efg456", "total" : 15 },
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "ijk123", "total" : 20 },
  { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc123", "total" : 45 },
  { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc456", "total" : 25 },
```

```
  { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
  { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc456", "total" : 25 } ]
```

The method call is analogous to the SQL statement:

```
SELECT ord_dt, item_sku, SUM(item_qty) as total
FROM orders
WHERE ord_dt > '01/01/2012'
GROUP BY ord_dt, item_sku
```

Calculate Sum, Count, and Average The following example groups by the calculated `day_of_week` field, those documents that have `ord_dt` greater than `01/01/2011` and calculates the sum, count, and average of the `qty` field for each grouping:

```
db.orders.group(
   {
     keyf: function(doc) {
               return { day_of_week: doc.ord_dt.getDay() };
           },
     cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
    reduce: function( curr, result ) {
                result.total += curr.item.qty;
                result.count++;
            },
    initial: { total : 0, count: 0 },
    finalize: function(result) {
                  var weekdays = [
                        "Sunday", "Monday", "Tuesday",
                        "Wednesday", "Thursday",
                        "Friday", "Saturday"
                      ];
                  result.day_of_week = weekdays[result.day_of_week];
                  result.avg = Math.round(result.total / result.count);
              }
   }
)
```

The result is an array of documents that contain the group by fields and the calculated aggregation field:

```
[
  { "day_of_week" : "Sunday", "total" : 70, "count" : 4, "avg" : 18 },
  { "day_of_week" : "Friday", "total" : 110, "count" : 6, "avg" : 18 },
  { "day_of_week" : "Tuesday", "total" : 70, "count" : 3, "avg" : 23 }
]
```

See also:

`http://docs.mongodb.org/manual/core/aggregation`

db.collection.indexStats()

Definition

`db.collection.`**`indexStats`**`(`*index*`)`

Aggregates statistics for the B-tree data structure that stores data for a MongoDB index. The `indexStats()` (page 54) method is a thin wrapper around the `indexStats` (page 360) command. The `indexStats()` (page 54) method is available only on `mongod` (page 583) instances running with the `--enableExperimentalIndexStatsCmd` option.

Important: The `indexStats()` (page 54) method is not intended for production deployments.

The `indexStats()` (page 54) method has the following form:

```
db.<collection>.indexStats( { index: "<index name>" } )
```

The `indexStats()` (page 54) method has the following parameter:

param document index *The index name*.

The method takes a read lock and pages into memory all the extents, or B-tree buckets, encountered. The method might be slow for large indexes if the underlying extents are not already in physical memory. Do not run `indexStats()` (page 54) on a *replica set primary*. When run on a *secondary*, the command causes the secondary to fall behind on replication.

The method aggregates statistics for the entire B-tree and for each individual level of the B-tree. For a description of the command's output, see *indexStats* (page 360).

For more information about running `indexStats()` (page 54), see https://github.com/10gen-labs/storage-viz#readme.

db.collection.insert()

Definition

`db.collection.insert()`

Inserts a document or documents into a collection.

The `insert()` (page 55) method has the following syntax:

Changed in version 2.6.

```
db.collection.insert(
   <document or array of documents>,
   {
     writeConcern: <document>,
     ordered: <boolean>
   }
)
```

param document,array document A document or array of documents to insert into the collection.

param document writeConcern A document expressing the `write concern`. Omit to use the default write concern. See *Safe Writes* (page 56).

New in version 2.6.

param boolean ordered If `true`, perform an ordered insert of the documents in the array, and if an error occurs with one of documents, MongoDB will return without processing the remaining documents in the array.

If `false`, perform an unordered insert, and if an error occurs with one of documents, continue processing the remaining documents in the array.

Defaults to `true`.

New in version 2.6.

Changed in version 2.6: The `insert()` (page 55) returns an object that contains the status of the operation.

Returns

- A *WriteResult* (page 57) object for single inserts.
- A *BulkWriteResult* (page 58) object for bulk inserts.

Behaviors

Safe Writes Changed in version 2.6.

The `insert()` (page 55) method uses the `insert` (page 247) command, which uses the default write concern. To specify a different write concern, include the write concern in the options parameter.

Create Collection If the collection does not exist, then the `insert()` (page 55) method will create the collection.

`_id` Field If the document does not specify an *_id* field, then MongoDB will add the `_id` field and assign a unique `http://docs.mongodb.org/manual/reference/object-id` for the document before inserting. Most drivers create an ObjectId and insert the `_id` field, but the `mongod` (page 583) will create and populate the `_id` if the driver or application does not.

If the document contains an `_id` field, the `_id` value must be unique within the collection to avoid duplicate key error.

Examples The following examples insert documents into the `products` collection. If the collection does not exist, the `insert()` (page 55) method creates the collection.

Insert a Document without Specifying an `_id` Field In the following example, the document passed to the `insert()` (page 55) method does not contain the `_id` field:

```
db.products.insert( { item: "card", qty: 15 } )
```

During the insert, `mongod` (page 583) will create the `_id` field and assign it a unique `http://docs.mongodb.org/manual/reference/object-id` value, as verified by the inserted document:

```
{ "_id" : ObjectId("5063114bd386d8fadbd6b004"), "item" : "card", "qty" : 15 }
```

The `ObjectId` values are specific to the machine and time when the operation is run. As such, your values may differ from those in the example.

Insert a Document Specifying an `_id` Field In the following example, the document passed to the `insert()` (page 55) method includes the `_id` field. The value of `_id` must be unique within the collection to avoid duplicate key error.

```
db.products.insert( { _id: 10, item: "box", qty: 20 } )
```

The operation inserts the following document in the `products` collection:

```
{ "_id" : 10, "item" : "box", "qty" : 20 }
```

Insert Multiple Documents The following example performs a bulk insert of three documents by passing an array of documents to the `insert()` (page 55) method. By default, MongoDB performs an *ordered* insert. With *ordered* inserts, if an error occurs during an insert of one of the documents, MongoDB returns on error without processing the remaining documents in the array.

The documents in the array do not need to have the same fields. For instance, the first document in the array has an `_id` field and a `type` field. Because the second and third documents do not contain an `_id` field, `mongod` (page 583) will create the `_id` field for the second and third documents during the insert:

```
db.products.insert(
   [
     { _id: 11, item: "pencil", qty: 50, type: "no.2" },
     { item: "pen", qty: 20 },
     { item: "eraser", qty: 25 }
   ]
)
```

The operation inserted the following three documents:

```
{ "_id" : 11, "item" : "pencil", "qty" : 50, "type" : "no.2" }
{ "_id" : ObjectId("51e0373c6f35bd826f47e9a0"), "item" : "pen", "qty" : 20 }
{ "_id" : ObjectId("51e0373c6f35bd826f47e9a1"), "item" : "eraser", "qty" : 25 }
```

Perform an Unordered Insert The following example performs an *unordered* insert of three documents. With *unordered* inserts, if an error occurs during an insert of one of the documents, MongoDB continues to insert the remaining documents in the array.

```
db.products.insert(
   [
     { _id: 20, item: "lamp", qty: 50, type: "desk" },
     { _id: 21, item: "lamp", qty: 20, type: "floor" },
     { _id: 22, item: "bulk", qty: 100 }
   ],
   { ordered: false }
)
```

Override Default Write Concern The following operation to a replica set specifies a `write concern` of `"w: majority"` with a `wtimeout` of 5000 milliseconds such that the method returns after the write propagates to a majority of the replica set members or the method times out after 5 seconds.

```
db.products.insert(
    { item: "envelopes", qty : 100, type: "Clasp" },
    { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

WriteResult Changed in version 2.6.

When passed a single document, `insert()` (page 55) returns a `WriteResult` object.

Successful Results The `insert()` (page 55) returns a `WriteResult` (page 201) object that contains the status of the operation. Upon success, the `WriteResult` (page 201) object contains information on the number of documents inserted:

```
WriteResult({ "nInserted" : 1 })
```

Write Concern Errors If the `insert()` (page 55) method encounters write concern errors, the results include the `WriteResult.writeConcernError` (page 201) field:

```
WriteResult({
   "nInserted" : 1,
   "writeConcernError" : {
      "code" : 64,
      "errmsg" : "waiting for replication timed out at shard-a"
   }
})
```

Errors Unrelated to Write Concern If the `insert()` (page 55) method encounters a non-write concern error, the results include the `WriteResult.writeError` (page 201) field:

```
WriteResult({
   "nInserted" : 0,
   "writeError" : {
      "code" : 11000,
      "errmsg" : "insertDocument :: caused by :: 11000 E11000 duplicate key error index: test.foo.$_
   }
})
```

BulkWriteResult Changed in version 2.6.

When passed an array of documents, `insert()` (page 55) returns a `BulkWriteResult()` (page 198) object. See `BulkWriteResult()` (page 198) for details.

db.collection.isCapped()

`db.collection.isCapped()`

Returns Returns `true` if the collection is a *capped collection*, otherwise returns `false`.

See also:

`http://docs.mongodb.org/manual/core/capped-collections`

db.collection.mapReduce()

`db.collection.mapReduce(`*map*, *reduce*, *{<out>*, *<query>*, *<sort>*, *<limit>*, *<finalize>*, *<scope>*, *<jsMode>*, *<verbose>}*`)`

The `db.collection.mapReduce()` (page 58) method provides a wrapper around the `mapReduce` (page 220) command.

```
db.collection.mapReduce(
                          <map>,
                          <reduce>,
                          {
                            out: <collection>,
                            query: <document>,
                            sort: <document>,
                            limit: <number>,
                            finalize: <function>,
                            scope: <document>,
                            jsMode: <boolean>,
```

```
        verbose: <boolean>
      }
    )
```

`db.collection.mapReduce()` (page 58) takes the following parameters:

field Javascript function map A JavaScript function that associates or "maps" a `value` with a `key` and emits the `key` and value `pair`.

See *Requirements for the map Function* (page 60) for more information.

field JavaScript function reduce A JavaScript function that "reduces" to a single object all the `values` associated with a particular `key`.

See *Requirements for the reduce Function* (page 61) for more information.

field document options A document that specifies additional parameters to `db.collection.mapReduce()` (page 58).

The following table describes additional arguments that `db.collection.mapReduce()` (page 58) can accept.

field string or document out Specifies the location of the result of the map-reduce operation. You can output to a collection, output to a collection with an action, or output inline. You may output to a collection when performing map reduce operations on the primary members of the set; on *secondary* members you may only use the `inline` output.

See *out Options* (page 62) for more information.

field document query Specifies the selection criteria using *query operators* (page 400) for determining the documents input to the `map` function.

field document sort Sorts the *input* documents. This option is useful for optimization. For example, specify the sort key to be the same as the emit key so that there are fewer reduce operations. The sort key must be in an existing index for this collection.

field number limit Specifies a maximum number of documents for the input into the `map` function.

field Javascript function finalize Follows the `reduce` method and modifies the output.

See *Requirements for the finalize Function* (page 63) for more information.

field document scope Specifies global variables that are accessible in the `map`, `reduce` and `finalize` functions.

field Boolean jsMode Specifies whether to convert intermediate data into BSON format between the execution of the `map` and `reduce` functions. Defaults to `false`.

If `false`:

- Internally, MongoDB converts the JavaScript objects emitted by the `map` function to BSON objects. These BSON objects are then converted back to JavaScript objects when calling the `reduce` function.
- The map-reduce operation places the intermediate BSON objects in temporary, on-disk storage. This allows the map-reduce operation to execute over arbitrarily large data sets.

If `true`:

- Internally, the JavaScript objects emitted during `map` function remain as JavaScript objects. There is no need to convert the objects for the `reduce` function, which can result in faster execution.

- You can only use `jsMode` for result sets with fewer than 500,000 distinct `key` arguments to the mapper's `emit()` function.

The `jsMode` defaults to false.

field Boolean verbose Specifies whether to include the `timing` information in the result information. The `verbose` defaults to `true` to include the `timing` information.

Note: Changed in version 2.4.

In MongoDB 2.4, `map-reduce operations` (page 220), the `group` (page 216) command, and `$where` (page 421) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 610) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 220), `group` (page 216) commands, or `$where` (page 421) operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations` (page 220), the `group` (page 216) command, and `$where` (page 421) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	

`args`
`MaxKey`
`MinKey`
	`assert()`
	`BinData()`
	`DBPointer()`
	`DBRef()`
	`doassert()`
	`emit()`
	`gc()`
	`HexData()`
	`hex_md5()`
	`isNumber()`
	`isObject()`
	`ISODate()`
	`isString()`
		`Map()`
		`MD5()`
		`NumberInt()`
		`NumberLong()`
		`ObjectId()`
		`print()`
		`printjson()`
		`printjsononeline()`
		`sleep()`
		`Timestamp()`
		`tojson()`
		`tojsononeline()`
		`tojsonObject()`
		`UUID()`
		`version()`

Requirements for the `map` Function The `map` function is responsible for transforming each input document into zero or more documents. It can access the variables defined in the `scope` parameter, and has the following prototype:

```
function() {
   ...
   emit(key, value);
}
```

The `map` function has the following requirements:

- In the `map` function, reference the current document as `this` within the function.
- The `map` function should *not* access the database for any reason.
- The `map` function should be pure, or have *no* impact outside of the function (i.e. side effects.)

- A single emit can only hold half of MongoDB's *maximum BSON document size* (page 692).
- The `map` function may optionally call `emit(key,value)` any number of times to create an output document associating `key` with `value`.

The following `map` function will call `emit(key,value)` either 0 or 1 times depending on the value of the input document's `status` field:

```
function() {
    if (this.status == 'A')
        emit(this.cust_id, 1);
}
```

The following `map` function may call `emit(key,value)` multiple times depending on the number of elements in the input document's `items` field:

```
function() {
    this.items.forEach(function(item){ emit(item.sku, 1); });
}
```

Requirements for the `reduce` Function The `reduce` function has the following prototype:

```
function(key, values) {
   ...
   return result;
}
```

The `reduce` function exhibits the following behaviors:

- The `reduce` function should *not* access the database, even to perform read operations.
- The `reduce` function should *not* affect the outside system.
- MongoDB will **not** call the `reduce` function for a key that has only a single value. The `values` argument is an array whose elements are the `value` objects that are "mapped" to the `key`.
- MongoDB can invoke the `reduce` function more than once for the same key. In this case, the previous output from the `reduce` function for that key will become one of the input values to the next `reduce` function invocation for that key.
- The `reduce` function can access the variables defined in the `scope` parameter.
- The inputs to `reduce` must not be larger than half of MongoDB's *maximum BSON document size* (page 692). This requirement may be violated when large documents are returned and then joined together in subsequent `reduce` steps.

Because it is possible to invoke the `reduce` function more than once for the same key, the following properties need to be true:

- the *type* of the return object must be **identical** to the type of the `value` emitted by the `map` function.
- the `reduce` function must be *associative*. The following statement must be true:

  ```
  reduce(key, [ C, reduce(key, [ A, B ]) ] ) == reduce( key, [ C, A, B ] )
  ```

- the `reduce` function must be *idempotent*. Ensure that the following statement is true:

  ```
  reduce( key, [ reduce(key, valuesArray) ] ) == reduce( key, valuesArray )
  ```

- the `reduce` function should be *commutative*: that is, the order of the elements in the `valuesArray` should not affect the output of the `reduce` function, so that the following statement is true:

```
reduce( key, [ A, B ] ) == reduce( key, [ B, A ] )
```

out Options You can specify the following options for the out parameter:

Output to a Collection This option outputs to a new collection, and is not available on secondary members of replica sets.

```
out: <collectionName>
```

Output to a Collection with an Action This option is only available when passing a collection that already exists to out. It is not available on secondary members of replica sets.

```
out: { <action>: <collectionName>
       [, db: <dbName>]
       [, sharded: <boolean> ]
       [, nonAtomic: <boolean> ] }
```

When you output to a collection with an action, the out has the following parameters:

- <action>: Specify one of the following actions:
  - replace

    Replace the contents of the <collectionName> if the collection with the <collectionName> exists.
  - merge

    Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, *overwrite* that existing document.
  - reduce

    Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, apply the reduce function to both the new and the existing documents and overwrite the existing document with the result.
- db:

  Optional. The name of the database that you want the map-reduce operation to write its output. By default this will be the same database as the input collection.
- sharded:

  Optional. If true *and* you have enabled sharding on output database, the map-reduce operation will shard the output collection using the _id field as the shard key.
- nonAtomic:

  New in version 2.2.

  Optional. Specify output operation as non-atomic. This applies **only** to the merge and reduce output modes, which may take minutes to execute.

  By default nonAtomic is false, and the map-reduce operation locks the database during post-processing.

  If nonAtomic is true, the post-processing step prevents MongoDB from locking the database: during this time, other clients will be able to read intermediate states of the output collection.

Output Inline Perform the map-reduce operation in memory and return the result. This option is the only available option for `out` on secondary members of replica sets.

```
out: { inline: 1 }
```

The result must fit within the *maximum size of a BSON document* (page 692).

Requirements for the `finalize` Function The `finalize` function has the following prototype:

```
function(key, reducedValue) {
   ...
   return modifiedObject;
}
```

The `finalize` function receives as its arguments a `key` value and the `reducedValue` from the `reduce` function. Be aware that:

- The `finalize` function should *not* access the database for any reason.
- The `finalize` function should be pure, or have *no* impact outside of the function (i.e. side effects.)
- The `finalize` function can access the variables defined in the `scope` parameter.

Map-Reduce Examples Consider the following map-reduce operations on a collection `orders` that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 25,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

Return the Total Price Per Customer Perform the map-reduce operation on the `orders` collection to group by the `cust_id`, and calculate the sum of the `price` for each `cust_id`:

1. Define the map function to process each input document:
   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - The function maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair.

   ```
   var mapFunction1 = function() {
                          emit(this.cust_id, this.price);
                      };
   ```

2. Define the corresponding reduce function with two arguments `keyCustId` and `valuesPrices`:
   - The `valuesPrices` is an array whose elements are the `price` values emitted by the map function and grouped by `keyCustId`.
   - The function reduces the `valuesPrice` array to the sum of its elements.

   ```
   var reduceFunction1 = function(keyCustId, valuesPrices) {
                             return Array.sum(valuesPrices);
                         };
   ```

3. Perform the map-reduce on all documents in the `orders` collection using the `mapFunction1` map function and the `reduceFunction1` reduce function.

```
db.orders.mapReduce(
                     mapFunction1,
                     reduceFunction1,
                     { out: "map_reduce_example" }
                   )
```

This operation outputs the results to a collection named `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will replace the contents with the results of this map-reduce operation:

Calculate Order and Total Quantity with Average Quantity Per Item In this example, you will perform a map-reduce operation on the `orders` collection for all documents that have an `ord_date` value greater than `01/01/2012`. The operation groups by the `item.sku` field, and calculates the number of orders and the total quantity ordered for each `sku`. The operation concludes by calculating the average quantity per order for each `sku` value:

1. Define the map function to process each input document:

   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

```
var mapFunction2 = function() {
                       for (var idx = 0; idx < this.items.length; idx++) {
                           var key = this.items[idx].sku;
                           var value = {
                                         count: 1,
                                         qty: this.items[idx].qty
                                       };
                           emit(key, value);
                       }
                   };
```

2. Define the corresponding reduce function with two arguments `keySKU` and `countObjVals`:

   - `countObjVals` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.
   - The function reduces the `countObjVals` array to a single object `reducedValue` that contains the `count` and the `qty` fields.
   - In `reducedVal`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

```
var reduceFunction2 = function(keySKU, countObjVals) {
                       reducedVal = { count: 0, qty: 0 };

                       for (var idx = 0; idx < countObjVals.length; idx++) {
                           reducedVal.count += countObjVals[idx].count;
                           reducedVal.qty += countObjVals[idx].qty;
                       }

                       return reducedVal;
                   };
```

3. Define a finalize function with two arguments `key` and `reducedVal`. The function modifies the `reducedVal` object to add a computed field named `avg` and returns the modified object:

```
var finalizeFunction2 = function (key, reducedVal) {

                          reducedVal.avg = reducedVal.qty/reducedVal.count;

                          return reducedVal;

                        };
```

4. Perform the map-reduce operation on the `orders` collection using the `mapFunction2`, `reduceFunction2`, and `finalizeFunction2` functions.

```
db.orders.mapReduce( mapFunction2,
                     reduceFunction2,
                     {
                       out: { merge: "map_reduce_example" },
                       query: { ord_date:
                                   { $gt: new Date('01/01/2012') }
                              },
                       finalize: finalizeFunction2
                     }
                   )
```

   This operation uses the `query` field to select only those documents with `ord_date` greater than `new Date(01/01/2012)`. Then it output the results to a collection `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will merge the existing contents with the results of this map-reduce operation.

Output The output of the `db.collection.mapReduce()` (page 58) method is identical to that of the `mapReduce` (page 220) command. See the *Output* (page 227) section of the `mapReduce` (page 220) command for information on the `db.collection.mapReduce()` (page 58) output.

Additional Information

- `http://docs.mongodb.org/manual/tutorial/troubleshoot-map-function`
- `http://docs.mongodb.org/manual/tutorial/troubleshoot-reduce-function`
- `mapReduce` (page 220) command
- `http://docs.mongodb.org/manual/core/aggregation`
- `Map-Reduce`
- `http://docs.mongodb.org/manual/tutorial/perform-incremental-map-reduce`

db.collection.reIndex()

`db.collection.reIndex()`

The `db.collection.reIndex()` (page 65) drops all indexes on a collection and recreates them. This operation may be expensive for collections that have a large amount of data and/or a large number of indexes.

Call this method, which takes no arguments, on a collection object. For example:

```
db.collection.reIndex()
```

Normally, MongoDB compacts indexes during routine updates. For most users, the db.collection.reIndex() (page 65) is unnecessary. However, it may be worth running if the collection size has changed significantly or if the indexes are consuming a disproportionate amount of disk space.

Behavior

Note: For replica sets, db.collection.reIndex() (page 65) will not propagate from the *primary* to *secondaries*. db.collection.reIndex() (page 65) will only affect a single mongod (page 583) instance.

Important: db.collection.reIndex() (page 65) will rebuild indexes in the *background if the index was originally specified with this option*. However, db.collection.reIndex() (page 65) will rebuild the _id index in the foreground, which takes the database's write lock.

Changed in version 2.6: Reindexing operations will error if the index entry for an indexed field exceeds the Maximum Index Key Length. Reindexing operations occur as part of compact (page 322) and repairDatabase (page 341) commands as well as the db.collection.reIndex() (page 65) method.

Because these operations drop *all* the indexes from a collection and then recreate them sequentially, the error from the Maximum Index Key Length prevents these operations from rebuilding any remaining indexes for the collection and, in the case of the repairDatabase (page 341) command, from continuing with the remainder of the process.

See

http://docs.mongodb.org/manual/core/index-creation for more information on the behavior of indexing operations in MongoDB.

db.collection.remove()

Definition

db.collection.**remove**()

Removes documents from a collection.

The db.collection.remove() (page 66) method can have one of two syntaxes. The remove() (page 66) method can take a query document and an optional justOne boolean:

```
db.collection.remove(
   <query>,
   <justOne>
)
```

Or the method can take a query document and an optional remove options document:

New in version 2.6.

```
db.collection.remove(
   <query>,
   {
     justOne: <boolean>,
     writeConcern: <document>
   }
)
```

param document query Specifies deletion criteria using *query operators* (page 400). To delete all documents in a collection, pass an empty document ({}).

Changed in version 2.6: In previous versions, the method invoked with no query parameter deleted all documents in a collection.

param boolean justOne To limit the deletion to just one document, set to `true`. Omit to use the default value of `false` and delete all documents matching the deletion criteria.

param document writeConcern A document expressing the `write concern`. Omit to use the default write concern. See *Safe Writes* (page 67).

New in version 2.6.

Changed in version 2.6: The `remove()` (page 66) returns an object that contains the status of the operation.

Returns A *WriteResult* (page 68) object that contains the status of the operation.

Behavior

Safe Writes Changed in version 2.6.

The `remove()` (page 66) method uses the `delete` (page 234) command, which uses the default write concern. To specify a different write concern, include the write concern in the options parameter.

Query Considerations By default, `remove()` (page 66) removes all documents that match the `query` expression. Specify the `justOne` option to limit the operation to removing a single document. To delete a single document sorted by a specified order, use the *findAndModify()* (page 46) method.

When removing multiple documents, the remove operation may interleave with other read and/or write operations to the collection. For *unsharded* collections, you can override this behavior with the `$isolated` (page 482) operator, which "isolates" the remove operation and disallows yielding during the operation. This ensures that no client can see the affected documents until they are all processed or an error stops the remove operation.

See *Isolate Remove Operations* (page 68) for an example.

Capped Collections You cannot use the `remove()` (page 66) method with a *capped collection*.

Sharded Collections All `remove()` (page 66) operations for a sharded collection that specify the `justOne` option must include the *shard key* *or* the `_id` field in the query specification. `remove()` (page 66) operations specifying `justOne` in a sharded collection without the *shard key* *or* the `_id` field return an error.

Examples The following are examples of the `remove()` (page 66) method.

Remove All Documents from a Collection To remove all documents in a collection, call the `remove` (page 66) method with an empty query document `{}`. The following operation deletes all documents from the `bios collection`:

```
db.bios.remove( { } )
```

This operation is not equivalent to the `drop()` (page 28) method.

To remove all documents from a collection, it may be more efficient to use the `drop()` (page 28) 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 To remove the documents that match a deletion criteria, call the `remove()` (page 66) method with the `<query>` parameter:

The following operation removes all the documents from the collection `products` where `qty` is greater than `20`:

```
db.products.remove( { qty: { $gt: 20 } } )
```

Override Default Write Concern The following operation to a replica set removes all the documents from the collection `products` where `qty` is greater than `20` and specifies a `write concern` of `"w: majority"` with a `wtimeout` of 5000 milliseconds such that the method returns after the write propagates to a majority of the replica set members or the method times out after 5 seconds.

```
db.products.remove(
    { qty: { $gt: 20 } },
    { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

Remove a Single Document that Matches a Condition To remove the first document that match a deletion criteria, call the `remove` (page 66) method with the `query` criteria and the `justOne` parameter set to `true` or `1`.

The following operation removes the first document from the collection `products` where `qty` is greater than `20`:

```
db.products.remove( { qty: { $gt: 20 } }, true )
```

Isolate Remove Operations To isolate the query, include `$isolated: 1` in the `<query>` parameter as in the following examples:

```
db.products.remove( { qty: { $gt: 20 }, $isolated: 1 } )
```

WriteResult Changed in version 2.6.

Successful Results The `remove()` (page 66) returns a `WriteResult` (page 201) object that contains the status of the operation. Upon success, the `WriteResult` (page 201) object contains information on the number of documents removed:

```
WriteResult({ "nRemoved" : 4 })
```

See also:

`WriteResult.nRemoved` (page 201)

Write Concern Errors If the `remove()` (page 66) method encounters write concern errors, the results include the `WriteResult.writeConcernError` (page 201) field:

```
WriteResult({
   "nRemoved" : 21,
   "writeConcernError" : {
      "code" : 64,
      "errInfo" : {
         "wtimeout" : true
      },
      "errmsg" : "waiting for replication timed out"
   }
})
```

See also:

`WriteResult.hasWriteConcernError()` (page 202)

Errors Unrelated to Write Concern If the `remove()` (page 66) method encounters a non-write concern error, the results include `WriteResult.writeError` (page 201) field:

```
WriteResult({
   "nRemoved" : 0,
   "writeError" : {
      "code" : 2,
      "errmsg" : "unknown top level operator: $invalidFieldName"
   }
})
```

See also:

`WriteResult.hasWriteError()` (page 202)

db.collection.renameCollection()

Definition

`db.collection.renameCollection`(*target*, *dropTarget*)

Renames a collection. Provides a wrapper for the `renameCollection` (page 340) *database command*.

param string target The new name of the collection. Enclose the string in quotes.

param boolean dropTarget If `true`, `mongod` (page 583) drops the *target* of `renameCollection` (page 340) prior to renaming the collection.

Example Call the `db.collection.renameCollection()` (page 69) method on a collection object. For example:

```
db.rrecord.renameCollection("record")
```

This operation will rename the `rrecord` collection to `record`. If the target name (i.e. `record`) is the name of an existing collection, then the operation will fail.

Limitations The method has the following limitations:

- `db.collection.renameCollection()` (page 69) cannot move a collection between databases. Use `renameCollection` (page 340) for these rename operations.
- `db.collection.renameCollection()` (page 69) is not supported on sharded collections.

The `db.collection.renameCollection()` (page 69) method operates within a collection by changing the metadata associated with a given collection.

Refer to the documentation `renameCollection` (page 340) for additional warnings and messages.

Warning: The `db.collection.renameCollection()` (page 69) method and `renameCollection` (page 340) command will invalidate open cursors which interrupts queries that are currently returning data.

db.collection.save()

Definition

`db.collection.save()`

Updates an existing document or inserts a new document, depending on its `document` parameter.

The `save()` (page 70) method has the following form:

Changed in version 2.6.

```
db.collection.save(
   <document>,
   {
     writeConcern: <document>
   }
)
```

param document document A document to save to the collection.

param document writeConcern A document expressing the `write concern`. Omit to use the default write concern. See *Safe Writes* (page 70).

New in version 2.6.

Changed in version 2.6: The `save()` (page 70) returns an object that contains the status of the operation.

Returns A *WriteResult* (page 71) object that contains the status of the operation.

Behavior

Safe Writes Changed in version 2.6.

The `save()` (page 70) method uses either the `insert` (page 247) or the `update` (page 251) command, which use the default write concern. To specify a different write concern, include the write concern in the options parameter.

Insert If the document does **not** contain an *_id* field, then the `save()` (page 70) method calls the `insert()` (page 55) method. During the operation, the `mongo` (page 610) shell will create an `http://docs.mongodb.org/manual/reference/object-id` and assign it to the `_id` field.

Note: Most MongoDB driver clients will include the `_id` field and generate an `ObjectId` before sending the insert operation to MongoDB; however, if the client sends a document without an `_id` field, the `mongod` (page 583) will add the `_id` field and generate the `ObjectId`.

Update If the document contains an *_id* field, then the `save()` (page 70) method is equivalent to an update with the *upsert option* (page 74) set to `true` and the query predicate on the `_id` field.

Examples

Save a New Document without Specifying an `_id` Field In the following example, `save()` (page 70) method performs an insert since the document passed to the method does not contain the `_id` field:

```
db.products.save( { item: "book", qty: 40 } )
```

During the insert, mongod (page 583) will create the _id field with a unique http://docs.mongodb.org/manual/reference/object-id value, as verified by the inserted document:

```
{ "_id" : ObjectId("50691737d386d8fadbd6b01d"), "item" : "book", "qty" : 40 }
```

The ObjectId values are specific to the machine and time when the operation is run. As such, your values may differ from those in the example.

Save a New Document Specifying an _id Field In the following example, save() (page 70) performs an update with upsert:true since the document contains an _id field:

```
db.products.save( { _id: 100, item: "water", qty: 30 } )
```

Because the _id field holds a value that *does not* exist in the collection, the update operation results in an insertion of the document. The results of these operations are identical to an *update() method with the upsert option* (page 74) set to true.

The operation results in the following new document in the products collection:

```
{ "_id" : 100, "item" : "water", "qty" : 30 }
```

Replace an Existing Document The products collection contains the following document:

```
{ "_id" : 100, "item" : "water", "qty" : 30 }
```

The save() (page 70) method performs an update with upsert:true since the document contains an _id field:

```
db.products.save( { _id : 100, item : "juice" } )
```

Because the _id field holds a value that exists in the collection, the operation performs an update to replace the document and results in the following document:

```
{ "_id" : 100, "item" : "juice" }
```

Override Default Write Concern The following operation to a replica set specifies a write concern of "w: majority" with a wtimeout of 5000 milliseconds such that the method returns after the write propagates to a majority of the replica set members or the method times out after 5 seconds.

```
db.products.save(
    { item: "envelopes", qty : 100, type: "Clasp" },
    { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

WriteResult Changed in version 2.6.

The save() (page 70) returns a WriteResult (page 201) object that contains the status of the insert or update operation. See *WriteResult for insert* (page 57) and *WriteResult for update* (page 78) for details.

db.collection.stats()

Definition

db.collection.**stats**(*scale*)

Returns statistics about the collection. The method includes the following parameter:

param number scale The scale used in the output to display the sizes of items. By default, output displays sizes in bytes. To display kilobytes rather than bytes, specify a `scale` value of `1024`.

Returns A *document* that contains statistics on the specified collection.

The `stats()` (page 71) method provides a wrapper around the database command `collStats` (page 348).

Example The following operation returns stats on the `people` collection:

```
db.people.stats()
```

See also:

collStats (page 348) for an overview of the output of this command.

db.collection.storageSize()

```
db.collection.storageSize()
```

Returns The total amount of storage allocated to this collection for document storage. Provides a wrapper around the `storageSize` (page 349) field of the `collStats` (page 348) (i.e. `db.collection.stats()` (page 71)) output.

db.collection.totalIndexSize()

```
db.collection.totalIndexSize()
```

Returns The total size of all indexes for the collection. This method provides a wrapper around the `totalIndexSize` (page 349) output of the `collStats` (page 348) (i.e. `db.collection.stats()` (page 71)) operation.

db.collection.totalSize()

```
db.collection.totalSize()
```

Returns The total size of the data in the collection plus the size of every indexes on the collection.

db.collection.update()

Definition

`db.collection.update(`*query*, *update*, *options*`)`

Modifies an existing document or documents in a collection. The method can modify specific fields of an existing document or documents or replace an existing document entirely, depending on the *update parameter* (page 73).

By default, the `update()` (page 72) method updates a **single** document. Set the *Multi Parameter* (page 75) to update all documents that match the query criteria.

The `update()` (page 72) method has the following form:

Changed in version 2.6.

```
db.collection.update(
   <query>,
   <update>,
   {
     upsert: <boolean>,
     multi: <boolean>,
     writeConcern: <document>
   }
)
```

The `update()` (page 72) method takes the following parameters:

param document query The selection criteria for the update. Use the same *query selectors* (page 400) as used in the `find()` (page 36) method.

param document update The modifications to apply. For details see *Update Parameter* (page 73).

param boolean upsert If set to `true`, creates a new document when no document matches the query criteria. The default value is `false`, which does *not* insert a new document when no match is found.

param boolean multi If set to `true`, updates multiple documents that meet the `query` criteria. If set to `false`, updates one document. The default value is `false`. For additional information, see *Multi Parameter* (page 75).

param document writeConcern A document expressing the `write concern`. Omit to use the default write concern. See *Safe Writes* (page 73).

New in version 2.6.

Changed in version 2.6: The `update()` (page 72) method returns an object that contains the status of the operation.

Returns A *WriteResult* (page 78) object that contains the status of the operation.

Behavior

Safe Writes Changed in version 2.6.

The `update()` (page 72) method uses the `update` (page 251) command, which uses the default write concern. To specify a different write concern, include the `writeConcern` option in the options parameter. See *Override Default Write Concern* (page 77) for an example.

Update Parameter The `update()` (page 72) method either modifies specific fields in existing documents or replaces an existing document entirely.

Update Specific Fields If the `<update>` document contains *update operator* (page 451) modifiers, such as those using the `$set` (page 459) modifier, then:

- The `<update>` document must contain *only* *update operator* (page 451) expressions.
- The `update()` (page 72) method updates only the corresponding fields in the document.

To update an embedded document or an array as a whole, specify the replacement value for the field. To update particular fields in an embedded document or in an array, use *dot notation* to specify the field.

Replace a Document Entirely If the <update> document contains *only* field:value expressions, then:

- The update() (page 72) method *replaces* the matching document with the <update> document. The update() (page 72) method *does not* replace the _id value. For an example, see *Replace All Fields* (page 76).
- update() (page 72) *cannot update multiple documents* (page 75).

Upsert Option

Upsert Behavior If upsert is true and no document matches the query criteria, update() (page 72) inserts a *single* document. The update creates the new document with either:

- The fields and values of the <update> parameter if the <update> parameter contains only field and value pairs, or
- The fields and values of both the <query> and <update> parameters if the <update> parameter contains *update operator* (page 451) expressions. The update creates a base document from the equality clauses in the <query> parameter, and then applies the update expressions from the <update> parameter.

If upsert is true and there are documents that match the query criteria, update() (page 72) performs an update.

See also:

$setOnInsert (page 458)

Use Unique Indexes

Warning: To avoid inserting the same document more than once, only use upsert: true if the query fi is uniquely indexed.

Given a collection named people where no documents have a name field that holds the value Andy. Consider when multiple clients issue the following *update* with upsert: true at the same time:

```
db.people.update(
   { name: "Andy" },
   {
      name: "Andy",
      rating: 1,
      score: 1
   },
   { upsert: true }
)
```

If all update() (page 72) operations complete the query portion before any client successfully inserts data, **and** there is no unique index on the name field, then each update operation may result in an insert.

To prevent MongoDB from inserting the same document more than once, create a *unique index* on the name field. With a unique index, if multiple applications issue the same update with upsert: true, *exactly one* update() (page 72) would successfully insert a new document.

The remaining operations would either:

- update the newly inserted document, or
- fail when they attempted to insert a duplicate.

  If the operation fails because of a duplicate index key error, applications may retry the operation which will succeed as an update operation.

Multi Parameter If `multi` is set to `true`, the `update()` (page 72) method updates all documents that meet the `<query>` criteria. The `multi` update operation may interleave with other operations, both read and/or write operations. For unsharded collections, you can override this behavior with the `$isolated` (page 482) operator, which isolates the update operation and disallows yielding during the operation. This isolates the update so that no client can see the updated documents until they are all processed, or an error stops the update operation.

If the *<update>* (page 73) document contains *only* `field:value` expressions, then `update()` (page 72) *cannot* update multiple documents.

For an example, see *Update Multiple Documents* (page 77).

Sharded Collections All `update()` (page 72) operations for a sharded collection that specify the `multi: false` option must include the *shard key* *or* the `_id` field in the query specification. `update()` (page 72) operations specifying `multi: false` in a sharded collection without the *shard key* *or* the `_id` field return an error.

See also:

`findAndModify()` (page 42)

Examples

Update Specific Fields To update specific fields in a document, use *update operators* (page 451) in the `<update>` parameter.

For example, given a `books` collection with the following document:

```
{
  _id: 1,
  item: "TBD",
  stock: 0,
  info: { publisher: "1111", pages: 430 },
  tags: [ "technology", "computer" ],
  ratings: [ { by: "ijk", rating: 4 }, { by: "lmn", rating: 5 } ],
  reorder: false
}
```

The following operation uses:

- the `$inc` (page 452) operator to increment the `stock` field; and
- the `$set` (page 459) operator to replace the value of the `item` field, the `publisher` field in the `info` embedded document, the `tags` field, and the second element in the `ratings` array.

```
db.books.update(
   { _id: 1 },
   {
     $inc: { stock: 5 },
     $set: {
       item: "ABC123",
       "info.publisher": "2222",
       tags: [ "software" ],
       "ratings.1": { by: "xyz", rating: 3 }
     }
   }
)
```

The updated document is the following:

```
{
  "_id" : 1,
  "item" : "ABC123",
  "stock" : 5,
  "info" : { "publisher" : "2222", "pages" : 430 },
  "tags" : [ "software" ],
  "ratings" : [ { "by" : "ijk", "rating" : 4 }, { "by" : "xyz", "rating" : 3 } ],
  "reorder" : false
}
```

See also:

$set (page 459), $inc (page 452), *Update Operators* (page 451), *dot notation*

Remove Fields The following operation uses the $unset (page 461) operator to remove the tags field:

```
db.books.update( { _id: 1 }, { $unset: { tags: 1 } } )
```

See also:

$unset (page 461), $rename (page 457), *Update Operators* (page 451)

Replace All Fields Given the following document in the books collection:

```
{
  _id: 2,
  item: "XYZ123",
  stock: 15,
  info: { publisher: "5555", pages: 150 },
  tags: [ ],
  ratings: [ { by: "xyz", rating: 5, comment: "ratings and reorder will go away after update"} ],
  reorder: false
}
```

The following operation passes an <update> document that contains only field and value pairs. The <update> document completely replaces the original document except for the _id field.

```
db.books.update(
   { item: "XYZ123" },
   {
     item: "XYZ123",
     stock: 10,
     info: { publisher: "2255", pages: 150 },
     tags: [ "baking", "cooking" ]
   }
)
```

The updated document contains *only* the fields from the replacement document and the _id field. That is, the fields ratings and reorder no longer exist in the updated document since the fields were not in the replacement document.

```
{
   "_id" : 2,
   "item" : "XYZ123",
   "stock" : 10,
   "info" : { "publisher" : "2255", "pages" : 150 },
   "tags" : [ "baking", "cooking" ]
}
```

Insert a New Document if No Match Exists The following update sets the *upsert* (page 74) option to `true` so that `update()` (page 72) creates a new document in the `books` collection if no document matches the `<query>` parameter:

```
db.books.update(
   { item: "ZZZ135" },
   {
     item: "ZZZ135",
     stock: 5,
     tags: [ "database" ]
   },
   { upsert: true }
)
```

If no document matches the `<query>` parameter, the update operation inserts a document with *only* the fields and values of the `<update>` document and a new unique `ObjectId` for the `_id` field:

```
{
  "_id" : ObjectId("542310906694ce357ad2a1a9"),
  "item" : "ZZZ135",
  "stock" : 5,
  "tags" : [ "database" ]
}
```

For more information on `upsert` option and the inserted document, *Upsert Option* (page 74).

Update Multiple Documents To update multiple documents, set the `multi` option to `true`. For example, the following operation updates all documents where `stock` is less than or equal to `10`:

```
db.books.update(
   { stock: { $lte: 10 } },
   { $set: { reorder: true } },
   { multi: true }
)
```

If the `reorder` field does not exist in the matching document(s), the `$set` (page 459) operator will add the field with the specified value. See `$set` (page 459) for more information.

Override Default Write Concern The following operation on a replica set specifies a `write concern` of `"w: majority"` with a `wtimeout` of 5000 milliseconds such that the method returns after the write propagates to a majority of the replica set members or the method times out after 5 seconds.

```
db.books.update(
   { stock: { $lte: 10 } },
   { $set: { reorder: true } },
   {
     multi: true,
     writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

Combine the `upsert` and `multi` Options Given a `books` collection that includes the following documents:

```
{
  _id: 5,
  item: "EFG222",
```

```
  stock: 18,
  info: { publisher: "0000", pages: 70 },
  reorder: true
}
{
  _id: 6,
  item: "EFG222",
  stock: 15,
  info: { publisher: "1111", pages: 72 },
  reorder: true
}
```

The following operation specifies both the `multi` option and the `upsert` option. If matching documents exist, the operation updates all matching documents. If no matching documents exist, the operation inserts a new document.

```
db.books.update(
   { item: "EFG222" },
   { $set: { reorder: false, tags: [ "literature", "translated" ] } },
   { upsert: true, multi: true }
)
```

The operation updates all matching documents and results in the following:

```
{
   "_id" : 5,
   "item" : "EFG222",
   "stock" : 18,
   "info" : { "publisher" : "0000", "pages" : 70 },
   "reorder" : false,
   "tags" : [ "literature", "translated" ]
}
{
   "_id" : 6,
   "item" : "EFG222",
   "stock" : 15,
   "info" : { "publisher" : "1111", "pages" : 72 },
   "reorder" : false,
   "tags" : [ "literature", "translated" ]
}
```

If the collection had *no* matching document, the operation would result in the insertion of a document using the fields from both the `<query>` and the `<update>` specifications:

```
{
   "_id" : ObjectId("5423200e6694ce357ad2a1ac"),
   "item" : "EFG222",
   "reorder" : false,
   "tags" : [ "literature", "translated" ]
}
```

For more information on `upsert` option and the inserted document, *Upsert Option* (page 74).

WriteResult Changed in version 2.6.

Successful Results The `update()` (page 72) method returns a `WriteResult` (page 201) object that contains the status of the operation. Upon success, the `WriteResult` (page 201) object contains the number of documents that matched the query condition, the number of documents inserted by the update, and the number of documents modified:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

See

`WriteResult.nMatched` (page 201), `WriteResult.nUpserted` (page 201), `WriteResult.nModified` (page 201)

Write Concern Errors If the `update()` (page 72) method encounters write concern errors, the results include the `WriteResult.writeConcernError` (page 201) field:

```
WriteResult({
   "nMatched" : 1,
   "nUpserted" : 0,
   "nModified" : 1,
   "writeConcernError" : {
      "code" : 64,
      "errmsg" : "waiting for replication timed out at shard-a"
   }
})
```

See also:

`WriteResult.hasWriteConcernError()` (page 202)

Errors Unrelated to Write Concern If the `update()` (page 72) method encounters a non-write concern error, the results include the `WriteResult.writeError` (page 201) field:

```
WriteResult({
   "nMatched" : 0,
   "nUpserted" : 0,
   "nModified" : 0,
   "writeError" : {
      "code" : 7,
      "errmsg" : "could not contact primary for replica set shard-a"
   }
})
```

See also:

`WriteResult.hasWriteError()` (page 202)

db.collection.validate()

Description

`db.collection.validate(full)`

Validates a collection. The method scans a collection's data structures for correctness and returns a single *document* that describes the relationship between the logical collection and the physical representation of the data.

The `validate()` (page 79) method has the following parameter:

param Boolean full Specify `true` to enable a full validation and to return full statistics. MongoDB disables full validation by default because it is a potentially resource-intensive operation.

The `validate()` (page 79) method output provides an in-depth view of how the collection uses storage. Be aware that this command is potentially resource intensive and may impact the performance of your MongoDB instance.

The `validate()` (page 79) method is a wrapper around the `validate` (page 389) *database command*.

See also:

validate (page 389)

2.1.2 Cursor

Cursor Methods

Name	Description
`cursor.addOption()` (page 81)	Adds special wire protocol flags that modify the behavior of the query.'
`cursor.batchSize()` (page 82)	Controls the number of documents MongoDB will return to the client in a single network message.
`cursor.count()` (page 83)	Returns a count of the documents in a cursor.
`cursor.explain()` (page 85)	Reports on the query execution plan, including index use, for a cursor.
`cursor.forEach()` (page 86)	Applies a JavaScript function for every document in a cursor.
`cursor.hasNext()` (page 87)	Returns true if the cursor has documents and can be iterated.
`cursor.hint()` (page 87)	Forces MongoDB to use a specific index for a query.
`cursor.limit()` (page 88)	Constrains the size of a cursor's result set.
`cursor.map()` (page 88)	Applies a function to each document in a cursor and collects the return values in an array.
`cursor.max()` (page 88)	Specifies an exclusive upper index bound for a cursor. For use with `cursor.hint()` (page 87)
`cursor.maxTimeMS()` (page 90)	Specifies a cumulative time limit in milliseconds for processing operations on a cursor.
`cursor.min()` (page 91)	Specifies an inclusive lower index bound for a cursor. For use with `cursor.hint()` (page 87)
`cursor.next()` (page 93)	Returns the next document in a cursor.
`cursor.objsLeftInBatch()` (page 93)	Returns the number of documents left in the current cursor batch.
`cursor.readPref()` (page 93)	Specifies a *read preference* to a cursor to control how the client directs queries to a *replica set*.
`cursor.showDiskLoc()` (page 93)	Returns a cursor with modified documents that include the on-disk location of the document.
`cursor.size()` (page 94)	Returns a count of the documents in the cursor after applying `skip()` (page 94) and `limit()` (page 88) methods.
`cursor.skip()` (page 94)	Returns a cursor that begins returning results only after passing or skipping a number of documents.
`cursor.snapshot()` (page 95)	Forces the cursor to use the index on the `_id` field. Ensures that the cursor returns each document, with regards to the value of the `_id` field, only once.
`cursor.sort()` (page 95)	Returns results ordered according to a sort specification.
`cursor.toArray()` (page 99)	Returns an array that contains all documents returned by the cursor.

cursor.addOption()

Definition

`cursor.addOption(`*flag*`)`

Adds `OP_QUERY` wire protocol flags, such as the `tailable` flag, to change the behavior of queries.

The `cursor.addOption()` (page 81) method has the following parameter:

param flag flag `OP_QUERY` wire protocol flag. See MongoDB wire protocol[6] for more information on MongoDB Wire Protocols and the OP_QUERY flags. For the `mongo` (page 610) shell, you can use *cursor flags* (page 82). For the driver-specific list, see your `driver` `documentation`.

Flags The `mongo` (page 610) shell provides several additional cursor flags to modify the behavior of the cursor.

`DBQuery.Option.`**`tailable`**

`DBQuery.Option.`**`slaveOk`**

`DBQuery.Option.`**`oplogReplay`**

`DBQuery.Option.`**`noTimeout`**

`DBQuery.Option.`**`awaitData`**

`DBQuery.Option.`**`exhaust`**

`DBQuery.Option.`**`partial`**

For a description of the flags, see MongoDB wire protocol[7].

Example The following example adds the `DBQuery.Option.tailable` flag and the `DBQuery.Option.awaitData` flag to ensure that the query returns a *tailable cursor*. The sequence creates a cursor that will wait for few seconds after returning the full result set so that it can capture and return additional data added during the query:

```
var t = db.myCappedCollection;
var cursor = t.find().addOption(DBQuery.Option.tailable).
                      addOption(DBQuery.Option.awaitData)
```

Warning: Adding incorrect wire protocol flags can cause problems and/or extra server load.

cursor.batchSize()

Definition

`cursor.`**`batchSize`**`(`*size*`)`

Specifies the number of documents to return in each batch of the response from the MongoDB instance. In most cases, modifying the batch size will not affect the user or the application, as the `mongo` (page 610) shell and most `drivers` return results as if MongoDB returned a single batch.

The `batchSize()` (page 82) method takes the following parameter:

param integer size The number of documents to return per batch. Do **not** use a batch size of `1`.

Note: Specifying `1` or a negative number is analogous to using the `limit()` (page 88) method.

[6] http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol/?pageVersion=106#op-query
[7] http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol/?pageVersion=106#op-query

Example The following example sets the batch size for the results of a query (i.e. `find()` (page 36)) to `10`. The `batchSize()` (page 82) method does not change the output in the `mongo` (page 610) shell, which, by default, iterates over the first 20 documents.

```
db.inventory.find().batchSize(10)
```

cursor.count()

Definition

`cursor.count()`

Counts the number of documents referenced by a cursor. Append the `count()` (page 83) method to a `find()` (page 36) query to return the number of matching documents. The operation does not perform the query but instead counts the results that would be returned by the query.

Changed in version 2.6: MongoDB supports the use of `hint()` (page 87) with `count()` (page 83). See *Specify the Index to Use* (page 84) for an example.

The `count()` (page 83) method has the following prototype form:

```
db.collection.find(<query>).count()
```

The `count()` (page 83) method has the following parameter:

param Boolean applySkipLimit Specifies whether to consider the effects of the `cursor.skip()` (page 94) and `cursor.limit()` (page 88) methods in the count. By default, the `count()` (page 83) method ignores the effects of the `cursor.skip()` (page 94) and `cursor.limit()` (page 88). Set `applySkipLimit` to `true` to consider the effect of these methods.

MongoDB also provides an equivalent `db.collection.count()` (page 26) as an alternative to the `db.collection.find(<query>).count()` construct.

See also:

`cursor.size()` (page 94)

Behavior

Sharded Clusters On a sharded cluster, `count()` (page 83) method can result in an *inaccurate* count if *orphaned documents* exist or if a `chunk migration` is in progress.

To avoid these situations, on a sharded cluster, use the `$group` (page 486) stage of the `db.collection.aggregate()` (page 22) method to `$sum` (page 554) the documents. For example, the following operation counts the documents in a collection:

```
db.collection.aggregate(
   [
      { $group: { _id: null, count: { $sum: 1 } } }
   ]
)
```

To get a count of documents that match a query condition, include the `$match` (page 490) stage as well:

```
db.collection.aggregate(
   [
      { $match: <query condition> },
      { $group: { _id: null, count: { $sum: 1 } } }
```

```
    ]
)
```

See *Perform a Count* (page 491) for an example.

Index Use Consider a collection with the following index:

```
{ a: 1, b: 1 }
```

When performing a count, MongoDB can return the count using only the index if:

- the query can use an index,
- the query only contains conditions on the keys of the index, *and*
- the query predicates access a single contiguous range of index keys.

For example, the following operations can return the count using only the index:

```
db.collection.find( { a: 5, b: 5 } ).count()
db.collection.find( { a: { $gt: 5 } } ).count()
db.collection.find( { a: 5, b: { $gt: 10 } } ).count()
```

If, however, the query can use an index but the query predicates do not access a single contiguous range of index keys or the query also contains conditions on fields outside the index, then in addition to using the index, MongoDB must also read the documents to return the count.

```
db.collection.find( { a: 5, b: { $in: [ 1, 2, 3 ] } } ).count()
db.collection.find( { a: { $gt: 5 }, b: 5 } ).count()
db.collection.find( { a: 5, b: 5, c: 5 } ).count()
```

In such cases, during the initial read of the documents, MongoDB pages the documents into memory such that subsequent calls of the same count operation will have better performance.

Examples The following are examples of the `count()` (page 83) method.

Count All Documents The following operation counts the number of all documents in the `orders` collection:

```
db.orders.find().count()
```

Count Documents That Match a Query The following operation counts the number of the documents in the `orders` collection with the field `ord_dt` greater than `new Date('01/01/2012')`:

```
db.orders.find( { ord_dt: { $gt: new Date('01/01/2012') } } ).count()
```

Limit Documents in Count The following operation counts the number of the documents in the `orders` collection with the field `ord_dt` greater than `new Date('01/01/2012')` *taking into account* the effect of the `limit(5)`:

```
db.orders.find( { ord_dt: { $gt: new Date('01/01/2012') } } ).limit(5).count(true)
```

Specify the Index to Use The following operation uses the index named `"status_1"`, which has the index key specification of `{ status: 1 }`, to return a count of the documents in the `orders` collection with the field `ord_dt` greater than `new Date('01/01/2012')` and the `status` field is equal to `"D"`:

```
db.orders.find(
   { ord_dt: { $gt: new Date('01/01/2012') }, status: "D" }
).hint( "status_1" ).count()
```

cursor.explain()

Definition

`cursor.explain(`*verbosity*`)`

Changed in version 2.8: The parameter to the method and the output format have changed in 2.8.

Provides information on the query plan for the `db.collection.find()` (page 36) method.

The `explain()` (page 85) method has the following form:

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

The `explain()` (page 85) method has the following parameter:

param String verbose Specifies the verbosity mode for the explain output. The mode affects the behavior of `explain()` and determines the amount of information to return. The possible modes are: `"queryPlanner"`, `"executionStats"`, and `"allPlansExecution"`.

Default mode is `"queryPlanner"`.

For backwards compatibility with earlier versions of `cursor.explain()` (page 85), MongoDB interprets `true` as `"allPlansExecution"` and `false` as `"queryPlanner"`.

For more information on the modes, see *Verbosity Modes* (page 85).

Changed in version 2.8.

The `explain()` (page 85) method returns a document with the query plan and, optionally, the execution statistics.

Behavior

Verbosity Modes The behavior of `cursor.explain()` (page 85) and the amount of information returned depend on the `verbosity` mode.

`queryPlanner` Mode By default, `cursor.explain()` (page 85) runs in `queryPlanner` verbosity mode.

MongoDB runs the `query optimizer` to choose the winning plan for the operation under evaluation. `cursor.explain()` (page 85) returns the `queryPlanner` information for the evaluated method.

`executionStats` Mode MongoDB runs the `query optimizer` to choose the winning plan, executes the winning plan to completion, and returns statistics describing the execution of the winning plan.

For write operations, `cursor.explain()` (page 85) returns information about the update or delete operations that *would* be performed, but does *not* apply the modifications to the database.

`cursor.explain()` (page 85) returns the `queryPlanner` and `executionStats` information for the evaluated method. However, `executionStats` does not provide query execution information for the rejected plans.

`allPlansExecution` Mode MongoDB runs the `query optimizer` to choose the winning plan and executes the winning plan to completion. In `"allPlansExecution"` mode, MongoDB returns statistics describing the execution of the winning plan as well as statistics for the other candidate plans captured during *plan selection*.

For write operations, `cursor.explain()` (page 85) returns information about the update or delete operations that *would* be performed, but does *not* apply the modifications to the database.

`cursor.explain()` (page 85) returns the `queryPlanner` and `executionStats` information for the evaluated method. The `executionStats` includes the *completed* query execution information for the *winning plan*.

If the query optimizer considered more than one plan, `executionStats` information also includes the *partial* execution information captured during the *plan selection phase* for both the winning and rejected candidate plans.

`db.collection.explain().find()` `db.collection.explain().find()` is similar to `db.collection.find().explain()` (page 85) with the following key differences:

- The `db.collection.explain().find()` construct allows for the additional chaining of query modifiers. For list of query modifiers, see *db.collection.explain().find().help()* (page 35).
- The `db.collection.explain().find()` returns a cursor, which requires a call to `.next()`, or its alias `.finish()`, to return the `explain()` results.

See `db.collection.explain()` (page 33) for more information.

Example The following example runs `cursor.explain()` (page 85) in *"executionStats"* (page 34) verbosity mode to return the query planning and execution information for the specified `db.collection.find()` (page 36) operation:

```
db.products.find(
   { quantity: { $gt: 50 }, category: "apparel" }
).explain("executionStats")
```

Output `cursor.explain()` (page 85) operations can return information regarding:

- *queryPlanner*, which details the plan selected by the `query optimizer` and lists the rejected plans;
- *executionStats*, which details the execution of the winning plan and the rejected plans; and
- *serverInfo*, which provides information on the MongoDB instance.

The verbosity mode (i.e. `queryPlanner`, `executionStats`, `allPlansExecution`) determines whether the results include *executionStats* and whether *executionStats* includes data captured during *plan selection*.

For details on the output, see `http://docs.mongodb.org/manual/reference/explain-results`.

For a mixed version sharded cluster with version 2.8 `mongos` (page 601) and at least one 2.6 `mongod` (page 583) shard, when you run `cursor.explain()` (page 85) in a version 2.8 `mongo` (page 610) shell, `cursor.explain()` (page 85) will retry with the `$explain` (page 556) operator to return results in the 2.6 format.

cursor.forEach()

Description

`cursor.forEach(function)`

Iterates the cursor to apply a JavaScript `function` to each document from the cursor.

The `forEach()` (page 86) method has the following prototype form:

```
db.collection.find().forEach(<function>)
```

The `forEach()` (page 86) method has the following parameter:

param JavaScript function A JavaScript function to apply to each document from the cursor. The `<function>` signature includes a single argument that is passed the current document to process.

Example The following example invokes the `forEach()` (page 86) method on the cursor returned by `find()` (page 36) to print the name of each user in the collection:

```
db.users.find().forEach( function(myDoc) { print( "user: " + myDoc.name ); } );
```

See also:

`cursor.map()` (page 88) for similar functionality.

cursor.hasNext()

```
cursor.hasNext()
```

Returns Boolean.

`cursor.hasNext()` (page 87) returns `true` if the cursor returned by the `db.collection.find()` (page 36) query can iterate further to return more documents.

cursor.hint()

Definition

`cursor.hint(`*index*`)`

Call this method on a query to override MongoDB's default index selection and `query optimization process`. Use `db.collection.getIndexes()` (page 48) to return the list of current indexes on a collection.

The `cursor.hint()` (page 87) method has the following parameter:

param string,document index The index to "hint" or force MongoDB to use when performing the query. Specify the index either by the index name or by the index specification document.

Behavior When an *index filter* exists for the query shape, MongoDB ignores the `hint()` (page 87).

You cannot use `hint()` (page 87) if the query includes a `$text` (page 417) query expression.

Example The following example returns all documents in the collection named `users` using the index on the `age` field.

```
db.users.find().hint( { age: 1 } )
```

You can also specify the index using the index name:

```
db.users.find().hint( "age_1" )
```

See also:

- `http://docs.mongodb.org/manual/core/indexes-introduction`

- http://docs.mongodb.org/manual/administration/indexes
- http://docs.mongodb.org/manual/core/query-plans
- *index-filters*
- $hint (page 556)

cursor.limit()

Definition

cursor.**limit**()

Use the limit() (page 88) method on a cursor to specify the maximum number of documents the cursor will return. limit() (page 88) is analogous to the LIMIT statement in a SQL database.

Note: You must apply limit() (page 88) to the cursor before retrieving any documents from the database.

Use limit() (page 88) to maximize performance and prevent MongoDB from returning more results than required for processing.

Behavior

Supported Values The behavior of limit() (page 88) is undefined for values less than -2^{31} and greater than 2^{31}.

Negative Values A limit() (page 88) value of 0 (i.e. .limit(0) (page 88)) is equivalent to setting no limit. A negative limit is similar to a positive limit, but a negative limit prevents the creation of a cursor such that only a single batch of results is returned. As such, with a negative limit, if the limited result set does not fit into a single batch, the number of documents received will be less than the limit.

cursor.map()

cursor.**map**(*function*)

Applies function to each document visited by the cursor and collects the return values from successive application into an array.

The cursor.map() (page 88) method has the following parameter:

param function function A function to apply to each document visited by the cursor.

Example

```
db.users.find().map( function(u) { return u.name; } );
```

See also:

cursor.forEach() (page 86) for similar functionality.

cursor.max()

Definition

`cursor.max()`

Specifies the *exclusive* upper bound for a specific index in order to constrain the results of `find()` (page 36). `max()` (page 88) provides a way to specify an upper bound on compound key indexes.

The `max()` (page 88) method has the following parameter:

param document indexBounds The exclusive upper bound for the index keys.

The `indexBounds` parameter has the following prototype form:

```
{ field1: <max value>, field2: <max value2> ... fieldN:<max valueN> }
```

The fields correspond to *all* the keys of a particular index *in order*. You can explicitly specify the particular index with the `hint()` (page 87) method. Otherwise, `mongod` (page 583) selects the index using the fields in the `indexBounds`; however, if multiple indexes exist on same fields with different sort orders, the selection of the index may be ambiguous.

See also:

`min()` (page 91).

`max()` (page 88) exists primarily to support the `mongos` (page 601) (sharding) process, and is a shell wrapper around the query modifier `$max` (page 558).

Behavior

Interaction with Index Selection Because `max()` (page 88) requires an index on a field, and forces the query to use this index, you may prefer the `$lt` (page 403) operator for the query if possible. Consider the following example:

```
db.products.find( { _id: 7 } ).max( { price: 1.39 } )
```

The query will use the index on the `price` field, even if the index on `_id` may be better.

Index Bounds If you use `max()` (page 88) with `min()` (page 91) to specify a range, the index bounds specified in `min()` (page 91) and `max()` (page 88) must both refer to the keys of the same index.

`max()` without `min()` The `min` and `max` operators indicate that the system should avoid normal query planning. Instead they construct an index scan where the index bounds are explicitly specified by the values given in `min` and `max`.

If one of the two boundaries is not specified, then the query plan will be an index scan that is unbounded on one side. This may degrade performance compared to a query containing neither operator, or one that uses both operators to more tightly constrain the index scan.

Example This example assumes a collection named `products` that holds the following documents:

```
{ "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
{ "_id" : 2, "item" : "apple", "type" : "fuji", "price" : 1.99 }
{ "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : 1.99 }
{ "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
{ "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
{ "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
{ "_id" : 7, "item" : "orange", "type" : "cara cara", "price" : 2.99 }
{ "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
{ "_id" : 9, "item" : "orange", "type" : "satsuma", "price" : 1.99 }
{ "_id" : 8, "item" : "orange", "type" : "valencia", "price" : 0.99 }
```

The collection has the following indexes:

```
{ "_id" : 1 }
{ "item" : 1, "type" : 1 }
{ "item" : 1, "type" : -1 }
{ "price" : 1 }
```

- Using the ordering of `{ item: 1, type: 1 }` index, `max()` (page 88) limits the query to the documents that are below the bound of `item` equal to `apple` and `type` equal to `jonagold`:

  ```
  db.products.find().max( { item: 'apple', type: 'jonagold' } ).hint( { item: 1, type: 1 } )
  ```

  The query returns the following documents:

  ```
  { "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
  { "_id" : 2, "item" : "apple", "type" : "fuji", "price" : 1.99 }
  { "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : 1.99 }
  ```

  If the query did not explicitly specify the index with the `hint()` (page 87) method, it is ambiguous as to whether `mongod` (page 583) would select the `{ item: 1, type: 1 }` index ordering or the `{ item: 1, type: -1 }` index ordering.

- Using the ordering of the index `{ price: 1 }`, `max()` (page 88) limits the query to the documents that are below the index key bound of `price` equal to `1.99` and `min()` (page 91) limits the query to the documents that are at or above the index key bound of `price` equal to `1.39`:

  ```
  db.products.find().min( { price: 1.39 } ).max( { price: 1.99 } ).hint( { price: 1 } )
  ```

  The query returns the following documents:

  ```
  { "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
  { "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
  { "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
  { "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
  { "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
  ```

cursor.maxTimeMS()

Definition New in version 2.6.

`cursor.`**`maxTimeMS`**`(`*<time limit>*`)`

Specifies a cumulative time limit in milliseconds for processing operations on a cursor.

The `maxTimeMS()` (page 90) method has the following parameter:

param integer milliseconds Specifies a cumulative time limit in milliseconds for processing operations on the cursor.

Important: `maxTimeMS()` (page 90) is not related to the `NoCursorTimeout` query flag. `maxTimeMS()` (page 90) relates to processing time, while `NoCursorTimeout` relates to idle time. A cursor's idle time does not contribute towards its processing time.

Behaviors MongoDB targets operations for termination if the associated cursor exceeds its allotted time limit. MongoDB terminates operations that exceed their allotted time limit, using the same mechanism as `db.killOp()` (page 119). MongoDB only terminates an operation at one of its designated interrupt points.

MongoDB does not count network latency towards a cursor's time limit.

Queries that generate multiple batches of results continue to return batches until the cursor exceeds its allotted time limit.

Examples

Example

The following query specifies a time limit of 50 milliseconds:

```
db.collection.find({description: /August [0-9]+, 1969/}).maxTimeMS(50)
```

cursor.min()

Definition

`cursor.min()`

Specifies the *inclusive* lower bound for a specific index in order to constrain the results of `find()` (page 36). `min()` (page 91) provides a way to specify lower bounds on compound key indexes.

The `min()` (page 91) method has the following parameter:

param document indexBounds The inclusive lower bound for the index keys.

The `indexBounds` parameter has the following prototype form:

```
{ field1: <min value>, field2: <min value2>, fieldN:<min valueN> }
```

The fields correspond to *all* the keys of a particular index *in order*. You can explicitly specify the particular index with the `hint()` (page 87) method. Otherwise, MongoDB selects the index using the fields in the `indexBounds`; however, if multiple indexes exist on same fields with different sort orders, the selection of the index may be ambiguous.

See also:

`max()` (page 88).

`min()` (page 91) exists primarily to support the `mongos` (page 601) process, and is a shell wrapper around the query modifier `$min` (page 559).

Behaviors

Interaction with Index Selection Because `min()` (page 91) requires an index on a field, and forces the query to use this index, you may prefer the `$gte` (page 401) operator for the query if possible. Consider the following example:

```
db.products.find( { _id: 7 } ).min( { price: 1.39 } )
```

The query will use the index on the `price` field, even if the index on `_id` may be better.

Index Bounds If you use `min()` (page 91) with `max()` (page 88) to specify a range, the index bounds specified in `min()` (page 91) and `max()` (page 88) must both refer to the keys of the same index.

`min()` without `max()` The `min` and `max` operators indicate that the system should avoid normal query planning. Instead they construct an index scan where the index bounds are explicitly specified by the values given in `min` and `max`.

If one of the two boundaries is not specified, then the query plan will be an index scan that is unbounded on one side. This may degrade performance compared to a query containing neither operator, or one that uses both operators to more tightly constrain the index scan.

Example This example assumes a collection named `products` that holds the following documents:

```
{ "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
{ "_id" : 2, "item" : "apple", "type" : "fuji", "price" : 1.99 }
{ "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : 1.99 }
{ "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
{ "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
{ "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
{ "_id" : 7, "item" : "orange", "type" : "cara cara", "price" : 2.99 }
{ "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
{ "_id" : 9, "item" : "orange", "type" : "satsuma", "price" : 1.99 }
{ "_id" : 8, "item" : "orange", "type" : "valencia", "price" : 0.99 }
```

The collection has the following indexes:

```
{ "_id" : 1 }
{ "item" : 1, "type" : 1 }
{ "item" : 1, "type" : -1 }
{ "price" : 1 }
```

- Using the ordering of the `{ item: 1, type: 1 }` index, `min()` (page 91) limits the query to the documents that are at or above the index key bound of `item` equal to `apple` and `type` equal to `jonagold`, as in the following:

  ```
  db.products.find().min( { item: 'apple', type: 'jonagold' } ).hint( { item: 1, type: 1 } )
  ```

  The query returns the following documents:

  ```
  { "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
  { "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
  { "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
  { "_id" : 7, "item" : "orange", "type" : "cara cara", "price" : 2.99 }
  { "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
  { "_id" : 9, "item" : "orange", "type" : "satsuma", "price" : 1.99 }
  { "_id" : 8, "item" : "orange", "type" : "valencia", "price" : 0.99 }
  ```

  If the query did not explicitly specify the index with the `hint()` (page 87) method, it is ambiguous as to whether `mongod` (page 583) would select the `{ item: 1, type: 1 }` index ordering or the `{ item: 1, type: -1 }` index ordering.

- Using the ordering of the index `{ price: 1 }`, `min()` (page 91) limits the query to the documents that are at or above the index key bound of `price` equal to `1.39` and `max()` (page 88) limits the query to the documents that are below the index key bound of `price` equal to `1.99`:

  ```
  db.products.find().min( { price: 1.39 } ).max( { price: 1.99 } ).hint( { price: 1 } )
  ```

  The query returns the following documents:

  ```
  { "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
  { "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
  { "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
  { "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
  { "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
  ```

cursor.next()

cursor.**next**()

Returns The next document in the cursor returned by the db.collection.find() (page 36) method. See cursor.hasNext() (page 87) related functionality.

cursor.objsLeftInBatch()

cursor.**objsLeftInBatch**()

cursor.objsLeftInBatch() (page 93) returns the number of documents remaining in the current batch.

The MongoDB instance returns response in batches. To retrieve all the documents from a cursor may require multiple batch responses from the MongoDB instance. When there are no more documents remaining in the current batch, the cursor will retrieve another batch to get more documents until the cursor exhausts.

cursor.readPref()

Definition

cursor.**readPref**(*mode*, *tagSet*)

Append readPref() (page 93) to a cursor to control how the client routes the query to members of the replica set.

param string mode One of the following *read preference* modes: primary, primaryPreferred, secondary, secondaryPreferred, or nearest

param array tagSet A tag set used to specify custom read preference modes. For details, see *replica-set-read-preference-tag-sets*.

Note: You must apply readPref() (page 93) to the cursor before retrieving any documents from the database.

---

cursor.showDiskLoc()

cursor.**showDiskLoc**()

Modifies the output of a query by adding a field $diskLoc to matching documents. $diskLoc contains disk location information and has the form:

```
"$diskLoc": {
  "file": <int>,
  "offset": <int>
}
```

cursor.showDiskLoc() (page 93) method is a wrapper around $showDiskLoc (page 561).

Returns A modified cursor object that contains documents with appended information that describes the on-disk location of the document.

Example The following operation appends the showDiskLoc() (page 93) method to the db.collection.find() (page 36) method in order to include in the matching documents the disk location information:

```
db.collection.find( { a: 1 } ).showDiskLoc()
```

The operation returns the following documents, which includes the `$diskLoc` field:

```
{
  "_id" : ObjectId("53908ccb18facd50a75bfbac"),
  "a" : 1,
  "b" : 1,
  "$diskLoc" : { "file" : 0, "offset" : 16195760 }
}
{
   "_id" : ObjectId("53908cd518facd50a75bfbad"),
   "a" : 1,
   "b" : 2,
   "$diskLoc" : { "file" : 0, "offset" : 16195824 }
}
```

The *projection* can also access the added field `$diskLoc`, as in the following example:

```
db.collection.find( { a: 1 }, { $diskLoc: 1 } ).showDiskLoc()
```

The operation returns just the `_id` field and the `$diskLoc` field in the matching documents:

```
{
  "_id" : ObjectId("53908ccb18facd50a75bfbac"),
  "$diskLoc" : { "file" : 0, "offset" : 16195760 }
}
{
   "_id" : ObjectId("53908cd518facd50a75bfbad"),
   "$diskLoc" : { "file" : 0, "offset" : 16195824 }
}
```

See also:

`$showDiskLoc` (page 561) for related functionality.

cursor.size()

`cursor.size()`

Returns A count of the number of documents that match the `db.collection.find()` (page 36) query after applying any `cursor.skip()` (page 94) and `cursor.limit()` (page 88) methods.

cursor.skip()

`cursor.skip()`

Call the `cursor.skip()` (page 94) method on a cursor to control where MongoDB begins returning results. This approach may be useful in implementing "paged" results.

Note: You must apply `cursor.skip()` (page 94) to the cursor before retrieving any documents from the database.

Consider the following JavaScript function as an example of the skip function:

```
function printStudents(pageNumber, nPerPage) {
   print("Page: " + pageNumber);
   db.students.find().skip(pageNumber > 0 ? ((pageNumber-1)*nPerPage) : 0).limit(nPerPage).forEa
}
```

The `cursor.skip()` (page 94) method is often expensive because it requires the server to walk from the beginning of the collection or index to get the offset or skip position before beginning to return results. As the offset (e.g. `pageNumber` above) increases, `cursor.skip()` (page 94) will become slower and more CPU intensive. With larger collections, `cursor.skip()` (page 94) may become IO bound.

Consider using range-based pagination for these kinds of tasks. That is, query for a range of objects, using logic within the application to determine the pagination rather than the database itself. This approach features better index utilization, if you do not need to easily jump to a specific page.

cursor.snapshot()

`cursor.snapshot()`

Append the `snapshot()` (page 95) method to a cursor to toggle the "snapshot" mode. This ensures that the query will not return a document multiple times, even if intervening write operations result in a move of the document due to the growth in document size.

Warning:

- You must apply `snapshot()` (page 95) to the cursor before retrieving any documents from the database.
- You can only use `snapshot()` (page 95) with **unsharded** collections.

The `snapshot()` (page 95) does not guarantee isolation from insertion or deletions.

The `snapshot()` (page 95) traverses the index on the `_id` field. As such, `snapshot()` (page 95) **cannot** be used with `sort()` (page 95) or `hint()` (page 87).

Queries with results of less than 1 megabyte are effectively implicitly snapshotted.

cursor.sort()

Definition

`cursor.sort(`*sort*`)`

Specifies the order in which the query returns matching documents. You must apply `sort()` (page 95) to the cursor before retrieving any documents from the database.

The `sort()` (page 95) method has the following parameter:

param document sort A document that defines the sort order of the result set.

The `sort` parameter contains field and value pairs, in the following form:

```
{ field: value }
```

The sort document can specify *ascending or descending sort on existing fields* (page 96) or *sort on computed metadata* (page 96).

Behaviors

Result Ordering Unless you specify the `sort()` (page 95) method or use the `$near` (page 429) operator, MongoDB does **not** guarantee the order of query results.

Ascending/Descending Sort Specify in the sort parameter the field or fields to sort by and a value of `1` or `-1` to specify an ascending or descending sort respectively.

The following sample document specifies a descending sort by the `age` field and then an ascending sort by the `posts` field:

```
{ age : -1, posts: 1 }
```

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectId
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

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

The comparison treats a non-existent field as it would an empty BSON Object. As such, a sort on the `a` field in documents `{ }` and `{ a: null }` would treat the documents as equivalent in sort order.

With arrays, a less-than comparison or an ascending sort compares the smallest element of arrays, and a greater-than comparison or a descending sort compares the largest element of the arrays. As such, when comparing a field whose value is a single-element array (e.g. `[ 1 ]`) with non-array fields (e.g. `2`), the comparison is between `1` and `2`. A comparison of an empty array (e.g. `[ ]`) treats the empty array as less than `null` or a missing field.

MongoDB sorts `BinData` in the following order:

1. First, the length or size of the data.
2. Then, by the BSON one-byte subtype.
3. Finally, by the data, performing a byte-by-byte comparison.

Metadata Sort Specify in the sort parameter a new field name for the computed metadata and specify the `$meta` (page 449) expression as its value.

The following sample document specifies a descending sort by the `"textScore"` metadata:

```
{ score: { $meta: "textScore" } }
```

The specified metadata determines the sort order. For example, the `"textScore"` metadata sorts in descending order. See `$meta` (page 449) for details.

Limit Results The sort operation requires that the entire sort be able to complete within 32 megabytes.

When the sort operation consumes more than 32 megabytes, MongoDB returns an error. To avoid this error, either create an index to support the sort operation or use `sort()` (page 95) in conjunction with `limit()` (page 88). The specified limit must result in a number of documents that fall within the 32 megabyte limit.

For example, if the following sort operation `stocks_quotes` exceeds the 32 megabyte limit:

```
db.stocks.find().sort( { ticker: 1, date: -1 } )
```

Either create an index to support the sort operation:

```
db.stocks.ensureIndex( { ticker: 1, date: -1 } )
```

Or use `sort()` (page 95) in conjunction with `limit()` (page 88):

```
db.stocks.find().sort( { ticker: 1, date: -1 } ).limit(100)
```

Interaction with *Projection* When a set of results are both sorted and projected, the MongoDB query engine will always apply the sorting **first**.

Examples A collection `orders` contain the following documents:

```
{ _id: 1, item: { category: "cake", type: "chiffon" }, amount: 10 }
{ _id: 2, item: { category: "cookies", type: "chocolate chip" }, amount: 50 }
{ _id: 3, item: { category: "cookies", type: "chocolate chip" }, amount: 15 }
{ _id: 4, item: { category: "cake", type: "lemon" }, amount: 30 }
{ _id: 5, item: { category: "cake", type: "carrot" }, amount: 20 }
{ _id: 6, item: { category: "brownies", type: "blondie" }, amount: 10 }
```

The following query, which returns all documents from the `orders` collection, does not specify a sort order:

```
db.orders.find()
```

The query returns the documents in indeterminate order:

```
{ "_id" : 1, "item" : { "category" : "cake", "type" : "chiffon" }, "amount" : 10 }
{ "_id" : 2, "item" : { "category" : "cookies", "type" : "chocolate chip" }, "amount" : 50 }
{ "_id" : 3, "item" : { "category" : "cookies", "type" : "chocolate chip" }, "amount" : 15 }
{ "_id" : 4, "item" : { "category" : "cake", "type" : "lemon" }, "amount" : 30 }
{ "_id" : 5, "item" : { "category" : "cake", "type" : "carrot" }, "amount" : 20 }
{ "_id" : 6, "item" : { "category" : "brownies", "type" : "blondie" }, "amount" : 10 }
```

The following query specifies a sort on the `amount` field in descending order.

```
db.orders.find().sort( { amount: -1 } )
```

The query returns the following documents, in descending order of `amount`:

```
{ "_id" : 2, "item" : { "category" : "cookies", "type" : "chocolate chip" }, "amount" : 50 }
{ "_id" : 4, "item" : { "category" : "cake", "type" : "lemon" }, "amount" : 30 }
{ "_id" : 5, "item" : { "category" : "cake", "type" : "carrot" }, "amount" : 20 }
{ "_id" : 3, "item" : { "category" : "cookies", "type" : "chocolate chip" }, "amount" : 15 }
{ "_id" : 1, "item" : { "category" : "cake", "type" : "chiffon" }, "amount" : 10 }
{ "_id" : 6, "item" : { "category" : "brownies", "type" : "blondie" }, "amount" : 10 }
```

The following query specifies the sort order using the fields from a sub-document `item`. The query sorts first by the `category` field in ascending order, and then within each `category`, by the `type` field in ascending order.

```
db.orders.find().sort( { "item.category": 1, "item.type": 1 } )
```

The query returns the following documents, ordered first by the `category` field, and within each category, by the `type` field:

```
{ "_id" : 6, "item" : { "category" : "brownies", "type" : "blondie" }, "amount" : 10 }
{ "_id" : 5, "item" : { "category" : "cake", "type" : "carrot" }, "amount" : 20 }
{ "_id" : 1, "item" : { "category" : "cake", "type" : "chiffon" }, "amount" : 10 }
{ "_id" : 4, "item" : { "category" : "cake", "type" : "lemon" }, "amount" : 30 }
{ "_id" : 2, "item" : { "category" : "cookies", "type" : "chocolate chip" }, "amount" : 50 }
{ "_id" : 3, "item" : { "category" : "cookies", "type" : "chocolate chip" }, "amount" : 15 }
```

Return in Natural Order The `$natural` (page 563) parameter returns items according to their *natural order* within the database. This ordering is an internal implementation feature, and you should not rely on any particular structure within it.

Typically, the natural order reflects insertion order, *except* when documents relocate because of *document growth due to updates* or remove operations free up space which are then taken up by newly inserted documents.

Consider the sequence of insert operations to the `trees` collection:

```
db.trees.insert( { _id: 1, common_name: "oak", genus: "quercus" } )
db.trees.insert( { _id: 2, common_name: "chestnut", genus: "castanea" } )
db.trees.insert( { _id: 3, common_name: "maple", genus: "aceraceae" } )
db.trees.insert( { _id: 4, common_name: "birch", genus: "betula" } )
```

The following query returns the documents in the natural order:

```
db.trees.find().sort( { $natural: 1 } )
```

The documents return in the following order:

```
{ "_id" : 1, "common_name" : "oak", "genus" : "quercus" }
{ "_id" : 2, "common_name" : "chestnut", "genus" : "castanea" }
{ "_id" : 3, "common_name" : "maple", "genus" : "aceraceae" }
{ "_id" : 4, "common_name" : "birch", "genus" : "betula" }
```

Update a document such that the document outgrows its current allotted space:

```
db.trees.update(
   { _id: 1 },
   { $set: { famous_oaks: [ "Emancipation Oak", "Goethe Oak" ] } }
)
```

Rerun the query to returns the documents in natural order:

```
db.trees.find().sort( { $natural: 1 } )
```

The documents return in the following natural order:

```
{ "_id" : 2, "common_name" : "chestnut", "genus" : "castanea" }
{ "_id" : 3, "common_name" : "maple", "genus" : "aceraceae" }
{ "_id" : 4, "common_name" : "birch", "genus" : "betula" }
{ "_id" : 1, "common_name" : "oak", "genus" : "quercus", "famous_oaks" : [ "Emancipation Oak", "Goeth
```

See also:

`$natural` (page 563)

cursor.toArray()

cursor.**toArray**()

The toArray() (page 99) method returns an array that contains all the documents from a cursor. The method iterates completely the cursor, loading all the documents into RAM and exhausting the cursor.

Returns An array of documents.

Consider the following example that applies toArray() (page 99) to the cursor returned from the find() (page 36) method:

```
var allProductsArray = db.products.find().toArray();

if (allProductsArray.length > 0) { printjson (allProductsArray[0]); }
```

The variable allProductsArray holds the array of documents returned by toArray() (page 99).

2.1.3 Database

Database Methods

Name	Description
db.auth() (page 100)	Authenticates a user to a database.
db.changeUserPassword() (page 152)	Changes an existing user's password.
db.cloneCollection() (page 101)	Copies data directly between MongoDB instances. Wraps cloneCollec
db.cloneDatabase() (page 101)	Copies a database from a remote host to the current host. Wraps clone (p
db.commandHelp() (page 101)	Returns help information for a *database command*.
db.copyDatabase() (page 102)	Copies a database to another database on the current host. Wraps copydb
db.createCollection() (page 104)	Creates a new collection. Commonly used to create a capped collection.
db.currentOp() (page 105)	Reports the current in-progress operations.
db.dropDatabase() (page 111)	Removes the current database.
db.eval() (page 112)	Passes a JavaScript function to the mongod (page 583) instance for server
db.fsyncLock() (page 113)	Flushes writes to disk and locks the database to prevent write operations ar
db.fsyncUnlock() (page 114)	Allows writes to continue on a database locked with db.fsyncLock()
db.getCollection() (page 114)	Returns a collection object. Used to access collections with names that are
db.getCollectionNames() (page 114)	Lists all collections in the current database.
db.getLastError() (page 114)	Checks and returns the status of the last operation. Wraps getLastErro
db.getLastErrorObj() (page 115)	Returns the status document for the last operation. Wraps getLastErro
db.getLogComponents() (page 115)	Returns the log message verbosity levels.
db.getMongo() (page 116)	Returns the Mongo() (page 202) connection object for the current connec
db.getName() (page 116)	Returns the name of the current database.
db.getPrevError() (page 116)	Returns a status document containing all errors since the last error reset. W
db.getProfilingLevel() (page 117)	Returns the current profiling level for database operations.
db.getProfilingStatus() (page 117)	Returns a document that reflects the current profiling level and the profilin
db.getReplicationInfo() (page 117)	Returns a document with replication statistics.
db.getSiblingDB() (page 118)	Provides access to the specified database.
db.help() (page 118)	Displays descriptions of common db object methods.
db.hostInfo() (page 119)	Returns a document with information about the system MongoDB runs on
db.isMaster() (page 119)	Returns a document that reports the state of the replica set.
db.killOp() (page 119)	Terminates a specified operation.
db.listCommands() (page 120)	Displays a list of common database commands.
db.loadServerScripts() (page 120)	Loads all scripts in the system.js collection for the current database int
db.logout() (page 120)	Ends an authenticated session.

Table 2.2 – continued from previous page

Name	Description
db.printCollectionStats() (page 121)	Prints statistics from every collection. Wraps db.collection.stats
db.printReplicationInfo() (page 121)	Prints a report of the status of the replica set from the perspective of the pr
db.printShardingStatus() (page 122)	Prints a report of the sharding configuration and the chunk ranges.
db.printSlaveReplicationInfo() (page 122)	Prints a report of the status of the replica set from the perspective of the se
db.removeUser() (page 157)	Removes a user from a database.
db.repairDatabase() (page 123)	Runs a repair routine on the current database.
db.resetError() (page 123)	Resets the error message returned by db.getPrevError() (page 116)
db.runCommand() (page 123)	Runs a *database command* (page 210).
db.serverBuildInfo() (page 124)	Returns a document that displays the compilation parameters for the mong
db.serverCmdLineOpts() (page 124)	Returns a document with information about the runtime used to start the M
db.serverStatus() (page 124)	Returns a document that provides an overview of the state of the database
db.setLogLevel() (page 124)	Sets a single log message verbosity level.
db.setProfilingLevel() (page 126)	Modifies the current level of database profiling.
db.shutdownServer() (page 126)	Shuts down the current mongod (page 583) or mongos (page 601) proces
db.stats() (page 126)	Returns a document that reports on the state of the current database.
db.upgradeCheck() (page 127)	Performs a preliminary check for upgrade preparedness for a specific datab
db.upgradeCheckAllDBs() (page 128)	Performs a preliminary check for upgrade preparedness for all databases a
db.version() (page 130)	Returns the version of the mongod (page 583) instance.

db.auth()

Definition

db.**auth**(*username*, *password*)

Allows a user to authenticate to the database from within the shell.

param string username Specifies an existing username with access privileges for this database.

param string password Specifies the corresponding password.

param string mechanism Specifies the authentication mechanism used. Defaults to MONGODB-CR. PLAIN is used for SASL/LDAP authentication, available only in MongoDB Enterprise.

Alternatively, you can use *mongo --username*, *--password*, and *--authenticationMechanism* to specify authentication credentials.

--authenticationMechanism supports additional mechanisms not available when using db.auth() (page 100).

Note: The mongo (page 610) shell excludes all db.auth() (page 100) operations from the saved history.

Returns db.auth() (page 100) returns 0 when authentication is **not** successful, and 1 when the operation is successful.

db.changeUserPassword()

Definition

db.**changeUserPassword**(*username*, *password*)

Updates a user's password.

param string username Specifies an existing username with access privileges for this database.

param string password Specifies the corresponding password.

param string mechanism Specifies the authentication mechanism used. Defaults to `MONGODB-CR`. `PLAIN` is used for `SASL/LDAP authentication`, available only in MongoDB Enterprise.

Example The following operation changes the `reporting` user's password to `SOh3TbYhx8ypJPxmt1oOfL`:

```
db.changeUserPassword("reporting", "SOh3TbYhx8ypJPxmt1oOfL")
```

db.cloneCollection()

Definition

`db.cloneCollection` (*from*, *collection*, *query*)

Copies data directly between MongoDB instances. The `db.cloneCollection()` (page 101) wraps the `cloneCollection` (page 320) database command and accepts the following arguments:

param string from Host name of the MongoDB instance that holds the collection to copy.

param string collection The collection in the MongoDB instance that you want to copy. `db.cloneCollection()` (page 101) will only copy the collection with this name from *database* of the same name as the current database the remote MongoDB instance. If you want to copy a collection from a different database name you must use the `cloneCollection` (page 320) directly.

param document query A standard query document that limits the documents copied as part of the `db.cloneCollection()` (page 101) operation. All *query selectors* (page 400) available to the `find()` (page 36) are available here.

`db.cloneCollection()` (page 101) does not allow you to clone a collection through a `mongos` (page 601). You must connect directly to the `mongod` (page 583) instance.

db.cloneDatabase()

Definition

`db.cloneDatabase` ("*hostname*")

Copies a remote database to the current database. The command assumes that the remote database has the same name as the current database.

param string hostname The hostname of the database to copy.

This method provides a wrapper around the MongoDB *database command* "`clone` (page 320)." The `copydb` (page 326) database command provides related functionality.

Example To clone a database named `importdb` on a host named `hostname`, issue the following:

```
use importdb
db.cloneDatabase("hostname")
```

New databases are implicitly created, so the current host does not need to have a database named `importdb` for this command to succeed.

db.commandHelp()

Description

db.**commandHelp**(*command*)

Displays help text for the specified *database command*. See the *Database Commands* (page 210).

The db.commandHelp() (page 101) method has the following parameter:

param string command The name of a *database command*.

db.copyDatabase()

Definition

db.**copyDatabase**(*fromdb*, *todb*, *fromhost*, *username*, *password*)

Copies a database from a remote host to the current host or copies a database to another database within the current host. db.copyDatabase() (page 102) wraps the copydb (page 326) command and takes the following arguments:

param string fromdb The name of the source database.

param string todb The name of the destination database.

param string fromhost The name of the source database host. Omit the hostname to copy from one database to another on the same server.

field string username The username credentials on the fromhost for authentication and authorization.

field string password The password on the fromhost for authentication and authorization. The method does not transmit the password in plaintext.

Behavior Be aware of the following properties of db.copyDatabase() (page 102):

- db.copyDatabase() (page 102) runs on the destination mongod (page 583) instance, i.e. the host receiving the copied data.
- If the destination mongod (page 583) has authorization enabled, db.copyDatabase() (page 102) *must* specify the credentials of a user present in the *source* database who has the privileges described in *Required Access* (page 102).
- db.copyDatabase() (page 102) creates the target database if it does not exist.
- db.copyDatabase() (page 102) requires enough free disk space on the host instance for the copied database. Use the db.stats() (page 126) operation to check the size of the database on the source mongod (page 583) instance.
- db.copyDatabase() (page 102) and clone (page 320) do not produce point-in-time snapshots of the source database. Write traffic to the source or destination database during the copy process will result in divergent data sets.
- db.copyDatabase() (page 102) does not lock the destination server during its operation, so the copy will occasionally yield to allow other operations to complete.

Required Access Changed in version 2.6.

The copydb (page 326) command requires the following authorization on the target and source databases.

Source Database (fromdb)

Source is non-admin Database If the source database is a non-admin database, you must have privileges that specify find action on the source database, and find action on the system.js collection in the source database. For example:

```
{ resource: { db: "mySourceDB", collection: "" }, actions: [ "find" ] }
{ resource: { db: "mySourceDB", collection: "system.js" }, actions: [ "find" ] }
```

If the source database is on a remote server, you also need the find action on the system.indexes and system.namespaces collections in the source database; e.g.

```
{ resource: { db: "mySourceDB", collection: "system.indexes" }, actions: [ "find" ] }
{ resource: { db: "mySourceDB", collection: "system.namespaces" }, actions: [ "find" ] }
```

Source is admin Database If the source database is the admin database, you must have privileges that specify find action on the admin database, and find action on the system.js, system.users, system.roles, and system.version collections in the admin database. For example:

```
{ resource: { db: "admin", collection: "" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.js" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.users" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.roles" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.version" }, actions: [ "find" ] }
```

If the source database is on a remote server, the you also need the find action on the system.indexes and system.namespaces collections in the admin database; e.g.

```
{ resource: { db: "admin", collection: "system.indexes" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.namespaces" }, actions: [ "find" ] }
```

Source Database is on a Remote Server If copying from a remote server and the remote server has authentication enabled, you must authenticate to the remote host as a user with the proper authorization. See *Authentication* (page 104).

Target Database (todb)

Copy from non-admin Database If the source database is not the admin database, you must have privileges that specify insert and createIndex actions on the target database, and insert action on the system.js collection in the target database. For example:

```
{ resource: { db: "myTargetDB", collection: "" }, actions: [ "insert", "createIndex" ] }
{ resource: { db: "myTargetDB", collection: "system.js" }, actions: [ "insert" ] }
```

Copy from admin Database If the source database is the admin database, you must have privileges that specify insert and createIndex actions on the target database, and insert action on the system.js, system.users, system.roles, and system.version collections in the target database. For example:

```
{ resource: { db: "myTargetDB", collection: "" }, actions: [ "insert", "createIndex" ] },
{ resource: { db: "myTargetDB", collection: "system.js" }, actions: [ "insert" ] },
{ resource: { db: "myTargetDB", collection: "system.users" }, actions: [ "insert" ] },
{ resource: { db: "myTargetDB", collection: "system.roles" }, actions: [ "insert" ] },
{ resource: { db: "myTargetDB", collection: "system.version" }, actions: [ "insert" ] }
```

Authentication If copying from a remote server and the remote server has authentication enabled, then you must include the `<username>` and `<password>`. The method does not transmit the password in plaintext.

Example To copy a database named `records` into a database named `archive_records`, use the following invocation of `db.copyDatabase()` (page 102):

```
db.copyDatabase('records', 'archive_records')
```

See also:

`clone` (page 320)

db.createCollection()

Definition

`db.createCollection`(*name*, *options*)

Creates a new collection explicitly.

Because MongoDB creates a collection implicitly when the collection is first referenced in a command, this method is used primarily for creating new *capped collections*. This is also used to pre-allocate space for an ordinary collection.

The `db.createCollection()` (page 104) method has the following prototype form:

```
db.createCollection(<name>, { capped: <boolean>,
                              autoIndexId: <boolean>,
                              size: <number>,
                              max: <number>,
                              storageEngine: <document> } )
```

The `db.createCollection()` (page 104) method has the following parameters:

param string name The name of the collection to create.

param document options Configuration options for creating a capped collection or for preallocating space in a new collection.

The `options` document creates a capped collection or preallocates space in a new ordinary collection. The `options` document contains the following fields:

field Boolean capped Enables a *capped collection*. To create a capped collection, specify `true`. If you specify `true`, you must also set a maximum size in the `size` field.

field Boolean autoIndexId Specify `false` to disable the automatic creation of an index on the `_id` field. Before 2.2, the default value for `autoIndexId` was `false`. See *_id Fields and Indexes on Capped Collections* (page 798) for more information.

Do **not** set `autoIndexId` to `true` for replicated collections.

field number size Specifies a maximum size in bytes for a capped collection. The `size` field is required for capped collections, and ignored for other collections.

field number max The maximum number of documents allowed in the capped collection. The `size` limit takes precedence over this limit. If a capped collection reaches its maximum `size` before it reaches the maximum number of documents, MongoDB removes old documents. If you prefer to use this limit, ensure that the `size` limit, which is required, is sufficient to contain the documents limit.

field boolean usePowerOf2Sizes Deprecated since version 2.8: For `mmapv1`, all collections have `usePowerOf2Sizes` (page 322) allocation unless the `noPadding` option is `true`.

field boolean noPadding Changed in version 2.8: For `mmapv1`, `noPadding` flag removes all additional space in the record allocation, so that the allocation does not permit documents to grow after insertion without a new allocation. Defaults to `false`. Use for insert-only workloads.

field document storageEngine New in version 2.8.

Allows users to specify configuration to the storage engine on a per-collection basis when creating a collection. The value of the `storageEngine` option should take the following form:e

```
{ <storage-engine-name>: <options> }
```

Storage engine configuration specified when creating collections are validated and logged to the *oplog* during replication to support replica sets with members that use different storage engines.

Examples

Create Capped Collection Capped collections have maximum size or document counts that prevent them from growing beyond maximum thresholds. All capped collections must specify a maximum size and may also specify a maximum document count. MongoDB removes older documents if a collection reaches the maximum size limit before it reaches the maximum document count. Consider the following example:

```
db.createCollection("log", { capped : true, size : 5242880, max : 5000 } )
```

This command creates a collection named `log` with a maximum size of 5 megabytes and a maximum of 5000 documents.

The following command simply pre-allocates a 2-gigabyte, uncapped collection named `people`:

```
db.createCollection("people", { size: 2147483648 } )
```

This command provides a wrapper around the database command `create` (page 333). See `http://docs.mongodb.org/manual/core/capped-collections` for more information about capped collections.

Specify Storage Engine Options New in version 2.8.

You can specify collection-specific storage engine configuration options when you create a collection with `db.createCollection()` (page 104). Consider the following operation:

```
db.createCollection( "users", { storageEngine: {
                                    wiredTiger: { configString: "<option>=<setting>" }})
```

This operation creates a new collection named `users` with a specific configuration string that MongoDB will pass to the `wiredTiger` storage engine. See the WiredTiger documentation of collection level options[8] for specific `wiredTiger` options.

db.currentOp()

Definition

`db.currentOp()`

Returns a *document* that contains information on in-progress operations for the database instance.

`db.currentOp()` (page 105) method has the following form:

[8] http://source.wiredtiger.com/2.4.1/struct_w_t___s_e_s_s_i_o_n.html#a358ca4141d59c345f401c58501276bbb

```
db.currentOp(<operations>)
```

The db.currentOp() (page 105) method can take the following *optional* argument:

param boolean or document operations Specifies the operations to report on. Can pass either a boolean or a document.

Specify true to include operations on idle connections and system operations. Specify a document with query conditions to report only on operations that match the conditions. See *Behavior* (page 106) for details.

Behavior If you pass in true to db.currentOp() (page 105), the method returns information on all operations, including operations on idle connections and system operations.

```
db.currentOp(true)
```

Passing in true is equivalent to passing in a query document of { '$all': true }.

If you pass a query document to db.currentOp() (page 105), the output returns information only for the current operations that match the query. You can query on the *Output Fields* (page 108). See *Examples* (page 106).

You can also specify { '$all': true } query document to return information on all in-progress operations, including operations on idle connections and system operations. If you specify in the query document other conditions as well as '$all': true, only the '$all': true applies.

Access Control On systems running with authorization, a user must have access that includes the inprog action. For example, see *create-role-to-manage-ops*.

Examples The following examples use the db.currentOp() (page 105) method with various query documents to filter the output.

Write Operations Waiting for a Lock The following example returns information on all write operations that are waiting for a lock:

```
db.currentOp(
   {
     "waitingForLock" : true,
     $or: [
        { "op" : { "$in" : [ "insert", "update", "remove" ] } },
        { "query.update": { $exists: true } },
        { "query.insert": { $exists: true } },
        { "query.remove": { $exists: true } }
    ]
   }
)
```

Active Operations with no Yields The following example returns information on all active running operations that have never yielded:

```
db.currentOp(
   {
     "active" : true,
     "numYields" : 0,
     "waitingForLock" : false
```

```
    }
)
```

Active Operations on a Specific Database The following example returns information on all active operations for database `db1` that have been running longer than 3 seconds:

```
db.currentOp(
   {
     "active" : true,
     "secs_running" : { "$gt" : 3 },
     "ns" : /^db1./
   }
)
```

Active Indexing Operations The following example returns information on index creation operations:

```
db.currentOp(
    {
      $or: [
        { op: "query", "query.createIndexes": { $exists: true } },
        { op: "insert", ns: /\.system\.indexes\b/ }
      ]
    }
)
```

Output Example The following is an example of `db.currentOp()` (page 105) output.

```
{
  "inprog": [
       {
         "opid" : <number>,
         "active" : <boolean>,
         "secs_running" : <NumberLong()>,
         "microsecs_running" : <number>,
         "op" : <string>,
         "ns" : <string>,
         "query" : <document>,
         "insert" : <document>,
         "planSummary": <string>,
         "client" : <string>,
         "desc" : <string>,
         "threadId" : <string>,
         "connectionId" : <number>,
         "locks" : {
             "^" : <string>,
             "^local" : <string>,
             "^<database>" : <string>
         },
         "waitingForLock" : <boolean>,
         "msg": <string>,
         "progress" : {
             "done" : <number>,
             "total" : <number>
         },
         "killPending" : <boolean>,
```

```
          "numYields" : <number>,
          "lockStats" : {
              "timeLockedMicros" : {
                  "R" : <NumberLong()>,
                  "W" : <NumberLong()>,
                  "r" : <NumberLong()>,
                  "w" : <NumberLong()>
              },
              "timeAcquiringMicros" : {
                  "R" : <NumberLong()>,
                  "W" : <NumberLong()>,
                  "r" : <NumberLong()>,
                  "w" : <NumberLong()>
              }
          }
        },
        ...
    ]
}
```

Output Fields

`currentOp.opid`
: The identifier for the operation. You can pass this value to `db.killOp()` (page 119) in the `mongo` (page 610) shell to terminate the operation.

  Warning: Terminate running operations with extreme caution. Only use `db.killOp()` (page 119) to terminate operations initiated by clients and *do not* terminate internal database operations.

`currentOp.active`
: A boolean value specifying whether the operation has started. Value is `true` if the operation has started or `false` if the operation is queued and waiting for a lock to run. `active` (page 108) may be `true` even if the operation has yielded to another operation.

`currentOp.secs_running`
: The duration of the operation in seconds. MongoDB calculates this value by subtracting the current time from the start time of the operation.

  Only appears if the operation is running, (i.e. if `active` (page 108) is `true`).

`currentOp.microsecs_running`
: New in version 2.6.

  The duration of the operation in microseconds. MongoDB calculates this value by subtracting the current time from the start time of the operation.

  Only appears if the operation is running, (i.e. if `active` (page 108) is `true`).

`currentOp.op`
: A string that identifies the type of operation. The possible values are:

  - `"none"`
  - `"update"`
  - `"insert"`
  - `"query"`
  - `"getmore"`

- "remove"
- "killcursors"

The "query" type includes operations that use the insert (page 247), update (page 251), and delete (page 234) commands. Write operations that do not use the aforementioned write commands will show with the appropriate "insert", "update", or "remove" value.

currentOp.**ns**
The *namespace* the operation targets. A namespace consists of the *database* name and the *collection* name concatenated with a dot (.); i.e., "<database>.<collection>".

currentOp.**insert**
Contains the document to be inserted for operations with op (page 108) value of "insert". Only appears for operations with op (page 108) value "insert".

Insert operations such as db.collection.insert() (page 55) that use the insert (page 247) command will have op (page 108) value of "query".

currentOp.**query**
A document containing information on current operation if op (page 108) value is *not* "insert". query (page 109) does not appear for op (page 108) of type "insert".

Write operations that use the insert (page 247), update (page 251), and delete (page 234) commands have op (page 108) value of "query" and the corresponding query (page 109) contains information on these operations.

For example, the following query (page 109) field contains information for an update operation:

```
"query" : {
            "update" : "grades",
            "updates" : [
               {
                  "q" : {
                     "x" : {
                        "$gt" : 70
                     }
                  },
                  "u" : {
                     "$set" : {
                        "y" : 1
                     }
                  },
                  "multi" : true,
                  "upsert" : false
               }
            ],
            "ordered" : true
         }
```

The document can be empty for op (page 108) types such as "getmore".

currentOp.**planSummary**
A string that contains the query plan to help debug slow queries.

currentOp.**client**
The IP address (or hostname) and the ephemeral port of the client connection where the operation originates. If your inprog array has operations from many different clients, use this string to relate operations to clients.

For some commands, including findAndModify (page 239) and db.eval() (page 112), the client will be 0.0.0.0:0, rather than an actual client.

currentOp.**desc**
A description of the client. This string includes the connectionId (page 110).

currentOp.**threadId**
An identifier for the thread that services the operation and its connection.

currentOp.**connectionId**
An identifier for the connection where the operation originated.

currentOp.**locks**
New in version 2.2.

The locks (page 110) document reports by databases the types of locks the operation currently holds. The possible lock types are:

- R represents the global read lock,
- W represents the global write lock,
- r represents the database specific read lock, and
- w represents the database specific write lock.

currentOp.locks.**^**
^ (page 110) reports on the use of the global lock for the mongod (page 583) instance. All operations must hold the global lock for some phases of operation.

currentOp.locks.**^local**
^local (page 110) reports on the lock for the local database. MongoDB uses the local database for a number of operations, but the most frequent use of the local database is for the *oplog* used in replication.

currentOp.locks.**^<database>**
locks.^<database> (page 110) reports on the lock state for the database that this operation targets.

currentOp.**waitingForLock**
Returns a boolean value. waitingForLock (page 110) is true if the operation is waiting for a lock and false if the operation has the required lock.

currentOp.**msg**
The msg (page 110) provides a message that describes the status and progress of the operation. In the case of indexing or mapReduce operations, the field reports the completion percentage.

currentOp.**progress**
Reports on the progress of mapReduce or indexing operations. The progress (page 110) fields corresponds to the completion percentage in the msg (page 110) field. The progress (page 110) specifies the following information:

currentOp.progress.**done**
Reports the number completed.

currentOp.progress.**total**
Reports the total number.

currentOp.**killPending**
Returns true if the operation is currently flagged for termination. When the operation encounters its next safe termination point, the operation will terminate.

currentOp.**numYields**
numYields (page 110) is a counter that reports the number of times the operation has yielded to allow other operations to complete.

Typically, operations yield when they need access to data that MongoDB has not yet fully read into memory. This allows other operations that have data in memory to complete quickly while MongoDB reads in data for the yielding operation.

currentOp.**lockStats**
New in version 2.2.

The lockStats (page 111) document reflects the amount of time the operation has spent both acquiring and holding locks. lockStats (page 111) reports data on a per-lock type, with the following possible lock types:

- R represents the global read lock,
- W represents the global write lock,
- r represents the database specific read lock, and
- w represents the database specific write lock.

currentOp.**timeLockedMicros**
The timeLockedMicros (page 111) document reports the amount of time the operation has spent holding a specific lock.

For operations that require more than one lock, like those that lock the local database to update the *oplog*, then the values in this document can be longer than this value may be longer than the total length of the operation (i.e. secs_running (page 108).)

currentOp.timeLockedMicros.**R**
Reports the amount of time in microseconds the operation has held the global read lock.

currentOp.timeLockedMicros.**W**
Reports the amount of time in microseconds the operation has held the global write lock.

currentOp.timeLockedMicros.**r**
Reports the amount of time in microseconds the operation has held the database specific read lock.

currentOp.timeLockedMicros.**w**
Reports the amount of time in microseconds the operation has held the database specific write lock.

currentOp.**timeAcquiringMicros**
The timeAcquiringMicros (page 111) document reports the amount of time the operation has spent *waiting* to acquire a specific lock.

currentOp.timeAcquiringMicros.**R**
Reports the mount of time in microseconds the operation has waited for the global read lock.

currentOp.timeAcquiringMicros.**W**
Reports the mount of time in microseconds the operation has waited for the global write lock.

currentOp.timeAcquiringMicros.**r**
Reports the mount of time in microseconds the operation has waited for the database specific read lock.

currentOp.timeAcquiringMicros.**w**
Reports the mount of time in microseconds the operation has waited for the database specific write lock.

db.dropDatabase()

db.**dropDatabase**()
Removes the current database. Does not change the current database, so the insertion of any documents in this database will allocate a fresh set of data files.

Changed in version 2.6: This command does not delete the *users* associated with the current database. To drop the associated users, run the `dropAllUsersFromDatabase` (page 266) command in the database you are deleting.

See also:

`dropDatabase` (page 334)

db.eval()

Definition

`db.eval` (*function*, *arguments*)

Deprecated since version 2.8.0.

Provides the ability to run JavaScript code on the MongoDB server.

The helper `db.eval()` (page 112) in the `mongo` (page 610) shell wraps the `eval` (page 237) command. Therefore, the helper method shares the characteristics and behavior of the underlying command with *one exception*: `db.eval()` (page 112) method does not support the `nolock` option.

The method accepts the following parameters:

param JavaScript function function A JavaScript function to execute.

param list arguments A list of arguments to pass to the JavaScript function. Omit if the function does not take arguments.

The JavaScript function need not take any arguments, as in the first example, or may optionally take arguments as in the second:

```
function () {
  // ...
}

function (arg1, arg2) {
   // ...
}
```

Behavior

Write Lock By default, `db.eval()` (page 112) takes a global write lock while evaluating the JavaScript function. As a result, `db.eval()` (page 112) blocks all other read and write operations to the database while the `db.eval()` (page 112) operation runs.

To prevent the taking of the global write lock while evaluating the JavaScript code, use the `eval` (page 237) *command* with `nolock` set to `true`. `nolock` does not impact whether the operations within the JavaScript code take write locks.

For long running `db.eval()` (page 112) operation, consider using either the **eval** command with `nolock: true` or using `other server side code execution options`.

Sharded Data You can not use `db.eval()` (page 112) with *sharded* collections. In general, you should avoid using `db.eval()` (page 112) in *sharded clusters*; nevertheless, it is possible to use `db.eval()` (page 112) with non-sharded collections and databases stored in a *sharded cluster*.

Access Control Changed in version 2.6.

If authorization is enabled, you must have access to all actions on all resources in order to run `db.eval()` (page 112). Providing such access is not recommended, but if your organization requires a user to run `db.eval()` (page 112), create a role that grants `anyAction` on *resource-anyresource*. Do not assign this role to any other user.

JavaScript Engine Changed in version 2.4.

The V8 JavaScript engine, which became the default in 2.4, allows multiple JavaScript operations to execute at the same time. Prior to 2.4, `db.eval()` (page 112) executed in a single thread.

Examples The following is an example of the `db.eval()` (page 112) method:

```
db.eval( function(name, incAmount) {
            var doc = db.myCollection.findOne( { name : name } );

            doc = doc || { name : name , num : 0 , total : 0 , avg : 0 };

            doc.num++;
            doc.total += incAmount;
            doc.avg = doc.total / doc.num;

            db.myCollection.save( doc );
            return doc;
         },
         "eliot", 5 );
```

- The `db` in the function refers to the current database.
- `"eliot"` is the argument passed to the function, and corresponds to the `name` argument.
- `5` is an argument to the function and corresponds to the `incAmount` field.

If you want to use the server's interpreter, you must run `db.eval()` (page 112). Otherwise, the `mongo` (page 610) shell's JavaScript interpreter evaluates functions entered directly into the shell.

If an error occurs, `db.eval()` (page 112) throws an exception. The following is an example of an invalid function that uses the variable `x` without declaring it as an argument:

```
db.eval( function() { return x + x; }, 3 );
```

The statement results in the following exception:

```
{
   "errmsg" : "exception: JavaScript execution failed: ReferenceError: x is not defined near '{ retu
   "code" : 16722,
   "ok" : 0
}
```

See also:

`http://docs.mongodb.org/manual/core/server-side-javascript`

db.fsyncLock()

`db.fsyncLock()`

Forces the `mongod` (page 583) to flush all pending write operations to the disk and locks the *entire* `mongod` (page 583) instance to prevent additional writes until the user releases the lock with the `db.fsyncUnlock()` (page 114) command. `db.fsyncLock()` (page 113) is an administrative command.

This command provides a simple wrapper around a `fsync` (page 336) database command with the following syntax:

```
{ fsync: 1, lock: true }
```

This function locks the database and create a window for `backup operations`.

Important: `db.fsyncLock()` (page 113) *may* block reads, including those necessary to verify authentication. Such reads are necessary to establish new connections to a `mongod` (page 583) that enforces authorization checks.

Warning: When calling `db.fsyncLock()` (page 113), ensure that the connection is kept open to allow a subsequent call to `db.fsyncUnlock()` (page 114).
Closing the connection may make it difficult to release the lock.

db.fsyncUnlock()

`db.fsyncUnlock()`

Unlocks a `mongod` (page 583) instance to allow writes and reverses the operation of a `db.fsyncLock()` (page 113) operation. Typically you will use `db.fsyncUnlock()` (page 114) following a database `backup operation`.

`db.fsyncUnlock()` (page 114) is an administrative command.

db.getCollection()

Description

`db.getCollection(`*name*`)`

Returns a collection name. This is useful for a collection whose name might interact with the shell itself, such names that begin with _ or that mirror the *database commands* (page 210).

The `db.getCollection()` (page 114) method has the following parameter:

param string name The name of the collection.

db.getCollectionNames()

Definition

`db.getCollectionNames()`

Returns An array containing all collections in the existing database.

Considerations Changed in version 2.8.0.

If you use `db.getCollectionNames()` (page 114) from a version of the `mongo` (page 610) shell before 2.8.0, on a `mongod` (page 583) instance that uses the‘‘wiredTiger‘‘ storage engine, `db.getCollectionNames()` (page 114) will return no data, even if there are existing collections.

db.getLastError()

`db.getLastError()`

Changed in version 2.6: A new protocol for *write operations* (page 745) integrates write concerns with the write

operations, eliminating the need for a separate db.getLastError() (page 114) method. Write methods now return the status of the write operation, including error information.

In previous versions, clients typically used the db.getLastError() (page 114) method in combination with the write operations to ensure that the write succeeds.

Returns The last error message string.

Sets the level of *write concern* for confirming the success of write operations.

See

getLastError (page 245) and http://docs.mongodb.org/manual/reference/write-concern for all options, *Write Concern* for a conceptual overview, http://docs.mongodb.org/manual/core/write-operati for information about all write operations in MongoDB.

---

db.getLastErrorObj()

db.**getLastErrorObj**()

Changed in version 2.6: A new protocol for *write operations* (page 745) integrates write concerns with the write operations, eliminating the need for a separate db.getLastError() (page 114) method. Write methods now return the status of the write operation, including error information.

In previous versions, clients typically used the db.getLastError() (page 114) method in combination with the write operations to ensure that the write succeeds.

Returns A full *document* with status information.

See also:

Write Concern, http://docs.mongodb.org/manual/reference/write-concern, and *replica-set-write-concern*.

db.getLogComponents()

Definition

db.**getLogComponents**()

New in version 2.8.

Returns the current verbosity settings. The verbosity settings determine the amount of http://docs.mongodb.org/manual/reference/log-messages that MongoDB produces for each *log message component*.

If a component inherits the verbosity level of its parent, db.getLogComponents() (page 115) displays -1 for the component's verbosity.

Output The db.getLogComponents() (page 115) returns a document with the verbosity settings. For example:

```
{
   "verbosity" : 0,
   "accessControl" : {
      "verbosity" : -1
   },
   "command" : {
      "verbosity" : -1
   },
   "control" : {
```

```
      "verbosity" : -1
   },
   "geo" : {
      "verbosity" : -1
   },
   "index" : {
      "verbosity" : -1
   },
   "network" : {
      "verbosity" : -1
   },
   "query" : {
      "verbosity" : 2
   },
   "replication" : {
      "verbosity" : -1
   },
   "sharding" : {
      "verbosity" : -1
   },
   "storage" : {
      "verbosity" : 2,
      "journal" : {
         "verbosity" : -1
      }
   },
   "write" : {
      "verbosity" : -1
   }
}
```

To modify these settings, you can configure the `systemLog.verbosity` and `systemLog.component.<name>.verbosity` settings in the `configuration file` or set the `logComponentVerbosity` parameter using the `setParameter` (page 343) command or use the `db.setLogLevel()` (page 124) method. For examples, see *log-messages-configure-verbosity*.

db.getMongo()

`db.getMongo()`

Returns The current database connection.

`db.getMongo()` (page 116) runs when the shell initiates. Use this command to test that the `mongo` (page 610) shell has a connection to the proper database instance.

db.getName()

`db.getName()`

Returns the current database name.

db.getPrevError()

`db.getPrevError()`

Returns A status document, containing the errors.

Deprecated since version 1.6.

This output reports all errors since the last time the database received a `resetError` (page 250) (also `db.resetError()` (page 123)) command.

This method provides a wrapper around the `getPrevError` (page 247) command.

db.getProfilingLevel()

`db.getProfilingLevel()`

This method provides a wrapper around the database command "`profile` (page 366)" and returns the current profiling level.

Deprecated since version 1.8.4: Use `db.getProfilingStatus()` (page 117) for related functionality.

db.getProfilingStatus()

`db.getProfilingStatus()`

Returns The current `profile` (page 366) level and `slowOpThresholdMs` setting.

db.getReplicationInfo()

Definition

`db.getReplicationInfo()`

Returns A document with the status of the replica status, using data polled from the "*oplog*". Use this output when diagnosing issues with replication.

Output

`db.getReplicationInfo.logSizeMB`
Returns the total size of the *oplog* in megabytes. This refers to the total amount of space allocated to the oplog rather than the current size of operations stored in the oplog.

`db.getReplicationInfo.usedMB`
Returns the total amount of space used by the *oplog* in megabytes. This refers to the total amount of space currently used by operations stored in the oplog rather than the total amount of space allocated.

`db.getReplicationInfo.errmsg`
Returns an error message if there are no entries in the oplog.

`db.getReplicationInfo.oplogMainRowCount`
Only present when there are no entries in the oplog. Reports a the number of items or rows in the *oplog* (e.g. `0`).

`db.getReplicationInfo.timeDiff`
Returns the difference between the first and last operation in the *oplog*, represented in seconds.

Only present if there are entries in the oplog.

`db.getReplicationInfo.timeDiffHours`
Returns the difference between the first and last operation in the *oplog*, rounded and represented in hours.

Only present if there are entries in the oplog.

`db.getReplicationInfo.tFirst`
Returns a time stamp for the first (i.e. earliest) operation in the *oplog*. Compare this value to the last write operation issued against the server.

Only present if there are entries in the oplog.

`db.getReplicationInfo.tLast`
Returns a time stamp for the last (i.e. latest) operation in the *oplog*. Compare this value to the last write operation issued against the server.

Only present if there are entries in the oplog.

`db.getReplicationInfo.now`
Returns a time stamp that reflects reflecting the current time. The shell process generates this value, and the datum may differ slightly from the server time if you're connecting from a remote host as a result. Equivalent to `Date()` (page 199).

Only present if there are entries in the oplog.

db.getSiblingDB()

Definition

`db.getSiblingDB(<database>)`

param string database The name of a MongoDB database.

Returns A database object.

Used to return another database without modifying the `db` variable in the shell environment.

Example You can use `db.getSiblingDB()` (page 118) as an alternative to the `use <database>` helper. This is particularly useful when writing scripts using the `mongo` (page 610) shell where the `use` helper is not available. Consider the following sequence of operations:

```
db = db.getSiblingDB('users')
db.active.count()
```

This operation sets the `db` object to point to the database named `users`, and then returns a *count* (page 26) of the collection named `active`. You can create multiple `db` objects, that refer to different databases, as in the following sequence of operations:

```
users = db.getSiblingDB('users')
records = db.getSiblingDB('records')

users.active.count()
users.active.findOne()

records.requests.count()
records.requests.findOne()
```

This operation creates two `db` objects referring to different databases (i.e. `users` and `records`) and then returns a *count* (page 26) and an *example document* (page 46) from one collection in that database (i.e. `active` and `requests` respectively.)

db.help()

`db.help()`

Returns Text output listing common methods on the `db` object.

db.hostInfo()

db.**hostInfo**()

New in version 2.2.

Returns A document with information about the underlying system that the mongod (page 583) or mongos (page 601) runs on. Some of the returned fields are only included on some platforms.

db.hostInfo() (page 119) provides a helper in the mongo (page 610) shell around the hostInfo (page 358) The output of db.hostInfo() (page 119) on a Linux system will resemble the following:

```
{
   "system" : {
          "currentTime" : ISODate("<timestamp>"),
          "hostname" : "<hostname>",
          "cpuAddrSize" : <number>,
          "memSizeMB" : <number>,
          "numCores" : <number>,
          "cpuArch" : "<identifier>",
          "numaEnabled" : <boolean>
   },
   "os" : {
          "type" : "<string>",
          "name" : "<string>",
          "version" : "<string>"
   },
   "extra" : {
          "versionString" : "<string>",
          "libcVersion" : "<string>",
          "kernelVersion" : "<string>",
          "cpuFrequencyMHz" : "<string>",
          "cpuFeatures" : "<string>",
          "pageSize" : <number>,
          "numPages" : <number>,
          "maxOpenFiles" : <number>
   },
   "ok" : <return>
}
```

See hostInfo (page 358) for full documentation of the output of db.hostInfo() (page 119).

db.isMaster()

db.**isMaster**()

Returns A document that describes the role of the mongod (page 583) instance.

If the mongod (page 583) is a member of a *replica set*, then the ismaster (page 290) and secondary (page 290) fields report if the instance is the *primary* or if it is a *secondary* member of the replica set.

See

isMaster (page 289) for the complete documentation of the output of db.isMaster() (page 119).

---

db.killOp()

Description

db.**killOp**(*opid*)

Terminates an operation as specified by the operation ID. To find operations and their corresponding IDs, see db.currentOp() (page 105).

The db.killOp() (page 119) method has the following parameter:

param number opid An operation ID.

Warning: Terminate running operations with extreme caution. Only use db.killOp() (page 119) to terminate operations initiated by clients and *do not* terminate internal database operations.

db.listCommands()

db.**listCommands**()

Provides a list of all database commands. See the *Database Commands* (page 210) document for a more extensive index of these options.

db.loadServerScripts()

db.**loadServerScripts**()

db.loadServerScripts() (page 120) loads all scripts in the system.js collection for the current database into the mongo (page 610) shell session.

Documents in the system.js collection have the following prototype form:

```
{ _id : "<name>" , value : <function> } }
```

The documents in the system.js collection provide functions that your applications can use in any JavaScript context with MongoDB in this database. These contexts include $where (page 421) clauses and mapReduce (page 220) operations.

db.logout()

db.**logout**()

Ends the current authentication session. This function has no effect if the current session is not authenticated.

Note: If you're not logged in and using authentication, db.logout() (page 120) has no effect.

Changed in version 2.4: Because MongoDB now allows users defined in one database to have privileges on another database, you must call db.logout() (page 120) while using the same database context that you authenticated to.

If you authenticated to a database such as users or $external, you must issue db.logout() (page 120) against this database in order to successfully log out.

Example

Use the use <database-name> helper in the interactive mongo (page 610) shell, or the following db.getSiblingDB() (page 118) in the interactive shell or in mongo (page 610) shell scripts to change the db object:

```
db = db.getSiblingDB('<database-name>')
```

When you have set the database context and `db` object, you can use the `db.logout()` (page 120) to log out of database as in the following operation:

```
db.logout()
```

`db.logout()` (page 120) function provides a wrapper around the database command `logout` (page 264).

db.printCollectionStats()

`db.printCollectionStats()`

Provides a wrapper around the `db.collection.stats()` (page 71) method. Returns statistics from every collection separated by three hyphen characters.

Note: The `db.printCollectionStats()` (page 121) in the `mongo` (page 610) shell does **not** return *JSON*. Use `db.printCollectionStats()` (page 121) for manual inspection, and `db.collection.stats()` (page 71) in scripts.

See also:

collStats (page 348)

db.printReplicationInfo()

`db.printReplicationInfo()`

Prints a formatted report of the status of a *replica set* from the perspective of the *primary* set member if run on the primary. [9] The displayed report formats the data returned by `db.getReplicationInfo()` (page 117).

Note: The `db.printReplicationInfo()` (page 121) in the `mongo` (page 610) shell does **not** return *JSON*. Use `db.printReplicationInfo()` (page 121) for manual inspection, and `db.getReplicationInfo()` (page 117) in scripts.

The output of `db.printReplicationInfo()` (page 121) is identical to that of `rs.printReplicationInfo()` (page 175).

Output Example The following example is a sample output from the `db.printReplicationInfo()` (page 121) method run on the primary:

```
configured oplog size:   192MB
log length start to end: 65422secs (18.17hrs)
oplog first event time:  Mon Jun 23 2014 17:47:18 GMT-0400 (EDT)
oplog last event time:   Tue Jun 24 2014 11:57:40 GMT-0400 (EDT)
now:                     Thu Jun 26 2014 14:24:39 GMT-0400 (EDT)
```

Output Fields `db.printReplicationInfo()` (page 121) formats and prints the data returned by `db.getReplicationInfo()` (page 117):

configured oplog size Displays the `db.getReplicationInfo.logSizeMB` (page 117) value.

log length start to end Displays the `db.getReplicationInfo.timeDiff` (page 117) and `db.getReplicationInfo.timeDiffHours` (page 117) values.

[9] If run on a secondary, the method calls `db.printSlaveReplicationInfo()` (page 122). See `db.printSlaveReplicationInfo()` (page 122) for details.

oplog first event time Displays the db.getReplicationInfo.tFirst (page 117).

oplog last event time Displays the db.getReplicationInfo.tLast (page 118).

now Displays the db.getReplicationInfo.now (page 118).

See db.getReplicationInfo() (page 117) for description of the data.

db.printShardingStatus()

Definition

db.**printShardingStatus**()

Prints a formatted report of the sharding configuration and the information regarding existing chunks in a *sharded cluster*.

Only use db.printShardingStatus() (page 122) when connected to a mongos (page 601) instance.

The db.printShardingStatus() (page 122) method has the following parameter:

param Boolean verbose If true, the method displays details of the document distribution across chunks when you have 20 or more chunks.

See *sh.status()* (page 191) for details of the output.

Note: The db.printShardingStatus() (page 122) in the mongo (page 610) shell does **not** return *JSON*. Use db.printShardingStatus() (page 122) for manual inspection, and *Config Database* (page 681) in scripts.

See also:

sh.status() (page 191)

db.printSlaveReplicationInfo()

Definition

db.**printSlaveReplicationInfo**()

Returns a formatted report of the status of a *replica set* from the perspective of the *secondary* member of the set. The output is identical to that of rs.printSlaveReplicationInfo() (page 176).

Output The following is example output from the rs.printSlaveReplicationInfo() (page 176) method issued on a replica set with two secondary members:

```
source: m1.example.net:27017
    syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
    0 secs (0 hrs) behind the primary
source: m2.example.net:27017
    syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
    0 secs (0 hrs) behind the primary
```

Note: The db.printSlaveReplicationInfo() (page 122) in the mongo (page 610) shell does **not** return *JSON*. Use db.printSlaveReplicationInfo() (page 122) for manual inspection, and rs.status() (page 179) in scripts.

A *delayed member* may show as 0 seconds behind the primary when the inactivity period on the primary is greater than the slaveDelay value.

db.removeUser()

Deprecated since version 2.6: Use `db.dropUser()` (page 155) instead of `db.removeUser()` (page 157)

Definition

`db.removeUser(username)`

Removes the specified username from the database.

The `db.removeUser()` (page 157) method has the following parameter:

param string username The database username.

db.repairDatabase()

`db.repairDatabase()`

`db.repairDatabase()` (page 123) provides a wrapper around the database command `repairDatabase` (page 341), and has the same effect as the run-time option *mongod --repair* option, limited to *only* the current database. See `repairDatabase` (page 341) for full documentation.

Behavior

> **Warning:** During normal operations, only use the `repairDatabase` (page 341) command and wrappers including `db.repairDatabase()` (page 123) in the `mongo` (page 610) shell and *mongod --repair*, to compact database files and/or reclaim disk space. Be aware that these operations remove and do not save any corrupt data during the repair process.
> If you are trying to repair a *replica set* member, and you have access to an intact copy of your data (e.g. a recent backup or an intact member of the *replica set*), you should restore from that intact copy, and **not** use `repairDatabase` (page 341).

When using *journaling*, there is almost never any need to run `repairDatabase` (page 341). In the event of an unclean shutdown, the server will be able to restore the data files to a pristine state automatically.

Changed in version 2.6: The `db.repairDatabase()` (page 123) is now available for secondary as well as primary members of replica sets.

db.resetError()

`db.resetError()`

Deprecated since version 1.6.

Resets the error message returned by `db.getPrevError` (page 116) or `getPrevError` (page 247). Provides a wrapper around the `resetError` (page 250) command.

db.runCommand()

Definition

`db.runCommand(command)`

Provides a helper to run specified *database commands* (page 210). This is the preferred method to issue database commands, as it provides a consistent interface between the shell and drivers.

param document,string command "A *database command*, specified either in *document* form or as a string. If specified as a string, `db.runCommand()` (page 123) transforms the string into a document."

New in version 2.6: To specify a time limit in milliseconds, see http://docs.mongodb.org/manual/tutorial/terminate-running-operations.

Behavior db.runCommand() (page 123) runs the command in the context of the current database. Some commands are only applicable in the context of the admin database, and you must change your db object to before running these commands.

db.serverBuildInfo()

db.**serverBuildInfo**()

Provides a wrapper around the buildInfo (page 346) *database command*. buildInfo (page 346) returns a document that contains an overview of parameters used to compile this mongod (page 583) instance.

db.serverCmdLineOpts()

db.**serverCmdLineOpts**()

Wraps the getCmdLineOpts (page 357) *database command*.

Returns a document that reports on the arguments and configuration options used to start the mongod (page 583) or mongos (page 601) instance.

See http://docs.mongodb.org/manual/reference/configuration-options, *mongod* (page 583), and *mongos* (page 601) for additional information on available MongoDB runtime options.

db.serverStatus()

db.**serverStatus**()

Returns a *document* that provides an overview of the database process's state.

This command provides a wrapper around the database command serverStatus (page 366).

Changed in version 2.4: In 2.4 you can dynamically suppress portions of the db.serverStatus() (page 124) output, or include suppressed sections in a document passed to the db.serverStatus() (page 124) method, as in the following example:

```
db.serverStatus( { repl: 0, indexCounters: 0, locks: 0 } )
db.serverStatus( { workingSet: 1, metrics: 0, locks: 0 } )
```

db.serverStatus() (page 124) includes all fields by default, except workingSet (page 381) rangeDeleter (page 377), and some content in the repl (page 374) document.

Note: You may only dynamically include top-level fields from the *serverStatus* (page 366) document that are not included by default. You can exclude any field that db.serverStatus() (page 124) includes by default.

See also:

serverStatus (page 366) for complete documentation of the output of this function.

db.setLogLevel()

Definition

db.**setLogLevel**()

New in version 2.8.

Sets a single verbosity level for `log messages`.

`db.setLogLevel()` (page 124) has the following form:

```
db.setLogLevel(<level>, <component>)
```

`db.setLogLevel()` (page 124) takes the following parameters:

param int level The log verbosity level.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* messages.
- `1` to `5` increases the verbosity level to include *Debug* messages.

To inherit the verbosity level of the component's parent, you can also specify `-1`.

param string component The name of the component for which to specify the log verbosity level. The component name corresponds to the `<name>` from the corresponding `systemLog.component.<name>.verbosity` setting:

- `accessControl`
- `command`
- `control`
- `geo`
- `index`
- `network`
- `query`
- `replication`
- `sharding`
- `storage`
- `storage.journal`
- `write`

Omit to specify the default verbosity level for all components.

Behavior `db.setLogLevel()` (page 124) sets a *single* verbosity level. To set multiple verbosity levels in a single operation, use either the `setParameter` (page 343) command to set the `logComponentVerbosity` parameter. You can also specify the verbosity settings in the `configuration file`. See *log-messages-configure-verbosity* for examples.

Examples

Set Default Verbosity Level Omit the `<component>` parameter to set the default verbosity for all components; i.e. the `systemLog.verbosity` setting. The operation sets the default verbosity to `1`:

```
db.setLogLevel(1)
```

Set Verbosity Level for a Component Specify the `<component>` parameter to set the verbosity for the component. The following operation updates the `systemLog.component.storage.journal.verbosity` to `2`:

```
db.setLogLevel(2, "storage.journal" )
```

db.setProfilingLevel()

Definition

`db.setProfilingLevel(level, slowms)`

Modifies the current *database profiler* level used by the database profiling system to capture data about performance. The method provides a wrapper around the *database command* `profile` (page 366).

param integer level Specifies a profiling level, which is either `0` for no profiling, `1` for only slow operations, or `2` for all operations.

param integer slowms Sets the threshold in milliseconds for the profile to consider a query or operation to be slow.

The level chosen can affect performance. It also can allow the server to write the contents of queries to the log, which might have information security implications for your deployment.

Configure the `slowOpThresholdMs` option to set the threshold for the profiler to consider a query "slow." Specify this value in milliseconds to override the default, 100ms.

`mongod` (page 583) writes the output of the database profiler to the `system.profile` collection.

`mongod` (page 583) prints information about queries that take longer than the `slowOpThresholdMs` to the log even when the database profiler is not active.

db.shutdownServer()

`db.shutdownServer()`

Shuts down the current `mongod` (page 583) or `mongos` (page 601) process cleanly and safely.

This operation fails when the current database *is not* the *admin database*.

This command provides a wrapper around the `shutdown` (page 344).

db.stats()

Description

`db.stats(scale)`

Returns statistics that reflect the use state of a single *database*.

The `db.stats()` (page 126) method has the following parameter:

param number scale The scale at which to deliver results. Unless specified, this command returns all data in bytes.

Returns A *document* with statistics reflecting the database system's state. For an explanation of the output, see *dbStats* (page 352).

The `db.stats()` (page 126) method is a wrapper around the `dbStats` (page 352) database command.

Example The following example converts the returned values to kilobytes:

```
db.stats(1024)
```

Note: The scale factor rounds values to whole numbers. This can produce unpredictable and unexpected results in some situations.

db.upgradeCheck()

Definition

db.**upgradeCheck**(*<document>*)

New in version 2.6.

Performs a preliminary check for upgrade preparedness to 2.6. The helper, available in the 2.6 mongo (page 610) shell, can run connected to either a 2.4 or a 2.6 server.

The method checks for:

- documents with index keys *longer than the index key limit* (page 749),
- documents with illegal field names (page 698),
- collections without an _id index, and
- indexes with invalid specifications, such as an index key with an empty or illegal field name.

The method can accept a document parameter which determine the scope of the check:

param document scope Document to limit the scope of the check to the specified collection in the database.

Omit to perform the check on all collections in the database.

The optional scope document has the following form:

```
{
   collection: <string>
}
```

Additional 2.6 changes that affect compatibility with older versions require manual checks and intervention. See *Compatibility Changes in MongoDB 2.6* (page 749) for details.

See also:

db.upgradeCheckAllDBs() (page 128)

Behavior db.upgradeCheck() (page 127) performs collection scans and has an impact on performance. To mitigate the performance impact:

- For sharded clusters, configure to read from secondaries and run the command on the mongos (page 601).
- For replica sets, run the command on the secondary members.

db.upgradeCheck() (page 127) can miss new data during the check when run on a live system with active write operations.

For index validation, db.upgradeCheck() (page 127) only supports the check of version 1 indexes and skips the check of version 0 indexes.

The db.upgradeCheck() (page 127) checks all of the data stored in the mongod (page 583) instance: the time to run db.upgradeCheck() (page 127) depends on the quantity of data stored by mongod (page 583).

Required Access On systems running with `authorization`, a user must have access that includes the `find` action on all collections, including the *system collections* (page 688).

Example The following example connects to a secondary running on `localhost` and runs `db.upgradeCheck()` (page 127) against the `employees` collection in the `records` database. Because the output from the method can be quite large, the example pipes the output to a file.

```
./mongo --eval "db.getMongo().setSlaveOk(); db.upgradeCheck( { collection: 'employees' } )"  localhos
```

Error Output The upgrade check can return the following errors when it encounters incompatibilities in your data:

Index Key Exceed Limit

```
Document Error: key for index '<indexName>' (<indexSpec>) too long on document: <doc>
```

To resolve, remove the document. Ensure that the query to remove the document does not specify a condition on the invalid field or field.

Documents with Illegal Field Names

```
Document Error: document is no longer valid in 2.6 because <errmsg>: <doc>
```

To resolve, remove the document and re-insert with the appropriate corrections.

Index Specification Invalid

```
Index Error: invalid index spec for index '<indexName>': <indexSpec>
```

To resolve, remove the invalid index and recreate with a valid index specification.

Missing `_id` Index

```
Collection Error: lack of _id index on collection: <collectionName>
```

To resolve, create a unique index on `_id`.

Warning Output

```
Warning: upgradeCheck only supports V1 indexes. Skipping index: <indexSpec>
```

To resolve, remove the invalid index and recreate the index omitting the version specification, or reindex the collection. Reindex operation may be expensive for collections that have a large amount of data and/or a large number of indexes.

db.upgradeCheckAllDBs()

Definition

`db.upgradeCheckAllDBs()`

New in version 2.6.

Performs a preliminary check for upgrade preparedness to 2.6. The helper, available in the 2.6 `mongo` (page 610) shell, can run connected to either a 2.4 or a 2.6 server in the `admin` database.

The method cycles through all the databases and checks for:

- documents with index keys *longer than the index key limit* (page 749),

- documents with `illegal field names` (page 698),
- collections without an `_id` index, and
- indexes with invalid specifications, such as an index key with an empty or illegal field name.

Additional 2.6 changes that affect compatibility with older versions require manual checks and intervention. See *Compatibility Changes in MongoDB 2.6* (page 749) for details.

See also:

`db.upgradeCheck()` (page 127)

Behavior `db.upgradeCheckAllDBs()` (page 128) performs collection scans and has an impact on performance. To mitigate the performance impact:

- For sharded clusters, configure to read from secondaries and run the command on the `mongos` (page 601).
- For replica sets, run the command on the secondary members.

`db.upgradeCheckAllDBs()` (page 128) can miss new data during the check when run on a live system with active write operations.

For index validation, `db.upgradeCheckAllDBs()` (page 128) only supports the check of version `1` indexes and skips the check of version `0` indexes.

The `db.upgradeCheckAllDBs()` (page 128) checks all of the data stored in the `mongod` (page 583) instance: the time to run `db.upgradeCheckAllDBs()` (page 128) depends on the quantity of data stored by `mongod` (page 583).

Required Access On systems running with `authorization`, a user must have access that includes the `listDatabases` action on all databases and the `find` action on all collections, including the *system collections* (page 688).

You *must* run the `db.upgradeCheckAllDBs()` (page 128) operation in the `admin` database.

Example The following example connects to a secondary running on `localhost` and runs `db.upgradeCheckAllDBs()` (page 128) against the `admin` database. Because the output from the method can be quite large, the example pipes the output to a file.

```
./mongo --eval "db.getMongo().setSlaveOk(); db.upgradeCheckAllDBs();" localhost/admin | tee /tmp/upgr
```

Error Output The upgrade check can return the following errors when it encounters incompatibilities in your data:

Index Key Exceed Limit

```
Document Error: key for index '<indexName>' (<indexSpec>) too long on document: <doc>
```

To resolve, remove the document. Ensure that the query to remove the document does not specify a condition on the invalid field or field.

Documents with Illegal Field Names

```
Document Error: document is no longer valid in 2.6 because <errmsg>: <doc>
```

To resolve, remove the document and re-insert with the appropriate corrections.

Index Specification Invalid

```
Index Error: invalid index spec for index '<indexName>': <indexSpec>
```

To resolve, remove the invalid index and recreate with a valid index specification.

Missing `_id` Index

```
Collection Error: lack of _id index on collection: <collectionName>
```

To resolve, create a unique index on `_id`.

Warning Output

```
Warning: upgradeCheck only supports V1 indexes. Skipping index: <indexSpec>
```

To resolve, remove the invalid index and recreate the index omitting the version specification, or reindex the collection. Reindex operation may be expensive for collections that have a large amount of data and/or a large number of indexes.

db.version()

`db.version()`

Returns The version of the `mongod` (page 583) or `mongos` (page 601) instance.

2.1.4 Query Plan Cache

Query Plan Cache Methods

The PlanCache methods are only accessible from a collection's plan cache object. To retrieve the plan cache object, use the `db.collection.getPlanCache()` (page 134) method.

Name	Description
`PlanCache.clear()` (page 130)	Clears all the cached query plans for a collection. Accessible through the plan cache object of a specific collection, i.e. `db.collection.getPlanCache().clear()`.
`PlanCache.clearPlansByQuery()` (page 131)	Clears the cached query plans for the specified query shape. Accessible through the plan cache object of a specific collection, i.e. `db.collection.getPlanCache().clearPlansByQuery()`
`PlanCache.getPlansByQuery()` (page 132)	Displays the cached query plans for the specified query shape. Accessible through the plan cache object of a specific collection, i.e. `db.collection.getPlanCache().getPlansByQuery()`.
`PlanCache.help()` (page 133)	Displays the methods available for a collection's query plan cache. Accessible through the plan cache object of a specific collection, i.e. `db.collection.getPlanCache().help()`.
`PlanCache.listQueryShapes()` (page 133)	Displays the query shapes for which cached query plans exist. Accessible through the plan cache object of a specific collection, i.e. `db.collection.getPlanCache().listQueryShapes()`.
`db.collection.getPlanCache()` (page 134)	Returns an interface to access the query plan cache object and associated PlanCache methods for a collection."

PlanCache.clear()

Definition

PlanCache.**clear**()
Removes all cached query plans for a collection.

The method is only available from the plan cache object (page 134) of a specific collection; i.e.

```
db.collection.getPlanCache().clear()
```

For example, to clear the cache for the orders collection:

```
db.orders.getPlanCache().clear()
```

Required Access On systems running with authorization, a user must have access that includes the planCacheWrite action.

See also:

- db.collection.getPlanCache() (page 134)
- PlanCache.clearPlansByQuery() (page 131)

PlanCache.clearPlansByQuery()

Definition

PlanCache.**clearPlansByQuery**(<*query*>, <*projection*>, <*sort*>)
Clears the cached query plans for the specified *query shape*.

The method is only available from the plan cache object (page 134) of a specific collection; i.e.

```
db.collection.getPlanCache().clearPlansByQuery( <query>, <projection>, <sort> )
```

The PlanCache.clearPlansByQuery() (page 131) method accepts the following parameters:

param document query The query predicate of the *query shape*. Only the structure of the predicate, including the field names, are significant to the shape; the values in the query predicate are insignificant.

param document projection The projection associated with the *query shape*. Required if specifying the sort parameter.

param document sort The sort associated with the *query shape*.

To see the query shapes for which cached query plans exist, use the PlanCache.listQueryShapes() (page 133) method.

Required Access On systems running with authorization, a user must have access that includes the planCacheWrite action.

Example If a collection orders has the following query shape:

```
{
  "query" : { "qty" : { "$gt" : 10 } },
  "sort" : { "ord_date" : 1 },
  "projection" : { }
}
```

The following operation removes the query plan cached for the shape:

```
db.orders.getPlanCache().clearPlansByQuery(
   { "qty" : { "$gt" : 10 } },
   { },
   { "ord_date" : 1 }
)
```

See also:

- db.collection.getPlanCache() (page 134)
- PlanCache.listQueryShapes() (page 133)
- PlanCache.clear() (page 130)

PlanCache.getPlansByQuery()

Definition

PlanCache.**getPlansByQuery**(*<query>*, *<projection>*, *<sort>*)

Displays the cached query plans for the specified *query shape*.

The query optimizer only caches the plans for those query shapes that can have more than one viable plan.

The method is only available from the plan cache object (page 134) of a specific collection; i.e.

```
db.collection.getPlanCache().getPlansByQuery( <query>, <projection>, <sort> )
```

The PlanCache.getPlansByQuery() (page 132) method accepts the following parameters:

param document query The query predicate of the *query shape*. Only the structure of the predicate, including the field names, are significant to the shape; the values in the query predicate are insignificant.

param document projection The projection associated with the *query shape*. Required if specifying the sort parameter.

param document sort The sort associated with the *query shape*.

Returns Array of cached query plans for a query shape.

To see the query shapes for which cached query plans exist, use the PlanCache.listQueryShapes() (page 133) method.

Required Access On systems running with authorization, a user must have access that includes the planCacheRead action.

Example If a collection orders has the following query shape:

```
{
  "query" : { "qty" : { "$gt" : 10 } },
  "sort" : { "ord_date" : 1 },
  "projection" : { }
}
```

The following operation displays the query plan cached for the shape:

```
db.orders.getPlanCache().getPlansByQuery(
   { "qty" : { "$gt" : 10 } },
   { },
   { "ord_date" : 1 }
)
```

See also:

- db.collection.getPlanCache() (page 134)
- PlanCache.listQueryShapes() (page 133)
- PlanCache.help() (page 133)

PlanCache.help()

Definition

PlanCache.**help**()

Displays the methods available to view and modify a collection's query plan cache.

The method is only available from the plan cache object (page 134) of a specific collection; i.e.

```
db.collection.getPlanCache().help()
```

See also:

db.collection.getPlanCache() (page 134)

PlanCache.listQueryShapes()

Definition

PlanCache.**listQueryShapes**()

Displays the *query shapes* for which cached query plans exist.

The query optimizer only caches the plans for those query shapes that can have more than one viable plan.

The method is only available from the plan cache object (page 134) of a specific collection; i.e.

```
db.collection.getPlanCache().listQueryShapes()
```

Returns Array of *query shape* documents.

The method wraps the planCacheListQueryShapes (page 260) command.

Required Access On systems running with authorization, a user must have access that includes the planCacheRead action.

Example The following returns the *query shapes* that have cached plans for the orders collection:

```
db.orders.getPlanCache().listQueryShapes()
```

The method returns an array of the query shapes currently in the cache. In the example, the orders collection had cached query plans associated with the following shapes:

```
[
  {
    "query" : { "qty" : { "$gt" : 10 } },
    "sort" : { "ord_date" : 1 },
    "projection" : { }
  },
  {
    "query" : { "$or" :
       [
         { "qty" : { "$gt" : 15 }, "item" : "xyz123" },
         { "status" : "A" }
       ]
    },
    "sort" : { },
    "projection" : { }
  },
  {
    "query" : { "$or" : [ { "qty" : { "$gt" : 15 } }, { "status" : "A" } ] },
    "sort" : { },
    "projection" : { }
  }
]
```

See also:

- `db.collection.getPlanCache()` (page 134)
- `PlanCache.getPlansByQuery()` (page 132)
- `PlanCache.help()` (page 133)
- `planCacheListQueryShapes` (page 260)

db.collection.getPlanCache()

Definition

`db.collection.getPlanCache()`

Returns an interface to access the query plan cache for a collection. The interface provides methods to view and clear the query plan cache.

Returns Interface to access the query plan cache.

The query optimizer only caches the plans for those query shapes that can have more than one viable plan.

Methods The following methods are available through the interface:

Name	Description
PlanCache.help() (page 133)	Displays the methods available for a collection's query plan cache. Accessible through the plan cache object of a specific collection, i.e. db.collection.getPlanCache().help().
PlanCache.listQueryShapes() (page 133)	Displays the query shapes for which cached query plans exist. Accessible through the plan cache object of a specific collection, i.e. db.collection.getPlanCache().listQueryShapes().
PlanCache.getPlansByQuery() (page 132)	Displays the cached query plans for the specified query shape. Accessible through the plan cache object of a specific collection, i.e. db.collection.getPlanCache().getPlansByQuery().
PlanCache.clearPlansByQuery() (page 131)	Clears the cached query plans for the specified query shape. Accessible through the plan cache object of a specific collection, i.e. db.collection.getPlanCache().clearPlansByQuery()
PlanCache.clear() (page 130)	Clears all the cached query plans for a collection. Accessible through the plan cache object of a specific collection, i.e. db.collection.getPlanCache().clear().

2.1.5 Bulk Write Operation

Bulk Operation Methods

New in version 2.6.

Name	Description
Bulk() (page 135)	Bulk operations builder.
Bulk.execute() (page 137)	Executes a list of operations in bulk.
Bulk.find() (page 139)	Specifies the query condition for an update or a remove operation.
Bulk.find.remove() (page 140)	Adds a multiple document remove operation to a list of operations.
Bulk.find.removeOne() (page 140)	Adds a single document remove operation to a list of operations.
Bulk.find.replaceOne() (page 141)	Adds a single document replacement operation to a list of operations.
Bulk.find.update() (page 142)	Adds a multi update operation to a list of operations.
Bulk.find.updateOne() (page 142)	Adds a single document update operation to a list of operations.
Bulk.find.upsert() (page 144)	Specifies upsert: true for an update operation.
Bulk.getOperations() (page 147)	Returns an array of write operations executed in the Bulk() (page 135) operations object.
Bulk.insert() (page 148)	Adds an insert operation to a list of operations.
Bulk.toString() (page 148)	Returns the Bulk.tojson() (page 149) results as a string.
Bulk.tojson() (page 149)	Returns a JSON document that contains the number of operations and batches in the Bulk() (page 135) operations object.
db.collection.initializeOrderedBulkOp() (page 149)	Initializes a Bulk() (page 135) operations builder for an ordered list of operations.
db.collection.initializeUnorderedBulkOp() (page 150)	Initializes a Bulk() (page 135) operations builder for an unordered list of operations.

Bulk()

Description

Bulk()

New in version 2.6.

Bulk operations builder used to construct a list of write operations to perform in bulk for a single collection. To instantiate the builder, use either the db.collection.initializeOrderedBulkOp() (page 149) or the db.collection.initializeUnorderedBulkOp() (page 150) method.

Ordered and Unordered Bulk Operations The builder can construct the list of operations as *ordered* or *unordered*.

Ordered Operations With an *ordered* operations list, MongoDB executes the write operations in the list serially. If an error occurs during the processing of one of the write operations, MongoDB will return without processing any remaining write operations in the list.

Use db.collection.initializeOrderedBulkOp() (page 149) to create a builder for an ordered list of write commands.

When executing an ordered (page 149) list of operations, MongoDB groups the operations by the operation type (page 148) and contiguity; i.e. *contiguous* operations of the same type are grouped together. For example, if an ordered list has two insert operations followed by an update operation followed by another insert operation, MongoDB groups the operations into three separate groups: first group contains the two insert operations, second group contains the update operation, and the third group contains the last insert operation. This behavior is subject to change in future versions.

Each group of operations can have at most 1000 operations (page 697). If a group exceeds this limit (page 697), MongoDB will divide the group into smaller groups of 1000 or less. For example, if the bulk operations list consists of 2000 insert operations, MongoDB creates 2 groups, each with 1000 operations.

The sizes and grouping mechanics are internal performance details and are subject to change in future versions.

To see how the operations are grouped for a bulk operation execution, call Bulk.getOperations() (page 147) *after* the execution.

Executing an ordered (page 149) list of operations on a sharded collection will generally be slower than executing an unordered (page 150) list since with an ordered list, each operation must wait for the previous operation to finish.

Unordered Operations With an *unordered* operations list, MongoDB can execute in parallel, as well as in a nondeterministic order, the write operations in the list. If an error occurs during the processing of one of the write operations, MongoDB will continue to process remaining write operations in the list.

Use db.collection.initializeUnorderedBulkOp() (page 150) to create a builder for an unordered list of write commands.

When executing an unordered (page 150) list of operations, MongoDB groups the operations. With an unordered bulk operation, the operations in the list may be reordered to increase performance. As such, applications should not depend on the ordering when performing unordered (page 150) bulk operations.

Each group of operations can have at most 1000 operations (page 697). If a group exceeds this limit (page 697), MongoDB will divide the group into smaller groups of 1000 or less. For example, if the bulk operations list consists of 2000 insert operations, MongoDB creates 2 groups, each with 1000 operations.

The sizes and grouping mechanics are internal performance details and are subject to change in future versions.

To see how the operations are grouped for a bulk operation execution, call Bulk.getOperations() (page 147) *after* the execution.

Methods The Bulk() (page 135) builder has the following methods:

Name	Description
Bulk.insert() (page 148)	Adds an insert operation to a list of operations.
Bulk.find() (page 139)	Specifies the query condition for an update or a remove operation.
Bulk.find.removeOne() (page 140)	Adds a single document remove operation to a list of operations.
Bulk.find.remove() (page 140)	Adds a multiple document remove operation to a list of operations.
Bulk.find.replaceOne() (page 141)	Adds a single document replacement operation to a list of operations.
Bulk.find.updateOne() (page 142)	Adds a single document update operation to a list of operations.
Bulk.find.update() (page 142)	Adds a `multi` update operation to a list of operations.
Bulk.find.upsert() (page 144)	Specifies `upsert: true` for an update operation.
Bulk.execute() (page 137)	Executes a list of operations in bulk.
Bulk.getOperations() (page 147)	Returns an array of write operations executed in the Bulk() (page 135) operations object.
Bulk.tojson() (page 149)	Returns a JSON document that contains the number of operations and batches in the Bulk() (page 135) operations object.
Bulk.toString() (page 148)	Returns the Bulk.tojson() (page 149) results as a string.

Bulk.execute()

Description

`Bulk.execute()`

New in version 2.6.

Executes the list of operations built by the `Bulk()` (page 135) operations builder.

`Bulk.execute()` (page 137) accepts the following parameter:

param document writeConcern `Write concern` document for the bulk operation as a whole. Omit to use default. For a standalone `mongod` (page 583) server, the write concern defaults to `{ w: 1 }`. With a replica set, the default write concern is `{ w: 1 }` unless modified as part of the *replica set configuration*.

See *Override Default Write Concern* (page 138) for an example.

Returns A `BulkWriteResult` (page 198) object that contains the status of the operation.

After execution, you cannot re-execute the `Bulk()` (page 135) object without reinitializing. See `db.collection.initializeUnorderedBulkOp()` (page 150) and `db.collection.initializeOrderedBulkOp()` (page 149).

Behavior

Ordered Operations When executing an `ordered` (page 149) list of operations, MongoDB groups the operations by the `operation type` (page 148) and contiguity; i.e. *contiguous* operations of the same type are grouped together. For example, if an ordered list has two insert operations followed by an update operation followed by another insert operation, MongoDB groups the operations into three separate groups: first group contains the two

insert operations, second group contains the update operation, and the third group contains the last insert operation. This behavior is subject to change in future versions.

Each group of operations can have at most 1000 operations (page 697). If a group exceeds this limit (page 697), MongoDB will divide the group into smaller groups of 1000 or less. For example, if the bulk operations list consists of 2000 insert operations, MongoDB creates 2 groups, each with 1000 operations.

The sizes and grouping mechanics are internal performance details and are subject to change in future versions.

To see how the operations are grouped for a bulk operation execution, call Bulk.getOperations() (page 147) *after* the execution.

Executing an ordered (page 149) list of operations on a sharded collection will generally be slower than executing an unordered (page 150) list since with an ordered list, each operation must wait for the previous operation to finish.

Unordered Operations When executing an unordered (page 150) list of operations, MongoDB groups the operations. With an unordered bulk operation, the operations in the list may be reordered to increase performance. As such, applications should not depend on the ordering when performing unordered (page 150) bulk operations.

Each group of operations can have at most 1000 operations (page 697). If a group exceeds this limit (page 697), MongoDB will divide the group into smaller groups of 1000 or less. For example, if the bulk operations list consists of 2000 insert operations, MongoDB creates 2 groups, each with 1000 operations.

The sizes and grouping mechanics are internal performance details and are subject to change in future versions.

To see how the operations are grouped for a bulk operation execution, call Bulk.getOperations() (page 147) *after* the execution.

Examples

Execute Bulk Operations The following initializes a Bulk() (page 135) operations builder on the items collection, adds a series of insert operations, and executes the operations:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.insert( { item: "abc123", status: "A", defaultQty: 500, points: 5 } );
bulk.insert( { item: "ijk123", status: "A", defaultQty: 100, points: 10 } );
bulk.execute( );
```

The operation returns the following BulkWriteResult() (page 198) object:

```
BulkWriteResult({
   "writeErrors" : [ ],
   "writeConcernErrors" : [ ],
   "nInserted" : 2,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

For details on the return object, see BulkWriteResult() (page 198). For details on the batches executed, see Bulk.getOperations() (page 147).

Override Default Write Concern The following operation to a replica set specifies a write concern of "w: majority" with a wtimeout of 5000 milliseconds such that the method returns after the writes propagate to a majority of the replica set members or the method times out after 5 seconds.

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.insert( { item: "efg123", status: "A", defaultQty: 100, points: 0 } );
bulk.insert( { item: "xyz123", status: "A", defaultQty: 100, points: 0 } );
bulk.execute( { w: "majority", wtimeout: 5000 } );
```

The operation returns the following `BulkWriteResult()` (page 198) object:

```
BulkWriteResult({
   "writeErrors" : [ ],
   "writeConcernErrors" : [ ],
   "nInserted" : 2,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

See

`Bulk()` (page 135) for a listing of methods available for bulk operations.

Bulk.find()

Description

`Bulk.find(<query>)`

New in version 2.6.

Specifies a query condition for an update or a remove operation.

`Bulk.find()` (page 139) accepts the following parameter:

param document query Specifies a query condition using *Query Selectors* (page 400) to select documents for an update or a remove operation. To specify all documents, use an empty document `{}`.

With update operations, the sum of the query document and the update document must be less than or equal to the `maximum BSON document size` (page 692).

With remove operations, the query document must be less than or equal to the `maximum BSON document size` (page 692).

Use `Bulk.find()` (page 139) with the following write operations:

- `Bulk.find.removeOne()` (page 140)
- `Bulk.find.remove()` (page 140)
- `Bulk.find.replaceOne()` (page 141)
- `Bulk.find.updateOne()` (page 142)
- `Bulk.find.update()` (page 142)

Example The following example initializes a `Bulk()` (page 135) operations builder for the `items` collection and adds a remove operation and an update operation to the list of operations. The remove operation and the update operation use the `Bulk.find()` (page 139) method to specify a condition for their respective actions:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "D" } ).remove();
bulk.find( { status: "P" } ).update( { $set: { points: 0 } } )
bulk.execute();
```

See also:

- db.collection.initializeUnorderedBulkOp() (page 150)
- db.collection.initializeOrderedBulkOp() (page 149)
- Bulk.execute() (page 137)

Bulk.find.remove()

Description

Bulk.find.**remove**()

New in version 2.6.

Adds a remove operation to a bulk operations list. Use the Bulk.find() (page 139) method to specify the condition that determines which documents to remove. The Bulk.find.remove() (page 140) method removes all matching documents in the collection. To limit the remove to a single document, see Bulk.find.removeOne() (page 140).

Example The following example initializes a Bulk() (page 135) operations builder for the items collection and adds a remove operation to the list of operations. The remove operation removes all documents in the collection where the status equals "D":

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "D" } ).remove();
bulk.execute();
```

See also:

- db.collection.initializeUnorderedBulkOp() (page 150)
- db.collection.initializeOrderedBulkOp() (page 149)
- Bulk.find() (page 139)
- Bulk.find.removeOne() (page 140)
- Bulk.execute() (page 137)

Bulk.find.removeOne()

Description

Bulk.find.**removeOne**()

New in version 2.6.

Adds a single document remove operation to a bulk operations list. Use the Bulk.find() (page 139) method to specify the condition that determines which document to remove. The Bulk.find.removeOne() (page 140) limits the removal to one document. To remove multiple documents, see Bulk.find.remove() (page 140).

Example The following example initializes a `Bulk()` (page 135) operations builder for the `items` collection and adds two `Bulk.find.removeOne()` (page 140) operations to the list of operations.

Each remove operation removes just one document: one document with the `status` equal to `"D"` and another document with the `status` equal to `"P"`.

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "D" } ).removeOne();
bulk.find( { status: "P" } ).removeOne();
bulk.execute();
```

See also:

- `db.collection.initializeUnorderedBulkOp()` (page 150)
- `db.collection.initializeOrderedBulkOp()` (page 149)
- `Bulk.find()` (page 139)
- `Bulk.find.remove()` (page 140)
- `Bulk.execute()` (page 137)
- *All Bulk Methods* (page 136)

Bulk.find.replaceOne()

Description

`Bulk.find.replaceOne(<document>)`

New in version 2.6.

Adds a single document replacement operation to a bulk operations list. Use the `Bulk.find()` (page 139) method to specify the condition that determines which document to replace. The `Bulk.find.replaceOne()` (page 141) method limits the replacement to a single document.

`Bulk.find.replaceOne()` (page 141) accepts the following parameter:

param document replacement A replacement document that completely replaces the existing document. Contains only field and value pairs.

The sum of the associated `<query>` document from the `Bulk.find()` (page 139) and the replacement document must be less than or equal to the `maximum BSON document size` (page 692).

To specify an *upsert* for this operation, see `Bulk.find.upsert()` (page 144).

Example The following example initializes a `Bulk()` (page 135) operations builder for the `items` collection, and adds various `replaceOne` (page 141) operations to the list of operations.

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { item: "abc123" } ).replaceOne( { item: "abc123", status: "P", points: 100 } );
bulk.execute();
```

See also:

- `db.collection.initializeUnorderedBulkOp()` (page 150)
- `db.collection.initializeOrderedBulkOp()` (page 149)
- `Bulk.find()` (page 139)
- `Bulk.execute()` (page 137)

- *All Bulk Methods* (page 136)

Bulk.find.update()

Description

`Bulk.find.update(<update>)`

New in version 2.6.

Adds a `multi` update operation to a bulk operations list. The method updates specific fields in existing documents.

Use the `Bulk.find()` (page 139) method to specify the condition that determines which documents to update. The `Bulk.find.update()` (page 142) method updates all matching documents. To specify a single document update, see `Bulk.find.updateOne()` (page 142).

`Bulk.find.update()` (page 142) accepts the following parameter:

param document update Specifies the fields to update. Only contains *update operator* (page 451) expressions.

The sum of the associated `<query>` document from the `Bulk.find()` (page 139) and the update document must be less than or equal to the `maximum BSON document size` (page 692).

To specify *upsert: true* for this operation, see `Bulk.find.upsert()` (page 144). With `Bulk.find.upsert()` (page 144), if no documents match the `Bulk.find()` (page 139) query condition, the update operation inserts only a single document.

Example The following example initializes a `Bulk()` (page 135) operations builder for the `items` collection, and adds various `multi` update operations to the list of operations.

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "D" } ).update( { $set: { status: "I", points: "0" } } );
bulk.find( { item: null } ).update( { $set: { item: "TBD" } } );
bulk.execute();
```

See also:

- `db.collection.initializeUnorderedBulkOp()` (page 150)
- `db.collection.initializeOrderedBulkOp()` (page 149)
- `Bulk.find()` (page 139)
- `Bulk.find.updateOne()` (page 142)
- `Bulk.execute()` (page 137)
- *All Bulk Methods* (page 136)

Bulk.find.updateOne()

Description

`Bulk.find.updateOne(<update>)`

New in version 2.6.

Adds a single document update operation to a bulk operations list. The operation can either replace an existing document or update specific fields in an existing document.

Use the `Bulk.find()` (page 139) method to specify the condition that determines which document to update. The `Bulk.find.updateOne()` (page 142) method limits the update or replacement to a single document. To update multiple documents, see `Bulk.find.update()` (page 142).

`Bulk.find.updateOne()` (page 142) accepts the following parameter:

param document update An update document that updates specific fields or a replacement document that completely replaces the existing document.

An update document only contains *update operator* (page 451) expressions. A replacement document contains only field and value pairs.

The sum of the associated `<query>` document from the `Bulk.find()` (page 139) and the update/replacement document must be less than or equal to the `maximum BSON document size`.

To specify an *upsert: true* for this operation, see `Bulk.find.upsert()` (page 144).

Behavior

Update Specific Fields If the `<update>` document contains only *update operator* (page 451) expressions, as in:

```
{
  $set: { status: "D" },
  points: { $inc: 2 }
}
```

Then, `Bulk.find.updateOne()` (page 142) updates only the corresponding fields, `status` and `points`, in the document.

Replace a Document If the `<update>` document contains only `field:value` expressions, as in:

```
{
  item: "TBD",
  points: 0,
  inStock: true,
  status: "I"
}
```

Then, `Bulk.find.updateOne()` (page 142) *replaces* the matching document with the `<update>` document with the exception of the `_id` field. The `Bulk.find.updateOne()` (page 142) method *does not* replace the `_id` value.

Example The following example initializes a `Bulk()` (page 135) operations builder for the `items` collection, and adds various `updateOne` (page 142) operations to the list of operations.

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "D" } ).updateOne( { $set: { status: "I", points: "0" } } );
bulk.find( { item: null } ).updateOne(
   {
      item: "TBD",
      points: 0,
      inStock: true,
      status: "I"
   }
);
bulk.execute();
```

See also:

- db.collection.initializeUnorderedBulkOp() (page 150)
- db.collection.initializeOrderedBulkOp() (page 149)
- Bulk.find() (page 139)
- Bulk.find.update() (page 142)
- Bulk.execute() (page 137)
- *All Bulk Methods* (page 136)

Bulk.find.upsert()

Description

`Bulk.find.upsert()`

New in version 2.6.

Sets the *upsert* option to true for an update or a replacement operation and has the following syntax:

```
Bulk.find(<query>).upsert().update(<update>);
Bulk.find(<query>).upsert().updateOne(<update>);
Bulk.find(<query>).upsert().replaceOne(<replacement>);
```

With the `upsert` option set to `true`, if no matching documents exist for the Bulk.find() (page 139) condition, then the update or the replacement operation performs an insert. If a matching document does exist, then the update or replacement operation performs the specified update or replacement.

Use Bulk.find.upsert() (page 144) with the following write operations:

- Bulk.find.replaceOne() (page 141)
- Bulk.find.updateOne() (page 142)
- Bulk.find.update() (page 142)

Behavior The following describe the insert behavior of various write operations when used in conjunction with Bulk.find.upsert() (page 144).

Insert for `Bulk.find.replaceOne()` The Bulk.find.replaceOne() (page 141) method accepts, as its parameter, a replacement document that only contains field and value pairs:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { item: "abc123" } ).upsert().replaceOne(
   {
     item: "abc123",
     status: "P",
     points: 100,
   }
);
bulk.execute();
```

If the replacement operation with the Bulk.find.upsert() (page 144) option performs an insert, the inserted document is the replacement document. If the replacement document does not specify an `_id` field, MongoDB adds the `_id` field:

```
{
  "_id" : ObjectId("52ded3b398ca567f5c97ac9e"),
  "item" : "abc123",
  "status" : "P",
  "points" : 100
}
```

Insert for `Bulk.find.updateOne()` The `Bulk.find.updateOne()` (page 142) method accepts, as its parameter, an `<update>` document that contains only field and value pairs or only *update operator* (page 451) expressions.

Field and Value Pairs If the `<update>` document contains only field and value pairs:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "P" } ).upsert().updateOne(
   {
     item: "TBD",
     points: 0,
     inStock: true,
     status: "I"
   }
);
bulk.execute();
```

Then, if the update operation with the `Bulk.find.upsert()` (page 144) option performs an insert, the inserted document is the `<update>` document. If the update document does not specify an `_id` field, MongoDB adds the `_id` field:

```
{
  "_id" : ObjectId("52ded5a898ca567f5c97ac9f"),
  "item" : "TBD",
  "points" : 0,
  "inStock" : true,
  "status" : "I"
}
```

Update Operator Expressions If the `<update>` document contains contains only *update operator* (page 451) expressions:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "P", item: null } ).upsert().updateOne(
   {
     $setOnInsert: { defaultQty: 0, inStock: true },
     $currentDate: { lastModified: true },
     $set: { points: "0" }
   }
);
bulk.execute();
```

Then, if the update operation with the `Bulk.find.upsert()` (page 144) option performs an insert, the update operation inserts a document with field and values from the `<query>` document of the `Bulk.find()` (page 139) method and then applies the specified update from the `<update>` document:

```
{
   "_id" : ObjectId("52ded68c98ca567f5c97aca0"),
```

```
    "item" : null,
    "status" : "P",
    "defaultQty" : 0,
    "inStock" : true,
    "lastModified" : ISODate("2014-01-21T20:20:28.786Z"),
    "points" : "0"
}
```

If neither the <query> document nor the <update> document specifies an _id field, MongoDB adds the _id field.

Insert for `Bulk.find.update()` When using `upsert()` (page 144) with the multiple document update method `Bulk.find.update()` (page 142), if no documents match the query condition, the update operation inserts a *single* document.

The `Bulk.find.update()` (page 142) method accepts, as its parameter, an <update> document that contains *only update operator* (page 451) expressions:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "P" } ).upsert().update(
    {
      $setOnInsert: { defaultQty: 0, inStock: true },
      $currentDate: { lastModified: true },
      $set: { status: "I", points: "0" }
    }
);
bulk.execute();
```

Then, if the update operation with the `Bulk.find.upsert()` (page 144) option performs an insert, the update operation inserts a single document with the fields and values from the <query> document of the `Bulk.find()` (page 139) method and then applies the specified update from the <update> document:

```
{
    "_id": ObjectId("52ded81a98ca567f5c97aca1"),
    "status": "I",
    "defaultQty": 0,
    "inStock": true,
    "lastModified": ISODate("2014-01-21T20:27:06.691Z"),
    "points": "0"
}
```

If neither the <query> document nor the <update> document specifies an _id field, MongoDB adds the _id field.

See also:

- `db.collection.initializeUnorderedBulkOp()` (page 150)
- `db.collection.initializeOrderedBulkOp()` (page 149)
- `Bulk.find()` (page 139)
- `Bulk.execute()` (page 137)
- *All Bulk Methods* (page 136)

Bulk.getOperations()

`Bulk.getOperations()`

New in version 2.6.

Returns an array of write operations executed through `Bulk.execute()` (page 137). The returned write operations are in groups as determined by MongoDB for execution. For information on how MongoDB groups the list of bulk write operations, see *Bulk.execute() Behavior* (page 137).

Only use `Bulk.getOperations()` (page 147) after a `Bulk.execute()` (page 137). Calling `Bulk.getOperations()` (page 147) before you call `Bulk.execute()` (page 137) will result in an *incomplete* list.

Example The following initializes a `Bulk()` (page 135) operations builder on the `items` collection, adds a series of write operations, executes the operations, and then calls `getOperations()` (page 147) on the `bulk` builder object:

```
var bulk = db.items.initializeUnorderedBulkOp();

for (var i = 1; i <= 1500; i++) {
    bulk.insert( { x: i } );
}

bulk.execute();
bulk.getOperations();
```

The `getOperations()` (page 147) method returns an array with the operations executed. The output shows that MongoDB divided the operations into 2 groups, one with 1000 operations and one with 500. For information on how MongoDB groups the list of bulk write operations, see *Bulk.execute() Behavior* (page 137)

Although the method returns all 1500 operations in the returned array, this page omits some of the results for brevity.

```
[
   {
      "originalZeroIndex" : 0,
      "batchType" : 1,
      "operations" : [
         { "_id" : ObjectId("53a8959f1990ca24d01c6165"), "x" : 1 },

         ... // Content omitted for brevity

         { "_id" : ObjectId("53a8959f1990ca24d01c654c"), "x" : 1000 }
      ]
   },
   {
      "originalZeroIndex" : 1000,
      "batchType" : 1,
      "operations" : [
         { "_id" : ObjectId("53a8959f1990ca24d01c654d"), "x" : 1001 },

         ... // Content omitted for brevity

         { "_id" : ObjectId("53a8959f1990ca24d01c6740"), "x" : 1500 }
      ]
   }
]
```

Returned Fields The array contains documents with the following fields:

originalZeroIndex
Specifies the order in which the operation was added to the bulk operations builder, based on a zero index; e.g. first operation added to the bulk operations builder will have originalZeroIndex (page 148) value of 0.

batchType
Specifies the write operations type.

batchType	Operation
1	Insert
2	Update
3	Remove

operations
Array of documents that contain the details of the operation.

See also:

Bulk() (page 135) and Bulk.execute() (page 137).

Bulk.insert()

Description

Bulk.**insert**(<*document*>)
New in version 2.6.

Adds an insert operation to a bulk operations list.

Bulk.insert() (page 148) accepts the following parameter:

param document doc Document to insert. The size of the document must be less than or equal to the maximum BSON document size (page 692).

Example The following initializes a Bulk() (page 135) operations builder for the items collection and adds a series of insert operations to add multiple documents:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.insert( { item: "abc123", defaultQty: 100, status: "A", points: 100 } );
bulk.insert( { item: "ijk123", defaultQty: 200, status: "A", points: 200 } );
bulk.insert( { item: "mop123", defaultQty: 0, status: "P", points: 0 } );
bulk.execute();
```

See also:

- db.collection.initializeUnorderedBulkOp() (page 150)
- db.collection.initializeOrderedBulkOp() (page 149)
- Bulk.execute() (page 137)

Bulk.toString()

Bulk.**toString**()
New in version 2.6.

Returns as a string a JSON document that contains the number of operations and batches in the Bulk() (page 135) object.

Example The following initializes a `Bulk()` (page 135) operations builder on the `items` collection, adds a series of write operations, and calls `Bulk.toString()` (page 148) on the `bulk` builder object.

```
var bulk = db.items.initializeOrderedBulkOp();
bulk.insert( { item: "abc123", status: "A", defaultQty: 500, points: 5 } );
bulk.insert( { item: "ijk123", status: "A", defaultQty: 100, points: 10 } );
bulk.find( { status: "D" } ).removeOne();
bulk.toString();
```

The `Bulk.toString()` (page 148) returns the following JSON document

{ “nInsertOps” : 2, “nUpdateOps” : 0, “nRemoveOps” : 1, “nBatches” : 2 }

See also:

`Bulk()` (page 135)

Bulk.tojson()

`Bulk.tojson()`

New in version 2.6.

Returns a JSON document that contains the number of operations and batches in the `Bulk()` (page 135) object.

Example The following initializes a `Bulk()` (page 135) operations builder on the `items` collection, adds a series of write operations, and calls `Bulk.tojson()` (page 149) on the `bulk` builder object.

```
var bulk = db.items.initializeOrderedBulkOp();
bulk.insert( { item: "abc123", status: "A", defaultQty: 500, points: 5 } );
bulk.insert( { item: "ijk123", status: "A", defaultQty: 100, points: 10 } );
bulk.find( { status: "D" } ).removeOne();
bulk.tojson();
```

The `Bulk.tojson()` (page 149) returns the following JSON document

{ “nInsertOps” : 2, “nUpdateOps” : 0, “nRemoveOps” : 1, “nBatches” : 2 }

See also:

`Bulk()` (page 135)

db.collection.initializeOrderedBulkOp()

Definition

`db.collection.initializeOrderedBulkOp()`

Initializes and returns a new `Bulk()` (page 135) operations builder for a collection. The builder constructs an ordered list of write operations that MongoDB executes in bulk.

Returns new `Bulk()` (page 135) operations builder object.

Behavior

Order of Operation With an *ordered* operations list, MongoDB executes the write operations in the list serially.

Execution of Operations When executing an `ordered` (page 149) list of operations, MongoDB groups the operations by the `operation type` (page 148) and contiguity; i.e. *contiguous* operations of the same type are grouped together. For example, if an ordered list has two insert operations followed by an update operation followed by another insert operation, MongoDB groups the operations into three separate groups: first group contains the two insert operations, second group contains the update operation, and the third group contains the last insert operation. This behavior is subject to change in future versions.

Each group of operations can have at most `1000 operations` (page 697). If a group exceeds this `limit` (page 697), MongoDB will divide the group into smaller groups of 1000 or less. For example, if the bulk operations list consists of 2000 insert operations, MongoDB creates 2 groups, each with 1000 operations.

The sizes and grouping mechanics are internal performance details and are subject to change in future versions.

To see how the operations are grouped for a bulk operation execution, call `Bulk.getOperations()` (page 147) *after* the execution.

Executing an `ordered` (page 149) list of operations on a sharded collection will generally be slower than executing an `unordered` (page 150) list since with an ordered list, each operation must wait for the previous operation to finish.

Error Handling If an error occurs during the processing of one of the write operations, MongoDB will return without processing any remaining write operations in the list.

Examples The following initializes a `Bulk()` (page 135) operations builder on the `users` collection, adds a series of write operations, and executes the operations:

```
var bulk = db.users.initializeOrderedBulkOp();
bulk.insert( { user: "abc123", status: "A", points: 0 } );
bulk.insert( { user: "ijk123", status: "A", points: 0 } );
bulk.insert( { user: "mop123", status: "P", points: 0 } );
bulk.find( { status: "D" } ).remove();
bulk.find( { status: "P" } ).update( { $set: { comment: "Pending" } } );
bulk.execute();
```

See also:

- `db.collection.initializeUnorderedBulkOp()` (page 150)
- `Bulk.find()` (page 139)
- `Bulk.find.removeOne()` (page 140)
- `Bulk.execute()` (page 137)

db.collection.initializeUnorderedBulkOp()

Definition

`db.collection.initializeUnorderedBulkOp()`

New in version 2.6.

Initializes and returns a new `Bulk()` (page 135) operations builder for a collection. The builder constructs an *unordered* list of write operations that MongoDB executes in bulk.

Behavior

Order of Operation With an *unordered* operations list, MongoDB can execute in parallel the write operations in the list and in any order. If the order of operations matter, use `db.collection.initializeOrderedBulkOp()` (page 149) instead.

Execution of Operations When executing an `unordered` (page 150) list of operations, MongoDB groups the operations. With an unordered bulk operation, the operations in the list may be reordered to increase performance. As such, applications should not depend on the ordering when performing `unordered` (page 150) bulk operations.

Each group of operations can have at most `1000 operations` (page 697). If a group exceeds this `limit` (page 697), MongoDB will divide the group into smaller groups of 1000 or less. For example, if the bulk operations list consists of 2000 insert operations, MongoDB creates 2 groups, each with 1000 operations.

The sizes and grouping mechanics are internal performance details and are subject to change in future versions.

To see how the operations are grouped for a bulk operation execution, call `Bulk.getOperations()` (page 147) *after* the execution.

Error Handling If an error occurs during the processing of one of the write operations, MongoDB will continue to process remaining write operations in the list.

Example The following initializes a `Bulk()` (page 135) operations builder and adds a series of insert operations to add multiple documents:

```
var bulk = db.users.initializeUnorderedBulkOp();
bulk.insert( { user: "abc123", status: "A", points: 0 } );
bulk.insert( { user: "ijk123", status: "A", points: 0 } );
bulk.insert( { user: "mop123", status: "P", points: 0 } );
bulk.execute();
```

See also:

- `db.collection.initializeOrderedBulkOp()` (page 149)
- `Bulk()` (page 135)
- `Bulk.insert()` (page 148)
- `Bulk.execute()` (page 137)

2.1.6 User Management

User Management Methods

Name	Description
`db.changeUserPassword()` (page 152)	Changes an existing user's password.
`db.createUser()` (page 152)	Creates a new user.
`db.dropAllUsers()` (page 154)	Deletes all users associated with a database.
`db.dropUser()` (page 155)	Removes a single user.
`db.getUser()` (page 155)	Returns information about the specified user.
`db.getUsers()` (page 156)	Returns information about all users associated with a database.
`db.grantRolesToUser()` (page 156)	Grants a role and its privileges to a user.
`db.removeUser()` (page 157)	Deprecated. Removes a user from a database.
`db.revokeRolesFromUser()` (page 157)	Removes a role from a user.
`db.updateUser()` (page 159)	Updates user data.

db.changeUserPassword()

Definition

db.**changeUserPassword**(*username*, *password*)

Updates a user's password.

param string username Specifies an existing username with access privileges for this database.

param string password Specifies the corresponding password.

param string mechanism Specifies the authentication mechanism used. Defaults to `MONGODB-CR`. `PLAIN` is used for `SASL/LDAP authentication`, available only in MongoDB Enterprise.

Example The following operation changes the `reporting` user's password to `SOh3TbYhx8ypJPxmt1oOfL`:

```
db.changeUserPassword("reporting", "SOh3TbYhx8ypJPxmt1oOfL")
```

db.createUser()

Definition

db.**createUser**(*user*, *writeConcern*)

Creates a new user for the database where the method runs. `db.createUser()` (page 152) returns a *duplicate user* error if the user already exists on the database.

The `db.createUser()` (page 152) method has the following syntax:

field document user The document with authentication and access information about the user to create.

field document writeConcern The level of `write concern` for the creation operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

The `user` document defines the user and has the following form:

```
{ user: "<name>",
  pwd: "<cleartext password>",
  customData: { <any information> },
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ]
}
```

The `user` document has the following fields:

field string user The name of the new user.

field string pwd The user's password. The `pwd` field is not required if you run `db.createUser()` (page 152) on the `$external` database to create users who have credentials stored externally to MongoDB.

any document customData Any arbitrary information. This field can be used to store any data an admin wishes to associate with this particular user. For example, this could be the user's full name or employee id.

field array roles The roles granted to the user. Can specify an empty array `[]` to create users without roles.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `db.createUser()` (page 152) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

The `db.createUser()` (page 152) method wraps the `createUser` (page 265) command.

Behavior

Encryption `db.createUser()` (page 152) sends password to the MongoDB instance *without* encryption. To encrypt the password during transmission, use `SSL`.

External Credentials Users created on the `$external` database should have credentials stored externally to MongoDB, as, for example, with `MongoDB Enterprise installations that use Kerberos`.

`local` Database You cannot create users on the local database.

Required Access You must have the `createUser` *action* on a database to create a new user on that database.

You must have the `grantRole` *action* on a role's database to grant the role to another user.

If you have the `userAdmin` or `userAdminAnyDatabase` role, you have those actions.

Examples The following `db.createUser()` (page 152) operation creates the `accountAdmin01` user on the `products` database.

```
use products
db.createUser( { "user" : "accountAdmin01",
                 "pwd": "cleartext password",
                 "customData" : { employeeId: 12345 },
                 "roles" : [ { role: "clusterAdmin", db: "admin" },
                             { role: "readAnyDatabase", db: "admin" },
                             "readWrite"
                             ] },
               { w: "majority" , wtimeout: 5000 } )
```

The operation gives `accountAdmin01` the following roles:

- the `clusterAdmin` and `readAnyDatabase` roles on the `admin` database
- the `readWrite` role on the `products` database

Create User with Roles The following operation creates `accountUser` in the `products` database and gives the user the `readWrite` and `dbAdmin` roles.

```
use products
db.createUser(
   {
     user: "accountUser",
     pwd: "password",
     roles: [ "readWrite", "dbAdmin" ]
   }
)
```

Create User Without Roles The following operation creates a user named `reportsUser` in the `admin` database but does not yet assign roles:

```
use admin
db.createUser(
   {
     user: "reportsUser",
     pwd: "password",
     roles: [ ]
   }
)
```

Create Administrative User with Roles The following operation creates a user named `appAdmin` in the `admin` database and gives the user `readWrite` access to the `config` database, which lets the user change certain settings for sharded clusters, such as to the balancer setting.

```
use admin
db.createUser(
   {
     user: "appAdmin",
     pwd: "password",
     roles:
       [
         { role: "readWrite", db: "config" },
         "clusterAdmin"
       ]
   }
)
```

db.dropAllUsers()

Definition

db.**dropAllUsers**(*writeConcern*)

Removes all users from the current database.

> **Warning:** The `dropAllUsers` method removes all users from the database.

The `dropAllUsers` method takes the following arguments:

field document writeConcern The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

The `db.dropAllUsers()` (page 154) method wraps the `dropAllUsersFromDatabase` (page 266) command.

Required Access You must have the `dropUser` *action* on a database to drop a user from that database.

Example The following `db.dropAllUsers()` (page 154) operation drops every user from the `products` database.

```
use products
db.dropAllUsers( {w: "majority", wtimeout: 5000} )
```

The `n` field in the results document shows the number of users removed:

```
{ "n" : 12, "ok" : 1 }
```

db.dropUser()

Definition

`db.dropUser(`*username*, *writeConcern*`)`

Removes the user from the current database.

The `db.dropUser()` (page 155) method takes the following arguments:

param string username The name of the user to remove from the database.

field document writeConcern The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

The `db.dropUser()` (page 155) method wraps the `dropUser` (page 267) command.

Required Access You must have the `dropUser` *action* on a database to drop a user from that database.

Example The following `db.dropUser()` (page 155) operation drops the `accountAdmin01` user on the `products` database.

```
use products
db.dropUser("accountAdmin01", {w: "majority", wtimeout: 5000})
```

db.getUser()

Definition

`db.getUser(`*username*`)`

Returns user information for a specified user. Run this method on the user's database. The user must exist on the database on which the method runs.

The `db.getUser()` (page 155) method has the following parameter:

param string username The name of the user for which to retrieve information.

`db.getUser()` (page 155) wraps the `usersInfo` (page 272) command.

Required Access You must have the `viewUser` *action* on another user's database to view the other user's credentials.

You can view your own information.

Example The following sequence of operations returns information about the `appClient` user on the `accounts` database:

```
use accounts
db.getUser("appClient")
```

db.getUsers()

Definition

`db.getUsers()`

Returns information for all the users in the database.

`db.getUsers()` (page 156) wraps the `usersInfo` (page 272) command.

Required Access You must have the `viewUser` *action* on another user's database to view the other user's credentials.

You can view your own information.

db.grantRolesToUser()

Definition

`db.grantRolesToUser(`*username*, *roles*, *writeConcern*`)`

Grants additional roles to a user.

The `grantRolesToUser` method uses the following syntax:

```
db.grantRolesToUser( "<username>", [ <roles> ], { <writeConcern> } )
```

The `grantRolesToUser` method takes the following arguments:

param string user The name of the user to whom to grant roles.

field array roles An array of additional roles to grant to the user.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `db.grantRolesToUser()` (page 156) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

The `db.grantRolesToUser()` (page 156) method wraps the `grantRolesToUser` (page 267) command.

Required Access You must have the `grantRole` *action* on a database to grant a role on that database.

Example Given a user `accountUser01` in the `products` database with the following roles:

```
"roles" : [
    { "role" : "assetsReader",
      "db" : "assets"
    }
]
```

The following `grantRolesToUser()` operation gives `accountUser01` the `readWrite` role on the `products` database and the `read` role on the `stock` database.

```
use products
db.grantRolesToUser(
   "accountUser01",
   [ "readWrite" , { role: "read", db: "stock" } ],
   { w: "majority" , wtimeout: 4000 }
)
```

The user `accountUser01` in the `products` database now has the following roles:

```
"roles" : [
    { "role" : "assetsReader",
      "db" : "assets"
    },
    { "role" : "read",
      "db" : "stock"
    },
    { "role" : "readWrite",
      "db" : "products"
    }
]
```

db.removeUser()

Deprecated since version 2.6: Use `db.dropUser()` (page 155) instead of `db.removeUser()` (page 157)

Definition

`db.removeUser(`*username*`)`

Removes the specified username from the database.

The `db.removeUser()` (page 157) method has the following parameter:

param string username The database username.

db.revokeRolesFromUser()

Definition

`db.revokeRolesFromUser()`

Removes a one or more roles from a user on the current database. The `db.revokeRolesFromUser()` (page 157) method uses the following syntax:

```
db.revokeRolesFromUser( "<username>", [ <roles> ], { <writeConcern> } )
```

The `revokeRolesFromUser` method takes the following arguments:

param string user The name of the user from whom to revoke roles.

field array roles The roles to remove from the user.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `db.revokeRolesFromUser()` (page 157) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

The `db.revokeRolesFromUser()` (page 157) method wraps the `revokeRolesFromUser` (page 269) command.

Required Access You must have the `revokeRole` *action* on a database to revoke a role on that database.

Example The `accountUser01` user in the `products` database has the following roles:

```
"roles" : [
    { "role" : "assetsReader",
      "db" : "assets"
    },
    { "role" : "read",
      "db" : "stock"
    },
    { "role" : "readWrite",
      "db" : "products"
    }
]
```

The following `db.revokeRolesFromUser()` (page 157) method removes the two of the user's roles: the `read` role on the `stock` database and the `readWrite` role on the `products` database, which is also the database on which the method runs:

```
use products
db.revokeRolesFromUser( "accountUser01",
                        [ { role: "read", db: "stock" }, "readWrite" ],
                        { w: "majority" }
                      )
```

The user `accountUser01` user in the `products` database now has only one remaining role:

```
"roles" : [
    { "role" : "assetsReader",
      "db" : "assets"
    }
]
```

db.updateUser()

Definition

db.**updateUser**(*username*, *update*, *writeConcern*)

Updates the user's profile on the database on which you run the method. An update to a field **completely replaces** the previous field's values. This includes updates to the user's `roles` array.

> **Warning:** When you update the `roles` array, you completely replace the previous array's values. To add or remove roles without replacing all the user's existing roles, use the `db.grantRolesToUser()` (page 156) or `db.revokeRolesFromUser()` (page 157) methods.

The `db.updateUser()` (page 159) method uses the following syntax:

```
db.updateUser(
   "<username>",
   {
     customData : { <any information> },
     roles : [
               { role: "<role>", db: "<database>" } | "<role>",
               ...
             ],
     pwd: "<cleartext password>"
    },
    writeConcern: { <write concern> }
)
```

The `db.updateUser()` (page 159) method has the following arguments.

param string username The name of the user to update.

param document update A document containing the replacement data for the user. This data completely replaces the corresponding data for the user.

field document writeConcern The level of `write concern` for the update operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

The `update` document specifies the fields to update and their new values. All fields in the `update` document are optional, but *must* include at least one field.

The `update` document has the following fields:

field document customData Any arbitrary information.

field array roles The roles granted to the user. An update to the `roles` array overrides the previous array's values.

field string pwd The user's password.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `db.updateUser()` (page 159) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

The `db.updateUser()` (page 159) method wraps the `updateUser` (page 270) command.

Behavior `db.updateUser()` (page 159) sends password to the MongoDB instance *without* encryption. To encrypt the password during transmission, use `SSL`.

Required Access You must have access that includes the `revokeRole` *action* on all databases in order to update a user's `roles` array.

You must have the `grantRole` *action* on a role's database to add a role to a user.

To change another user's `pwd` or `customData` field, you must have the `changeAnyPassword` and `changeAnyCustomData` *actions* respectively on that user's database.

To modify your own password and custom data, you must have privileges that grant `changeOwnPassword` and `changeOwnCustomData` *actions* respectively on the user's database.

Example Given a user `appClient01` in the `products` database with the following user info:

```
{
   "_id" : "products.appClient01",
   "user" : "appClient01",
   "db" : "products",
   "customData" : { "empID" : "12345", "badge" : "9156" },
   "roles" : [
       { "role" : "readWrite",
         "db" : "products"
       },
       { "role" : "read",
         "db" : "inventory"
       }
   ]
}
```

The following `db.updateUser()` (page 159) method **completely** replaces the user's `customData` and `roles` data:

```
use products
db.updateUser( "appClient01",
               {
                 customData : { employeeId : "0x3039" },
                 roles : [
                           { role : "read", db : "assets"  }
                         ]
                }
             )
```

The user `appClient01` in the `products` database now has the following user information:

```
{
   "_id" : "products.appClient01",
   "user" : "appClient01",
   "db" : "products",
   "customData" : { "employeeId" : "0x3039" },
   "roles" : [
       { "role" : "read",
         "db" : "assets"
       }
   ]
}
```

2.1.7 Role Management

Role Management Methods

Name	Description
db.createRole() (page 161)	Creates a role and specifies its privileges.
db.dropAllRoles() (page 162)	Deletes all user-defined roles associated with a database.
db.dropRole() (page 163)	Deletes a user-defined role.
db.getRole() (page 163)	Returns information for the specified role.
db.getRoles() (page 164)	Returns information for all the user-defined roles in a database.
db.grantPrivilegesToRole() (page 165)	Assigns privileges to a user-defined role.
db.grantRolesToRole() (page 166)	Specifies roles from which a user-defined role inherits privileges.
db.revokePrivilegesFromRole() (page 167)	Removes the specified privileges from a user-defined role.
db.revokeRolesFromRole() (page 169)	Removes a role from a user.
db.updateRole() (page 170)	Updates a user-defined role.

db.createRole()

Definition

db.**createRole**(*role*, *writeConcern*)

Creates a role and specifies its *privileges*. The role applies to the database on which you run the method. The db.createRole() (page 161) method returns a *duplicate role* error if the role already exists in the database.

The db.createRole() (page 161) method takes the following arguments:

param document role A document containing the name of the role and the role definition.

field document writeConcern The level of write concern to apply to this operation. The writeConcern document uses the same fields as the getLastError (page 245) command.

The role document has the following form:

```
{ role: "<name>",
  privileges: [
     { resource: { <resource> }, actions: [ "<action>", ... ] },
     ...
  ],
  roles: [
     { role: "<role>", db: "<database>" } | "<role>",
      ...
  ]
}
```

The role document has the following fields:

field string role The name of the new role.

field array privileges The privileges to grant the role. A privilege consists of a resource and permitted actions. You must specify the privileges field. Use an empty array to specify *no* privileges. For the syntax of a privilege, see the privileges array.

field array roles An array of roles from which this role inherits privileges. You must specify the roles field. Use an empty array to specify *no* roles.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `db.createRole()` (page 161) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

The `db.createRole()` (page 161) method wraps the `createRole` (page 274) command.

Behavior A role's privileges apply to the database where the role is created. The role can inherit privileges from other roles in its database. A role created on the `admin` database can include privileges that apply to all databases or to the *cluster* and can inherit privileges from roles in other databases.

Required Access To create a role in a database, the user must have:

- the `createRole` *action* on that *database resource*.
- the `grantRole` *action* on that database to specify privileges for the new role as well as to specify roles to inherit from.

Built-in roles `userAdmin` and `userAdminAnyDatabase` provide `createRole` and `grantRole` actions on their respective `resources`.

Example The following `db.createRole()` (page 161) method creates the `myClusterwideAdmin` role on the `admin` database:

```
use admin
db.createRole({ role: "myClusterwideAdmin",
  privileges: [
    { resource: { cluster: true }, actions: [ "addShard" ] },
    { resource: { db: "config", collection: "" }, actions: [ "find", "update", "insert", "remove" ] }
    { resource: { db: "users", collection: "usersCollection" }, actions: [ "update", "insert", "remov
    { resource: { db: "", collection: "" }, actions: [ "find" ] }
  ],
  roles: [
    { role: "read", db: "admin" }
  ],
  writeConcern: { w: "majority" , wtimeout: 5000 }
})
```

db.dropAllRoles()

Definition

`db.dropAllRoles(writeConcern)`

Deletes all *user-defined* roles on the database where you run the method.

> **Warning:** The `dropAllRoles` method removes *all user-defined* roles from the database.

The `dropAllRoles` method takes the following argument:

field document writeConcern The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

Returns The number of *user-defined* roles dropped.

The `db.dropAllRoles()` (page 162) method wraps the `dropAllRolesFromDatabase` (page 275) command.

Required Access You must have the `dropRole` *action* on a database to drop a role from that database.

Example The following operations drop all *user-defined* roles from the `products` database and uses a *write concern* of `majority`.

```
use products
db.dropAllRoles( { w: "majority" } )
```

The method returns the number of roles dropped:

```
4
```

db.dropRole()

Definition

`db.dropRole(`*rolename*, *writeConcern*`)`

Deletes a *user-defined* role from the database on which you run the method.

The `db.dropRole()` (page 163) method takes the following arguments:

param string rolename The name of the *user-defined role* to remove from the database.

field document writeConcern The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

The `db.dropRole()` (page 163) method wraps the `dropRole` (page 276) command.

Required Access You must have the `dropRole` *action* on a database to drop a role from that database.

Example The following operations remove the `readPrices` role from the `products` database:

```
use products
db.dropRole( "readPrices", { w: "majority" } )
```

db.getRole()

Definition

`db.getRole(`*rolename*, *showPrivileges*`)`

Returns the roles from which this role inherits privileges. Optionally, the method can also return all the role's privileges.

Run `db.getRole()` (page 163) from the database that contains the role. The command can retrieve information for both *user-defined roles* and *built-in roles*.

The db.getRole() (page 163) method takes the following arguments:

param string rolename The name of the role.

param document showPrivileges If `true`, returns the role's privileges. Pass this argument as a document: `{showPrivileges: true}`.

db.getRole() (page 163) wraps the rolesInfo (page 283) command.

Required Access To view a role's information, you must be explicitly granted the role or must have the `viewRole` *action* on the role's database.

Examples The following operation returns role inheritance information for the role `associate` defined on the `products` database:

```
use products
db.getRole( "associate" )
```

The following operation returns role inheritance information *and privileges* for the role `associate` defined on the `products` database:

```
use products
db.getRole( "associate", { showPrivileges: true } )
```

db.getRoles()

Definition

db.**getRoles**()

Returns information for all the roles in the database on which the command runs. The method can be run with or without an argument.

If run without an argument, db.getRoles() (page 164) returns inheritance information for the database's *user-defined* roles.

To return more information, pass the db.getRoles() (page 164) a document with the following fields:

field integer rolesInfo Set this field to `1` to retrieve all user-defined roles.

field Boolean showBuiltinRoles Set to true to display *built-in roles* as well as user-defined roles.

field Boolean showPrivileges Set the field to `true` to show role privileges, including both privileges inherited from other roles and privileges defined directly. By default, the command returns only the roles from which this role inherits privileges and does not return specific privileges.

db.getRoles() (page 164) wraps the rolesInfo (page 283) command.

Required Access To view a role's information, you must be explicitly granted the role or must have the `viewRole` *action* on the role's database.

Example The following operations return documents for all the roles on the `products` database, including role privileges and built-in roles:

```
db.getRoles(
    {
      rolesInfo: 1,
      showPrivileges:true,
```

```
      showBuiltinRoles: true
    }
)
```

db.grantPrivilegesToRole()

Definition

db.**grantPrivilegesToRole**(*rolename*, *privileges*, *writeConcern*)

Grants additional *privileges* to a *user-defined* role.

The `grantPrivilegesToRole()` method uses the following syntax:

```
db.grantPrivilegesToRole(
    "< rolename >",
    [
        { resource: { <resource> }, actions: [ "<action>", ... ] },
        ...
    ],
    { < writeConcern > }
)
```

The `grantPrivilegesToRole()` method takes the following arguments:

param string rolename The name of the role to grant privileges to.

field array privileges The privileges to add to the role. For the format of a privilege, see `privileges`.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

The `grantPrivilegesToRole()` method can grant one or more privileges. Each `<privilege>` has the following syntax:

```
{ resource: { <resource> }, actions: [ "<action>", ... ] }
```

The `db.grantPrivilegesToRole()` (page 165) method wraps the `grantPrivilegesToRole` (page 277) command.

Behavior A role's privileges apply to the database where the role is created. A role created on the `admin` database can include privileges that apply to all databases or to the *cluster*.

Required Access You must have the `grantRole` *action* on the database a privilege targets in order to grant the privilege. To grant a privilege on multiple databases or on the `cluster` resource, you must have the `grantRole` action on the `admin` database.

Example The following `db.grantPrivilegesToRole()` (page 165) operation grants two additional privileges to the role `inventoryCntrl01`, which exists on the `products` database. The operation is run on that database:

```
use products
db.grantPrivilegesToRole(
  "inventoryCntrl01",
  [
```

```
    {
      resource: { db: "products", collection: "" },
      actions: [ "insert" ]
    },
    {
      resource: { db: "products", collection: "system.indexes" },
      actions: [ "find" ]
    }
  ],
  { w: "majority" }
)
```

The first privilege permits users with this role to perform the `insert` *action* on all collections of the `products` database, except the *system collections* (page 688). To access a system collection, a privilege must explicitly specify the system collection in the resource document, as in the second privilege.

The second privilege permits users with this role to perform the `find` *action* on the `product` database's system collection named `system.indexes` (page 689).

db.grantRolesToRole()

Definition

`db.grantRolesToRole(`*rolename*, *roles*, *writeConcern*`)`

Grants roles to a *user-defined role*.

The `grantRolesToRole` method uses the following syntax:

```
db.grantRolesToRole( "<rolename>", [ <roles> ], { <writeConcern> } )
```

The `grantRolesToRole` method takes the following arguments:

param string rolename The name of the role to which to grant sub roles.

field array roles An array of roles from which to inherit.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `db.grantRolesToRole()` (page 166) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

The `db.grantRolesToRole()` (page 166) method wraps the `grantRolesToRole` (page 278) command.

Behavior A role can inherit privileges from other roles in its database. A role created on the `admin` database can inherit privileges from roles in any database.

Required Access You must have the `grantRole` *action* on a database to grant a role on that database.

Example The following `grantRolesToRole()` operation updates the `productsReaderWriter` role in the `products` database to *inherit* the *privileges* of `productsReader` role:

```
use products
db.grantRolesToRole(
    "productsReaderWriter",
    [ "productsReader" ],
    { w: "majority" , wtimeout: 5000 }
)
```

db.revokePrivilegesFromRole()

Definition

`db.revokePrivilegesFromRole(rolename, privileges, writeConcern)`

Removes the specified privileges from the *user-defined* role on the database where the method runs. The `revokePrivilegesFromRole` method has the following syntax:

```
db.revokePrivilegesFromRole(
    "<rolename>",
    [
        { resource: { <resource> }, actions: [ "<action>", ... ] },
        ...
    ],
    { <writeConcern> }
)
```

The `revokePrivilegesFromRole` method takes the following arguments:

param string rolename The name of the *user-defined* role from which to revoke privileges.

field array privileges An array of privileges to remove from the role. See `privileges` for more information on the format of the privileges.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

The `db.revokePrivilegesFromRole()` (page 167) method wraps the `revokePrivilegesFromRole` (page 279) command.

Behavior To revoke a privilege, the `resource document` pattern must match **exactly** the `resource` field of that privilege. The `actions` field can be a subset or match exactly.

For example, given the role `accountRole` in the `products` database with the following privilege that specifies the `products` database as the resource:

```
{
  "resource" : {
      "db" : "products",
      "collection" : ""
  },
  "actions" : [
      "find",
      "update"
  ]
}
```

You *cannot* revoke `find` and/or `update` from just *one* collection in the `products` database. The following operations result in no change to the role:

```
use products
db.revokePrivilegesFromRole(
   "accountRole",
   [
     {
       resource : {
          db : "products",
          collection : "gadgets"
       },
       actions : [
          "find",
          "update"
       ]
     }
   ]
)

db.revokePrivilegesFromRole(
   "accountRole",
   [
     {
       resource : {
          db : "products",
          collection : "gadgets"
       },
       actions : [
          "find"
       ]
     }
   ]
)
```

To revoke the `"find"` and/or the `"update"` action from the role `accountRole`, you must match the resource document exactly. For example, the following operation revokes just the `"find"` action from the existing privilege.

```
use products
db.revokePrivilegesFromRole(
   "accountRole",
   [
     {
       resource : {
          db : "products",
          collection : ""
       },
       actions : [
          "find"
       ]
     }
   ]
)
```

Required Access You must have the `revokeRole` *action* on the database a privilege targets in order to revoke that privilege. If the privilege targets multiple databases or the `cluster` resource, you must have the `revokeRole` action on the `admin` database.

Example The following operation removes multiple privileges from the `associates` role:

```
db.revokePrivilegesFromRole(
   "associate",
   [
     {
       resource: { db: "products", collection: "" },
       actions: [ "createCollection", "createIndex", "find" ]
     },
     {
       resource: { db: "products", collection: "orders" },
       actions: [ "insert" ]
     }
   ],
   { w: "majority" }
)
```

db.revokeRolesFromRole()

Definition

db.**revokeRolesFromRole**(*rolename*, *roles*, *writeConcern*)

Removes the specified inherited roles from a role.

The `revokeRolesFromRole` method uses the following syntax:

```
db.revokeRolesFromRole( "<rolename>", [ <roles> ], { <writeConcern> } )
```

The `revokeRolesFromRole` method takes the following arguments:

param string rolename The name of the role from which to revoke roles.

field array roles The inherited roles to remove.

field document writeConcern The level of `write concern` to apply to this operation. The `writeConcern` document uses the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `db.revokeRolesFromRole()` (page 169) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

The `db.revokeRolesFromRole()` (page 169) method wraps the `revokeRolesFromRole` (page 282) command.

Required Access You must have the `revokeRole` *action* on a database to revoke a role on that database.

Example The `purchaseAgents` role in the `emea` database inherits privileges from several other roles, as listed in the `roles` array:

```
{
   "_id" : "emea.purchaseAgents",
   "role" : "purchaseAgents",
   "db" : "emea",
   "privileges" : [],
   "roles" : [
      {
         "role" : "readOrdersCollection",
         "db" : "emea"
      },
      {
         "role" : "readAccountsCollection",
         "db" : "emea"
      },
      {
         "role" : "writeOrdersCollection",
         "db" : "emea"
      }
   ]
}
```

The following `db.revokeRolesFromRole()` (page 169) operation on the `emea` database removes two roles from the `purchaseAgents` role:

```
use emea
db.revokeRolesFromRole( "purchaseAgents",
                        [
                          "writeOrdersCollection",
                          "readOrdersCollection"
                        ],
                        { w: "majority" , wtimeout: 5000 }
                      )
```

The `purchaseAgents` role now contains just one role:

```
{
   "_id" : "emea.purchaseAgents",
   "role" : "purchaseAgents",
   "db" : "emea",
   "privileges" : [],
   "roles" : [
      {
         "role" : "readAccountsCollection",
         "db" : "emea"
      }
   ]
}
```

db.updateRole()

Definition

db.**updateRole**(*rolename*, *update*, *writeConcern*)

Updates a *user-defined role*. The `db.updateRole()` (page 170) method must run on the role's database.

An update to a field **completely replaces** the previous field's values. To grant or remove roles or *privileges* without replacing all values, use one or more of the following methods:

- `db.grantRolesToRole()` (page 166)

- db.grantPrivilegesToRole() (page 165)
- db.revokeRolesFromRole() (page 169)
- db.revokePrivilegesFromRole() (page 167)

Warning: An update to the privileges or roles array completely replaces the previous array's values.

The updateRole() method uses the following syntax:

```
db.updateRole(
    "<rolename>",
    {
      privileges:
          [
            { resource: { <resource> }, actions: [ "<action>", ... ] },
            ...
          ],
      roles:
          [
            { role: "<role>", db: "<database>" } | "<role>",
            ...
          ]
    },
    { <writeConcern> }
)
```

The db.updateRole() (page 170) method takes the following arguments.

param string rolename The name of the *user-defined role* to update.

param document update A document containing the replacement data for the role. This data completely replaces the corresponding data for the role.

field document writeConcern The level of write concern for the update operation. The writeConcern document takes the same fields as the getLastError (page 245) command.

The update document specifies the fields to update and the new values. Each field in the update document is optional, but the document must include at least one field. The update document has the following fields:

field array privileges Required if you do not specify roles array. The privileges to grant the role. An update to the privileges array overrides the previous array's values. For the syntax for specifying a privilege, see the privileges array.

field array roles Required if you do not specify privileges array. The roles from which this role inherits privileges. An update to the roles array overrides the previous array's values.

In the roles field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where db.updateRole() (page 170) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

The db.updateRole() (page 170) method wraps the updateRole (page 286) command.

Behavior A role's privileges apply to the database where the role is created. The role can inherit privileges from other roles in its database. A role created on the `admin` database can include privileges that apply to all databases or to the *cluster* and can inherit privileges from roles in other databases.

Required Access You must have the `revokeRole` *action* on all databases in order to update a role.

You must have the `grantRole` *action* on the database of each role in the `roles` array to update the array.

You must have the `grantRole` action on the database of each privilege in the `privileges` array to update the array. If a privilege's resource spans databases, you must have `grantRole` on the `admin` database. A privilege spans databases if the privilege is any of the following:

- a collection in all databases
- all collections and all database
- the `cluster` resource

Example The following `db.updateRole()` (page 170) method replaces the `privileges` and the `roles` for the `inventoryControl` role that exists in the `products` database. The method runs on the database that contains `inventoryControl`:

```
use products
db.updateRole(
    "inventoryControl",
    {
      privileges:
          [
            {
              resource: { db:"products", collection:"clothing" },
              actions: [ "update", "createCollection", "createIndex"]
            }
          ],
      roles:
          [
            {
              role: "read",
              db: "products"
            }
          ]
    },
    { w:"majority" }
)
```

To view a role's privileges, use the `rolesInfo` (page 283) command.

2.1.8 Replication

Replication Methods

Name	Description
rs.add() (page 173)	Adds a member to a replica set.
rs.addArb() (page 174)	Adds an *arbiter* to a replica set.
rs.conf() (page 174)	Returns the replica set configuration document.
rs.freeze() (page 175)	Prevents the current member from seeking election as primary for a period of time.
rs.help() (page 175)	Returns basic help text for *replica set* functions.
rs.initiate() (page 175)	Initializes a new replica set.
rs.printReplicationInfo() (page 175)	Prints a report of the status of the replica set from the perspective of the primary.
rs.printSlaveReplicationInfo() (page 176)	Prints a report of the status of the replica set from the perspective of the secondaries.
rs.reconfig() (page 176)	Re-configures a replica set by applying a new replica set configuration object.
rs.remove() (page 178)	Remove a member from a replica set.
rs.slaveOk() (page 178)	Sets the slaveOk property for the current connection. Deprecated. Use readPref() (page 93) and Mongo.setReadPref() (page 204) to set *read preference*.
rs.status() (page 179)	Returns a document with information about the state of the replica set.
rs.stepDown() (page 179)	Causes the current *primary* to become a secondary which forces an *election*.
rs.syncFrom() (page 179)	Sets the member that this replica set member will sync from, overriding the default sync target selection logic.

rs.add()

Definition

rs.**add**()

Adds a member to a *replica set*. To run the method, you must connect to the *primary* of the replica set.

param string,document host The new member to add to the replica set.

If a string, specify the hostname and optionally the port number for the new member. See *Pass a Hostname String to rs.add()* (page 174) for an example.

If a document, specify a replica set member configuration document as found in the members array. You must specify _id and the host fields in the member configuration document. See *Pass a Member Configuration Document to rs.add()* (page 174) for an example.

See http://docs.mongodb.org/manual/reference/replica-configuration document for full documentation of all replica set configuration options

param boolean arbiterOnly Applies only if the <host> value is a string. If true, the added host is an arbiter.

rs.add() (page 173) provides a wrapper around some of the functionality of the replSetReconfig (page 300) *database command* and the corresponding mongo (page 610) shell helper rs.reconfig() (page 176). See the http://docs.mongodb.org/manual/reference/replica-configuration document for full documentation of all replica set configuration options.

Behavior `rs.add()` (page 173) can, in some cases, force an election for primary which will disconnect the shell. In such cases, the `mongo` (page 610) shell displays an error even if the operation succeeds.

Example

Pass a Hostname String to `rs.add()` The following operation adds a `mongod` (page 583) instance, running on the host `mongodb3.example.net` and accessible on the default port `27017`:

```
rs.add('mongodb3.example.net:27017')
```

If `mongodb3.example.net` is an arbiter, use the following form:

```
rs.add('mongodb3.example.net:27017', true)
```

Pass a Member Configuration Document to `rs.add()` Changed in version 2.8.0: Previous version required an `_id` file in the document you passed to `rs.add()` (page 173). After 2.8.0 you can omit the `_id` field in this document.

The following operation adds a `mongod` (page 583) instance, running on the host `mongodb4.example.net` and accessible on the default port `27017`, as a `priority 0` secondary member:

```
rs.add( { host: "mongodbd4.example.net:27017", priority: 0 } )
```

You must specify the `host` field in the member configuration document.

See the `http://docs.mongodb.org/manual/reference/replica-configuration` for the available replica set member configuration settings.

See `http://docs.mongodb.org/manual/administration/replica-sets` for more examples and information.

rs.addArb()

Description

`rs.addArb(`*host*`)`

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

The `rs.addArb()` (page 174) method takes the following parameter:

param string host Specifies the hostname and optionally the port number of the arbiter member to add to replica set.

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

rs.conf()

`rs.conf()`

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

Wraps `replSetGetConfig` (page 292). See `http://docs.mongodb.org/manual/reference/replica-config` for more information on the replica set configuration document.

`rs.config()`

`rs.config()` (page 174) is an alias of `rs.conf()` (page 174).

rs.freeze()

Description

`rs.freeze(seconds)`

Makes the current *replica set* member ineligible to become *primary* for the period specified.

The `rs.freeze()` (page 175) method has the following parameter:

param number seconds The duration the member is ineligible to become primary.

`rs.freeze()` (page 175) provides a wrapper around the *database command* `replSetFreeze` (page 291).

rs.help()

`rs.help()`

Returns a basic help text for all of the `replication` related shell functions.

rs.initiate()

Description

`rs.initiate(configuration)`

Initiates a *replica set*. Optionally takes a configuration argument in the form of a *document* that holds the configuration of a replica set.

The `rs.initiate()` (page 175) method has the following parameter:

param document configuration A *document* that specifies `configuration settings` for the new replica set. If a configuration is not specified, MongoDB uses a default configuration.

The `rs.initiate()` (page 175) method provides a wrapper around the "`replSetInitiate` (page 299)" *database command*.

Replica Set Configuration See `http://docs.mongodb.org/manual/administration/replica-set-member-con` and `http://docs.mongodb.org/manual/reference/replica-configuration` for examples of replica set configuration and invitation objects.

rs.printReplicationInfo()

`rs.printReplicationInfo()`

New in version 2.6.

Prints a formatted report of the status of a *replica set* from the perspective of the *primary* member of the set if run on the primary. [10] The displayed report formats the data returned by `db.getReplicationInfo()` (page 117).

Note: The `rs.printReplicationInfo()` (page 175) in the `mongo` (page 610) shell does **not** return *JSON*. Use `rs.printReplicationInfo()` (page 175) for manual inspection, and `db.getReplicationInfo()` (page 117) in scripts.

The output of `rs.printReplicationInfo()` (page 175) is identical to that of `db.printReplicationInfo()` (page 121).

[10] If run on a secondary, the method calls `db.printSlaveReplicationInfo()` (page 122). See `db.printSlaveReplicationInfo()` (page 122) for details.

Output Example The following example is a sample output from the `rs.printReplicationInfo()` (page 175) method run on the primary:

```
configured oplog size:   192MB
log length start to end: 65422secs (18.17hrs)
oplog first event time:  Mon Jun 23 2014 17:47:18 GMT-0400 (EDT)
oplog last event time:   Tue Jun 24 2014 11:57:40 GMT-0400 (EDT)
now:                     Thu Jun 26 2014 14:24:39 GMT-0400 (EDT)
```

Output Fields `rs.printReplicationInfo()` (page 175) formats and prints the data returned by `db.getReplicationInfo()` (page 117):

configured oplog size Displays the `db.getReplicationInfo.logSizeMB` (page 117) value.

log length start to end Displays the `db.getReplicationInfo.timeDiff` (page 117) and `db.getReplicationInfo.timeDiffHours` (page 117) values.

oplog first event time Displays the `db.getReplicationInfo.tFirst` (page 117).

oplog last event time Displays the `db.getReplicationInfo.tLast` (page 118).

now Displays the `db.getReplicationInfo.now` (page 118).

See `db.getReplicationInfo()` (page 117) for description of the data.

rs.printSlaveReplicationInfo()

Definition

`rs.printSlaveReplicationInfo()`

Returns a formatted report of the status of a *replica set* from the perspective of the *secondary* member of the set. The output is identical to that of `db.printSlaveReplicationInfo()` (page 122).

Output The following is example output from the `rs.printSlaveReplicationInfo()` (page 176) method issued on a replica set with two secondary members:

```
source: m1.example.net:27017
    syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
    0 secs (0 hrs) behind the primary
source: m2.example.net:27017
    syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
    0 secs (0 hrs) behind the primary
```

A *delayed member* may show as `0` seconds behind the primary when the inactivity period on the primary is greater than the `slaveDelay` value.

rs.reconfig()

Definition

`rs.reconfig(`*configuration*, *force*`)`

Reconfigures an existing replica set, overwriting the existing replica set configuration. To run the method, you must connect to the *primary* of the replica set.

param document configuration A `document` that specifies the configuration of a replica set.

param document force If set as `{ force: true }`, this forces the replica set to accept the new configuration even if a majority of the members are not accessible. Use with caution, as this can lead to *rollback* situations.

When used to reconfigure an existing replica set, first retrieve the current configuration with `rs.conf()` (page 174), modify the configuration document as needed, and then pass the modified document to `rs.reconfig()` (page 176).

`rs.reconfig()` (page 176) provides a wrapper around the `replSetReconfig` (page 300) *database command*.

Behavior The method disconnects the `mongo` (page 610) shell and forces a reconnection as the replica set renegotiates which member will be *primary*. As a result, the shell will display an error even if this command succeeds.

The `rs.reconfig()` (page 176) shell method can force the current primary to step down and triggers an election in some situations. When the primary steps down, the primary closes all client connections. This is by design. Since this typically takes 10-20 seconds, attempt to make such changes during scheduled maintenance periods.

Warning: Using `rs.reconfig()` (page 176) with `{ force: true }` can lead to *rollback* situations and other difficult-to-recover-from situations. Exercise caution when using this option.

Examples A replica set named `rs0` has the following configuration:

```
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [
         {
            "_id" : 0,
            "host" : "mongodb0.example.net:27017"
         },
         {
            "_id" : 1,
            "host" : "mongodb1.example.net:27017"
         },
         {
            "_id" : 2,
            "host" : "mongodb2.example.net:27017"
         }
     ]
}
```

The following sequence of operations updates the `priority` of the second member. The operations are issued through a `mongo` (page 610) shell connected to the primary.

```
cfg = rs.conf();
cfg.members[1].priority = 2;
rs.reconfig(cfg);
```

1. The first statement uses the `rs.conf()` (page 174) method to retrieve a document containing the current `configuration` for the replica set and sets the document to the local variable `cfg`.
2. The second statement sets a `priority` value to the second document in the `members` array. For additional settings, see *replica set configuration settings*.

   To access the member configuration document in the array, the statement uses the array index and **not** the replica set member's `_id` field.
3. The last statement calls the `rs.reconfig()` (page 176) method with the modified `cfg` to initialize this new configuration. Upon successful reconfiguration, the replica set configuration will resemble the following:

```
{
    "_id" : "rs0",
    "version" : 2,
    "members" : [
         {
            "_id" : 0,
            "host" : "mongodb0.example.net:27017"
         },
         {
            "_id" : 1,
            "host" : "mongodb1.example.net:27017",
            "priority" : 2
         },
         {
            "_id" : 2,
            "host" : "mongodb2.example.net:27017"
         }
     ]
}
```

See also:

`http://docs.mongodb.org/manual/reference/replica-configuration` and `http://docs.mongodb.org/manual/administration/replica-sets`.

rs.remove()

Definition

`rs.remove(`*hostname*`)`

Removes the member described by the `hostname` parameter from the current *replica set*. This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates which member will be *primary*. As a result, the shell will display an error even if this command succeeds.

The `rs.remove()` (page 178) method has the following parameter:

param string hostname The hostname of a system in the replica set.

Note: Before running the `rs.remove()` (page 178) 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 178), but it remains good practice.

---

rs.slaveOk()

`rs.slaveOk()`

Provides a shorthand for the following operation:

```
db.getMongo().setSlaveOk()
```

This allows the current connection to allow read operations to run on *secondary* members. See the `readPref()` (page 93) method for more fine-grained control over `read preference` in the `mongo` (page 610) shell.

rs.status()

`rs.status()`

Returns A *document* with status information.

This output reflects the current status of the replica set, using data derived from the heartbeat packets sent by the other members of the replica set.

This method provides a wrapper around the `replSetGetStatus` (page 296) command. See the documentation of the command for a complete description of the *output* (page 296).

rs.stepDown()

Description

`rs.stepDown(seconds)`

Forces the current *replica set* member to step down as *primary* and then attempt to avoid election as primary for the designated number of seconds. Produces an error if the current member is not the primary.

The `rs.stepDown()` (page 179) method has the following parameter:

param number seconds The duration of time that the stepped-down member attempts to avoid re-election as primary. If this parameter is not specified, the method uses the default value of 60 seconds.

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

`rs.stepDown()` (page 179) provides a wrapper around the *database command* `replSetStepDown` (page 301).

rs.syncFrom()

`rs.syncFrom()`

New in version 2.2.

Provides a wrapper around the `replSetSyncFrom` (page 302), 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 302) for more details.

See `http://docs.mongodb.org/manual/tutorial/configure-replica-set-secondary-sync-target` for details how to use this command.

2.1.9 Sharding

Sharding Methods

Name	Description
sh._adminCommand (page 182)	Runs a *database command* against the admin database, like db.runCommand() (page 123), but can confirm that it is issued against a mongos (page 601).
sh._checkFullName() (page 182)	Tests a namespace to determine if its well formed.
sh._checkMongos() (page 182)	Tests to see if the mongo (page 610) shell is connected to a mongos (page 601) instance.
sh._lastMigration() (page 182)	Reports on the last *chunk* migration.
sh.addShard() (page 183)	Adds a *shard* to a sharded cluster.
sh.addShardTag() (page 184)	Associates a shard with a tag, to support tag aware sharding.
sh.addTagRange() (page 184)	Associates range of shard keys with a shard tag, to support tag aware sharding.
sh.disableBalancing() (page 185)	Disable balancing on a single collection in a sharded database. Does not affect balancing of other collections in a sharded cluster.
sh.enableBalancing() (page 185)	Activates the sharded collection balancer process if previously disabled using sh.disableBalancing() (page 185).
sh.enableSharding() (page 186)	Enables sharding on a specific database.
sh.getBalancerHost() (page 186)	Returns the name of a mongos (page 601) that's responsible for the balancer process.
sh.getBalancerState() (page 186)	Returns a boolean to report if the *balancer* is currently enabled.
sh.help() (page 187)	Returns help text for the sh methods.
sh.isBalancerRunning() (page 187)	Returns a boolean to report if the balancer process is currently migrating chunks.
sh.moveChunk() (page 187)	Migrates a *chunk* in a *sharded cluster*.
sh.removeShardTag() (page 188)	Removes the association between a shard and a shard tag.
sh.removeTagRange() (page 188)	Removes an association between a range shard keys and a shard tag. Use to manage tag aware sharding.
sh.setBalancerState() (page 189)	Enables or disables the *balancer* which migrates *chunks* between *shards*.
sh.shardCollection() (page 189)	Enables sharding for a collection.
sh.splitAt() (page 190)	Divides an existing *chunk* into two chunks using a specific value of the *shard key* as the dividing point.
sh.splitFind() (page 190)	Divides an existing *chunk* that contains a document matching a query into two approximately equal chunks.
sh.startBalancer() (page 191)	Enables the *balancer* and waits for balancing to start.
sh.status() (page 191)	Reports on the status of a *sharded cluster*, as db.printShardingStatus() (page 122).
sh.stopBalancer() (page 194)	Disables the *balancer* and waits for any in progress balancing rounds to complete.
sh.waitForBalancer() (page 195)	Internal. Waits for the balancer state to change.
sh.waitForBalancerOff() (page 195)	Internal. Waits until the balancer stops running.
sh.waitForDLock() (page 195)	Internal. Waits for a specified distributed *sharded cluster* lock.
sh.waitForPingChange() (page 196)	Internal. Waits for a change in ping state from one of the mongos (page 601) in the sharded cluster.

sh._adminCommand()

Definition

`sh._adminCommand(command, checkMongos)`

Runs a database command against the admin database of a `mongos` (page 601) instance.

param string command A database command to run against the `admin` database.

param boolean checkMongos Require verification that the shell is connected to a `mongos` (page 601) instance.

See also:

`db.runCommand()` (page 123)

sh._checkFullName()

Definition

`sh._checkFullName(namespace)`

Verifies that a *namespace* name is well formed. If the namespace is well formed, the `sh._checkFullName()` (page 182) method exits *with no message*.

Throws If the namespace is not well formed, `sh._checkFullName()` (page 182) throws: "name needs to be fully qualified <db>.<collection>"

The `sh._checkFullName()` (page 182) method has the following parameter:

param string namespace The *namespace* of a collection. The namespace is the combination of the database name and the collection name. Enclose the namespace in quotation marks.

sh._checkMongos()

`sh._checkMongos()`

Returns nothing

Throws "not connected to a mongos"

The `sh._checkMongos()` (page 182) method throws an error message if the `mongo` (page 610) shell is not connected to a `mongos` (page 601) instance. Otherwise it exits (no return document or return code).

sh._lastMigration()

Definition

`sh._lastMigration(namespace)`

Returns information on the last migration performed on the specified database or collection.

The `sh._lastMigration()` (page 182) method has the following parameter:

param string namespace The *namespace* of a database or collection within the current database.

Output The `sh._lastMigration()` (page 182) method returns a document with details about the last migration performed on the database or collection. The document contains the following output:

`sh._lastMigration._id`

The id of the migration task.

`sh._lastMigration.server`
The name of the server.

`sh._lastMigration.clientAddr`
The IP address and port number of the server.

`sh._lastMigration.time`
The time of the last migration, formatted as *ISODate*.

`sh._lastMigration.what`
The specific type of migration.

`sh._lastMigration.ns`
The complete *namespace* of the collection affected by the migration.

`sh._lastMigration.details`
A document containing details about the migrated chunk. The document includes `min` and `max` sub-documents with the bounds of the migrated chunk.

sh.addShard()

Definition

`sh.addShard(host)`
Adds a database instance or replica set to a *sharded cluster*. The optimal configuration is to deploy shards across *replica sets*. This method must be run on a `mongos` (page 601) instance.

The `sh.addShard()` (page 183) method has the following parameter:

param string host The hostname of either a standalone database instance or of a replica set. Include the port number if the instance is running on a non-standard port. Include the replica set name if the instance is a replica set, as explained below.

The `sh.addShard()` (page 183) method has the following prototype form:

```
sh.addShard("<host>")
```

The `host` parameter can be in any of the following forms:

```
[hostname]
[hostname]:[port]
[replica-set-name]/[hostname]
[replica-set-name]/[hostname]:port
```

> **Warning:** Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.
> New in version 2.6: `mongos` (page 601) installed from official `.deb` and `.rpm` packages have the `bind_ip` configuration set to `127.0.0.1` by default.

The `sh.addShard()` (page 183) method is a helper for the `addShard` (page 303) command. The `addShard` (page 303) command has additional options which are not available with this helper.

Considerations

Balancing When you add a shard to a sharded cluster, you affect the balance of *chunks* among the shards of a cluster for all existing sharded collections. The balancer will begin migrating chunks so that the cluster will achieve balance. See `http://docs.mongodb.org/manual/core/sharding-balancing` for more information.

Hidden Members

Important: You cannot include a `hidden member` in the seed list provided to `sh.addShard()` (page 183).

Example To add a shard on a replica set, specify the name of the replica set and the hostname of at least one member of the replica set, as a seed. If you specify additional hostnames, all must be members of the same replica set.

The following example adds a replica set named `repl0` and specifies one member of the replica set:

```
sh.addShard("repl0/mongodb3.example.net:27327")
```

sh.addShardTag()

Definition

`sh.addShardTag(shard, tag)`

New in version 2.2.

Associates a shard with a tag or identifier. MongoDB uses these identifiers to direct *chunks* that fall within a tagged range to specific shards. `sh.addTagRange()` (page 184) associates chunk ranges with tag ranges.

param string shard The name of the shard to which to give a specific tag.

param string tag The name of the tag to add to the shard.

Only issue `sh.addShardTag()` (page 184) when connected to a `mongos` (page 601) 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 184) and `sh.removeShardTag()` (page 188).

sh.addTagRange()

Definition

`sh.addTagRange(namespace, minimum, maximum, tag)`

New in version 2.2.

Attaches a range of shard key values to a shard tag created using the `sh.addShardTag()` (page 184) method. `sh.addTagRange()` (page 184) takes the following arguments:

param string namespace The *namespace* of the sharded collection to tag.

param document minimum The minimum value of the *shard key* range to include in the tag. The minimum is an inclusive match. Specify the minimum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.

param document maximum The maximum value of the shard key range to include in the tag. The maximum is an exclusive match. Specify the maximum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.

param string tag The name of the tag to attach the range specified by the `minimum` and `maximum` arguments to.

Use `sh.addShardTag()` (page 184) to ensure that the balancer migrates documents that exist within the specified range to a specific shard or set of shards.

Only issue `sh.addTagRange()` (page 184) when connected to a `mongos` (page 601) instance.

Behavior

Bounds Shard ranges are always inclusive of the lower value and exclusive of the upper boundary.

Dropped Collections If you add a tag range to a collection using `sh.addTagRange()` (page 184) and then later drop the collection or its database, MongoDB does not remove the tag association. If you later create a new collection with the same name, the old tag association will apply to the new collection.

Example Given a shard key of `{state: 1, zip: 1}`, the following operation creates a tag range covering zip codes in New York State:

```
sh.addTagRange( "exampledb.collection",
                { state: "NY", zip: MinKey },
                { state: "NY", zip: MaxKey },
                "NY"
              )
```

sh.disableBalancing()

Description

`sh.disableBalancing(namespace)`

Disables the balancer for the specified sharded collection. This does not affect the balancing of *chunks* for other sharded collections in the same cluster.

The `sh.disableBalancing()` (page 185) method has the following parameter:

param string namespace The *namespace* of the collection.

For more information on the balancing process, see `http://docs.mongodb.org/manual/tutorial/manage-sharde` and *sharding-balancing*.

sh.enableBalancing()

Description

`sh.enableBalancing(namespace)`

Enables the balancer for the specified namespace of the sharded collection.

The `sh.enableBalancing()` (page 185) method has the following parameter:

param string namespace The *namespace* of the collection.

Important: `sh.enableBalancing()` (page 185) does not *start* balancing. Rather, it allows balancing of this collection the next time the balancer runs.

For more information on the balancing process, see `http://docs.mongodb.org/manual/tutorial/manage-sharde` and *sharding-balancing*.

sh.enableSharding()

Definition

`sh.enableSharding(database)`

Enables sharding on the specified database. This does not automatically shard any collections but makes it possible to begin sharding collections using `sh.shardCollection()` (page 189).

The `sh.enableSharding()` (page 186) method has the following parameter:

param string database The name of the database shard. Enclose the name in quotation marks.

See also:

`sh.shardCollection()` (page 189)

sh.getBalancerHost()

`sh.getBalancerHost()`

Returns String in form *hostname*:*port*

`sh.getBalancerHost()` (page 186) returns the name of the server that is running the balancer.

See also:

- `sh.enableBalancing()` (page 185)
- `sh.disableBalancing()` (page 185)
- `sh.getBalancerState()` (page 186)
- `sh.isBalancerRunning()` (page 187)
- `sh.setBalancerState()` (page 189)
- `sh.startBalancer()` (page 191)
- `sh.stopBalancer()` (page 194)
- `sh.waitForBalancer()` (page 195)
- `sh.waitForBalancerOff()` (page 195)

sh.getBalancerState()

`sh.getBalancerState()`

Returns boolean

`sh.getBalancerState()` (page 186) returns `true` when the *balancer* is enabled and false if the balancer is disabled. This does not reflect the current state of balancing operations: use `sh.isBalancerRunning()` (page 187) to check the balancer's current state.

See also:

- `sh.enableBalancing()` (page 185)
- `sh.disableBalancing()` (page 185)
- `sh.getBalancerHost()` (page 186)
- `sh.isBalancerRunning()` (page 187)
- `sh.setBalancerState()` (page 189)

- sh.startBalancer() (page 191)
- sh.stopBalancer() (page 194)
- sh.waitForBalancer() (page 195)
- sh.waitForBalancerOff() (page 195)

sh.help()

sh.**help**()

Returns a basic help text for all sharding related shell functions.

sh.isBalancerRunning()

sh.**isBalancerRunning**()

Returns boolean

Returns true if the *balancer* process is currently running and migrating chunks and false if the balancer process is not running. Use sh.getBalancerState() (page 186) to determine if the balancer is enabled or disabled.

See also:

- sh.enableBalancing() (page 185)
- sh.disableBalancing() (page 185)
- sh.getBalancerHost() (page 186)
- sh.getBalancerState() (page 186)
- sh.setBalancerState() (page 189)
- sh.startBalancer() (page 191)
- sh.stopBalancer() (page 194)
- sh.waitForBalancer() (page 195)
- sh.waitForBalancerOff() (page 195)

sh.moveChunk()

Definition

sh.**moveChunk**(*namespace*, *query*, *destination*)

Moves the *chunk* that contains the document specified by the query to the destination shard. sh.moveChunk() (page 187) provides a wrapper around the moveChunk (page 310) database command and takes the following arguments:

param string namespace The *namespace* of the sharded collection that contains the chunk to migrate.

param document query An equality match on the shard key that selects the chunk to move.

param string destination The name of the shard to move.

Important: In most circumstances, allow the *balancer* to automatically migrate *chunks*, and avoid calling sh.moveChunk() (page 187) directly.

See also:

moveChunk (page 310), sh.splitAt() (page 190), sh.splitFind() (page 190), http://docs.mongodb.org/manual/sharding, and *chunk migration*.

Example Given the people collection in the records database, the following operation finds the chunk that contains the documents with the zipcode field set to 53187 and then moves that chunk to the shard named shard0019:

```
sh.moveChunk("records.people", { zipcode: "53187" }, "shard0019")
```

sh.removeShardTag()

Definition

sh.**removeShardTag**(*shard*, *tag*)

New in version 2.2.

Removes the association between a tag and a shard. Only issue sh.removeShardTag() (page 188) when connected to a mongos (page 601) instance.

param string shard The name of the shard from which to remove a tag.

param string tag The name of the tag to remove from the shard.

See also:

sh.addShardTag() (page 184), sh.addTagRange() (page 184)

sh.removeTagRange()

Definition

sh.**removeTagRange**(*namespace*, *minimum*, *maximum*, *tag*)

New in version 2.8.

Removes a range of shard key values to a shard tag created using the sh.removeShardTag() (page 188) method. sh.removeTagRange() (page 188) takes the following arguments:

param string namespace The *namespace* of the sharded collection to tag.

param document minimum The minimum value of the *shard key* from the tag. Specify the minimum value in the form of <fieldname>:<value>. This value must be of the same BSON type or types as the shard key.

param document maximum The maximum value of the shard key range from the tag. Specify the maximum value in the form of <fieldname>:<value>. This value must be of the same BSON type or types as the shard key.

param string tag The name of the tag attached to the range specified by the minimum and maximum arguments to.

Use sh.removeShardTag() (page 188) to ensure that unused or out of date ranges are removed and hence chunks are balanced as required.

Only issue sh.removeTagRange() (page 188) when connected to a mongos (page 601) instance.

Example Given a shard key of `{state: 1, zip: 1}`, the following operation removes an existing tag range covering zip codes in New York State:

```
sh.removeTagRange( "exampledb.collection",
                   { state: "NY", zip: MinKey },
                   { state: "NY", zip: MaxKey },
                   "NY"
                 )
```

sh.setBalancerState()

Description

`sh.setBalancerState(state)`

Enables or disables the *balancer*. Use `sh.getBalancerState()` (page 186) to determine if the balancer is currently enabled or disabled and `sh.isBalancerRunning()` (page 187) to check its current state.

The `sh.getBalancerState()` (page 186) method has the following parameter:

param Boolean state Set this to `true` to enable the balancer and `false` to disable it.

See also:

- `sh.enableBalancing()` (page 185)
- `sh.disableBalancing()` (page 185)
- `sh.getBalancerHost()` (page 186)
- `sh.getBalancerState()` (page 186)
- `sh.isBalancerRunning()` (page 187)
- `sh.startBalancer()` (page 191)
- `sh.stopBalancer()` (page 194)
- `sh.waitForBalancer()` (page 195)
- `sh.waitForBalancerOff()` (page 195)

sh.shardCollection()

Definition

`sh.shardCollection(namespace, key, unique)`

Shards a collection using the `key` as a the *shard key*. `sh.shardCollection()` (page 189) takes the following arguments:

param string namespace The *namespace* of the collection to shard.

param document key A *document* that specifies the *shard key* to use to *partition* and distribute objects among the shards. A shard key may be one field or multiple fields. A shard key with multiple fields is called a "compound shard key."

param Boolean unique When true, ensures that the underlying index enforces a unique constraint. *Hashed shard keys* do not support unique constraints.

New in version 2.4: Use the form `{field: "hashed"}` to create a *hashed shard key*. Hashed shard keys may not be compound indexes.

Considerations MongoDB provides no method to deactivate sharding for a collection after calling `shardCollection` (page 314). Additionally, after `shardCollection` (page 314), you cannot change shard keys or modify the value of any field used in your shard key index.

Example Given the `people` collection in the `records` database, the following command shards the collection by the `zipcode` field:

```
sh.shardCollection("records.people", { zipcode: 1} )
```

Additional Information `shardCollection` (page 314) for additional options, `http://docs.mongodb.org/manual/sharding` and `http://docs.mongodb.org/manual/core/sharding-int` for an overview of sharding, `http://docs.mongodb.org/manual/tutorial/deploy-shard-cluster` for a tutorial, and *sharding-shard-key* for choosing a shard key.

sh.splitAt()

Definition

`sh.splitAt(`*namespace*, *query*`)`

Splits the chunk containing the document specified by the query as if that document were at the "middle" of the collection, even if the specified document is not the actual median of the collection.

param string namespace The namespace (i.e. `<database>.<collection>`) of the sharded collection that contains the chunk to split.

param document query A query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Use this command to manually split chunks unevenly. Use the "`sh.splitFind()` (page 190)" 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 190).

sh.splitFind()

Definition

`sh.splitFind(`*namespace*, *query*`)`

Splits the chunk containing the document specified by the `query` at its median point, creating two roughly equal chunks. Use `sh.splitAt()` (page 190) 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 190).

param string namespace The namespace (i.e. `<database>.<collection>`) of the sharded collection that contains the chunk to split.

param document query A query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

sh.startBalancer()

Definition

`sh.startBalancer(timeout, interval)`

Enables the balancer in a sharded cluster and waits for balancing to initiate.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep each cycle of waiting.

See also:

- `sh.enableBalancing()` (page 185)
- `sh.disableBalancing()` (page 185)
- `sh.getBalancerHost()` (page 186)
- `sh.getBalancerState()` (page 186)
- `sh.isBalancerRunning()` (page 187)
- `sh.setBalancerState()` (page 189)
- `sh.stopBalancer()` (page 194)
- `sh.waitForBalancer()` (page 195)
- `sh.waitForBalancerOff()` (page 195)

sh.status()

Definition

`sh.status()`

Prints a formatted report of the sharding configuration and the information regarding existing chunks in a *sharded cluster*. The default behavior suppresses the detailed chunk information if the total number of chunks is greater than or equal to 20.

The `sh.status()` (page 191) method has the following parameter:

param Boolean verbose If `true`, the method displays details of the document distribution across chunks when you have 20 or more chunks.

See also:

`db.printShardingStatus()` (page 122)

Output Examples The *Sharding Version* (page 193) section displays information on the *config database*:

```
--- Sharding Status ---
  sharding version: {
   "_id" : <num>,
   "minCompatibleVersion" : <num>,
   "currentVersion" : <num>,
   "clusterId" : <ObjectId>
}
```

The *Shards* (page 193) section lists information on the shard(s). For each shard, the section displays the name, host, and the associated tags, if any.

```
shards:
 {  "_id" : <shard name1>,
    "host" : <string>,
    "tags" : [ <string> ... ]
 }
 {  "_id" : <shard name2>,
    "host" : <string>,
    "tags" : [ <string> ... ]
 }
 ...
```

New in version 2.8.0: The *Balancer* (page 193) section lists information about the state of the *balancer*. This provides insight into current balancer operation and can be useful when troubleshooting an unbalanced sharded cluster.

```
balancer:
      Currently enabled:  yes
      Currently running:  yes
          Balancer lock taken at Wed Dec 10 2014 12:00:16 GMT+1100 (AEDT) by
          Pixl.local:27017:1418172757:16807:Balancer:282475249
      Collections with active migrations:
          test.t2 started at Wed Dec 10 2014 11:54:51 GMT+1100 (AEDT)
      Failed balancer rounds in last 5 attempts:  1
      Last reported error:  tag ranges not valid for: test.t2
      Time of Reported error:  Wed Dec 10 2014 12:00:33 GMT+1100 (AEDT)
      Migration Results for the last 24 hours:
          96 : Success
          15 : Failed with error 'ns not found, should be impossible', from
          shard01 to shard02
```

The *Databases* (page 194) section lists information on the database(s). For each database, the section displays the name, whether the database has sharding enabled, and the *primary shard* for the database.

```
databases:
 {  "_id" : <dbname1>,
    "partitioned" : <boolean>,
    "primary" : <string>
 }
 {  "_id" : <dbname2>,
    "partitioned" : <boolean>,
    "primary" : <string>
 }
 ...
```

The *Sharded Collection* (page 194) section provides information on the sharding details for sharded collection(s). For each sharded collection, the section displays the shard key, the number of chunks per shard(s), the distribution of documents across chunks [11], and the tag information, if any, for shard key range(s).

```
<dbname>.<collection>
   shard key: { <shard key> : <1 or hashed> }
   chunks:
      <shard name1> <number of chunks>
      <shard name2> <number of chunks>
      ...
   { <shard key>: <min range1> } -->> { <shard key> : <max range1> } on : <shard name> <last modifie
   { <shard key>: <min range2> } -->> { <shard key> : <max range2> } on : <shard name> <last modifie
   ...
```

[11] The sharded collection section, by default, displays the chunk information if the total number of chunks is less than 20. To display the information when you have 20 or more chunks, call the `sh.status()` (page 191) methods with the `verbose` parameter set to `true`, i.e. `sh.status(true)`.

```
tag: <tag1>  { <shard key> : <min range1> } -->> { <shard key> : <max range1> }
...
```

Output Fields

Sharding Version

`sh.status.sharding-version.`**`_id`**
The `_id` (page 193) is an identifier for the version details.

`sh.status.sharding-version.`**`minCompatibleVersion`**
The `minCompatibleVersion` (page 193) is the minimum compatible version of the config server.

`sh.status.sharding-version.`**`currentVersion`**
The `currentVersion` (page 193) is the current version of the config server.

`sh.status.sharding-version.`**`clusterId`**
The `clusterId` (page 193) is the identification for the sharded cluster.

Shards

`sh.status.shards.`**`_id`**
The `_id` (page 193) displays the name of the shard.

`sh.status.shards.`**`host`**
The `host` (page 193) displays the host location of the shard.

`sh.status.shards.`**`tags`**
The `tags` (page 193) displays all the tags for the shard. The field only displays if the shard has tags.

Balancer New in version 2.8.0: `sh.status()` (page 191) added the `balancer` field.

`sh.status.balancer.`**`currently-enabled`**
`currently-enabled` (page 193) indicates if the *balancer* is currently enabled on the sharded cluster.

`sh.status.balancer.`**`currently-running`**
`currently-running` (page 193) indicates whether the *balancer* is currently running, and therefore currently balancing the cluster.

If the *balancer* is running, `currently-running` (page 193) lists the process that holds the balancer lock, and the date and time that the process obtained the lock.

`sh.status.balancer.`**`collections-with-active-migrations`**
`collections-with-active-migrations` (page 193) lists the names of any collections with active migrations, and specifies when the migration began. If there are no active migrations, this field will not appear in the `sh.status()` (page 191) output.

`sh.status.balancer.`**`failed-balancer-rounds-in-last-5-attempts`**
`failed-balancer-rounds-in-last-5-attempts` (page 193) displays the number of *balancer* rounds that failed, from among the last five attempted rounds. A balancer round will fail when a chunk migration fails.

`sh.status.balancer.`**`last-reported-error`**
`last-reported-error` (page 193) lists the most recent balancer error message. If there have been no errors, this field will not appear in the `sh.status()` (page 191) output.

`sh.status.balancer.`**`time-of-reported-error`**
`time-of-reported-error` (page 193) provides the date and time of the most recently-reported error.

sh.status.balancer.**migration-results-for-the-last-24-hours**
migration-results-for-the-last-24-hours (page 193) displays the number of migrations in the last 24 hours, and the error messages from failed migrations . If there have been no recent migrations, migration-results-for-the-last-24-hours (page 193) displays `No recent migrations`.

migration-results-for-the-last-24-hours (page 193) includes *all* migrations, including those not initiated by the balancer.

Databases

sh.status.databases.**_id**
The _id (page 194) displays the name of the database.

sh.status.databases.**partitioned**
The partitioned (page 194) displays whether the database has sharding enabled. If `true`, the database has sharding enabled.

sh.status.databases.**primary**
The primary (page 194) displays the *primary shard* for the database.

Sharded Collection

sh.status.databases.**shard-key**
The shard-key (page 194) displays the shard key specification document.

sh.status.databases.**chunks**
The chunks (page 194) lists all the shards and the number of chunks that reside on each shard.

sh.status.databases.**chunk-details**
The chunk-details (page 194) lists the details of the chunks [1]:

- The range of shard key values that define the chunk,
- The shard where the chunk resides, and
- The last modified timestamp for the chunk.

sh.status.databases.**tag**
The tag (page 194) lists the details of the tags associated with a range of shard key values.

sh.stopBalancer()

Definition

sh.**stopBalancer**(*timeout*, *interval*)
Disables the balancer in a sharded cluster and waits for balancing to complete.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep each cycle of waiting.

See also:

- sh.enableBalancing() (page 185)
- sh.disableBalancing() (page 185)
- sh.getBalancerHost() (page 186)
- sh.getBalancerState() (page 186)
- sh.isBalancerRunning() (page 187)
- sh.setBalancerState() (page 189)
- sh.startBalancer() (page 191)

- sh.waitForBalancer() (page 195)
- sh.waitForBalancerOff() (page 195)

sh.waitForBalancer()

Definition

sh.**waitForBalancer**(*wait*, *timeout*, *interval*)

Waits for a change in the state of the balancer. sh.waitForBalancer() (page 195) is an internal method, which takes the following arguments:

param Boolean wait Set to true to ensure the balancer is now active. The default is false, which waits until balancing stops and becomes inactive.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep.

sh.waitForBalancerOff()

Definition

sh.**waitForBalancerOff**(*timeout*, *interval*)

Internal method that waits until the balancer is not running.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep.

See also:

- sh.enableBalancing() (page 185)
- sh.disableBalancing() (page 185)
- sh.getBalancerHost() (page 186)
- sh.getBalancerState() (page 186)
- sh.isBalancerRunning() (page 187)
- sh.setBalancerState() (page 189)
- sh.startBalancer() (page 191)
- sh.stopBalancer() (page 194)
- sh.waitForBalancer() (page 195)

sh.waitForDLock()

Definition

sh.**waitForDLock**(*lockname*, *wait*, *timeout*, *interval*)

Waits until the specified distributed lock changes state. sh.waitForDLock() (page 195) is an internal method that takes the following arguments:

param string lockname The name of the distributed lock.

param Boolean wait Set to true to ensure the balancer is now active. Set to false to wait until balancing stops and becomes inactive.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep in each waiting cycle.

sh.waitForPingChange()

Definition

`sh.`**`waitForPingChange`**(*activePings*, *timeout*, *interval*)

`sh.waitForPingChange()` (page 196) waits for a change in ping state of one of the `activepings`, and only returns when the specified ping changes state.

param array activePings An array of active pings from the `mongos` (page 685) collection.

param integer timeout Number of milliseconds to wait for a change in ping state.

param integer interval Number of milliseconds to sleep in each waiting cycle.

2.1.10 Subprocess

Subprocess Methods

Name	Description
`clearRawMongoProgramOutput()` (page 196)	For internal use.
`rawMongoProgramOutput()` (page 196)	For internal use.
`run()`	For internal use.
`runMongoProgram()` (page 197)	For internal use.
`runProgram()` (page 197)	For internal use.
`startMongoProgram()`	For internal use.
`stopMongoProgram()` (page 197)	For internal use.
`stopMongoProgramByPid()` (page 197)	For internal use.
`stopMongod()` (page 197)	For internal use.
`waitMongoProgramOnPort()` (page 197)	For internal use.
`waitProgram()` (page 197)	For internal use.

clearRawMongoProgramOutput()

`clearRawMongoProgramOutput`()

For internal use.

rawMongoProgramOutput()

`rawMongoProgramOutput`()

For internal use.

run()

`run`()

For internal use.

runMongoProgram()

runMongoProgram()
For internal use.

runProgram()

runProgram()
For internal use.

startMongoProgram()

_startMongoProgram()
For internal use.

stopMongoProgram()

stopMongoProgram()
For internal use.

stopMongoProgramByPid()

stopMongoProgramByPid()
For internal use.

stopMongod()

stopMongod()
For internal use.

waitMongoProgramOnPort()

waitMongoProgramOnPort()
For internal use.

waitProgram()

waitProgram()
For internal use.

2.1.11 Constructors

Object Constructors and Methods

Name	Description
BulkWriteResult() (page 198)	Wrapper around the result set from Bulk.execute() (page 137).
Date() (page 199)	Creates a date object. By default creates a date object including the current date.
ObjectId.getTimestamp() (page 199)	Returns the timestamp portion of an *ObjectId*.
ObjectId.toString() (page 200)	Displays the string representation of an *ObjectId*.
ObjectId.valueOf() (page 200)	Displays the str attribute of an ObjectId as a hexadecimal string.
UUID() (page 200)	Converts a 32-byte hexadecimal string to the UUID BSON subtype.
WriteResult() (page 201)	Wrapper around the result set from write methods.
WriteResult.hasWriteConcernError() (page 202)	Returns a boolean specifying whether whether the results include WriteResult.writeConcernError (page 201).
WriteResult.hasWriteError() (page 202)	Returns a boolean specifying whether the results include WriteResult.writeError (page 201).

BulkWriteResult()

BulkWriteResult()
New in version 2.6.

A wrapper that contains the results of the Bulk.execute() (page 137) method.

Properties The BulkWriteResult (page 198) has the following properties:

BulkWriteResult.**nInserted**
The number of documents inserted using the Bulk.insert() (page 148) method. For documents inserted through operations with the Bulk.find.upsert() (page 144) option, see the nUpserted (page 198) field instead.

BulkWriteResult.**nMatched**
The number of existing documents selected for update or replacement. If the update/replacement operation results in no change to an existing document, e.g. $set (page 459) expression updates the value to the current value, nMatched (page 198) can be greater than nModified (page 198).

BulkWriteResult.**nModified**
The number of existing documents updated or replaced. If the update/replacement operation results in no change to an existing document, such as setting the value of the field to its current value, nModified (page 198) can be less than nMatched (page 198). Inserted documents do not affect the number of nModified (page 198); refer to the nInserted (page 198) and nUpserted (page 198) fields instead.

BulkWriteResult.**nRemoved**
The number of documents removed.

BulkWriteResult.**nUpserted**
The number of documents inserted through operations with the Bulk.find.upsert() (page 144) option.

BulkWriteResult.**upserted**
An array of documents that contains information for each document inserted through operations with the Bulk.find.upsert() (page 144) option.

Each document contains the following information:

`BulkWriteResult.upserted.`**`index`**
An integer that identifies the operation in the bulk operations list, which uses a zero-based index.

`BulkWriteResult.upserted.`**`_id`**
The `_id` value of the inserted document.

`BulkWriteResult.`**`writeErrors`**
An array of documents that contains information regarding any error, unrelated to write concerns, encountered during the update operation. The `writeErrors` (page 199) array contains an error document for each write operation that errors.

Each error document contains the following fields:

`BulkWriteResult.writeErrors.`**`index`**
An integer that identifies the write operation in the bulk operations list, which uses a zero-based index. See also `Bulk.getOperations()` (page 147).

`BulkWriteResult.writeErrors.`**`code`**
An integer value identifying the error.

`BulkWriteResult.writeErrors.`**`errmsg`**
A description of the error.

`BulkWriteResult.writeErrors.`**`op`**
A document identifying the operation that failed. For instance, an update/replace operation error will return a document specifying the query, the update, the `multi` and the `upsert` options; an insert operation will return the document the operation tried to insert.

`BulkWriteResult.`**`writeConcernError`**
Document that describe error related to write concern and contains the field:

`BulkWriteResult.writeConcernError.`**`code`**
An integer value identifying the cause of the write concern error.

`BulkWriteResult.writeConcernError.`**`errInfo`**
A document identifying the write concern setting related to the error.

`BulkWriteResult.writeConcernError.`**`errmsg`**
A description of the cause of the write concern error.

Date()

`Date``()`

Returns Current date, as a string.

ObjectId.getTimestamp()

`ObjectId.`**`getTimestamp`**`()`

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

In the following example, call the `getTimestamp()` (page 199) method on an ObjectId (e.g. `ObjectId("507c7f79bcf86cd7994f6c0e")`):

```
ObjectId("507c7f79bcf86cd7994f6c0e").getTimestamp()
```

This will return the following output:

```
ISODate("2012-10-15T21:26:17Z")
```

ObjectId.toString()

`ObjectId.toString()`

Returns The string representation of the *ObjectId()* object. This value has the format of `ObjectId(...)`.

Changed in version 2.2: In previous versions `ObjectId.toString()` (page 200) returns the value of the ObjectId as a hexadecimal string.

In the following example, call the `toString()` (page 200) method on an ObjectId (e.g. `ObjectId("507c7f79bcf86cd7994f6c0e")`):

```
ObjectId("507c7f79bcf86cd7994f6c0e").toString()
```

This will return the following string:

```
ObjectId("507c7f79bcf86cd7994f6c0e")
```

You can confirm the type of this object using the following operation:

```
typeof ObjectId("507c7f79bcf86cd7994f6c0e").toString()
```

ObjectId.valueOf()

`ObjectId.valueOf()`

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

Changed in version 2.2: In previous versions `ObjectId.valueOf()` (page 200) returns the `ObjectId()` object.

In the following example, call the `valueOf()` (page 200) method on an ObjectId (e.g. `ObjectId("507c7f79bcf86cd7994f6c0e")`):

```
ObjectId("507c7f79bcf86cd7994f6c0e").valueOf()
```

This will return the following string:

```
507c7f79bcf86cd7994f6c0e
```

You can confirm the type of this object using the following operation:

```
typeof ObjectId("507c7f79bcf86cd7994f6c0e").valueOf()
```

UUID()

Definition

UUID(<*string*>)

Generates a BSON UUID object.

param string hex Specify a 32-byte hexadecimal string to convert to the UUID BSON subtype.

Returns A BSON UUID object.

Example Create a 32 byte hexadecimal string:

```
var myuuid = '0123456789abcdeffedcba9876543210'
```

Convert it to the BSON UUID subtype:

```
UUID(myuuid)
BinData(3,"ASNFZ4mrze/+3LqYdlQyEA==")
```

WriteResult()

Definition

WriteResult()

A wrapper that contains the result status of the mongo (page 610) shell write methods.

See

db.collection.insert() (page 55), db.collection.update() (page 72), db.collection.remove() (page 66), and db.collection.save() (page 70).

Properties The WriteResult (page 201) has the following properties:

WriteResult.**nInserted**

The number of documents inserted, excluding upserted documents. See WriteResult.nUpserted (page 201) for the number of documents inserted through an upsert.

WriteResult.**nMatched**

The number of documents selected for update. If the update operation results in no change to the document, e.g. $set (page 459) expression updates the value to the current value, nMatched (page 201) can be greater than nModified (page 201).

WriteResult.**nModified**

The number of existing documents updated. If the update/replacement operation results in no change to the document, such as setting the value of the field to its current value, nModified (page 201) can be less than nMatched (page 201).

WriteResult.**nUpserted**

The number of documents inserted by an *upsert* (page 74).

WriteResult.**_id**

The _id of the document inserted by an upsert. Returned only if an upsert results in an insert.

WriteResult.**nRemoved**

The number of documents removed.

WriteResult.**writeError**

A document that contains information regarding any error, excluding write concern errors, encountered during the write operation.

WriteResult.writeError.**code**

An integer value identifying the error.

WriteResult.writeError.**errmsg**

A description of the error.

WriteResult.**writeConcernError**

A document that contains information regarding any write concern errors encountered during the write operation.

`WriteResult.writeConcernError.`**`code`**
An integer value identifying the write concern error.

`WriteResult.writeConcernError.`**`errInfo`**
A document identifying the write concern setting related to the error.

`WriteResult.writeError.`**`errmsg`**
A description of the error.

See also:

`WriteResult.hasWriteError()` (page 202), `WriteResult.hasWriteConcernError()` (page 202)

WriteResult.hasWriteConcernError()

Definition

`WriteResult.`**`hasWriteConcernError`**`()`
Returns `true` if the result of a `mongo` (page 610) shell write method has `WriteResult.writeConcernError` (page 201). Otherwise, the method returns `false`.

See also:

`WriteResult()` (page 201)

WriteResult.hasWriteError()

Definition

`WriteResult.`**`hasWriteError`**`()`
Returns `true` if the result of a `mongo` (page 610) shell write method has `WriteResult.writeError` (page 201). Otherwise, the method returns `false`.

See also:

`WriteResult()` (page 201)

2.1.12 Connection

Connection Methods

Name	Description
`Mongo()` (page 202)	Creates a new connection object.
`Mongo.getDB()` (page 203)	Returns a database object.
`Mongo.getReadPrefMode()` (page 203)	Returns the current read preference mode for the MongoDB connection.
`Mongo.getReadPrefTagSet()` (page 204)	Returns the read preference tag set for the MongoDB connection.
`Mongo.setReadPref()` (page 204)	Sets the *read preference* for the MongoDB connection.
`Mongo.setSlaveOk()` (page 205)	Allows operations on the current connection to read from *secondary* members.
`connect()`	Connects to a MongoDB instance and to a specified database on that instance.

Mongo()

Description

Mongo (*host*)

JavaScript constructor to instantiate a database connection from the mongo (page 610) shell or from a JavaScript file.

The Mongo() (page 202) method has the following parameter:

param string host The host, either in the form of <host> or <host><:port>.

Instantiation Options Use the constructor without a parameter to instantiate a connection to the localhost interface on the default port.

Pass the <host> parameter to the constructor to instantiate a connection to the <host> and the default port.

Pass the <host><:port> parameter to the constructor to instantiate a connection to the <host> and the <port>.

See also:

Mongo.getDB() (page 203) and db.getMongo() (page 116).

Mongo.getDB()

Description

Mongo.**getDB**(*<database>*)

Provides access to database objects from the mongo (page 610) shell or from a JavaScript file.

The Mongo.getDB() (page 203) method has the following parameter:

param string database The name of the database to access.

Example The following example instantiates a new connection to the MongoDB instance running on the localhost interface and returns a reference to "myDatabase":

```
db = new Mongo().getDB("myDatabase");
```

See also:

Mongo() (page 202) and *connect()* (page 205)

Mongo.getReadPrefMode()

Mongo.**getReadPrefMode**()

Returns The current *read preference* mode for the Mongo() (page 116) connection object.

See http://docs.mongodb.org/manual/core/read-preference for an introduction to read preferences in MongoDB. Use getReadPrefMode() (page 203) to return the current read preference mode, as in the following example:

```
db.getMongo().getReadPrefMode()
```

Use the following operation to return and print the current read preference mode:

```
print(db.getMongo().getReadPrefMode());
```

This operation will return one of the following read preference modes:

- primary
- primaryPreferred

- secondary
- secondaryPreferred
- nearest

See also:

http://docs.mongodb.org/manual/core/read-preference, readPref() (page 93), setReadPref() (page 204), and getReadPrefTagSet() (page 204).

Mongo.getReadPrefTagSet()

Mongo.**getReadPrefTagSet**()

Returns The current *read preference* tag set for the Mongo() (page 116) connection object.

See http://docs.mongodb.org/manual/core/read-preference for an introduction to read preferences and tag sets in MongoDB. Use getReadPrefTagSet() (page 204) to return the current read preference tag set, as in the following example:

```
db.getMongo().getReadPrefTagSet()
```

Use the following operation to return and print the current read preference tag set:

```
printjson(db.getMongo().getReadPrefTagSet());
```

See also:

http://docs.mongodb.org/manual/core/read-preference, readPref() (page 93), setReadPref() (page 204), and getReadPrefTagSet() (page 204).

Mongo.setReadPref()

Definition

Mongo.**setReadPref**(*mode*, *tagSet*)

Call the setReadPref() (page 204) method on a Mongo (page 116) connection object to control how the client will route all queries to members of the replica set.

param string mode One of the following *read preference* modes: primary, primaryPreferred, secondary, secondaryPreferred, or nearest.

param array tagSet A tag set used to specify custom read preference modes. For details, see *replica-set-read-preference-tag-sets*.

Examples To set a read preference mode in the mongo (page 610) shell, use the following operation:

```
db.getMongo().setReadPref('primaryPreferred')
```

To set a read preference that uses a tag set, specify an array of tag sets as the second argument to Mongo.setReadPref() (page 204), as in the following:

```
db.getMongo().setReadPref('primaryPreferred', [ { "dc": "east" } ] )
```

You can specify multiple tag sets, in order of preference, as in the following:

```
db.getMongo().setReadPref('secondaryPreferred',
                          [ { "dc": "east", "use": "production" },
                            { "dc": "east", "use": "reporting" },
                            { "dc": "east" },
                            {}
                          ] )
```

If the replica set cannot satisfy the first tag set, the client will attempt to use the second read preference. Each tag set can contain zero or more field/value tag pairs, with an "empty" document acting as a wildcard which matches a replica set member with any tag set or no tag set.

Note: You must call `Mongo.setReadPref()` (page 204) on the connection object before retrieving documents using that connection to use that read preference.

---

mongo.setSlaveOk()

`Mongo.setSlaveOk()`

For the current session, this command permits read operations from non-master (i.e. *slave* or *secondary*) instances. Practically, use this method in the following form:

```
db.getMongo().setSlaveOk()
```

Indicates that "*eventually consistent*" read operations are acceptable for the current application. This function provides the same functionality as `rs.slaveOk()` (page 178).

See the `readPref()` (page 93) method for more fine-grained control over `read preference` in the `mongo` (page 610) shell.

connect()

`connect``(<hostname><:port>/<database>)`

The `connect()` method creates a connection to a MongoDB instance. However, use the `Mongo()` (page 202) object and its `getDB()` (page 203) method in most cases.

`connect()` accepts a string `<hostname><:port>/<database>` parameter to connect to the MongoDB instance on the `<hostname><:port>` and return the reference to the database `<database>`.

The following example instantiates a new connection to the MongoDB instance running on the localhost interface and returns a reference to `myDatabase`:

```
db = connect("localhost:27017/myDatabase")
```

See also:

`Mongo.getDB()` (page 203)

2.1.13 Native

Native Methods

Name	Description
_isWindows() (page 206)	Returns true if the shell runs on a Windows system; false if a Unix or Linux system.
_rand() (page 206)	Returns a random number between 0 and 1.
_srand() (page 207)	For internal use.
cat()	Returns the contents of the specified file.
cd()	Changes the current working directory to the specified path.
copyDbpath() (page 207)	Copies a local dbPath. For internal use.
fuzzFile() (page 207)	For internal use to support testing.
getHostName() (page 207)	Returns the hostname of the system running the mongo (page 610) shell.
getMemInfo() (page 207)	Returns a document that reports the amount of memory used by the shell.
hostname()	Returns the hostname of the system running the shell.
listFiles() (page 208)	Returns an array of documents that give the name and size of each object in the directory.
load()	Loads and runs a JavaScript file in the shell.
ls()	Returns a list of the files in the current directory.
md5sumFile() (page 208)	The *md5* hash of the specified file.
mkdir()	Creates a directory at the specified path.
pwd()	Returns the current directory.
quit()	Exits the current shell session.
removeFile() (page 209)	Removes the specified file from the local file system.
resetDbpath() (page 209)	Removes a local dbPath. For internal use.
version()	Returns the current version of the mongo (page 610) shell instance.

_isWindows()

_isWindows()

Returns boolean.

Returns "true" if the mongo (page 610) shell is running on a system that is Windows, or "false" if the shell is running on a Unix or Linux systems.

rand()

_rand()

Returns A random number between 0 and 1.

This function provides functionality similar to the Math.rand() function from the standard library.

_srand()

_srand()
For internal use.

cat()

Definition
cat(*filename*)
Returns the contents of the specified file. The method returns with output relative to the current shell session and does not impact the server.

param string filename Specify a path and file name on the local file system.

cd()

Definition
cd(*path*)

param string path A path on the file system local to the mongo (page 610) shell context.

cd() changes the directory context of the mongo (page 610) shell and has no effect on the MongoDB server.

copyDbpath()

copyDbpath()
For internal use.

fuzzFile()

Description
fuzzFile(*filename*)
For internal use.

param string filename A filename or path to a local file.

getHostName()

getHostName()

Returns The hostname of the system running the mongo (page 610) shell process.

getMemInfo()

getMemInfo()
Returns a document with two fields that report the amount of memory used by the JavaScript shell process. The fields returned are *resident* and *virtual*.

hostname()

hostname()

Returns The hostname of the system running the mongo (page 610) shell process.

listFiles()

listFiles()

Returns an array, containing one document per object in the directory. This function operates in the context of the mongo (page 610) process. The included fields are:

name
Returns a string which contains the name of the object.

isDirectory
Returns true or false if the object is a directory.

size
Returns the size of the object in bytes. This field is only present for files.

load()

Definition

load(*file*)

Loads and runs a JavaScript file into the current shell environment.

The load() method has the following parameter:

param string filename Specifies the path of a JavaScript file to execute.

Specify filenames with relative or absolute paths. When using relative path names, confirm the current directory using the pwd() method.

After executing a file with load(), you may reference any functions or variables defined the file from the mongo (page 610) shell environment.

Example Consider the following examples of the load() method:

```
load("scripts/myjstest.js")
load("/data/db/scripts/myjstest.js")
```

ls()

ls()

Returns a list of the files in the current directory.

This function returns with output relative to the current shell session, and does not impact the server.

md5sumFile()

Description

md5sumFile(*filename*)

Returns a *md5* hash of the specified file.

The `md5sumFile()` (page 208) method has the following parameter:

param string filename A file name.

Note: The specified filename must refer to a file located on the system running the `mongo` (page 610) shell.

mkdir()

Description

mkdir(*path*)

Creates a directory at the specified path. This method creates the entire path specified if the enclosing directory or directories do not already exit.

This method is equivalent to **mkdir -p** with BSD or GNU utilities.

The `mkdir()` method has the following parameter:

param string path A path on the local filesystem.

pwd()

pwd()

Returns the current directory.

This function returns with output relative to the current shell session, and does not impact the server.

quit()

quit()

Exits the current shell session.

removeFile()

Description

removeFile(*filename*)

Removes the specified file from the local file system.

The `removeFile()` (page 209) method has the following parameter:

param string filename A filename or path to a local file.

resetDbpath()

resetDbpath()

For internal use.

version()

version()

Returns The version of the mongo (page 610) shell as a string.

Changed in version 2.4: In previous versions of the shell, version() would print the version instead of returning a string.

2.2 Database Commands

All command documentation outlined below describes a command and its available parameters and provides a document template or prototype for each command. Some command documentation also includes the relevant mongo (page 610) shell helpers.

2.2.1 User Commands

Aggregation Commands

Aggregation Commands

Name	Description
aggregate (page 210)	Performs aggregation tasks such as group using the aggregation framework.
count (page 213)	Counts the number of documents in a collection.
distinct (page 215)	Displays the distinct values found for a specified key in a collection.
group (page 216)	Groups documents in a collection by the specified key and performs simple aggregation.
mapReduce (page 220)	Performs map-reduce aggregation for large data sets.

aggregate

aggregate

New in version 2.2.

Performs aggregation operation using the *aggregation pipeline* (page 484). The pipeline allows users to process data from a collection with a sequence of stage-based manipulations.

Changed in version 2.6.

- The aggregate (page 210) command adds support for returning a cursor, supports the explain option, and enhances its sort operations with an external sort facility.
- *aggregation pipeline* (page 484) introduces the $out (page 491) operator to allow aggregate (page 210) command to store results to a collection.

The command has following syntax:

Changed in version 2.6.

```
{
  aggregate: "<collection>",
  pipeline: [ <stage>, <...> ],
  explain: <boolean>,
```

```
  allowDiskUse: <boolean>,
  cursor: <document>
}
```

The aggregate (page 210) command takes the following fields as arguments:

field string aggregate The name of the collection to as the input for the aggregation pipeline.

field array pipeline An array of *aggregation pipeline stages* (page 484) that process and transform the document stream as part of the aggregation pipeline.

field boolean explain Specifies to return the information on the processing of the pipeline.

New in version 2.6.

field boolean allowDiskUse Enables writing to temporary files. When set to `true`, aggregation stages can write data to the `_tmp` subdirectory in the `dbPath` directory.

New in version 2.6.

field document cursor Specify a document that contains options that control the creation of the cursor object.

New in version 2.6.

For more information about the aggregation pipeline `http://docs.mongodb.org/manual/core/aggregation-pipeline` *Aggregation Reference* (page 564), and `http://docs.mongodb.org/manual/core/aggregation-pipeline-limits`.

Example

Aggregate Data with Multi-Stage Pipeline A collection `articles` contains documents such as the following:

```
{
   _id: ObjectId("52769ea0f3dc6ead47c9a1b2"),
   author: "abc123",
   title: "zzz",
   tags: [ "programming", "database", "mongodb" ]
}
```

The following example performs an aggregate (page 210) operation on the `articles` collection to calculate the count of each distinct element in the `tags` array that appears in the collection.

```
db.runCommand(
   { aggregate: "articles",
     pipeline: [
                 { $project: { tags: 1 } },
                 { $unwind: "$tags" },
                 { $group: {
                             _id: "$tags",
                             count: { $sum : 1 }
                           }
                 }
               ]
   }
)
```

In the mongo (page 610) shell, this operation can use the `aggregate()` (page 22) helper as in the following:

```
db.articles.aggregate(
                       [
                          { $project: { tags: 1 } },
```

```
                        { $unwind: "$tags" },
                        { $group: {
                                    _id: "$tags",
                                    count: { $sum : 1 }
                                  }
                        }
                      ]
)
```

Note: In 2.6 and later, the `aggregate()` (page 22) helper always returns a cursor.

Changed in version 2.4: If an error occurs, the `aggregate()` (page 22) helper throws an exception. In previous versions, the helper returned a document with the error message and code, and `ok` status field not equal to `1`, same as the `aggregate` (page 210) command.

Return Information on the Aggregation Operation The following aggregation operation sets the optional field `explain` to `true` to return information about the aggregation operation.

```
db.runCommand( { aggregate: "orders",
                 pipeline: [
                             { $match: { status: "A" } },
                             { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
                             { $sort: { total: -1 } }
                           ],
                 explain: true
               } )
```

Note: The intended readers of the `explain` output document are humans, and not machines, and the output format is subject to change between releases.

See also:

`db.collection.aggregate()` (page 22) method

Aggregate Data using External Sort Aggregation pipeline stages have *maximum memory use limit*. To handle large datasets, set `allowDiskUse` option to `true` to enable writing data to temporary files, as in the following example:

```
db.runCommand(
   { aggregate: "stocks",
     pipeline: [
                 { $project : { cusip: 1, date: 1, price: 1, _id: 0 } },
                 { $sort : { cusip : 1, date: 1 } }
               ],
     allowDiskUse: true
   }
)
```

See also:

`db.collection.aggregate()` (page 22)

Aggregate Command Returns a Cursor

Note: Using the `aggregate` (page 210) command to return a cursor is a low-level operation, intended for authors of drivers. Most users should use the `db.collection.aggregate()` (page 22) helper provided in the `mongo` (page 610) shell or in their driver. In 2.6 and later, the `aggregate()` (page 22) helper always returns a cursor.

The following command returns a document that contains results with which to instantiate a cursor object.

```
db.runCommand(
   { aggregate: "records",
     pipeline: [
        { $project: { name: 1, email: 1, _id: 0 } },
        { $sort: { name: 1 } }
     ],
     cursor: { }
   }
)
```

To specify an *initial* batch size, specify the `batchSize` in the `cursor` field, as in the following example:

```
db.runCommand(
   { aggregate: "records",
     pipeline: [
        { $project: { name: 1, email: 1, _id: 0 } },
        { $sort: { name: 1 } }
     ],
     cursor: { batchSize: 0 }
   }
)
```

The `{batchSize: 0 }` document specifies the size of the *initial* batch size only. Specify subsequent batch sizes to *OP_GET_MORE*[12] operations as with other MongoDB cursors. A `batchSize` of `0` means an empty first batch and is useful if you want to quickly get back a cursor or failure message, without doing significant server-side work.

See also:

`db.collection.aggregate()` (page 22)

count

Definition

`count`

Counts the number of documents in a collection. Returns a document that contains this count and as well as the command status. `count` (page 213) has the following form:

Changed in version 2.6: `count` (page 213) now accepts the `hint` option to specify an index.

```
{ count: <collection>, query: <query>, limit: <limit>, skip: <skip>, hint: <hint> }
```

`count` (page 213) has the following fields:

field string count The name of the collection to count.

field document query A query that selects which documents to count in a collection.

field integer limit The maximum number of matching documents to return.

field integer skip The number of matching documents to skip before returning results.

field String,document hint The index to use. Specify either the index name as a string or the index specification document.

New in version 2.6.

[12] http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol/#wire-op-get-more

MongoDB also provides the count() (page 83) and db.collection.count() (page 26) wrapper methods in the mongo (page 610) shell.

Behavior On a sharded cluster, count (page 213) can result in an *inaccurate* count if *orphaned documents* exist or if a chunk migration is in progress.

To avoid these situations, on a sharded cluster, use the $group (page 486) stage of the db.collection.aggregate() (page 22) method to $sum (page 554) the documents. For example, the following operation counts the documents in a collection:

```
db.collection.aggregate(
   [
      { $group: { _id: null, count: { $sum: 1 } } }
   ]
)
```

To get a count of documents that match a query condition, include the $match (page 490) stage as well:

```
db.collection.aggregate(
   [
      { $match: <query condition> },
      { $group: { _id: null, count: { $sum: 1 } } }
   ]
)
```

See *Perform a Count* (page 491) for an example.

Examples The following sections provide examples of the count (page 213) command.

Count All Documents The following operation counts the number of all documents in the orders collection:

```
db.runCommand( { count: 'orders' } )
```

In the result, the n, which represents the count, is 26, and the command status ok is 1:

```
{ "n" : 26, "ok" : 1 }
```

Count Documents That Match a Query The following operation returns a count of the documents in the orders collection where the value of the ord_dt field is greater than Date('01/01/2012'):

```
db.runCommand( { count:'orders',
                 query: { ord_dt: { $gt: new Date('01/01/2012') } }
               } )
```

In the result, the n, which represents the count, is 13 and the command status ok is 1:

```
{ "n" : 13, "ok" : 1 }
```

Skip Documents in Count The following operation returns a count of the documents in the orders collection where the value of the ord_dt field is greater than Date('01/01/2012') and skip the first 10 matching documents:

```
db.runCommand( { count:'orders',
                 query: { ord_dt: { $gt: new Date('01/01/2012') } },
                 skip: 10 }  )
```

In the result, the `n`, which represents the count, is `3` and the command status `ok` is `1`:

```
{ "n" : 3, "ok" : 1 }
```

Specify the Index to Use The following operation uses the index `{ status: 1 }` to return a count of the documents in the `orders` collection where the value of the `ord_dt` field is greater than `Date('01/01/2012')` and the `status` field is equal to `"D"`:

```
db.runCommand(
   {
     count:'orders',
     query: {
               ord_dt: { $gt: new Date('01/01/2012') },
               status: "D"
             },
     hint: { status: 1 }
   }
)
```

In the result, the `n`, which represents the count, is `1` and the command status `ok` is `1`:

```
{ "n" : 1, "ok" : 1 }
```

distinct

Definition

`distinct`

Finds the distinct values for a specified field across a single collection. `distinct` (page 215) returns a document that contains an array of the distinct values. The return document also contains a subdocument with query statistics and the query plan.

When possible, the `distinct` (page 215) command uses an index to find documents and return values.

The command takes the following form:

```
{ distinct: "<collection>", key: "<field>", query: <query> }
```

The command contains the following fields:

field string distinct The name of the collection to query for distinct values.

field string key The field to collect distinct values from.

field document query A query specification to limit the input documents in the *distinct* analysis.

Examples Return an array of the distinct values of the field `ord_dt` from all documents in the `orders` collection:

```
db.runCommand ( { distinct: "orders", key: "ord_dt" } )
```

Return an array of the distinct values of the field `sku` in the subdocument `item` from all documents in the `orders` collection:

```
db.runCommand ( { distinct: "orders", key: "item.sku" } )
```

Return an array of the distinct values of the field `ord_dt` from the documents in the `orders` collection where the `price` is greater than `10`:

```
db.runCommand ( { distinct: "orders",
                  key: "ord_dt",
                  query: { price: { $gt: 10 } }
                } )
```

Note: MongoDB also provides the shell wrapper method `db.collection.distinct()` (page 28) for the `distinct` (page 215) command. Additionally, many MongoDB *drivers* also provide a wrapper method. Refer to the specific driver documentation.

group

Definition

group

Groups documents in a collection by the specified key and performs simple aggregation functions, such as computing counts and sums. The command is analogous to a `SELECT <...> GROUP BY` statement in SQL. The command returns a document with the grouped records as well as the command meta-data.

The `group` (page 216) command takes the following prototype form:

```
{
  group:
   {
     ns: <namespace>,
     key: <key>,
     $reduce: <reduce function>,
     $keyf: <key function>,
     cond: <query>,
     finalize: <finalize function>
   }
}
```

The command accepts a document with the following fields:

field string ns The collection from which to perform the group by operation.

field document key The field or fields to group. Returns a "key object" for use as the grouping key.

field function $reduce An aggregation function that operates on the documents during the grouping operation. These functions may return a sum or a count. The function takes two arguments: the current document and an aggregation result document for that group.

field document initial Initializes the aggregation result document.

field function $keyf Alternative to the `key` field. Specifies a function that creates a "key object" for use as the grouping key. Use `$keyf` instead of `key` to group by calculated fields rather than existing document fields.

field document cond The selection criteria to determine which documents in the collection to process. If you omit the `cond` field, `group` (page 216) processes all the documents in the collection for the group operation.

field function finalize A function that runs each item in the result set before `group` (page 216) returns the final value. This function can either modify the result document or replace the result document as a whole. Unlike the `$keyf` and `$reduce` fields that also specify a function, this field name is `finalize`, *not* `$finalize`.

For the shell, MongoDB provides a wrapper method `db.collection.group()` (page 51). However, the `db.collection.group()` (page 51) method takes the `keyf` field and the `reduce` field whereas the `group` (page 216) command takes the `$keyf` field and the `$reduce` field.

Behavior

Limits and Restrictions The `group` (page 216) command does not work with *sharded clusters*. Use the *aggregation framework* or *map-reduce* in *sharded environments*.

The result set must fit within the *maximum BSON document size* (page 692).

Additionally, in version 2.2, the returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. For group by operations that results in more than 20,000 unique groupings, use `mapReduce` (page 220). Previous versions had a limit of 10,000 elements.

Prior to 2.4, the `group` (page 216) command took the `mongod` (page 583) instance's JavaScript lock which blocked all other JavaScript execution.

`mongo` Shell JavaScript Functions/Properties Changed in version 2.4.

In MongoDB 2.4, `map-reduce operations` (page 220), the `group` (page 216) command, and `$where` (page 421) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 610) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 220), `group` (page 216) commands, or `$where` (page 421) operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations` (page 220), the `group` (page 216) command, and `$where` (page 421) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	

`args`
`MaxKey`
`MinKey`
	`assert()`
	`BinData()`
	`DBPointer()`
	`DBRef()`
	`doassert()`
	`emit()`
	`gc()`
	`HexData()`
	`hex_md5()`
	`isNumber()`
	`isObject()`
	`ISODate()`
	`isString()`
		`Map()`
		`MD5()`
		`NumberInt()`
		`NumberLong()`
		`ObjectId()`
		`print()`
		`printjson()`
		`printjsononeline()`
		`sleep()`
		`Timestamp()`
		`tojson()`
		`tojsononeline()`
		`tojsonObject()`
		`UUID()`
		`version()`

JavaScript in MongoDB

Although `group` (page 216) uses JavaScript, most interactions with MongoDB do not use JavaScript but use an `idiomatic driver` in the language of the interacting application.

Examples The following are examples of the `db.collection.group()` (page 51) method. The examples assume an `orders` collection with documents of the following prototype:

```
{
  _id: ObjectId("5085a95c8fada716c89d0021"),
  ord_dt: ISODate("2012-07-01T04:00:00Z"),
  ship_dt: ISODate("2012-07-02T04:00:00Z"),
  item:
    {
      sku: "abc123",
      price: 1.99,
      uom: "pcs",
      qty: 25
    }
}
```

Group by Two Fields The following example groups by the `ord_dt` and `item.sku` fields those documents that have `ord_dt` greater than `01/01/2012`:

```
db.runCommand(
   {
     group:
       {
         ns: 'orders',
         key: { ord_dt: 1, 'item.sku': 1 },
         cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
         $reduce: function ( curr, result ) { },
         initial: { }
       }
   }
)
```

The result is a document that contain the `retval` field which contains the group by records, the `count` field which contains the total number of documents grouped, the `keys` field which contains the number of unique groupings (i.e. number of elements in the `retval`), and the `ok` field which contains the command status:

```
{ "retval" :
      [ { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc123"},
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc456"},
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "bcd123"},
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "efg456"},
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "abc123"},
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "efg456"},
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "ijk123"},
        { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc123"},
        { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc456"},
        { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc123"},
        { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc456"}
      ],
  "count" : 13,
  "keys" : 11,
  "ok" : 1 }
```

The method call is analogous to the SQL statement:

```
SELECT ord_dt, item_sku
FROM orders
WHERE ord_dt > '01/01/2012'
GROUP BY ord_dt, item_sku
```

Calculate the Sum The following example groups by the `ord_dt` and `item.sku` fields those documents that have `ord_dt` greater than `01/01/2012` and calculates the sum of the `qty` field for each grouping:

```
db.runCommand(
   { group:
       {
         ns: 'orders',
         key: { ord_dt: 1, 'item.sku': 1 },
         cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
         $reduce: function ( curr, result ) {
                     result.total += curr.item.qty;
                   },
         initial: { total : 0 }
       }
    }
)
```

The `retval` field of the returned document is an array of documents that contain the group by fields and the calculated aggregation field:

```
{ "retval" :
      [ { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc456", "total" : 25 },
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "bcd123", "total" : 10 },
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "efg456", "total" : 10 },
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "efg456", "total" : 15 },
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "ijk123", "total" : 20 },
        { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc123", "total" : 45 },
        { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc456", "total" : 25 },
        { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
        { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc456", "total" : 25 }
      ],
  "count" : 13,
  "keys" : 11,
  "ok" : 1 }
```

The method call is analogous to the SQL statement:

```
SELECT ord_dt, item_sku, SUM(item_qty) as total
FROM orders
WHERE ord_dt > '01/01/2012'
GROUP BY ord_dt, item_sku
```

Calculate Sum, Count, and Average The following example groups by the calculated `day_of_week` field, those documents that have `ord_dt` greater than `01/01/2012` and calculates the sum, count, and average of the `qty` field for each grouping:

```
db.runCommand(
   {
     group:
       {
```

```
        ns: 'orders',
        $keyf: function(doc) {
                    return { day_of_week: doc.ord_dt.getDay() };
                },
        cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
        $reduce: function( curr, result ) {
                    result.total += curr.item.qty;
                    result.count++;
                 },
        initial: { total : 0, count: 0 },
        finalize: function(result) {
                    var weekdays = [
                          "Sunday", "Monday", "Tuesday",
                          "Wednesday", "Thursday",
                          "Friday", "Saturday"
                        ];
                    result.day_of_week = weekdays[result.day_of_week];
                    result.avg = Math.round(result.total / result.count);
                  }
      }
   }
)
```

The `retval` field of the returned document is an array of documents that contain the group by fields and the calculated aggregation field:

```
{
  "retval" :
      [
        { "day_of_week" : "Sunday", "total" : 70, "count" : 4, "avg" : 18 },
        { "day_of_week" : "Friday", "total" : 110, "count" : 6, "avg" : 18 },
        { "day_of_week" : "Tuesday", "total" : 70, "count" : 3, "avg" : 23 }
      ],
  "count" : 13,
  "keys" : 3,
  "ok" : 1
}
```

See also:

`http://docs.mongodb.org/manual/core/aggregation`

mapReduce

mapReduce

The `mapReduce` (page 220) command allows you to run *map-reduce* aggregation operations over a collection.

The `mapReduce` (page 220) command has the following prototype form:

```
db.runCommand(
                {
                  mapReduce: <collection>,
                  map: <function>,
                  reduce: <function>,
                  finalize: <function>,
                  out: <output>,
                  query: <document>,
                  sort: <document>,
                  limit: <number>,
                  scope: <document>,
```

```
    jsMode: <boolean>,
    verbose: <boolean>
  }
)
```

Pass the name of the collection to the `mapReduce` command (i.e. `<collection>`) to use as the source documents to perform the map reduce operation. The command also accepts the following parameters:

field collection mapReduce The name of the collection on which you want to perform map-reduce. This collection will be filtered using `query` before being processed by the `map` function.

field Javascript function map A JavaScript function that associates or "maps" a `value` with a `key` and emits the `key` and value `pair`.

See *Requirements for the map Function* (page 223) for more information.

field JavaScript function reduce A JavaScript function that "reduces" to a single object all the `values` associated with a particular `key`.

See *Requirements for the reduce Function* (page 223) for more information.

field string or document out Specifies where to output the result of the map-reduce operation. You can either output to a collection or return the result inline. On a *primary* member of a replica set you can output either to a collection or inline, but on a *secondary*, only inline output is possible.

See *out Options* (page 224) for more information.

field document query Specifies the selection criteria using *query operators* (page 400) for determining the documents input to the `map` function.

field document sort Sorts the *input* documents. This option is useful for optimization. For example, specify the sort key to be the same as the emit key so that there are fewer reduce operations. The sort key must be in an existing index for this collection.

field number limit Specifies a maximum number of documents for the input into the `map` function.

field Javascript function finalize Follows the `reduce` method and modifies the output.

See *Requirements for the finalize Function* (page 224) for more information.

field document scope Specifies global variables that are accessible in the `map`, `reduce` and `finalize` functions.

field Boolean jsMode Specifies whether to convert intermediate data into BSON format between the execution of the `map` and `reduce` functions. Defaults to `false`.

If `false`:

- Internally, MongoDB converts the JavaScript objects emitted by the `map` function to BSON objects. These BSON objects are then converted back to JavaScript objects when calling the `reduce` function.
- The map-reduce operation places the intermediate BSON objects in temporary, on-disk storage. This allows the map-reduce operation to execute over arbitrarily large data sets.

If `true`:

- Internally, the JavaScript objects emitted during `map` function remain as JavaScript objects. There is no need to convert the objects for the `reduce` function, which can result in faster execution.
- You can only use `jsMode` for result sets with fewer than 500,000 distinct `key` arguments to the mapper's `emit()` function.

The `jsMode` defaults to false.

field Boolean verbose Specifies whether to include the `timing` information in the result information. The `verbose` defaults to `true` to include the `timing` information.

The following is a prototype usage of the `mapReduce` (page 220) command:

```
var mapFunction = function() { ... };
var reduceFunction = function(key, values) { ... };

db.runCommand(
               {
                 mapReduce: <input-collection>,
                 map: mapFunction,
                 reduce: reduceFunction,
                 out: { merge: <output-collection> },
                 query: <query>
               }
             )
```

JavaScript in MongoDB

Although `mapReduce` (page 220) uses JavaScript, most interactions with MongoDB do not use JavaScript but use an `idiomatic driver` in the language of the interacting application.

Note: Changed in version 2.4.

In MongoDB 2.4, `map-reduce operations` (page 220), the `group` (page 216) command, and `$where` (page 421) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 610) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 220), `group` (page 216) commands, or `$where` (page 421) operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations` (page 220), the `group` (page 216) command, and `$where` (page 421) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	
args MaxKey MinKey	assert() BinData() DBPointer() DBRef() doassert() emit() gc() HexData() hex_md5() isNumber() isObject() ISODate() isString()	Map() MD5() NumberInt() NumberLong() ObjectId() print() printjson() printjsononeline() sleep() Timestamp() tojson() tojsononeline() tojsonObject() UUID() version()

Requirements for the `map` Function The `map` function is responsible for transforming each input document into zero or more documents. It can access the variables defined in the `scope` parameter, and has the following prototype:

```
function() {
   ...
   emit(key, value);
}
```

The `map` function has the following requirements:

- In the `map` function, reference the current document as `this` within the function.
- The `map` function should *not* access the database for any reason.
- The `map` function should be pure, or have *no* impact outside of the function (i.e. side effects.)
- A single emit can only hold half of MongoDB's *maximum BSON document size* (page 692).
- The `map` function may optionally call `emit(key,value)` any number of times to create an output document associating `key` with `value`.

The following `map` function will call `emit(key,value)` either 0 or 1 times depending on the value of the input document's `status` field:

```
function() {
    if (this.status == 'A')
        emit(this.cust_id, 1);
}
```

The following `map` function may call `emit(key,value)` multiple times depending on the number of elements in the input document's `items` field:

```
function() {
    this.items.forEach(function(item){ emit(item.sku, 1); });
}
```

Requirements for the `reduce` Function The `reduce` function has the following prototype:

```
function(key, values) {
   ...
   return result;
}
```

The `reduce` function exhibits the following behaviors:

- The `reduce` function should *not* access the database, even to perform read operations.
- The `reduce` function should *not* affect the outside system.
- MongoDB will **not** call the `reduce` function for a key that has only a single value. The `values` argument is an array whose elements are the `value` objects that are "mapped" to the `key`.
- MongoDB can invoke the `reduce` function more than once for the same key. In this case, the previous output from the `reduce` function for that key will become one of the input values to the next `reduce` function invocation for that key.
- The `reduce` function can access the variables defined in the `scope` parameter.

- The inputs to `reduce` must not be larger than half of MongoDB's *maximum BSON document size* (page 692). This requirement may be violated when large documents are returned and then joined together in subsequent `reduce` steps.

Because it is possible to invoke the `reduce` function more than once for the same key, the following properties need to be true:

- the *type* of the return object must be **identical** to the type of the `value` emitted by the `map` function.
- the `reduce` function must be *associative*. The following statement must be true:

  ```
  reduce(key, [ C, reduce(key, [ A, B ]) ] ) == reduce( key, [ C, A, B ] )
  ```

- the `reduce` function must be *idempotent*. Ensure that the following statement is true:

  ```
  reduce( key, [ reduce(key, valuesArray) ] ) == reduce( key, valuesArray )
  ```

- the `reduce` function should be *commutative*: that is, the order of the elements in the `valuesArray` should not affect the output of the `reduce` function, so that the following statement is true:

  ```
  reduce( key, [ A, B ] ) == reduce( key, [ B, A ] )
  ```

Requirements for the `finalize` Function The `finalize` function has the following prototype:

```
function(key, reducedValue) {
   ...
   return modifiedObject;
}
```

The `finalize` function receives as its arguments a `key` value and the `reducedValue` from the `reduce` function. Be aware that:

- The `finalize` function should *not* access the database for any reason.
- The `finalize` function should be pure, or have *no* impact outside of the function (i.e. side effects.)
- The `finalize` function can access the variables defined in the `scope` parameter.

`out` Options You can specify the following options for the `out` parameter:

Output to a Collection This option outputs to a new collection, and is not available on secondary members of replica sets.

```
out: <collectionName>
```

Output to a Collection with an Action This option is only available when passing a collection that already exists to `out`. It is not available on secondary members of replica sets.

```
out: { <action>: <collectionName>
        [, db: <dbName>]
        [, sharded: <boolean> ]
        [, nonAtomic: <boolean> ] }
```

When you output to a collection with an action, the `out` has the following parameters:

- `<action>`: Specify one of the following actions:

  - replace

    Replace the contents of the <collectionName> if the collection with the <collectionName> exists.

  - merge

    Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, *overwrite* that existing document.

  - reduce

    Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, apply the reduce function to both the new and the existing documents and overwrite the existing document with the result.

- db:

  Optional. The name of the database that you want the map-reduce operation to write its output. By default this will be the same database as the input collection.

- sharded:

  Optional. If true *and* you have enabled sharding on output database, the map-reduce operation will shard the output collection using the _id field as the shard key.

- nonAtomic:

  New in version 2.2.

  Optional. Specify output operation as non-atomic. This applies **only** to the merge and reduce output modes, which may take minutes to execute.

  By default nonAtomic is false, and the map-reduce operation locks the database during post-processing.

  If nonAtomic is true, the post-processing step prevents MongoDB from locking the database: during this time, other clients will be able to read intermediate states of the output collection.

Output Inline Perform the map-reduce operation in memory and return the result. This option is the only available option for out on secondary members of replica sets.

```
out: { inline: 1 }
```

The result must fit within the *maximum size of a BSON document* (page 692).

Map-Reduce Examples In the mongo (page 610) shell, the db.collection.mapReduce() (page 58) method is a wrapper around the mapReduce (page 220) command. The following examples use the db.collection.mapReduce() (page 58) method:

Consider the following map-reduce operations on a collection orders that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 25,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

Return the Total Price Per Customer Perform the map-reduce operation on the `orders` collection to group by the `cust_id`, and calculate the sum of the `price` for each `cust_id`:

1. Define the map function to process each input document:

   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - The function maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair.

   ```
   var mapFunction1 = function() {
                          emit(this.cust_id, this.price);
                      };
   ```

2. Define the corresponding reduce function with two arguments `keyCustId` and `valuesPrices`:

   - The `valuesPrices` is an array whose elements are the `price` values emitted by the map function and grouped by `keyCustId`.
   - The function reduces the `valuesPrice` array to the sum of its elements.

   ```
   var reduceFunction1 = function(keyCustId, valuesPrices) {
                             return Array.sum(valuesPrices);
                         };
   ```

3. Perform the map-reduce on all documents in the `orders` collection using the `mapFunction1` map function and the `reduceFunction1` reduce function.

   ```
   db.orders.mapReduce(
                        mapFunction1,
                        reduceFunction1,
                        { out: "map_reduce_example" }
                      )
   ```

   This operation outputs the results to a collection named `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will replace the contents with the results of this map-reduce operation:

Calculate Order and Total Quantity with Average Quantity Per Item In this example, you will perform a map-reduce operation on the `orders` collection for all documents that have an `ord_date` value greater than `01/01/2012`. The operation groups by the `item.sku` field, and calculates the number of orders and the total quantity ordered for each `sku`. The operation concludes by calculating the average quantity per order for each `sku` value:

1. Define the map function to process each input document:

   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

   ```
   var mapFunction2 = function() {
                          for (var idx = 0; idx < this.items.length; idx++) {
                              var key = this.items[idx].sku;
                              var value = {
                                            count: 1,
                                            qty: this.items[idx].qty
                                          };
                              emit(key, value);
                          }
                      };
   ```

2. Define the corresponding reduce function with two arguments `keySKU` and `countObjVals`:

   - `countObjVals` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.
   - The function reduces the `countObjVals` array to a single object `reducedValue` that contains the `count` and the `qty` fields.
   - In `reducedVal`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

   ```
   var reduceFunction2 = function(keySKU, countObjVals) {
                          reducedVal = { count: 0, qty: 0 };

                          for (var idx = 0; idx < countObjVals.length; idx++) {
                              reducedVal.count += countObjVals[idx].count;
                              reducedVal.qty += countObjVals[idx].qty;
                          }

                          return reducedVal;
                        };
   ```

3. Define a finalize function with two arguments `key` and `reducedVal`. The function modifies the `reducedVal` object to add a computed field named `avg` and returns the modified object:

   ```
   var finalizeFunction2 = function (key, reducedVal) {

                             reducedVal.avg = reducedVal.qty/reducedVal.count;

                             return reducedVal;

                           };
   ```

4. Perform the map-reduce operation on the `orders` collection using the `mapFunction2`, `reduceFunction2`, and `finalizeFunction2` functions.

   ```
   db.orders.mapReduce( mapFunction2,
                        reduceFunction2,
                        {
                          out: { merge: "map_reduce_example" },
                          query: { ord_date:
                                       { $gt: new Date('01/01/2012') }
                                 },
                          finalize: finalizeFunction2
                        }
                      )
   ```

   This operation uses the `query` field to select only those documents with `ord_date` greater than `new Date(01/01/2012)`. Then it output the results to a collection `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will merge the existing contents with the results of this map-reduce operation.

For more information and examples, see the `Map-Reduce` page and `http://docs.mongodb.org/manual/tutorial/perfo`

Output If you set the *out* (page 224) parameter to write the results to a collection, the `mapReduce` (page 220) command returns a document in the following form:

```
{
    "result" : <string or document>,
    "timeMillis" : <int>,
```

```
    "counts" : {
        "input" : <int>,
        "emit" : <int>,
        "reduce" : <int>,
        "output" : <int>
    },
    "ok" : <int>,
}
```

If you set the *out* (page 224) parameter to output the results inline, the `mapReduce` (page 220) command returns a document in the following form:

```
{
    "results" : [
       {
          "_id" : <key>,
          "value" :<reduced or finalizedValue for key>
       },
       ...
    ],
    "timeMillis" : <int>,
    "counts" : {
       "input" : <int>,
       "emit" : <int>,
       "reduce" : <int>,
       "output" : <int>
    },
    "ok" : <int>
}
```

`mapReduce.`**`result`**
For output sent to a collection, this value is either:

- a string for the collection name if *out* (page 224) did not specify the database name, or
- a document with both `db` and `collection` fields if *out* (page 224) specified both a database and collection name.

`mapReduce.`**`results`**
For output written inline, an array of resulting documents. Each resulting document contains two fields:

- `_id` field contains the `key` value,
- `value` field contains the reduced or finalized value for the associated `key`.

`mapReduce.`**`timeMillis`**
The command execution time in milliseconds.

`mapReduce.`**`counts`**
Various count statistics from the `mapReduce` (page 220) command.

`mapReduce.counts.`**`input`**
The number of documents the `mapReduce` (page 220) command called the `map` function.

`mapReduce.counts.`**`emit`**
The number of times the `mapReduce` (page 220) command called the `emit` function.

`mapReduce.counts.`**`reduce`**
The number of times the `mapReduce` (page 220) command called the `reduce` function.

`mapReduce.counts.`**`output`**
The number of output values produced.

`mapReduce.ok`
A value of `1` indicates the `mapReduce` (page 220) command ran successfully. A value of `0` indicates an error.

Additional Information

- `http://docs.mongodb.org/manual/tutorial/troubleshoot-map-function`
- `http://docs.mongodb.org/manual/tutorial/troubleshoot-reduce-function`
- `db.collection.mapReduce()` (page 58)
- `http://docs.mongodb.org/manual/core/aggregation`

For a detailed comparison of the different approaches, see *Aggregation Commands Comparison* (page 569).

Geospatial Commands

Geospatial Commands

Name	Description
`geoNear` (page 229)	Performs a geospatial query that returns the documents closest to a given point.
`geoSearch` (page 233)	Performs a geospatial query that uses MongoDB's *haystack index* functionality.
`geoWalk` (page 233)	An internal command to support geospatial queries.

geoNear

Definition

`geoNear`

Specifies a point for which a *geospatial* query returns the closest documents first. The query returns the documents from nearest to farthest. The `geoNear` (page 229) command provides an alternative to the `$near` (page 429) operator. In addition to the functionality of `$near` (page 429), `geoNear` (page 229) returns additional diagnostic information.

The `geoNear` (page 229) command accepts a *document* that contains the following fields. Specify all distances in the same units as the document coordinate system:

field string geoNear The collection to query.

:field GeoJSON point,:term:*legacy coordinate pairs* <*legacy coordinate pairs*> near:

The point for which to find the closest documents.

field number limit The maximum number of documents to return. The default value is `100`. See also the `num` option.

field number num The `num` option provides the same function as the `limit` option. Both define the maximum number of documents to return. If both options are included, the `num` value overrides the `limit` value.

field number minDistance The minimum distance from the center point that the documents *must* be. MongoDB filters the results to those documents that are *at least* the specified distance from the center point.

Only available for use with `2dsphere` index.

Specify the distance in meters for *GeoJSON* data and in radians for *legacy coordinate pairs*.

New in version 2.6.

field number maxDistance The maximum distance from the center point that the documents *can* be. MongoDB limits the results to those documents that fall within the specified distance from the center point.

Specify the distance in meters for *GeoJSON* data and in radians for *legacy coordinate pairs*.

field document query Limits the results to the documents that match the query. The query syntax is the usual MongoDB *read operation query* syntax.

You cannot specify a `$near` (page 429) predicate in the `query` field of the `geoNear` (page 229) command.

field Boolean spherical Required *if* using a `2dsphere` index. For use with `2dsphere` indexes, `spherical` must be `true`.

The default value is `false`.

field number distanceMultiplier The factor to multiply all distances returned by the query. For example, use the `distanceMultiplier` to convert radians, as returned by a spherical query, to kilometers by multiplying by the radius of the Earth.

field Boolean includeLocs If this is `true`, the query returns the location of the matching documents in the results. The default is `false`. This option is useful when a location field contains multiple locations. To specify a field within a subdocument, use *dot notation*.

field Boolean uniqueDocs If this value is `true`, the query returns a matching document once, even if more than one of the document's location fields match the query.

Deprecated since version 2.6: Geospatial queries no longer return duplicate results. The `$uniqueDocs` (page 437) operator has no impact on results.

Considerations The `geoNear` (page 229) command can use either a *GeoJSON* point or *legacy coordinate pairs*.

The `geoNear` (page 229) command requires that a collection have *at most* only one `2d index` and/or only one `2dsphere`.

You cannot specify a `$near` (page 429) predicate in the `query` field of the `geoNear` (page 229) command.

Behavior `geoNear` (page 229) always returns the documents sorted by distance. Any other sort order requires to sort the documents in memory, which can be inefficient. To return results in a different sort order, use the `$geoWithin` operator and the `sort()` method.

Because `geoNear` (page 229) orders the documents from nearest to farthest, the `minDistance` field effectively skips over the first *n* documents where *n* is determined by the distance requirement.

Command Format

`2dsphere` Index To query a `2dsphere` index, use the following syntax:

```
db.runCommand( {
   geoNear: <collection> ,
   near: { type: "Point" , coordinates: [ <coordinates> ] } ,
   spherical: true,
   ...
} )
```

You must include `spherical: true`.

2d Index To query a `2d` index, use:

```
db.runCommand( {
   geoNear: <collection>,
   near : [ <coordinates> ],
   ...
} )
```

Examples The following examples run the `geoNear` (page 229) command on the collection `places` that has a `2dsphere` index.

Specify a Query Condition The following `geoNear` (page 229) command queries for documents whose `category` equals `"public"` and returns the matching documents in order of nearest to farthest to the specified point:

```
db.runCommand(
   {
     geoNear: "places",
     near: { type: "Point", coordinates: [ -73.9667, 40.78 ] },
     spherical: true,
     query: { category: "public" }
   }
)
```

The operation returns the following output, the documents in the `results` from nearest to farthest:

```
{
  "results" : [
     {
       "dis" : 0,
       "obj" : {
          "_id" : 2,
          "location" : { "type" : "Point", "coordinates" : [ -73.9667, 40.78 ] },
          "name" : "Central Park",
          "category" : "public"
       }
     },
     {
       "dis" : 3245.988787957091,
       "obj" : {
          "_id" : 3,
          "location" : { "type" : "Point", "coordinates" : [ -73.9836, 40.7538 ] },
          "name" : "Bryant Park",
          "category" : "public"
       }
     },
     {
       "dis" : 7106.506152782733,
       "obj" : {
          "_id" : 4,
          "location" : { "type" : "Point", "coordinates" : [ -73.9928, 40.7193 ] },
          "name" : "Sara D. Roosevelt Park",
          "category" : "public"
       }
     },

  ],
```

```
  "stats" : {
     "nscanned" : NumberLong(47),
     "objectsLoaded" : NumberLong(47),
     "avgDistance" : 3450.8316469132747,
     "maxDistance" : 7106.506152782733,
     "time" : 4
  },
  "ok" : 1
}
```

Specify a `minDistance` and `maxDistance` The following example specifies a `minDistance` of `3000` meters and `maxDistance` of `7000` meters:

```
db.runCommand(
   {
     geoNear: "places",
     near: { type: "Point", coordinates: [ -73.9667, 40.78 ] },
     spherical: true,
     query: { category: "public" },
     minDistance: 3000,
     maxDistance: 7000
   }
)
```

The operation returns the following output:

```
{
  "results" : [
     {
       "dis" : 3245.988787957091,
       "obj" : {
          "_id" : 3,
          "location" : { "type" : "Point", "coordinates" : [ -73.9836, 40.7538 ] },
          "name" : "Bryant Park",
          "category" : "public"
       }
     }
  ],
  "stats" : {
      "nscanned" : NumberLong(11),
      "objectsLoaded" : NumberLong(11),
      "avgDistance" : 3245.988787957091,
      "maxDistance" : 3245.988787957091,
      "time" : 0
  },
  "ok" : 1
}
```

Output The `geoNear` (page 229) command returns a document with the following fields:

`geoNear.results`
: An array with the results of the `geoNear` (page 229) command, sorted by distance with the nearest result listed first and farthest last.

`geoNear.results[n].dis`
: For each document in the results, the distance from the coordinates defined in the `geoNear` (page 229) command.

`geoNear.results[n].obj`
The document from the collection.

`geoNear.stats`
An object with statistics about the query used to return the results of the `geoNear` (page 229) search.

`geoNear.stats.nscanned`
The total number of index entries scanned during the database operation.

`geoNear.stats.objectsLoaded`
The total number of documents read from disk during the database operation.

`geoNear.stats.avgDistance`
The average distance between the coordinates defined in the `geoNear` (page 229) command and coordinates of the documents returned as results.

`geoNear.stats.maxDistance`
The maximum distance between the coordinates defined in the `geoNear` (page 229) command and coordinates of the documents returned as results.

`geoNear.stats.time`
The execution time of the database operation, in milliseconds.

`geoNear.ok`
A value of `1` indicates the `geoNear` (page 229) search succeeded. A value of `0` indicates an error.

geoSearch

`geoSearch`

The `geoSearch` (page 233) command provides an interface to MongoDB's *haystack index* functionality. These indexes are useful for returning results based on location coordinates *after* collecting results based on some other query (i.e. a "haystack.") Consider the following example:

```
{ geoSearch : "places", near : [33, 33], maxDistance : 6, search : { type : "restaurant" }, limi
```

The above command returns all documents with a `type` of `restaurant` having a maximum distance of 6 units from the coordinates `[30,33]` in the collection `places` up to a maximum of 30 results.

Unless specified otherwise, the `geoSearch` (page 233) command limits results to 50 documents.

Important: `geoSearch` (page 233) is not supported for sharded clusters.

---

geoWalk

`geoWalk`

`geoWalk` (page 233) is an internal command.

Query and Write Operation Commands

Query and Write Operation Commands

Name	Description
delete (page 234)	Deletes one or more documents.
eval (page 237)	Runs a JavaScript function on the database server.
findAndModify (page 239)	Returns and modifies a single document.
getLastError (page 245)	Returns the success status of the last operation.
getPrevError (page 247)	Returns status document containing all errors since the last resetError (page 250) command.
insert (page 247)	Inserts one or more documents.
parallelCollectionScan (page 249)	Lets applications use multiple parallel cursors when reading documents from a collection.
resetError (page 250)	Resets the last error status.
text (page 251)	Performs a text search.
update (page 251)	Updates one or more documents.

delete

Definition

delete

New in version 2.6.

The delete (page 234) command removes documents from a collection. A single delete (page 234) command can contain multiple delete specifications. The command cannot operate on capped collections. The remove methods provided by the MongoDB drivers use this command internally.

The delete (page 234) command has the following syntax:

```
{
   delete: <collection>,
   deletes: [
      { q : <query>, limit : <integer> },
      { q : <query>, limit : <integer> },
      { q : <query>, limit : <integer> },
      ...
   ],
   ordered: <boolean>,
   writeConcern: { <write concern> }
}
```

The command takes the following fields:

field string delete The name of the target collection.

field array deletes An array of one or more delete statements to perform in the named collection.

field boolean ordered If true, then when a delete statement fails, return without performing the remaining delete statements. If false, then when a delete statement fails, continue with the remaining delete statements, if any. Defaults to true.

field document writeConcern A document expressing the write concern of the delete (page 234) command. Omit to use the default write concern.

Each element of the deletes array contains the following sub-fields:

field document q The query that matches documents to delete.

field integer limit The number of matching documents to delete. Specify either a `0` to delete all matching documents or `1` to delete a single document.

Returns A document that contains the status of the operation. See *Output* (page 236) for details.

Behavior The total size of all the queries (i.e. the `q` field values) in the `deletes` array must be less than or equal to the `maximum BSON document size` (page 692).

The total number of delete documents in the `deletes` array must be less than or equal to the `maximum bulk size`.

Examples

Limit the Number of Documents Deleted The following example deletes from the `orders` collection one document that has the `status` equal to `D` by specifying the `limit` of `1`:

```
db.runCommand(
   {
      delete: "orders",
      deletes: [ { q: { status: "D" }, limit: 1 } ]
   }
)
```

The returned document shows that the command deleted `1` document. See *Output* (page 236) for details.

```
{ "ok" : 1, "n" : 1 }
```

Delete All Documents That Match a Condition The following example deletes from the `orders` collection all documents that have the `status` equal to `D` by specifying the `limit` of `0`:

```
db.runCommand(
   {
      delete: "orders",
      deletes: [ { q: { status: "D" }, limit: 0 } ],
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

The returned document shows that the command found and deleted `13` documents. See *Output* (page 236) for details.

```
{ "ok" : 1, "n" : 13 }
```

Delete All Documents from a Collection Delete all documents in the `orders` collection by specifying an empty query condition *and* a `limit` of `0`:

```
db.runCommand(
   {
      delete: "orders",
      deletes: [ { q: { }, limit: 0 } ],
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

The returned document shows that the command found and deleted `35` documents in total. See *Output* (page 236) for details.

```
{ "ok" : 1, "n" : 35 }
```

Bulk Delete The following example performs multiple delete operations on the `orders` collection:

```
db.runCommand(
   {
      delete: "orders",
      deletes: [
         { q: { status: "D" }, limit: 0 },
         { q: { cust_num: 99999, item: "abc123", status: "A" }, limit: 1 }
      ],
      ordered: false,
      writeConcern: { w: 1 }
   }
)
```

The returned document shows that the command found and deleted `21` documents in total for the two delete statements. See *Output* (page 236) for details.

```
{ "ok" : 1, "n" : 21 }
```

Output The returned document contains a subset of the following fields:

`delete.`**`ok`**
: The status of the command.

`delete.`**`n`**
: The number of documents deleted.

`delete.`**`writeErrors`**
: An array of documents that contains information regarding any error encountered during the delete operation. The `writeErrors` (page 236) array contains an error document for each delete statement that errors.

  Each error document contains the following information:

  `delete.writeErrors.`**`index`**
  : An integer that identifies the delete statement in the `deletes` array, which uses a zero-based index.

  `delete.writeErrors.`**`code`**
  : An integer value identifying the error.

  `delete.writeErrors.`**`errmsg`**
  : A description of the error.

`delete.`**`writeConcernError`**
: Document that describe error related to write concern and contains the field:

  `delete.writeConcernError.`**`code`**
  : An integer value identifying the cause of the write concern error.

  `delete.writeConcernError.`**`errmsg`**
  : A description of the cause of the write concern error.

The following is an example document returned for a successful `delete` (page 234) command:

```
{ ok: 1, n: 1 }
```

The following is an example document returned for a `delete` (page 234) command that encountered an error:

```
{
   "ok" : 1,
   "n" : 0,
   "writeErrors" : [
      {
         "index" : 0,
         "code" : 10101,
         "errmsg" : "can't remove from a capped collection: test.cappedLog"
      }
   ]
}
```

eval

eval

The `eval` (page 237) command evaluates JavaScript functions on the database server.

The `eval` (page 237) command has the following form:

```
{
  eval: <function>,
  args: [ <arg1>, <arg2> ... ],
  nolock: <boolean>
}
```

The command contains the following fields:

field function eval A JavaScript function.

field array args An array of arguments to pass to the JavaScript function. Omit if the function does not take arguments.

field boolean nolock By default, `eval` (page 237) takes a global write lock before evaluating the JavaScript function. As a result, `eval` (page 237) blocks all other read and write operations to the database while the `eval` (page 237) operation runs. Set `nolock` to `true` on the `eval` (page 237) command to prevent the `eval` (page 237) command from taking the global write lock before evaluating the JavaScript. `nolock` does not impact whether operations within the JavaScript code itself takes a write lock.

JavaScript in MongoDB

Although `eval` (page 237) uses JavaScript, most interactions with MongoDB do not use JavaScript but use an `idiomatic driver` in the language of the interacting application.

Behavior

Write Lock By default, `eval` (page 237) takes a global write lock while evaluating the JavaScript function. As a result, `eval` (page 237) blocks all other read and write operations to the database while the `eval` (page 237) operation runs.

To prevent the taking of the global write lock while evaluating the JavaScript code, use the `eval` (page 237) command with `nolock` set to `true`. `nolock` does not impact whether the operations within the JavaScript code take write locks.

For long running `eval` (page 237) operation, consider using either the **eval** command with `nolock: true` or using `other server side code execution options`.

Sharded Data You can not use `eval` (page 237) with *sharded* collections. In general, you should avoid using `eval` (page 237) in *sharded clusters*; nevertheless, it is possible to use `eval` (page 237) with non-sharded collections and databases stored in a *sharded cluster*.

Access Control Changed in version 2.6.

If authorization is enabled, you must have access to all actions on all resources in order to run `eval` (page 237). Providing such access is not recommended, but if your organization requires a user to run `eval` (page 237), create a role that grants `anyAction` on *resource-anyresource*. Do not assign this role to any other user.

JavaScript Engine Changed in version 2.4.

The V8 JavaScript engine, which became the default in 2.4, allows multiple JavaScript operations to execute at the same time. Prior to 2.4, `eval` (page 237) executed in a single thread.

Example The following example uses `eval` (page 237) to perform an increment and calculate the average on the server:

```
db.runCommand( {
      eval: function(name, incAmount) {
                var doc = db.myCollection.findOne( { name : name } );

                doc = doc || { name : name , num : 0 , total : 0 , avg : 0 };

                doc.num++;
                doc.total += incAmount;
                doc.avg = doc.total / doc.num;

                db.myCollection.save( doc );
                return doc;
            },
      args: [ "eliot", 5 ]
   }
);
```

The `db` in the function refers to the current database.

The `mongo` (page 610) shell provides a helper method `db.eval()` (page 112) [13], so you can express the above as follows:

```
db.eval( function(name, incAmount) {
            var doc = db.myCollection.findOne( { name : name } );

            doc = doc || { name : name , num : 0 , total : 0 , avg : 0 };

            doc.num++;
            doc.total += incAmount;
            doc.avg = doc.total / doc.num;

            db.myCollection.save( doc );
            return doc;
         },
         "eliot", 5 );
```

[13] The helper `db.eval()` (page 112) in the `mongo` (page 610) shell wraps the `eval` (page 237) command. Therefore, the helper method shares the characteristics and behavior of the underlying command with *one exception*: `db.eval()` (page 112) method does not support the `nolock` option.

If you want to use the server's interpreter, you must run `eval` (page 237). Otherwise, the `mongo` (page 610) shell's JavaScript interpreter evaluates functions entered directly into the shell.

If an error occurs, `eval` (page 237) throws an exception. The following invalid function uses the variable `x` without declaring it as an argument:

```
db.runCommand(
                {
                  eval: function() { return x + x; },
                  args: [ 3 ]
                }
              )
```

The statement will result in the following exception:

```
{
   "errmsg" : "exception: JavaScript execution failed: ReferenceError: x is not defined near '{ retu
   "code" : 16722,
   "ok" : 0
}
```

See also:

`http://docs.mongodb.org/manual/core/server-side-javascript`

findAndModify

Definition

`findAndModify`

The `findAndModify` (page 239) command modifies and returns a single document. By default, the returned document does not include the modifications made on the update. To return the document with the modifications made on the update, use the `new` option.

The command has the following syntax:

```
{
  findAndModify: <collection-name>,
  query: <document>,
  sort: <document>,
  remove: <boolean>,
  update: <document>,
  new: <boolean>,
  fields: <document>,
  upsert: <boolean>
}
```

The `findAndModify` (page 239) command takes the following fields:

field string findAndModify The collection against which to run the command.

param document query The selection criteria for the modification. The `query` field employs the same *query selectors* (page 400) as used in the `db.collection.find()` (page 36) method. Although the query may match multiple documents, `findAndModify` (page 239) **will only select one document to modify**.

param document sort Determines which document the operation modifies if the query selects multiple documents. `findAndModify` (page 239) modifies the first document in the sort order specified by this argument.

param Boolean remove Must specify either the `remove` or the `update` field. Removes the document specified in the `query` field. Set this to `true` to remove the selected document . The default is `false`.

param document update Must specify either the `remove` or the `update` field. Performs an update of the selected document. The `update` field employs the same *update operators* (page 451) or `field: value` specifications to modify the selected document.

param Boolean new When `true`, returns the modified document rather than the original. The `findAndModify` (page 239) method ignores the `new` option for `remove` operations. The default is `false`.

param document fields A subset of fields to return. The `fields` document specifies an inclusion of a field with `1`, as in: `fields: { <field1>: 1, <field2>: 1, ... }`. See *projection*.

param Boolean upsert Used in conjunction with the `update` field.

When `true`, `findAndModify` (page 239) creates a new document if no document matches the `query`, or if documents match the `query`, `findAndModify` (page 239) performs an update. To avoid multiple upserts, ensure that the `query` fields are *uniquely indexed*.

The default is `false`.

Output The return document contains the following fields:

- The `lastErrorObject` field that returns the details of the command:
  - The `updatedExisting` field **only** appears if the command specifies an `update` or an `update` with `upsert: true`; i.e. the field does not appear for a `remove`.
  - The `upserted` field **only** appears if the `update` with the `upsert: true` operation results in an insertion.
- The `value` field that returns either:
  - the original (i.e. pre-modification) document if `new` is false, or
  - the modified or inserted document if `new: true`.
- The `ok` field that returns the status of the command.

Note: If the `findAndModify` (page 239) finds no matching document, then:

- for `update` or `remove` operations, `lastErrorObject` does not appear in the return document and the `value` field holds a `null`.

```
{ "value" : null, "ok" : 1 }
```

- for `update` with `upsert: true` operation that results in an insertion, if the command also specifies `new` is `false` **and** specifies a `sort`, the return document has a `lastErrorObject`, `value`, and `ok` fields, but the `value` field holds an empty document `{}`.
- for `update` with `upsert: true` operation that results in an insertion, if the command also specifies `new` is `false` but does **not** specify a `sort`, the return document has a `lastErrorObject`, `value`, and `ok` fields, but the `value` field holds a `null`.

Behavior

Upsert and Unique Index When the `findAndModify` (page 239) command includes the `upsert: true` option **and** the query field(s) is not uniquely indexed, the command could insert a document multiple times in certain circumstances.

Consider an example where no document with the name `Andy` exists and multiple clients issue the following command:

```
db.runCommand(
   {
     findAndModify: "people",
     query: { name: "Andy" },
     sort: { rating: 1 },
     update: { $inc: { score: 1 } },
     upsert: true
   }
)
```

If all the commands finish the `query` phase before any command starts the `modify` phase, **and** there is no unique index on the `name` field, the commands may each perform an upsert, creating multiple duplicate documents.

To prevent the creation of multiple duplicate documents, create a *unique index* on the `name` field. With the unique index in place, then the multiple `findAndModify` (page 239) commands will exhibit one of the following behaviors:

- Exactly one `findAndModify` (page 239) successfully inserts a new document.
- Zero or more `findAndModify` (page 239) commands update the newly inserted document.
- Zero or more `findAndModify` (page 239) commands fail when they attempt to insert a duplicate. If the command fails due to a unique index constraint violation, you can retry the command. Absent a delete of the document, the retry should not fail.

Sharded Collections When using `findAndModify` (page 239) in a *sharded* environment, the `query` must contain the *shard key* for all operations against the shard cluster. `findAndModify` (page 239) operations issued against `mongos` (page 601) instances for non-sharded collections function normally.

Concurrency This command obtains a write lock on the affected database and will block other operations until it has completed; however, typically the write lock is short lived and equivalent to other similar `update()` (page 72) operations.

Comparisons with the `update` Method When updating a document, `findAndModify` (page 239) and the `update()` (page 72) method operate differently:

- By default, both operations modify a single document. However, the `update()` (page 72) method with its `multi` option can modify more than one document.
- If multiple documents match the update criteria, for `findAndModify` (page 239), you can specify a `sort` to provide some measure of control on which document to update.

  With the default behavior of the `update()` (page 72) method, you cannot specify which single document to update when multiple documents match.
- By default, `findAndModify` (page 239) method returns an object that contains the pre-modified version of the document, as well as the status of the operation. To obtain the updated document, use the `new` option.

  The `update()` (page 72) method returns a `WriteResult` (page 201) object that contains the status of the operation. To return the updated document, use the `find()` (page 36) method. However, other updates may have modified the document between your update and the document retrieval. Also, if the update modified only

a single document but multiple documents matched, you will need to use additional logic to identify the updated document.

- You cannot specify a `write concern` to `findAndModify` (page 239) to override the default write concern whereas, starting in MongoDB 2.6, you can specify a write concern to the `update()` (page 72) method.

When modifying a *single* document, both `findAndModify` (page 239) and the `update()` (page 72) method *atomically* update the document. See `http://docs.mongodb.org/manual/core/write-operations-atomicity` for more details about interactions and order of operations of these methods.

Examples

Update and Return The following command updates an existing document in the `people` collection where the document matches the `query` criteria:

```
db.runCommand(
   {
     findAndModify: "people",
     query: { name: "Tom", state: "active", rating: { $gt: 10 } },
     sort: { rating: 1 },
     update: { $inc: { score: 1 } }
   }
)
```

This command performs the following actions:

1. The `query` finds a document in the `people` collection where the `name` field has the value `Tom`, the `state` field has the value `active` and the `rating` field has a value `greater than` 10.
2. The `sort` orders the results of the query in ascending order. If multiple documents meet the `query` condition, the command will select for modification the first document as ordered by this `sort`.
3. The `update` `increments` the value of the `score` field by 1.
4. The command returns a document with the following fields:
   - The `lastErrorObject` field that contains the details of the command, including the field `updatedExisting` which is `true`, and
   - The `value` field that contains the original (i.e. pre-modification) document selected for this update:

```
{
  "lastErrorObject" : {
     "updatedExisting" : true,
     "n" : 1,
     "connectionId" : 1,
     "err" : null,
     "ok" : 1
  },
  "value" : {
     "_id" : ObjectId("50f1d54e9beb36a0f45c6452"),
     "name" : "Tom",
     "state" : "active",
     "rating" : 100,
     "score" : 5
  },
  "ok" : 1
}
```

To return the modified document in the `value` field, add the `new:true` option to the command.

If no document match the `query` condition, the command returns a document that contains `null` in the `value` field:

```
{ "value" : null, "ok" : 1 }
```

The `mongo` (page 610) shell and many *drivers* provide a `findAndModify()` (page 42) helper method. Using the shell helper, this previous operation can take the following form:

```
db.people.findAndModify( {
   query: { name: "Tom", state: "active", rating: { $gt: 10 } },
   sort: { rating: 1 },
   update: { $inc: { score: 1 } }
} );
```

However, the `findAndModify()` (page 42) shell helper method returns only the unmodified document, or if `new` is `true`, the modified document.

```
{
   "_id" : ObjectId("50f1d54e9beb36a0f45c6452"),
   "name" : "Tom",
   "state" : "active",
   "rating" : 100,
   "score" : 5
}
```

`upsert: true` The following `findAndModify` (page 239) command includes the `upsert: true` option for the `update` operation to either update a matching document or, if no matching document exists, create a new document:

```
db.runCommand(
                {
                  findAndModify: "people",
                  query: { name: "Gus", state: "active", rating: 100 },
                  sort: { rating: 1 },
                  update: { $inc: { score: 1 } },
                  upsert: true
                }
              )
```

If the command finds a matching document, the command performs an update.

If the command does **not** find a matching document, the `update` with *upsert: true* operation results in an insertion and returns a document with the following fields:

- The `lastErrorObject` field that contains the details of the command, including the field `upserted` that contains the `ObjectId` of the newly inserted document, and
- The `value` field that contains an empty document `{}` as the original document because the command included the `sort` option:

```
{
  "lastErrorObject" : {
     "updatedExisting" : false,
     "upserted" : ObjectId("50f2329d0092b46dae1dc98e"),
     "n" : 1,
     "connectionId" : 1,
     "err" : null,
     "ok" : 1
  },
```

```
  "value" : {

  },
  "ok" : 1
}
```

If the command did **not** include the `sort` option, the `value` field would contain `null`:

```
{
  "value" : null,
  "lastErrorObject" : {
     "updatedExisting" : false,
     "n" : 1,
     "upserted" : ObjectId("5102f7540cb5c8be998c2e99")
  },
  "ok" : 1
}
```

Return New Document The following `findAndModify` (page 239) command includes both `upsert: true` option and the `new:true` option. The command either updates a matching document and returns the updated document or, if no matching document exists, inserts a document and returns the newly inserted document in the `value` field.

In the following example, no document in the `people` collection matches the `query` condition:

```
db.runCommand(
   {
     findAndModify: "people",
     query: { name: "Pascal", state: "active", rating: 25 },
     sort: { rating: 1 },
     update: { $inc: { score: 1 } },
     upsert: true,
     new: true
   }
)
```

The command returns the newly inserted document in the `value` field:

```
{
  "lastErrorObject" : {
     "updatedExisting" : false,
     "upserted" : ObjectId("50f47909444c11ac2448a5ce"),
     "n" : 1,
     "connectionId" : 1,
     "err" : null,
     "ok" : 1
  },
  "value" : {
     "_id" : ObjectId("50f47909444c11ac2448a5ce"),
     "name" : "Pascal",
     "rating" : 25,
     "score" : 1,
     "state" : "active"
  },
  "ok" : 1
}
```

getLastError

Definition

getLastError

Changed in version 2.6: A new protocol for *write operations* (page 745) integrates write concerns with the write operations, eliminating the need for a separate `getLastError` (page 245) command. Write methods now return the status of the write operation, including error information.

In previous versions, clients typically used the `getLastError` (page 245) command in combination with the write operations to ensure that the write succeeds.

Returns the error status of the preceding write operation on the *current connection*.

`getLastError` (page 245) uses the following prototype form:

```
{ getLastError: 1 }
```

`getLastError` (page 245) uses the following fields:

field Boolean j If `true`, wait for the next journal commit before returning, rather than waiting for a full disk flush. If `mongod` (page 583) does not have journaling enabled, this option has no effect. If this option is enabled for a write operation, `mongod` (page 583) will wait *no more* than 1/3 of the current `commitIntervalMs` before writing data to the journal.

field integer,string w When running with replication, this is the number of servers to replicate to before returning. A `w` value of 1 indicates the primary only. A `w` value of 2 includes the primary and at least one secondary, etc. In place of a number, you may also set `w` to `majority` to indicate that the command should wait until the latest write propagates to a majority of replica set members. If using `w`, you should also use `wtimeout`. Specifying a value for `w` without also providing a `wtimeout` may cause `getLastError` (page 245) to block indefinitely.

field Boolean fsync If `true`, wait for `mongod` (page 583) to write this data to disk before returning. Defaults to false. In most cases, use the `j` option to ensure durability and consistency of the data set.

field integer wtimeout Milliseconds. Specify a value in milliseconds to control how long to wait for write propagation to complete. If replication does not complete in the given timeframe, the `getLastError` (page 245) command will return with an error status.

See also:

`Write Concern`, `http://docs.mongodb.org/manual/reference/write-concern`, and *replica-set-write-concern*.

Output Each `getLastError()` command returns a document containing a subset of the fields listed below.

`getLastError.ok`

`ok` (page 245) is `true` when the `getLastError` (page 245) command completes successfully.

Note: A value of `true` does *not* indicate that the preceding operation did not produce an error.

`getLastError.err`

`err` (page 245) is `null` unless an error occurs. When there was an error with the preceding operation, `err` contains a textual description of the error.

`getLastError.code`

`code` (page 245) reports the preceding operation's error code.

getLastError.**connectionId**
: The identifier of the connection.

getLastError.**lastOp**
: When issued against a replica set member and the preceding operation was a write or update, lastOp (page 246) is the *optime* timestamp in the *oplog* of the change.

getLastError.**n**
: n (page 246) reports the number of documents updated or removed, if the preceding operation was an update or remove operation.

getLastError.**shards**
: When issued against a sharded cluster after a write operation, shards (page 246) identifies the shards targeted in the write operation. shards (page 246) is present in the output only if the write operation targets multiple shards.

getLastError.**singleShard**
: When issued against a sharded cluster after a write operation, identifies the shard targeted in the write operation. singleShard (page 246) is only present if the write operation targets exactly one shard.

getLastError.**updatedExisting**
: updatedExisting (page 246) is true when an update affects at least one document and does not result in an *upsert*.

getLastError.**upserted**
: If the update results in an insert, upserted (page 246) is the value of _id field of the document.

  Changed in version 2.6: Earlier versions of MongoDB included upserted (page 246) only if _id was an *ObjectId*.

getLastError.**wnote**
: If set, wnote indicates that the preceding operation's error relates to using the w parameter to getLastError (page 245).

See

http://docs.mongodb.org/manual/reference/write-concern for more information about w values.

---

getLastError.**wtimeout**
: wtimeout (page 246) is true if the getLastError (page 245) timed out because of the wtimeout setting to getLastError (page 245).

getLastError.**waited**
: If the preceding operation specified a timeout using the wtimeout setting to getLastError (page 245), then waited (page 246) reports the number of milliseconds getLastError (page 245) waited before timing out.

getLastError.**wtime**
: getLastError.wtime (page 246) is the number of milliseconds spent waiting for the preceding operation to complete. If getLastError (page 245) timed out, wtime (page 246) and getLastError.waited are equal.

Examples

Confirm Replication to Two Replica Set Members The following example ensures the operation has replicated to two members (the primary and one other member):

```
db.runCommand( { getLastError: 1, w: 2 } )
```

Confirm Replication to a Majority of a Replica Set The following example ensures the write operation has replicated to a majority of the configured members of the set.

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

Changed in version 2.6: In `Master/Slave` deployments, MongoDB treats `w: "majority"` as equivalent to `w: 1`. In earlier versions of MongoDB, `w: "majority"` produces an error in `master/slave` deployments.

Set a Timeout for a `getLastError` Response Unless you specify a timeout, a `getLastError` (page 245) command may block forever if MongoDB cannot satisfy the requested write concern. To specify a timeout of 5000 milliseconds, use an invocation that resembles the following:

```
db.runCommand( { getLastError: 1, w: 2, wtimeout:5000 } )
```

When `wtimeout` is `0`, the `getLastError` (page 245) operation will never time out.

getPrevError

`getPrevError`

The `getPrevError` (page 247) command returns the errors since the last `resetError` (page 250) command.

See also:

`db.getPrevError()` (page 116)

insert

Definition

`insert`

New in version 2.6.

The `insert` (page 247) command inserts one or more documents and returns a document containing the status of all inserts. The insert methods provided by the MongoDB drivers use this command internally.

The command has the following syntax:

```
{
   insert: <collection>,
   documents: [ <document>, <document>, <document>, ... ],
   ordered: <boolean>,
   writeConcern: { <write concern> }
}
```

The `insert` (page 247) command takes the following fields:

field string insert The name of the target collection.

field array documents An array of one or more documents to insert into the named collection.

field boolean ordered If `true`, then when an insert of a document fails, return without inserting any remaining documents listed in the `inserts` array. If `false`, then when an insert of a document fails, continue to insert the remaining documents. Defaults to `true`.

field document writeConcern A document that expresses the `write concern` of the `insert` (page 247) command. Omit to use the default write concern.

Returns A document that contains the status of the operation. See *Output* (page 248) for details.

Behavior The total size of all the `documents` array elements must be less than or equal to the `maximum BSON document size` (page 692).

The total number of documents in the `documents` array must be less than or equal to the `maximum bulk size`.

Examples

Insert a Single Document Insert a document into the `users` collection:

```
db.runCommand(
   {
      insert: "users",
      documents: [ { _id: 1, user: "abc123", status: "A" } ]
   }
)
```

The returned document shows that the command successfully inserted a document. See *Output* (page 248) for details.

```
{ "ok" : 1, "n" : 1 }
```

Bulk Insert Insert three documents into the `users` collection:

```
db.runCommand(
   {
      insert: "users",
      documents: [
         { _id: 2, user: "ijk123", status: "A" },
         { _id: 3, user: "xyz123", status: "P" },
         { _id: 4, user: "mop123", status: "P" }
      ],
      ordered: false,
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

The returned document shows that the command successfully inserted the three documents. See *Output* (page 248) for details.

```
{ "ok" : 1, "n" : 3 }
```

Output The returned document contains a subset of the following fields:

`insert.ok`
The status of the command.

`insert.n`
The number of documents inserted.

`insert.writeErrors`
An array of documents that contains information regarding any error encountered during the insert operation. The `writeErrors` (page 248) array contains an error document for each insert that errors.

Each error document contains the following fields:

`insert.writeErrors.`**`index`**
An integer that identifies the document in the `documents` array, which uses a zero-based index.

`insert.writeErrors.`**`code`**
An integer value identifying the error.

`insert.writeErrors.`**`errmsg`**
A description of the error.

`insert.`**`writeConcernError`**
Document that describe error related to write concern and contains the field:

`insert.writeConcernError.`**`code`**
An integer value identifying the cause of the write concern error.

`insert.writeConcernError.`**`errmsg`**
A description of the cause of the write concern error.

The following is an example document returned for a successful `insert` (page 247) of a single document:

```
{ ok: 1, n: 1 }
```

The following is an example document returned for an `insert` (page 247) of two documents that successfully inserted one document but encountered an error with the other document:

```
{
   "ok" : 1,
   "n" : 1,
   "writeErrors" : [
      {
         "index" : 1,
         "code" : 11000,
         "errmsg" : "insertDocument :: caused by :: 11000 E11000 duplicate key error index: test.user
      }
   ]
}
```

parallelCollectionScan

`parallelCollectionScan`
New in version 2.6.

Allows applications to use multiple parallel cursors when reading all the documents from a collection, thereby increasing throughput. The `parallelCollectionScan` (page 249) command returns a document that contains an array of cursor information.

Each cursor provides access to the return of a partial set of documents from a collection. Iterating each cursor returns every document in the collection. Cursors do not contain the results of the database command. The result of the database command identifies the cursors, but does not contain or constitute the cursors.

The server may return fewer cursors than requested.

The command has the following syntax:

```
{
  parallelCollectionScan: "<collection>",
  numCursors: <integer>
}
```

The `parallelCollectionScan` (page 249) command takes the following fields:

field string parallelCollectionScan The name of the collection.

field integer numCursors The maximum number of cursors to return. Must be between 1 and 10000, inclusive.

Output The `parallelCollectionScan` (page 249) command returns a document containing the array of cursor information:

```
{
    "cursors" : [
        {
            "cursor" : {
                "firstBatch" : [ ],
                "ns" : "<database name>.<collection name>",
                "id" : NumberLong("155949257847")
            },
            "ok" : true
        }
    ],
    "ok" : 1
}
```

`parallelCollectionScan.`**`cursors`**
An array with one or more cursors returned with the command.

`parallelCollectionScan.cursors.`**`cursor`**
For each cursor returned, a document with details about the cursor.

`parallelCollectionScan.cursors.cursor.`**`firstBatch`**
An empty first batch is useful for quickly returning a cursor or failure message without doing significant server-side work. See *cursor batches*.

`parallelCollectionScan.cursors.cursor.`**`ns`**
The namespace for each cursor.

`parallelCollectionScan.cursors.cursor.`**`id`**
The unique id for each cursor.

`parallelCollectionScan.cursors.`**`ok`**
The status of each cursor returned with the command.

`parallelCollectionScan.`**`ok`**
A value of `1` indicates the `parallelCollectionScan` (page 249) command succeeded. A value of `0` indicates an error.

resetError

`resetError`

The `resetError` (page 250) command resets the last error status.

See also:

`db.resetError()` (page 123)

text

Definition

text

Deprecated since version 2.6: Use `$text` (page 417) query operator instead.

For document on the `text` (page 251), refer to the 2.4 Manual 2.4 Manual[14].

update

Definition

update

New in version 2.6.

The `update` (page 251) command modifies documents in a collection. A single `update` (page 251) command can contain multiple update statements. The update methods provided by the MongoDB drivers use this command internally.

The `update` (page 251) command has the following syntax:

```
{
   update: <collection>,
   updates:
      [
         { q: <query>, u: <update>, upsert: <boolean>, multi: <boolean> },
         { q: <query>, u: <update>, upsert: <boolean>, multi: <boolean> },
         { q: <query>, u: <update>, upsert: <boolean>, multi: <boolean> },
         ...
      ],
   ordered: <boolean>,
   writeConcern: { <write concern> }
}
```

The command takes the following fields:

field string update The name of the target collection.

field array updates An array of one or more update statements to perform in the named collection.

field boolean ordered If `true`, then when an update statement fails, return without performing the remaining update statements. If `false`, then when an update fails, continue with the remaining update statements, if any. Defaults to `true`.

field document writeConcern A document expressing the `write concern` of the `update` (page 251) command. Omit to use the default write concern.

Each element of the `updates` array contains the following sub-fields:

field string document The query that matches documents to update. Use the same *query selectors* (page 400) as used in the `find()` (page 36) method.

field document u The modifications to apply. For details, see *Behaviors* (page 252).

field boolean upsert If `true`, perform an insert if no documents match the query. If both `upsert` and `multi` are true and no documents match the query, the update operation inserts only a single document.

field boolean multi If `true`, updates all documents that meet the query criteria. If `false`, limit the update to one document that meet the query criteria. Defaults to `false`.

Returns A document that contains the status of the operation. See *Output* (page 254) for details.

[14] http://docs.mongodb.org/v2.4/reference/command/text

Behaviors The <update> document can contain either all *update operator* (page 451) expressions or all field:value expressions.

Update Operator Expressions If the <update> document contains all *update operator* (page 451) expressions, as in:

```
{
  $set: { status: "D" },
  $inc: { quantity: 2 }
}
```

Then, the update (page 251) command updates only the corresponding fields in the document.

Field: Value Expressions If the <update> document contains *only* field:value expressions, as in:

```
{
  status: "D",
  quantity: 4
}
```

Then the update (page 251) command *replaces* the matching document with the update document. The update (page 251) command can only replace a *single* matching document; i.e. the multi field cannot be true. The update (page 251) command *does not* replace the _id value.

Limits For each update element in the updates array, the sum of the query and the update sizes (i.e. q and u) must be less than or equal to the maximum BSON document size (page 692).

The total number of update statements in the updates array must be less than or equal to the maximum bulk size.

Examples

Update Specific Fields of One Document Use *update operators* (page 451) to update only the specified fields of a document.

For example, given a users collection, the following command uses the $set (page 459) and $inc (page 452) operators to modify the status and the points fields respectively of a document where the user equals "abc123":

```
db.runCommand(
   {
      update: "users",
      updates: [
         {
           q: { user: "abc123" }, u: { $set: { status: "A" }, $inc: { points: 1 } }
         }
      ],
      ordered: false,
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

Because <update> document does not specify the optional multi field, the update only modifies one document, even if more than one document matches the q match condition.

The returned document shows that the command found and updated a single document. See *Output* (page 254) for details.

```
{ "ok" : 1, "nModified" : 1, "n" : 1 }
```

Update Specific Fields of Multiple Documents Use *update operators* (page 451) to update only the specified fields of a document, and include the `multi` field set to `true` in the update statement.

For example, given a `users` collection, the following command uses the `$set` (page 459) and `$inc` (page 452) operators to modify the `status` and the `points` fields respectively of all documents in the collection:

```
db.runCommand(
   {
      update: "users",
      updates: [
         { q: { }, u: { $set: { status: "A" }, $inc: { points: 1 } }, multi: true }
      ],
      ordered: false,
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

The update modifies all documents that match the query specified in the `q` field, namely the empty query which matches all documents in the collection.

The returned document shows that the command found and updated multiple documents. See *Output* (page 254) for details.

```
{ "ok" : 1, "nModified" : 100, "n" : 100 }
```

Bulk Update The following example performs multiple update operations on the `users` collection:

```
db.runCommand(
   {
      update: "users",
      updates: [
         { q: { status: "P" }, u: { $set: { status: "D" } }, multi: true },
         { q: { _id: 5 }, u: { _id: 5, name: "abc123", status: "A" }, upsert: true }
      ],
      ordered: false,
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

The returned document shows that the command modified `10` documents and inserted a document with the `_id` value `5`. See *Output* (page 254) for details.

```
{
   "ok" : 1,
   "nModified" : 10,
   "n" : 11,
   "upserted" : [
      {
         "index" : 1,
         "_id" : 5
      }
   ]
}
```

Output The returned document contains a subset of the following fields:

`update.ok`
: The status of the command.

`update.n`
: The number of documents selected for update. If the update operation results in no change to the document, e.g. `$set` (page 459) expression updates the value to the current value, `n` (page 254) can be greater than `nModified` (page 254).

`update.nModified`
: The number of documents updated. If the update operation results in no change to the document, such as setting the value of the field to its current value, `nModified` (page 254) can be less than `n` (page 254).

`update.upserted`
: An array of documents that contains information for each document inserted through the update with `upsert: true`.

  Each document contains the following information:

  `update.upserted.index`
  : An integer that identifies the update with `upsert:true` statement in the `updates` array, which uses a zero-based index.

  `update.upserted._id`
  : The `_id` value of the added document.

`update.writeErrors`
: An array of documents that contains information regarding any error encountered during the update operation. The `writeErrors` (page 254) array contains an error document for each update statement that errors.

  Each error document contains the following fields:

  `update.writeErrors.index`
  : An integer that identifies the update statement in the `updates` array, which uses a zero-based index.

  `update.writeErrors.code`
  : An integer value identifying the error.

  `update.writeErrors.errmsg`
  : A description of the error.

`update.writeConcernError`
: Document that describe error related to write concern and contains the field:

  `update.writeConcernError.code`
  : An integer value identifying the cause of the write concern error.

  `update.writeConcernError.errmsg`
  : A description of the cause of the write concern error.

The following is an example document returned for a successful `update` (page 251) command that performed an upsert:

```
{
   "ok" : 1,
   "nModified" : 0,
   "n" : 1,
   "upserted" : [
      {
         "index" : 0,
         "_id" : ObjectId("52ccb2118908ccd753d65882")
      }
```

```
   ]
}
```

The following is an example document returned for a bulk update involving three update statements, where one update statement was successful and two other update statements encountered errors:

```
{
   "ok" : 1,
   "nModified" : 1,
   "n" : 1,
   "writeErrors" : [
      {
         "index" : 1,
         "code" : 16837,
         "errmsg" : "The _id field cannot be changed from {_id: 1.0} to {_id: 5.0}."
      },
      {
         "index" : 2,
         "code" : 16837,
         "errmsg" : "The _id field cannot be changed from {_id: 2.0} to {_id: 6.0}."
      },
   ]
}
```

Query Plan Cache Commands

Query Plan Cache Commands

Name	Description
`planCacheClearFilters` (page 255)	Clears index filter(s) for a collection.
`planCacheClear` (page 257)	Removes cached query plan(s) for a collection.
`planCacheListFilters` (page 258)	Lists the index filters for a collection.
`planCacheListPlans` (page 259)	Displays the cached query plans for the specified query shape.
`planCacheListQueryShapes` (page 260)	Displays the query shapes for which cached query plans exist.
`planCacheSetFilter` (page 262)	Sets an index filter for a collection.

planCacheClearFilters

Definition

`planCacheClearFilters`

New in version 2.6.

Removes *index filters* on a collection. Although index filters only exist for the duration of the server process and do not persist after shutdown, you can also clear existing index filters with the `planCacheClearFilters` (page 255) command.

Specify the *query shape* to remove a specific index filter. Omit the query shape to clear all index filters on a collection.

The command has the following syntax:

```
db.runCommand(
   {
      planCacheClearFilters: <collection>,
      query: <query pattern>,
```

```
        sort: <sort specification>,
        projection: <projection specification>
    }
)
```

The `planCacheClearFilters` (page 255) command has the following field:

field string planCacheClearFilters The name of the collection.

field document query The query predicate associated with the filter to remove. If omitted, clears all filters from the collection.

The values in the `query` predicate are insignificant in determining the *query shape*, so the values used in the query need not match the values shown using `planCacheListFilters` (page 258).

field document sort The sort associated with the filter to remove, if any.

field document projection The projection associated with the filter to remove, if any.

Required Access A user must have access that includes the `planCacheIndexFilter` action.

Examples

Clear Specific Index Filter on Collection The `orders` collection contains the following two filters:

```
{
  "query" : { "status" : "A" },
  "sort" : { "ord_date" : -1 },
  "projection" : { },
  "indexes" : [ { "status" : 1, "cust_id" : 1 } ]
}

{
  "query" : { "status" : "A" },
  "sort" : { },
  "projection" : { },
  "indexes" : [ { "status" : 1, "cust_id" : 1 } ]
}
```

The following command removes the second index filter only:

```
db.runCommand(
   {
      planCacheClearFilters: "orders",
      query: { "status" : "A" }
   }
)
```

Because the values in the `query` predicate are insignificant in determining the *query shape*, the following command would also remove the second index filter:

```
db.runCommand(
   {
      planCacheClearFilters: "orders",
      query: { "status" : "P" }
   }
)
```

Clear all Index Filters on a Collection The following example clears all index filters on the `orders` collection:

```
db.runCommand(
   {
      planCacheClearFilters: "orders"
   }
)
```

See also:

`planCacheListFilters` (page 258), `planCacheSetFilter` (page 262)

planCacheClear

Definition

`planCacheClear`

New in version 2.6.

Removes cached query plans for a collection. Specify a *query shape* to remove cached query plans for that shape. Omit the query shape to clear all cached query plans.

The command has the following syntax:

```
db.runCommand(
   {
      planCacheClear: <collection>,
      query: <query>,
      sort: <sort>,
      projection: <projection>
   }
)
```

The `planCacheClear` (page 257) command has the following field:

field document query The query predicate of the *query shape*. Only the structure of the predicate, including the field names, are significant to the shape; the values in the query predicate are insignificant.

field document projection The projection associated with the *query shape*.

field document sort The sort associated with the *query shape*.

To see the query shapes for which cached query plans exist, use the `planCacheListQueryShapes` (page 260) command.

Required Access On systems running with `authorization`, a user must have access that includes the `planCacheWrite` action.

Examples

Clear Cached Plans for a Query Shape If a collection `orders` has the following query shape:

```
{
  "query" : { "qty" : { "$gt" : 10 } },
  "sort" : { "ord_date" : 1 },
  "projection" : { }
}
```

The following operation clears the query plan cached for the shape:

```
db.runCommand(
   {
      planCacheClear: "orders",
      query: { "qty" : { "$gt" : 10 } },
      sort: { "ord_date" : 1 }
   }
)
```

Clear All Cached Plans for a Collection The following example clears all the cached query plans for the `orders` collection:

```
db.runCommand(
   {
      planCacheClear: "orders"
   }
)
```

See also:

- `PlanCache.clearPlansByQuery()` (page 131)
- `PlanCache.clear()` (page 130)

planCacheListFilters

Definition

`planCacheListFilters`

New in version 2.6.

Lists the *index filters* associated with *query shapes* for a collection.

The command has the following syntax:

```
db.runCommand( { planCacheListFilters: <collection> } )
```

The `planCacheListFilters` (page 258) command has the following field:

field string planCacheListFilters The name of the collection.

Returns Document listing the index filters. See *Output* (page 258).

Required Access A user must have access that includes the `planCacheIndexFilter` action.

Output The `planCacheListFilters` (page 258) command returns the document with the following form:

```
{
   "filters" : [
      {
         "query" : <query>
         "sort" : <sort>,
         "projection" : <projection>,
         "indexes" : [
            <index1>,
            ...
```

```
            ]
        },
        ...
    ],
    "ok" : 1
}
```

`planCacheListFilters.`**`filters`**
: The array of documents that contain the index filter information.

  Each document contains the following fields:

  `planCacheListFilters.filters.`**`query`**
  : The query predicate associated with this filter. Although the `query` (page 259) shows the specific values used to create the index filter, the values in the predicate are insignificant; i.e. query predicates cover similar queries that differ only in the values.

    For instance, a `query` (page 259) predicate of `{ "type": "electronics", "status" : "A" }` covers the following query predicates:

    ```
    { type: "food", status: "A" }
    { type: "utensil", status: "D" }
    ```

    Together with the `sort` (page 259) and the `projection` (page 259), the `query` (page 259) make up the *query shape* for the specified index filter.

  `planCacheListFilters.filters.`**`sort`**
  : The sort associated with this filter. Can be an empty document.

    Together with the `query` (page 259) and the `projection` (page 259), the `sort` (page 259) make up the *query shape* for the specified index filter.

  `planCacheListFilters.filters.`**`projection`**
  : The projection associated with this filter. Can be an empty document.

    Together with the `query` (page 259) and the `sort` (page 259), the `projection` (page 259) make up the *query shape* for the specified index filter.

  `planCacheListFilters.filters.`**`indexes`**
  : The array of indexes for this *query shape*. To choose the optimal query plan, the query optimizer evaluates only the listed `indexes` *and* the collection scan.

`planCacheListFilters.`**`ok`**
: The status of the command.

See also:

`planCacheClearFilters` (page 255), `planCacheSetFilter` (page 262)

planCacheListPlans

Definition

`planCacheListPlans`
: New in version 2.6.

  Displays the cached query plans for the specified *query shape*.

  The query optimizer only caches the plans for those query shapes that can have more than one viable plan.

  The `mongo` (page 610) shell provides the wrapper `PlanCache.getPlansByQuery()` (page 132) for this command.

The planCacheListPlans (page 259) command has the following syntax:

```
db.runCommand(
   {
      planCacheListPlans: <collection>,
      query: <query>,
      sort: <sort>,
      projection: <projection>
   }
```

The planCacheListPlans (page 259) command has the following field:

field document query The query predicate of the *query shape*. Only the structure of the predicate, including the field names, are significant to the shape; the values in the query predicate are insignificant.

field document projection The projection associated with the *query shape*.

field document sort The sort associated with the *query shape*.

To see the query shapes for which cached query plans exist, use the planCacheListQueryShapes (page 260) command.

Required Access On systems running with authorization, a user must have access that includes the planCacheRead action.

Example If a collection orders has the following query shape:

```
{
  "query" : { "qty" : { "$gt" : 10 } },
  "sort" : { "ord_date" : 1 },
  "projection" : { }
}
```

The following operation displays the query plan cached for the shape:

```
db.runCommand(
   {
      planCacheListPlans: "orders",
      query: { "qty" : { "$gt" : 10 } },
      sort: { "ord_date" : 1 }
   }
)
```

See also:

- planCacheListQueryShapes (page 260)
- PlanCache.getPlansByQuery() (page 132)
- PlanCache.listQueryShapes() (page 133)

planCacheListQueryShapes

Definition

planCacheListQueryShapes

New in version 2.6.

Displays the *query shapes* for which cached query plans exist for a collection.

The query optimizer only caches the plans for those query shapes that can have more than one viable plan.

The `mongo` (page 610) shell provides the wrapper `PlanCache.listQueryShapes()` (page 133) for this command.

The command has the following syntax:

```
db.runCommand(
   {
      planCacheListQueryShapes: <collection>
   }
```

The `planCacheListQueryShapes` (page 260) command has the following field:

field string planCacheListQueryShapes The name of the collection.

Returns A document that contains an array of *query shapes* for which cached query plans exist.

Required Access On systems running with `authorization`, a user must have access that includes the `planCacheRead` action.

Example The following returns the *query shapes* that have cached plans for the `orders` collection:

```
db.runCommand(
   {
      planCacheListQueryShapes: "orders"
   }
)
```

The command returns a document that contains the field `shapes` that contains an array of the *query shapes* currently in the cache. In the example, the `orders` collection had cached query plans associated with the following shapes:

```
{
  "shapes" : [
     {
       "query" : { "qty" : { "$gt" : 10 } },
        "sort" : { "ord_date" : 1 },
        "projection" : { }
     },
     {
        "query" : { "$or" : [ { "qty" : { "$gt" : 15 } }, { "status" : "A" } ] },
        "sort" : { },
        "projection" : { }
     },
     {
       "query" : { "$or" :
          [
            { "qty" : { "$gt" : 15 }, "item" : "xyz123" },
            { "status" : "A" }
          ]
        },
        "sort" : { },
        "projection" : { }
      }
   ],
   "ok" : 1
}
```

See also:

- PlanCache.listQueryShapes() (page 133)

planCacheSetFilter

Definition

planCacheSetFilter

New in version 2.6.

Set an *index filter* for a collection. If an index filter already exists for the *query shape*, the command overrides the previous index filter.

The command has the following syntax:

```
db.runCommand(
   {
      planCacheSetFilter: <collection>,
      query: <query>,
      sort: <sort>,
      projection: <projection>,
      indexes: [ <index1>, <index2>, ...]
   }
)
```

The planCacheSetFilter (page 262) command has the following field:

field string planCacheSetFilter The name of the collection.

field document query The query predicate associated with the index filter. Together with the sort and the projection, the query predicate make up the *query shape* for the specified index filter.

Only the structure of the predicate, including the field names, are significant; the values in the query predicate are insignificant. As such, query predicates cover similar queries that differ only in the values.

field document sort The sort associated with the filter. Together with the query and the projection, the sort make up the *query shape* for the specified index filter.

field document projection The projection associated with the filter. Together with the query and the sort, the projection make up the *query shape* for the specified index filter.

field array indexes An array of index specification documents that act as the index filter for the specified *query shape*. Because the query optimizer chooses among the collection scan and these indexes, if the indexes are non-existent, the optimizer will choose the collection scan.

Index filters only exist for the duration of the server process and do not persist after shutdown; however, you can also clear existing index filters using the planCacheClearFilters (page 255) command.

Required Access A user must have access that includes the planCacheIndexFilter action.

Examples The following example creates an index filter on the orders collection such that for queries that consists only of an equality match on the status field without any projection and sort, the query optimizer evaluates only the two specified indexes and the collection scan for the winning plan:

```
db.runCommand(
   {
      planCacheSetFilter: "orders",
      query: { status: "A" },
```

```
      indexes: [
         { cust_id: 1, status: 1 },
         { status: 1, order_date: -1 }
      ]
   }
)
```

In the query predicate, only the structure of the predicate, including the field names, are significant; the values are insignificant. As such, the created filter applies to the following operations:

```
db.orders.find( { status: "D" } )
db.orders.find( { status: "P" } )
```

See also:

planCacheClearFilters (page 255), planCacheListFilters (page 258)

2.2.2 Database Operations

Authentication Commands

Authentication Commands

Name	Description
authSchemaUpgrade (page 263)	Supports the upgrade process for user data between version 2.4 and 2.6.
authenticate (page 263)	Starts an authenticated session using a username and password.
copydbgetnonce (page 264)	This is an internal command to generate a one-time password for use with the copydb (page 326) command.
getnonce (page 264)	This is an internal command to generate a one-time password for authentication.
logout (page 264)	Terminates the current authenticated session.

authSchemaUpgrade New in version 2.6.

authSchemaUpgrade

authSchemaUpgrade (page 263) supports the upgrade from 2.4 to 2.6 process for existing systems that use *authentication* and *authorization*. Between 2.4 and 2.6 the schema for user credential documents changed requiring the authSchemaUpgrade (page 263) process.

See *Upgrade User Authorization Data to 2.6 Format* (page 764) for more information.

authenticate

authenticate

Clients use authenticate (page 263) to authenticate a connection. When using the shell, use the db.auth() (page 100) helper as follows:

```
db.auth( "username", "password" )
```

See

db.auth() (page 100) and http://docs.mongodb.org/manual/core/security for more information.

---

copydbgetnonce

copydbgetnonce

Client libraries use copydbgetnonce (page 264) to get a one-time password for use with the copydb (page 326) command.

Note: This command obtains a write lock on the affected database and will block other operations until it has completed; however, the write lock for this operation is short lived.

---

getnonce

getnonce

Client libraries use getnonce (page 264) to generate a one-time password for authentication.

logout

logout

The logout (page 264) command terminates the current authenticated session:

```
{ logout: 1 }
```

Note: If you're not logged in and using authentication, logout (page 264) has no effect.

Changed in version 2.4: Because MongoDB now allows users defined in one database to have privileges on another database, you must call logout (page 264) while using the same database context that you authenticated to.

If you authenticated to a database such as users or $external, you must issue logout (page 264) against this database in order to successfully log out.

Example

Use the use <database-name> helper in the interactive mongo (page 610) shell, or the following db.getSiblingDB() (page 118) in the interactive shell or in mongo (page 610) shell scripts to change the db object:

```
db = db.getSiblingDB('<database-name>')
```

When you have set the database context and db object, you can use the logout (page 264) to log out of database as in the following operation:

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

User Management Commands

User Management Commands

Name	Description
createUser (page 265)	Creates a new user.
dropAllUsersFromDatabase (page 266)	Deletes all users associated with a database.
dropUser (page 267)	Removes a single user.
grantRolesToUser (page 267)	Grants a role and its privileges to a user.
revokeRolesFromUser (page 269)	Removes a role from a user.
updateUser (page 270)	Updates a user's data.
usersInfo (page 272)	Returns information about the specified users.

createUser

Definition

createUser

Creates a new user on the database where you run the command. The createUser (page 265) command returns a *duplicate user* error if the user exists. The createUser (page 265) command uses the following syntax:

```
{ createUser: "<name>",
  pwd: "<cleartext password>",
  customData: { <any information> },
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  writeConcern: { <write concern> }
}
```

createUser (page 265) has the following fields:

field string createUser The name of the new user.

field string pwd The user's password. The pwd field is not required if you run createUser (page 265) on the $external database to create users who have credentials stored externally to MongoDB.

any document customData Any arbitrary information. This field can be used to store any data an admin wishes to associate with this particular user. For example, this could be the user's full name or employee id.

field array roles The roles granted to the user. Can specify an empty array [] to create users without roles.

field boolean digestPassword When true, the mongod (page 583) instance will create the hash of the user password; otherwise, the client is responsible for creating the hash of the password. Defaults to true.

field document writeConcern The level of write concern for the creation operation. The writeConcern document takes the same fields as the getLastError (page 245) command.

In the roles field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where createUser (page 265) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

Behavior

Encryption createUser (page 265) sends password to the MongoDB instance in cleartext. To encrypt the password in transit, use SSL.

External Credentials Users created on the `$external` database should have credentials stored externally to MongoDB, as, for example, with `MongoDB Enterprise installations that use Kerberos`.

`local` Database You cannot create users on the local database.

Required Access You must have the `createUser` *action* on a database to create a new user on that database.

You must have the `grantRole` *action* on a role's database to grant the role to another user.

If you have the `userAdmin` or `userAdminAnyDatabase` role, you have those actions.

Example The following `createUser` (page 265) command creates a user `accountAdmin01` on the `products` database. The command gives `accountAdmin01` the `clusterAdmin` and `readAnyDatabase` roles on the `admin` database and the `readWrite` role on the `products` database:

```
db.getSiblingDB("products").runCommand( { createUser: "accountAdmin01",
                                          pwd: "cleartext password",
                                          customData: { employeeId: 12345 },
                                          roles: [
                                                   { role: "clusterAdmin", db: "admin" },
                                                   { role: "readAnyDatabase", db: "admin" },
                                                   "readWrite"
                                                 ],
                                          writeConcern: { w: "majority" , wtimeout: 5000 }
                                        } )
```

dropAllUsersFromDatabase

Definition

`dropAllUsersFromDatabase`

Removes all users from the database on which you run the command.

> **Warning:** The `dropAllUsersFromDatabase` (page 266) removes all users from the database.

The `dropAllUsersFromDatabase` (page 266) command has the following syntax:

```
{ dropAllUsersFromDatabase: 1,
  writeConcern: { <write concern> }
}
```

The `dropAllUsersFromDatabase` (page 266) document has the following fields:

field integer dropAllUsersFromDatabase Specify `1` to drop all the users from the current database.

field document writeConcern The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

Required Access You must have the `dropUser` *action* on a database to drop a user from that database.

Example The following sequence of operations in the mongo (page 610) shell drops every user from the products database:

```
use products
db.runCommand( { dropAllUsersFromDatabase: 1, writeConcern: { w: "majority" } } )
```

The n field in the results document shows the number of users removed:

```
{ "n" : 12, "ok" : 1 }
```

dropUser

Definition

dropUser

Removes the user from the database on which you run the command. The dropUser (page 267) command has the following syntax:

```
{
  dropUser: "<user>",
  writeConcern: { <write concern> }
}
```

The dropUser (page 267) command document has the following fields:

field string dropUser The name of the user to delete. You must issue the dropUser (page 267) command while using the database where the user exists.

field document writeConcern The level of write concern for the removal operation. The writeConcern document takes the same fields as the getLastError (page 245) command.

Required Access You must have the dropUser *action* on a database to drop a user from that database.

Example The following sequence of operations in the mongo (page 610) shell removes accountAdmin01 from the products database:

```
use products
db.runCommand( { dropUser: "accountAdmin01",
                 writeConcern: { w: "majority", wtimeout: 5000 }
                 } )
```

grantRolesToUser

Definition

grantRolesToUser

Grants additional roles to a user.

The grantRolesToUser (page 267) command uses the following syntax:

```
{ grantRolesToUser: "<user>",
  roles: [ <roles> ],
  writeConcern: { <write concern> }
}
```

The command has the following fields:

field string grantRolesToUser The name of the user to give additional roles.

field array roles An array of additional roles to grant to the user.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `grantRolesToUser` (page 267) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

Required Access You must have the `grantRole` *action* on a database to grant a role on that database.

Example Given a user `accountUser01` in the `products` database with the following roles:

```
"roles" : [
    { "role" : "assetsReader",
      "db" : "assets"
    }
]
```

The following `grantRolesToUser` (page 267) operation gives `accountUser01` the `read` role on the `stock` database and the `readWrite` role on the `products` database.

```
use products
db.runCommand( { grantRolesToUser: "accountUser01",
                 roles: [
                    { role: "read", db: "stock"},
                    "readWrite"
                 ],
                 writeConcern: { w: "majority" , wtimeout: 2000 }
               } )
```

The user `accountUser01` in the `products` database now has the following roles:

```
"roles" : [
    { "role" : "assetsReader",
      "db" : "assets"
    },
    { "role" : "read",
      "db" : "stock"
    },
    { "role" : "readWrite",
      "db" : "products"
    }
]
```

revokeRolesFromUser

Definition

revokeRolesFromUser

Removes a one or more roles from a user on the database where the roles exist. The `revokeRolesFromUser` (page 269) command uses the following syntax:

```
{ revokeRolesFromUser: "<user>",
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  writeConcern: { <write concern> }
}
```

The command has the following fields:

field string revokeRolesFromUser The user to remove roles from.

field array roles The roles to remove from the user.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `revokeRolesFromUser` (page 269) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

Required Access You must have the `revokeRole` *action* on a database to revoke a role on that database.

Example The `accountUser01` user in the `products` database has the following roles:

```
"roles" : [
    { "role" : "assetsReader",
      "db" : "assets"
    },
    { "role" : "read",
      "db" : "stock"
    },
    { "role" : "readWrite",
      "db" : "products"
    }
]
```

The following `revokeRolesFromUser` (page 269) command removes the two of the user's roles: the `read` role on the `stock` database and the `readWrite` role on the `products` database, which is also the database on which the command runs:

```
use products
db.runCommand( { revokeRolesFromUser: "accountUser01",
                 roles: [
                          { role: "read", db: "stock" },
```

```
                    "readWrite"
                ],
                writeConcern: { w: "majority" }
            } )
```

The user `accountUser01` in the `products` database now has only one remaining role:

```
"roles" : [
    { "role" : "assetsReader",
      "db" : "assets"
    }
]
```

updateUser

Definition

`updateUser`

Updates the user's profile on the database on which you run the command. An update to a field **completely replaces** the previous field's values, including updates to the user's `roles` array.

> **Warning:** When you update the `roles` array, you completely replace the previous array's values. To add or remove roles without replacing all the user's existing roles, use the `grantRolesToUser` (page 267) or `revokeRolesFromUser` (page 269) commands.

The `updateUser` (page 270) command uses the following syntax. To update a user, you must specify the `updateUser` field and at least one other field, other than `writeConcern`:

```
{ updateUser: "<username>",
  pwd: "<cleartext password>",
  customData: { <any information> },
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  writeConcern: { <write concern> }
}
```

The command has the following fields:

field string updateUser The name of the user to update.

field string pwd The user's password.

field document customData Any arbitrary information.

field array roles The roles granted to the user. An update to the `roles` array overrides the previous array's values.

field boolean digestPassword When `true`, the `mongod` (page 583) instance will create the hash of the user password; otherwise, the client is responsible for creating the hash of the password. Defaults to `true`.

field document writeConcern The level of `write concern` for the update operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `updateUser` (page 270) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

Behavior `updateUser` (page 270) sends the password to the MongoDB instance in cleartext. To encrypt the password in transit, use `SSL`.

Required Access You must have access that includes the `revokeRole` *action* on all databases in order to update a user's `roles` array.

You must have the `grantRole` *action* on a role's database to add a role to a user.

To change another user's `pwd` or `customData` field, you must have the `changeAnyPassword` and `changeAnyCustomData` *actions* respectively on that user's database.

To modify your own password and custom data, you must have privileges that grant `changeOwnPassword` and `changeOwnCustomData` *actions* respectively on the user's database.

Example Given a user `appClient01` in the `products` database with the following user info:

```
{
   "_id" : "products.appClient01",
   "user" : "appClient01",
   "db" : "products",
   "customData" : { "empID" : "12345", "badge" : "9156" },
   "roles" : [
       { "role" : "readWrite",
         "db" : "products"
       },
       { "role" : "read",
         "db" : "inventory"
       }
   ]
}
```

The following `updateUser` (page 270) command **completely** replaces the user's `customData` and `roles` data:

```
use products
db.runCommand( { updateUser : "appClient01",
                 customData : { employeeId : "0x3039" },
                 roles : [
                           { role : "read", db : "assets"  }
                         ]
             } )
```

The user `appClient01` in the `products` database now has the following user information:

```
{
   "_id" : "products.appClient01",
   "user" : "appClient01",
   "db" : "products",
```

```
   "customData" : { "employeeId" : "0x3039" },
   "roles" : [
       { "role" : "read",
         "db" : "assets"
       }
   ]
}
```

usersInfo

Definition

usersInfo

Returns information about one or more users. To match a single user on the database, use the following form:

```
{ usersInfo: { user: <name>, db: <db> },
  showCredentials: <Boolean>,
  showPrivileges: <Boolean>
}
```

The command has the following fields:

field various usersInfo The user(s) about whom to return information. See *Behavior* (page 272) for type and syntax.

field Boolean showCredentials Set the field to true to display the user's password hash. By default, this field is `false`.

field Boolean showPrivileges Set the field to true to show the user's full set of privileges, including expanded information for the inherited roles. By default, this field is `false`. If viewing all users, you cannot specify this field.

Required Access Users can always view their own information.

To view another user's information, the user running the command must have privileges that include the `viewUser` action on the other user's database.

Behavior The argument to the `usersInfo` (page 272) command has multiple forms depending on the requested information:

Specify a Single User In the `usersInfo` field, specify a document with the user's name and database:

```
{ usersInfo: { user: <name>, db: <db> } }
```

Alternatively, for a user that exists on the same database where the command runs, you can specify the user by its name only.

```
{ usersInfo: <name> }
```

Specify Multiple Users In the `usersInfo` field, specify an array of documents:

```
{ usersInfo: [ { user: <name>, db: <db> },  { user: <name>, db: <db> }, ... ] }
```

Specify All Users for a Database In the `usersInfo` field, specify `1`:

```
{ usersInfo: 1 }
```

Examples

View Specific Users To see information and privileges, but not the credentials, for the user `"Kari"` defined in `"home"` database, run the following command:

```
db.runCommand(
                {
                  usersInfo:  { user: "Kari", db: "home" },
                  showPrivileges: true
                }
)
```

To view a user that exists in the *current* database, you can specify the user by name only. For example, if you are in the `home` database and a user named `"Kari"` exists in the `home` database, you can run the following command:

```
db.getSiblingDB("home").runCommand(
                                     {
                                       usersInfo:  "Kari",
                                       showPrivileges: true
                                     }
)
```

View Multiple Users To view info for several users, use an array, with or without the optional fields `showPrivileges` and `showCredentials`. For example:

```
db.runCommand( { usersInfo: [ { user: "Kari", db: "home" }, { user: "Li", db: "myApp" } ],
                  showPrivileges: true
              } )
```

View All Users for a Database To view all users on the database the command is run, use a command document that resembles the following:

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

When viewing all users, you can specify the `showCredentials` field but not the `showPrivileges` field.

Role Management Commands

Role Management Commands

Name	Description
createRole (page 274)	Creates a role and specifies its privileges.
dropAllRolesFromDatabase (page 275)	Deletes all user-defined roles from a database.
dropRole (page 276)	Deletes the user-defined role.
grantPrivilegesToRole (page 277)	Assigns privileges to a user-defined role.
grantRolesToRole (page 278)	Specifies roles from which a user-defined role inherits privileges.
invalidateUserCache (page 279)	Flushes the in-memory cache of user information, including credentials and roles.
revokePrivilegesFromRole (page 279)	Removes the specified privileges from a user-defined role.
revokeRolesFromRole (page 282)	Removes specified inherited roles from a user-defined role.
rolesInfo (page 283)	Returns information for the specified role or roles.
updateRole (page 286)	Updates a user-defined role.

createRole

Definition

createRole

Creates a role and specifies its *privileges*. The role applies to the database on which you run the command. The createRole (page 274) command returns a *duplicate role* error if the role already exists in the database.

The createRole (page 274) command uses the following syntax:

```
{ createRole: "<new role>",
  privileges: [
    { resource: { <resource> }, actions: [ "<action>", ... ] },
    ...
  ],
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  writeConcern: <write concern document>
}
```

The createRole (page 274) command has the following fields:

field string createRole The name of the new role.

field array privileges The privileges to grant the role. A privilege consists of a resource and permitted actions. You must specify the privileges field. Use an empty array to specify *no* privileges. For the syntax of a privilege, see the privileges array.

field array roles An array of roles from which this role inherits privileges. You must specify the roles field. Use an empty array to specify *no* roles.

field document writeConcern The level of write concern to apply to this operation. The writeConcern document uses the same fields as the getLastError (page 245) command.

In the roles field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `createRole` (page 274) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

Behavior A role's privileges apply to the database where the role is created. The role can inherit privileges from other roles in its database. A role created on the `admin` database can include privileges that apply to all databases or to the *cluster* and can inherit privileges from roles in other databases.

Required Access To create a role in a database, the user must have:

- the `createRole` *action* on that *database resource*.
- the `grantRole` *action* on that database to specify privileges for the new role as well as to specify roles to inherit from.

Built-in roles `userAdmin` and `userAdminAnyDatabase` provide `createRole` and `grantRole` actions on their respective `resources`.

Example The following `createRole` (page 274) command creates the `myClusterwideAdmin` role on the `admin` database:

```
use admin
db.runCommand({ createRole: "myClusterwideAdmin",
  privileges: [
    { resource: { cluster: true }, actions: [ "addShard" ] },
    { resource: { db: "config", collection: "" }, actions: [ "find", "update", "insert", "remove" ] }
    { resource: { db: "users", collection: "usersCollection" }, actions: [ "update", "insert", "remov
    { resource: { db: "", collection: "" }, actions: [ "find" ] }
  ],
  roles: [
    { role: "read", db: "admin" }
  ],
  writeConcern: { w: "majority" , wtimeout: 5000 }
})
```

dropAllRolesFromDatabase

Definition

`dropAllRolesFromDatabase`

Deletes all *user-defined* roles on the database where you run the command.

> **Warning:** The `dropAllRolesFromDatabase` (page 275) removes *all user-defined* roles from the database.

The `dropAllRolesFromDatabase` (page 275) command takes the following form:

```
{
  dropAllRolesFromDatabase: 1,
  writeConcern: { <write concern> }
}
```

The command has the following fields:

field integer dropAllRolesFromDatabase Specify `1` to drop all *user-defined* roles from the database where the command is run.

field document writeConcern The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

Required Access You must have the `dropRole` *action* on a database to drop a role from that database.

Example The following operations drop all *user-defined* roles from the `products` database:

```
use products
db.runCommand(
   {
     dropAllRolesFromDatabase: 1,
     writeConcern: { w: "majority" }
   }
)
```

The `n` field in the results document reports the number of roles dropped:

```
{ "n" : 4, "ok" : 1 }
```

dropRole

Definition

`dropRole`

Deletes a *user-defined* role from the database on which you run the command.

The `dropRole` (page 276) command uses the following syntax:

```
{
  dropRole: "<role>",
  writeConcern: { <write concern> }
}
```

The `dropRole` (page 276) command has the following fields:

field string dropRole The name of the *user-defined role* to remove from the database.

field document writeConcern The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

Required Access You must have the `dropRole` *action* on a database to drop a role from that database.

Example The following operations remove the `readPrices` role from the `products` database:

```
use products
db.runCommand(
   {
     dropRole: "readPrices",
     writeConcern: { w: "majority" }
   }
)
```

grantPrivilegesToRole

Definition

`grantPrivilegesToRole`

Assigns additional *privileges* to a *user-defined* role defined on the database on which the command is run. The `grantPrivilegesToRole` (page 277) command uses the following syntax:

```
{
  grantPrivilegesToRole: "<role>",
  privileges: [
      {
        resource: { <resource> }, actions: [ "<action>", ... ]
      },
      ...
  ],
  writeConcern: { <write concern> }
}
```

The `grantPrivilegesToRole` (page 277) command has the following fields:

field string grantPrivilegesToRole The name of the user-defined role to grant privileges to.

field array privileges The privileges to add to the role. For the format of a privilege, see `privileges`.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

Behavior A role's privileges apply to the database where the role is created. A role created on the `admin` database can include privileges that apply to all databases or to the *cluster*.

Required Access You must have the `grantRole` *action* on the database a privilege targets in order to grant the privilege. To grant a privilege on multiple databases or on the `cluster` resource, you must have the `grantRole` action on the `admin` database.

Example The following `grantPrivilegesToRole` (page 277) command grants two additional privileges to the `service` role that exists in the `products` database:

```
use products
db.runCommand(
   {
     grantPrivilegesToRole: "service",
     privileges: [
         {
```

```
            resource: { db: "products", collection: "" }, actions: [ "find" ]
          },
          {
            resource: { db: "products", collection: "system.indexes" }, actions: [ "find" ]
          }
      ],
      writeConcern: { w: "majority" , wtimeout: 5000 }
    }
)
```

The first privilege in the `privileges` array allows the user to search on all non-system collections in the `products` database. The privilege does not allow searches on *system collections* (page 688), such as the `system.indexes` (page 689) collection. To grant access to these system collections, explicitly provision access in the `privileges` array. See `http://docs.mongodb.org/manual/reference/resource-document`.

The second privilege explicitly allows the `find` action on `system.indexes` (page 689) collections on all databases.

grantRolesToRole

Definition

`grantRolesToRole`

Grants roles to a *user-defined role*.

The `grantRolesToRole` (page 278) command affects roles on the database where the command runs. `grantRolesToRole` (page 278) has the following syntax:

```
{ grantRolesToRole: "<role>",
  roles: [
              { role: "<role>", db: "<database>" },
              ...
         ],
  writeConcern: { <write concern> }
}
```

The `grantRolesToRole` (page 278) command has the following fields:

field string grantRolesToRole The name of a role to add subsidiary roles.

field array roles An array of roles from which to inherit.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `grantRolesToRole` (page 278) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

Behavior A role can inherit privileges from other roles in its database. A role created on the `admin` database can inherit privileges from roles in any database.

Required Access You must have the `grantRole` *action* on a database to grant a role on that database.

Example The following `grantRolesToRole` (page 278) command updates the `productsReaderWriter` role in the `products` database to *inherit* the *privileges* of the `productsReader` role in the `products` database:

```
use products
db.runCommand(
   { grantRolesToRole: "productsReaderWriter",
     roles: [
              "productsReader"
     ],
     writeConcern: { w: "majority" , wtimeout: 5000 }
   }
)
```

invalidateUserCache

Definition

`invalidateUserCache`

New in version 2.6.

Flushes user information from in-memory cache, including removal of each user's credentials and roles. This allows you to purge the cache at any given moment, regardless of the interval set in the `userCacheInvalidationIntervalSecs` parameter.

`invalidateUserCache` (page 279) has the following syntax:

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

Required Access You must have privileges that include the `invalidateUserCache` action on the cluster resource in order to use this command.

revokePrivilegesFromRole

Definition

`revokePrivilegesFromRole`

Removes the specified privileges from the *user-defined* role on the database where the command is run. The `revokePrivilegesFromRole` (page 279) command has the following syntax:

```
{
  revokePrivilegesFromRole: "<role>",
  privileges:
      [
        { resource: { <resource> }, actions: [ "<action>", ... ] },
        ...
      ],
  writeConcern: <write concern document>
}
```

The `revokePrivilegesFromRole` (page 279) command has the following fields:

field string revokePrivilegesFromRole The *user-defined* role to revoke privileges from.

field array privileges An array of privileges to remove from the role. See `privileges` for more information on the format of the privileges.

field document writeConcern The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

Behavior To revoke a privilege, the `resource document` pattern must match **exactly** the `resource` field of that privilege. The `actions` field can be a subset or match exactly.

For example, consider the role `accountRole` in the `products` database with the following privilege that specifies the `products` database as the resource:

```
{
  "resource" : {
      "db" : "products",
      "collection" : ""
  },
  "actions" : [
      "find",
      "update"
  ]
}
```

You *cannot* revoke `find` and/or `update` from just *one* collection in the `products` database. The following operations result in no change to the role:

```
use products
db.runCommand(
    {
      revokePrivilegesFromRole: "accountRole",
      privileges:
        [
          {
            resource : {
                db : "products",
                collection : "gadgets"
            },
            actions : [
                "find",
                "update"
            ]
          }
        ]
    }
)

db.runCommand(
    {
      revokePrivilegesFromRole: "accountRole",
      privileges:
        [
          {
            resource : {
                db : "products",
                collection : "gadgets"
            },
```

```
            actions : [
                "find"
            ]
          }
        ]
    }
)
```

To revoke the `"find"` and/or the `"update"` action from the role `accountRole`, you must match the resource document exactly. For example, the following operation revokes just the `"find"` action from the existing privilege.

```
use products
db.runCommand(
    {
      revokePrivilegesFromRole: "accountRole",
      privileges:
        [
          {
            resource : {
                db : "products",
                collection : ""
            },
            actions : [
                "find"
            ]
          }
        ]
    }
)
```

Required Access You must have the `revokeRole` *action* on the database a privilege targets in order to revoke that privilege. If the privilege targets multiple databases or the `cluster` resource, you must have the `revokeRole` action on the `admin` database.

Example The following operation removes multiple privileges from the `associates` role in the `products` database:

```
use products
db.runCommand(
   {
     revokePrivilegesFromRole: "associate",
     privileges:
      [
        {
          resource: { db: "products", collection: "" },
          actions: [ "createCollection", "createIndex", "find" ]
        },
        {
          resource: { db: "products", collection: "orders" },
          actions: [ "insert" ]
        }
      ],
     writeConcern: { w: "majority" }
   }
)
```

revokeRolesFromRole

Definition

revokeRolesFromRole

Removes the specified inherited roles from a role. The `revokeRolesFromRole` (page 282) command has the following syntax:

```
{ revokeRolesFromRole: "<role>",
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  writeConcern: { <write concern> }
}
```

The command has the following fields:

field string revokeRolesFromRole The role from which to remove inherited roles.

field array roles The inherited roles to remove.

field document writeConcern The level of `write concern` to apply to this operation. The `writeConcern` document uses the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `revokeRolesFromRole` (page 282) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

Required Access You must have the `revokeRole` *action* on a database to revoke a role on that database.

Example The `purchaseAgents` role in the `emea` database inherits privileges from several other roles, as listed in the `roles` array:

```
{
   "_id" : "emea.purchaseAgents",
   "role" : "purchaseAgents",
   "db" : "emea",
   "privileges" : [],
   "roles" : [
      {
         "role" : "readOrdersCollection",
         "db" : "emea"
      },
      {
         "role" : "readAccountsCollection",
         "db" : "emea"
      },
      {
         "role" : "writeOrdersCollection",
```

```
         "db" : "emea"
      }
   ]
}
```

The following `revokeRolesFromRole` (page 282) operation on the `emea` database removes two roles from the `purchaseAgents` role:

```
use emea
db.runCommand( { revokeRolesFromRole: "purchaseAgents",
                 roles: [
                          "writeOrdersCollection",
                          "readOrdersCollection"
                        ],
                  writeConcern: { w: "majority" , wtimeout: 5000 }
               } )
```

The `purchaseAgents` role now contains just one role:

```
{
   "_id" : "emea.purchaseAgents",
   "role" : "purchaseAgents",
   "db" : "emea",
   "privileges" : [],
   "roles" : [
      {
         "role" : "readAccountsCollection",
         "db" : "emea"
      }
   ]
}
```

rolesInfo

Definition

`rolesInfo`

Returns inheritance and privilege information for specified roles, including both *user-defined roles* and *built-in roles*.

The `rolesInfo` (page 283) command can also retrieve all roles scoped to a database.

The command has the following fields:

field string,document,array,integer rolesInfo The role(s) to return information about. For the syntax for specifying roles, see *Behavior* (page 283).

field Boolean showPrivileges Set the field to `true` to show role privileges, including both privileges inherited from other roles and privileges defined directly. By default, the command returns only the roles from which this role inherits privileges and does not return specific privileges.

field Boolean showBuiltinRoles When the `rolesInfo` field is set to `1`, set `showBuiltinRoles` to `true` to include *built-in roles* in the output. By default this field is set to `false`, and the output for `rolesInfo: 1` displays only *user-defined roles*.

Behavior

Return Information for a Single Role To specify a role from the current database, specify the role by its name:

```
{ rolesInfo: "<rolename>" }
```

To specify a role from another database, specify the role by a document that specifies the role and database:

```
{ rolesInfo: { role: "<rolename>", db: "<database>" } }
```

Return Information for Multiple Roles To specify multiple roles, use an array. Specify each role in the array as a document or string. Use a string only if the role exists on the database on which the command runs:

```
{
  rolesInfo: [
     "<rolename>",
     { role: "<rolename>", db: "<database>" },
     ...
  ]
}
```

Return Information for All Roles in the Database To specify all roles in the database on which the command runs, specify `rolesInfo: 1`. By default MongoDB displays all the *user-defined roles* in the database. To include *built-in roles* as well, include the parameter-value pair `showBuiltinRoles: true`:

```
{ rolesInfo: 1, showBuiltinRoles: true }
```

Required Access To view a role's information, you must be explicitly granted the role or must have the `viewRole` *action* on the role's database.

Output

`rolesInfo.role`
: The name of the role.

`rolesInfo.db`
: The database on which the role is defined. Every database has *built-in roles*. A database might also have *user-defined roles*.

`rolesInfo.isBuiltin`
: A value of `true` indicates the role is a *built-in role*. A value of `false` indicates the role is a *user-defined role*.

`rolesInfo.roles`
: The roles that directly provide privileges to this role and the databases on which the roles are defined.

`rolesInfo.inheritedRoles`
: All roles from which this role inherits privileges. This includes the roles in the `rolesInfo.roles` (page 284) array as well as the roles from which the roles in the `rolesInfo.roles` (page 284) array inherit privileges. All privileges apply to the current role. The documents in this field list the roles and the databases on which they are defined.

`rolesInfo.privileges`
: The privileges directly specified by this role; i.e. the array excludes privileges inherited from other roles. By default the output does not include the `privileges` (page 284) field. To include the field, specify `showPrivileges: true` when running the `rolesInfo` (page 283) command.

  Each privilege document specifies the *resources* and the `actions` allowed on the resources.

rolesInfo.**inheritedPrivileges**
: All privileges granted by this role, including those inherited from other roles. By default the output does not include the `inheritedPrivileges` (page 284) field. To include the field, specify `showPrivileges: true` when running the `rolesInfo` (page 283) command.

  Each privilege document specifies the *resources* and the `actions` allowed on the resources.

Examples

View Information for a Single Role The following command returns the role inheritance information for the role `associate` defined in the `products` database:

```
db.runCommand(
    {
      rolesInfo: { role: "associate", db: "products" }
    }
)
```

The following command returns the role inheritance information for the role `siteManager` on the database on which the command runs:

```
db.runCommand(
    {
      rolesInfo: "siteManager"
    }
)
```

The following command returns *both* the role inheritance and the privileges for the role `associate` defined on the `products` database:

```
db.runCommand(
    {
      rolesInfo: { role: "associate", db: "products" },
      showPrivileges: true
    }
)
```

View Information for Several Roles The following command returns information for two roles on two different databases:

```
db.runCommand(
    {
      rolesInfo: [
         { role: "associate", db: "products" },
         { role: "manager", db: "resources" }
      ]
    }
)
```

The following returns *both* the role inheritance and the privileges:

```
db.runCommand(
    {
      rolesInfo: [
         { role: "associate", db: "products" },
         { role: "manager", db: "resources" }
      ],
```

```
      showPrivileges: true
    }
)
```

View All User-Defined Roles for a Database The following operation returns all *user-defined roles* on the database on which the command runs and includes privileges:

```
db.runCommand(
    {
      rolesInfo: 1,
      showPrivileges: true
    }
)
```

View All User-Defined and Built-In Roles for a Database The following operation returns all roles on the database on which the command runs, including both built-in and user-defined roles:

```
db.runCommand(
    {
      rolesInfo: 1,
      showBuiltinRoles: true
    }
)
```

updateRole

Definition

updateRole

Updates a *user-defined role*. The updateRole (page 286) command must run on the role's database.

An update to a field **completely replaces** the previous field's values. To grant or remove roles or *privileges* without replacing all values, use one or more of the following commands:

- grantRolesToRole (page 278)
- grantPrivilegesToRole (page 277)
- revokeRolesFromRole (page 282)
- revokePrivilegesFromRole (page 279)

> **Warning:** An update to the privileges or roles array completely replaces the previous array's values.

The updateRole (page 286) command uses the following syntax. To update a role, you must provide the privileges array, roles array, or both:

```
{
  updateRole: "<role>",
  privileges:
      [
        { resource: { <resource> }, actions: [ "<action>", ... ] },
        ...
      ],
  roles:
      [
```

```
        { role: "<role>", db: "<database>" } | "<role>",
        ...
      ],
  writeConcern: <write concern document>
}
```

The `updateRole` (page 286) command has the following fields:

field string updateRole The name of the *user-defined role* role to update.

field array privileges Required if you do not specify `roles` array. The privileges to grant the role. An update to the `privileges` array overrides the previous array's values. For the syntax for specifying a privilege, see the `privileges` array.

field array roles Required if you do not specify `privileges` array. The roles from which this role inherits privileges. An update to the `roles` array overrides the previous array's values.

field document writeConcern The level of `write concern` for the update operation. The `writeConcern` document takes the same fields as the `getLastError` (page 245) command.

In the `roles` field, you can specify both *built-in roles* and *user-defined role*.

To specify a role that exists in the same database where `updateRole` (page 286) runs, you can either specify the role with the name of the role:

```
"readWrite"
```

Or you can specify the role with a document, as in:

```
{ role: "<role>", db: "<database>" }
```

To specify a role that exists in a different database, specify the role with a document.

Behavior A role's privileges apply to the database where the role is created. The role can inherit privileges from other roles in its database. A role created on the `admin` database can include privileges that apply to all databases or to the *cluster* and can inherit privileges from roles in other databases.

Required Access You must have the `revokeRole` *action* on all databases in order to update a role.

You must have the `grantRole` *action* on the database of each role in the `roles` array to update the array.

You must have the `grantRole` action on the database of each privilege in the `privileges` array to update the array. If a privilege's resource spans databases, you must have `grantRole` on the `admin` database. A privilege spans databases if the privilege is any of the following:

- a collection in all databases
- all collections and all database
- the `cluster` resource

Example The following is an example of the `updateRole` (page 286) command that updates the `myClusterwideAdmin` role on the `admin` database. While the `privileges` and the `roles` arrays are both optional, at least one of the two is required:

```
use admin
db.runCommand(
   {
```

```
    updateRole: "myClusterwideAdmin",
    privileges:
        [
          {
            resource: { db: "", collection: "" },
            actions: [ "find" , "update", "insert", "remove" ]
          }
        ],
    roles:
        [
          { role: "dbAdminAnyDatabase", db: "admin" }
        ],
    writeConcern: { w: "majority" }
  }
)
```

To view a role's privileges, use the `rolesInfo` (page 283) command.

Replication Commands

Replication Commands

Name	Description
`applyOps` (page 288)	Internal command that applies *oplog* entries to the current data set.
`isMaster` (page 289)	Displays information about this member's role in the replica set, including whether it is the master.
`replSetFreeze` (page 291)	Prevents the current member from seeking election as *primary* for a period of time.
`replSetGetConfig` (page 292)	Returns the replica set's configuration object.
`replSetGetStatus` (page 296)	Returns a document that reports on the status of the replica set.
`replSetInitiate` (page 299)	Initializes a new replica set.
`replSetMaintenance` (page 300)	Enables or disables a maintenance mode, which puts a *secondary* node in a `RECOVERING` state.
`replSetReconfig` (page 300)	Applies a new configuration to an existing replica set.
`replSetStepDown` (page 301)	Forces the current *primary* to *step down* and become a *secondary*, forcing an election.
`replSetSyncFrom` (page 302)	Explicitly override the default logic for selecting a member to replicate from.
`resync` (page 302)	Forces a `mongod` (page 583) to re-synchronize from the *master*. For master-slave replication only.

applyOps

Definition

`applyOps`

Applies specified *oplog* entries to a `mongod` (page 583) instance. The `applyOps` (page 288) command is primarily an internal command.

If authorization is enabled, you must have access to all actions on all resources in order to run `applyOps` (page 288). Providing such access is not recommended, but if your organization requires a user to run `applyOps` (page 288), create a role that grants `anyAction` on *resource-anyresource*. Do not assign this role to any other user.

The `applyOps` (page 288) command has the following prototype form:

```
db.runCommand( { applyOps: [ <operations> ],
                 preCondition: [ { ns: <namespace>, q: <query>, res: <result> } ] } )
```

The `applyOps` (page 288) command takes a document with the following fields:

field array applyOps The oplog entries to apply.

field array preCondition An array of documents that contain the conditions that must be true in order to apply the oplog entry. Each document contains a set of conditions, as described in the next table.

field array alwaysUpsert A flag that indicates whether to apply update operations in the oplog as ref:*upserts <write-operations-upsert-behavior>*. When `true`, all updates become upserts to prevent failures as a results of sequences of updates followed by deletes: this is the same mode of operation as normal replication in secondaries. When `false`, updates are applied unmodified: this is the same mode of operation used during initial sync operations. `true` by default.

The `preCondition` array takes one or more documents with the following fields:

field string ns A *namespace*. If you use this field, `applyOps` (page 288) applies oplog entries only for the *collection* described by this namespace.

param string q Specifies the *query* that produces the results specified in the `res` field.

param string res The results of the query in the `q` field that must match to apply the oplog entry.

Behavior

> **Warning:** This command obtains a global write lock and will block other operations until it has completed.

isMaster

Definition

`isMaster`

`isMaster` (page 289) returns a document that describes the role of the `mongod` (page 583) instance.

If the instance is a member of a replica set, then `isMaster` (page 289) returns a subset of the replica set configuration and status including whether or not the instance is the *primary* of the replica set.

When sent to a `mongod` (page 583) instance that is not a member of a replica set, `isMaster` (page 289) returns a subset of this information.

MongoDB *drivers* and *clients* use `isMaster` (page 289) to determine the state of the replica set members and to discover additional members of a *replica set*.

The `db.isMaster()` (page 119) method in the `mongo` (page 610) shell provides a wrapper around `isMaster` (page 289).

The command takes the following form:

```
{ isMaster: 1 }
```

See also:

`db.isMaster()` (page 119)

Output

All Instances The following `isMaster` (page 289) fields are common across all roles:

`isMaster.`**`ismaster`**
A boolean value that reports when this node is writable. If `true`, then this instance is a *primary* in a *replica set*, or a *master* in a master-slave configuration, or a `mongos` (page 601) instance, or a standalone `mongod` (page 583).

This field will be `false` if the instance is a *secondary* member of a replica set or if the member is an *arbiter* of a replica set.

`isMaster.`**`rbid`**
New in version 2.8.

An ObjectId value that changes any time that a rollback may have occurred. For internal use.

`isMaster.`**`maxBsonObjectSize`**
The maximum permitted size of a *BSON* object in bytes for this `mongod` (page 583) process. If not provided, clients should assume a max size of "`16 * 1024 * 1024`".

`isMaster.`**`maxMessageSizeBytes`**
New in version 2.4.

The maximum permitted size of a *BSON* wire protocol message. The default value is `48000000` bytes.

`isMaster.`**`localTime`**
New in version 2.2.

Returns the local server time in UTC. This value is an *ISO date*.

`isMaster.`**`minWireVersion`**
New in version 2.6.

The earliest version of the wire protocol that this `mongod` (page 583) or `mongos` (page 601) instance is capable of using to communicate with clients.

Clients may use `minWireVersion` (page 290) to help negotiate compatibility with MongoDB.

`isMaster.`**`maxWireVersion`**
New in version 2.6.

The latest version of the wire protocol that this `mongod` (page 583) or `mongos` (page 601) instance is capable of using to communicate with clients.

Clients may use `maxWireVersion` (page 290) to help negotiate compatibility with MongoDB.

Sharded Instances `mongos` (page 601) instances add the following field to the `isMaster` (page 289) response document:

`isMaster.`**`msg`**
Contains the value `isdbgrid` when `isMaster` (page 289) returns from a `mongos` (page 601) instance.

Replica Sets `isMaster` (page 289) contains these fields when returned by a member of a replica set:

`isMaster.`**`setName`**
The name of the current :replica set.

`isMaster.`**`secondary`**
A boolean value that, when `true`, indicates if the `mongod` (page 583) is a *secondary* member of a *replica set*.

isMaster.**hosts**
An array of strings in the format of "[hostname]:[port]" that lists all members of the *replica set* that are neither *hidden*, *passive*, nor *arbiters*.

Drivers use this array and the isMaster.passives (page 291) to determine which members to read from.

isMaster.**passives**
An array of strings in the format of "[hostname]:[port]" listing all members of the *replica set* which have a priority of 0.

This field only appears if there is at least one member with a priority of 0.

Drivers use this array and the isMaster.hosts (page 290) to determine which members to read from.

isMaster.**arbiters**
An array of strings in the format of "[hostname]:[port]" listing all members of the *replica set* that are *arbiters*.

This field only appears if there is at least one arbiter in the replica set.

isMaster.**primary**
A string in the format of "[hostname]:[port]" listing the current *primary* member of the replica set.

isMaster.**arbiterOnly**
A boolean value that , when true, indicates that the current instance is an *arbiter*. The arbiterOnly (page 291) field is only present, if the instance is an arbiter.

isMaster.**passive**
A boolean value that, when true, indicates that the current instance is *passive*. The passive (page 291) field is only present for members with a priority of 0.

isMaster.**hidden**
A boolean value that, when true, indicates that the current instance is *hidden*. The hidden (page 291) field is only present for hidden members.

isMaster.**tags**
A document that lists any tags assigned to this member. This field is only present if there are tags assigned to the member. See http://docs.mongodb.org/manual/tutorial/configure-replica-set-tag-sets for more information.

isMaster.**me**
The [hostname]:[port] of the member that returned isMaster (page 289).

isMaster.**electionId**
New in version 2.8.0.

A unique identifier for each election. Included only in the output of isMaster (page 289) for the *primary*. Used by clients to determine when elections occur.

replSetFreeze

replSetFreeze
The replSetFreeze (page 291) command prevents a replica set member from seeking election for the specified number of seconds. Use this command in conjunction with the replSetStepDown (page 301) command to make a different node in the replica set a primary.

The replSetFreeze (page 291) command uses the following syntax:

```
{ replSetFreeze: <seconds> }
```

If you want to unfreeze a replica set member before the specified number of seconds has elapsed, you can issue the command with a seconds value of `0`:

```
{ replSetFreeze: 0 }
```

Restarting the `mongod` (page 583) process also unfreezes a replica set member.

`replSetFreeze` (page 291) is an administrative command, and you must issue it against the *admin database*.

replSetGetConfig New in version 2.8.0.

Definition

`replSetGetConfig`

Returns a document that describes the current configuration of the *replica set*. Invoke the directly, using the following operation:

```
db.runCommand( { replSetGetConfig: 1 } );
```

Access the data provided by `replSetGetConfig` (page 292) using `rs.conf()` (page 174) in the `mongo` (page 610) shell, as in the following:

```
rs.conf();
```

Output

`replSetGetConfig.`**`_id`**

Type: string

The name of the replica set. Once set, you cannot change the name of a replica set.

See

`replSetName` or `--replSet` (page 594) for information on setting the replica set name.

`replSetGetConfig.`**`version`**

An incrementing number used to distinguish revisions of the replica set configuration object from previous iterations of the configuration.

`members`

`replSetGetConfig.`**`members`**

Type: array

An array of member configuration documents, one for each member of the replica set. The `members` (page 292) array is a zero-indexed array.

Each member-specific configuration document can contain the following fields:

`replSetGetConfig.members[n].`**`_id`**

Type: integer

A numeric identifier of every member in the replica set. Once set, you cannot change the `_id` (page 292) of a member.

Note: When updating the replica configuration object, access the replica set members in the `members` array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` field in each document in the `members` array.

---

replSetGetConfig.members[n].**host**
Type: string

The hostname and, if specified, the port number, of the set member.

The hostname name must be resolvable for every host in the replica set.

> **Warning:** host (page 292) cannot hold a value that resolves to localhost or the local interface unless *all* members of the set are on hosts that resolve to localhost.

replSetGetConfig.members[n].**arbiterOnly**
Optional.

Type: boolean

Default: false

A boolean that identifies an arbiter. A value of true indicates that the member is an arbiter.

When using the rs.addArb() (page 174) method to add an arbiter, the method automatically sets arbiterOnly (page 293) to true for the added member.

replSetGetConfig.members[n].**buildIndexes**
Optional.

Type: boolean

Default: true

A boolean that indicates whether the mongod (page 583) builds *indexes* on this member. You can only set this value when adding a member to a replica set. You cannot change buildIndexes (page 293) field after the member has been added to the set. To add a member, see rs.add() (page 173) and rs.reconfig() (page 176).

Do not set to false for mongod (page 583) instances that receive queries from clients.

Setting buildIndexes to false may be useful if **all** the following conditions are true:

- you are only using this instance to perform backups using mongodump (page 622), *and*
- this member will receive no queries, *and*
- index creation and maintenance overburdens the host system.

Even if set to false, secondaries *will* build indexes on the _id field in order to facilitate operations required for replication.

> **Warning:** If you set buildIndexes (page 293) to false, you must also set priority (page 294) to 0. If priority (page 294) is not 0, MongoDB will return an error when attempting to add a member with buildIndexes (page 293) equal to false.
> To ensure the member receives no queries, you should make all instances that do not build indexes hidden.
> Other secondaries cannot replicate from a member where buildIndexes (page 293) is false.

replSetGetConfig.members[n].**hidden**
Optional.

Type: boolean

Default: false

When this value is `true`, the replica set hides this instance and does not include the member in the output of `db.isMaster()` (page 119) or `isMaster` (page 289). This prevents read operations (i.e. queries) from ever reaching this host by way of secondary *read preference*.

See also:

Hidden Replica Set Members

`replSetGetConfig.members[n].priority`
Optional.

Type: Number, between 0 and 1000.

Default: 1.0

A number that indicates the relative eligibility of a member to become a *primary*.

Specify higher values to make a member *more* eligible to become *primary*, and lower values to make the member *less* eligible. Priorities are only used in comparison to each other. Members of the set will veto election requests from members when another eligible member has a higher priority value. Changing the balance of priority in a replica set will trigger an election.

A `priority` (page 294) of `0` makes it impossible for a member to become primary.

See also:

Replica Set Elections.

`replSetGetConfig.members[n].tags`
Optional.

Type: document

Default: none

A document that contains arbitrary field and value pairs for describing or *tagging* members in order to extend `write concern` and `read preference` and thereby allowing configurable data center awareness.

Use `tags` to configure write concerns in conjunction with `getLastErrorModes` (page 295) and `getLastErrorDefaults` (page 295).

Important: In tag sets, all tag values must be strings.

For more information on configuring tag sets for read preference and write concern, see `http://docs.mongodb.org/manual/tutorial/configure-replica-set-tag-sets`.

`replSetGetConfig.members[n].slaveDelay`
Optional.

Type: integer

Default: 0

The number of seconds "behind" the primary that this replica set member should "lag".

Use this option to create *delayed members*. Delayed members maintain a copy of the data that reflects the state of the data at some time in the past.

See also:

`http://docs.mongodb.org/manual/core/replica-set-delayed-member`

`replSetGetConfig.members[n].votes`
Optional.

Type: integer

Default: 1

The number of votes a server will cast in a *replica set election*. The number of votes each member has is either `1` or `0`, and *arbiters* always have exactly `1` vote.

A replica set can have up to 12 members, but can have at most only 7 *voting* members. If you need more than 7 members in one replica set, set `votes` (page 294) to `0` for the additional non-voting members.

Note: Deprecated since version 2.6: `votes` values greater than `1`.

Earlier versions of MongoDB allowed a member to have more than `1` vote by setting `votes` to a value greater than `1`. Setting `votes` to value greater than `1` now produces a warning message.

---

settings

`replSetGetConfig.`**settings**

Optional.

Type: document

A document that contains configuration options that apply to the whole replica set.

The `settings` (page 295) document contain the following fields:

`replSetGetConfig.settings.`**chainingAllowed**

New in version 2.2.4.

Optional.

Type: boolean

Default: true

When `chainingAllowed` (page 295) is `true`, the replica set allows *secondary* members to replicate from other secondary members. When `chainingAllowed` (page 295) is `false`, secondaries can replicate only from the *primary*.

When you run `rs.conf()` (page 174) to view a replica set's configuration, the `chainingAllowed` (page 295) field appears only when set to `false`. If not set, `chainingAllowed` (page 295) is `true`.

See also:

`http://docs.mongodb.org/manual/tutorial/manage-chained-replication`

`replSetGetConfig.settings.`**getLastErrorDefaults**

Optional.

Type: document

A document that specifies the `write concern` for the replica set. The replica set will use this write concern only when *write operations* (page 751) or `getLastError` (page 245) specify no other write concern.

If `getLastErrorDefaults` (page 295) is not set, the default write concern for the replica set only requires confirmation from the primary.

`replSetGetConfig.settings.`**getLastErrorModes**

Optional.

Type: document

A document used to define an extended *write concern* through the use of `tags` (page 294). The extended *write concern* can provide *data-center awareness*.

For example, the following document defines an extended write concern named `eastCoast` and associates with a write to a member that has the `east` tag.

```
{ getLastErrorModes: { eastCoast: { "east": 1 } } }
```

Write operations to the replica set can use the extended write concern, e.g. `{ w: "eastCoast" }`.

See `http://docs.mongodb.org/manual/tutorial/configure-replica-set-tag-sets` for more information and example.

`replSetGetConfig.settings.`**`heartbeatTimeoutSecs`**

Optional.

Type: int

Default: 10

Number of seconds that the replica set members wait for a successful heartbeat from each other. If a member does not respond in time, other members mark the delinquent member as inaccessible.

replSetGetStatus

Definition

`replSetGetStatus`

The `replSetGetStatus` command returns the status of the replica set from the point of view of the current server. You must run the command against the *admin database*. The command has the following prototype format:

```
{ replSetGetStatus: 1 }
```

The value specified does not affect the output of the command. Data provided by this command derives from data included in heartbeats sent to the current instance by other members of the replica set. Because of the frequency of heartbeats, these data can be several seconds out of date.

You can also access this functionality through the `rs.status()` (page 179) helper in the `mongo` (page 610) shell.

The `mongod` (page 583) must have replication enabled and be a member of a replica set for the for `replSetGetStatus` (page 296) to return successfully.

Example The following example runs the **replSetGetStatus** command on the *admin database* of the replica set primary:

```
use admin
db.runCommand( { replSetGetStatus : 1 } )
```

Consider the following example output:

```
{
    "set" : "replset",
    "date" : ISODate("2014-05-01T14:44:03Z"),
    "myState" : 1,
    "members" : [
        {
            "_id" : 0,
            "name" : "m1.example.net:27017",
            "health" : 1,
            "state" : 1,
```

```
            "stateStr" : "PRIMARY",
            "uptime" : 269,
            "optime" : Timestamp(1404225575, 11),
            "optimeDate" : ISODate("2014-05-01T14:39:35Z"),
            "electionTime" : Timestamp(1404225586, 1),
            "electionDate" : ISODate("2014-05-01T14:39:46Z"),
            "self" : true
        },
        {
            "_id" : 1,
            "name" : "m2.example.net:27017",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 265,
            "optime" : Timestamp(1404225575, 11),
            "optimeDate" : ISODate("2014-05-01T14:39:35Z"),
            "lastHeartbeat" : ISODate("2014-05-01T14:44:03Z"),
            "lastHeartbeatRecv" : ISODate("2014-05-01T14:44:02Z"),
            "pingMs" : 0,
            "syncingTo" : "m1.example.net:27017"
        },
        {
            "_id" : 2,
            "name" : "m3.example.net:27017",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 265,
            "optime" : Timestamp(1404225575, 11),
            "optimeDate" : ISODate("2014-05-01T14:39:35Z"),
            "lastHeartbeat" : ISODate("2014-05-01T14:44:02Z"),
            "lastHeartbeatRecv" : ISODate("2014-05-01T14:44:02Z"),
            "pingMs" : 0,
            "syncingTo" : "m1.example.net:27017"
        }
    ],
    "ok" : 1
}
```

Output The **replSetGetStatus** command returns a document with the following fields:

`replSetGetStatus.set`
: The `set` value is the name of the replica set, configured in the `replSetName` setting. This is the same value as `_id` in `rs.conf()` (page 174).

`replSetGetStatus.date`
: The value of the `date` field is an *ISODate* of the current time, according to the current server. Compare this to the value of the `lastHeartbeat` (page **??**) to find the operational lag between the current host and the other hosts in the set.

`replSetGetStatus.myState`
: The value of `myState` (page 297) is an integer between `0` and `10` that represents the `replica state` of the current member.

`replSetGetStatus.members`
: The `members` field holds an array that contains a document for every member in the replica set.

replSetGetStatus.members[n].**name**
The name field holds the name of the server.

replSetGetStatus.members[n].**self**
The self field is only included in the document for the current mongod instance in the members array. It's value is true.

replSetGetStatus.members[n].**health**
The health value is only present for the other members of the replica set (i.e. not the member that returns rs.status (page 179).) This field conveys if the member is up (i.e. 1) or down (i.e. 0.)

replSetGetStatus.members.**state**
The value of state is an integer between 0 and 10 that represents the replica state of the member.

replSetGetStatus.members[n].**stateStr**
A string that describes state.

replSetGetStatus.members[n].**uptime**
The uptime (page 298) field holds a value that reflects the number of seconds that this member has been online.

This value does not appear for the member that returns the rs.status() (page 179) data.

replSetGetStatus.members[n].**optime**
Information regarding the last operation from the operation log that this member has applied.

replSetGetStatus.members[n].optime.**t**
A 32-bit timestamp of the last operation applied to this member of the replica set from the *oplog*.

replSetGetStatus.members[n].optime.**i**
An incremented field, which reflects the number of operations in since the last time stamp. This value only increases if there is more than one operation per second.

replSetGetStatus.members[n].**optimeDate**
An *ISODate* formatted date string that reflects the last entry from the *oplog* that this member applied. If this differs significantly from lastHeartbeat (page 298) this member is either experiencing "replication lag" *or* there have not been any new operations since the last update. Compare members.optimeDate between all of the members of the set.

replSetGetStatus.members[n].**electionTime**
For the current primary, information regarding the election time from the operation log. See http://docs.mongodb.org/manual/core/replica-set-elections for more information about elections.

replSetGetStatus.members[n].electionTime.**t**
For the current primary, a 32-bit timestamp of the election time applied to this member of the replica set from the *oplog*.

replSetGetStatus.members[n].electionTime.**i**
For the current primary, an incremented field which reflects the number of operations in since the last time stamp. This value only increases if there is more than one operation per second.

replSetGetStatus.members[n].**electionDate**
For the current primary, an *ISODate* formatted date string that reflects the election date. See http://docs.mongodb.org/manual/core/replica-set-elections for more information about elections.

replSetGetStatus.members[n].**self**
Indicates which replica set member processed the **replSetGetStatus** command.

replSetGetStatus.members[n].**lastHeartbeat**
The lastHeartbeat value provides an *ISODate* formatted date and time of the transmission time of

last heartbeat received from this member. Compare this value to the value of the `date` (page 297) and `lastHeartBeatRecv` field to track latency between these members.

This value does not appear for the member that returns the `rs.status()` (page 179) data.

`replSetGetStatus.members[n].lastHeartbeatRecv`
The `lastHeartbeatRecv` value provides an *ISODate* formatted date and time that the last heartbeat was received from this member. Compare this value to the value of the `date` (page 297) and `lastHeartBeat` field to track latency between these members.

`replSetGetStatus.members[n].lastHeartbeatMessage`
When the last heartbeat included an extra message, the `lastHeartbeatMessage` (page 299) contains a string representation of that message.

`replSetGetStatus.members[n].pingMs`
The `pingMs` represents the number of milliseconds (ms) that a round-trip packet takes to travel between the remote member and the local instance.

This value does not appear for the member that returns the `rs.status()` (page 179) data.

`replSetGetStatus.members[n].syncingTo`
The `syncingTo` field is only present on the output of `rs.status()` (page 179) on *secondary* and recovering members, and holds the hostname of the member from which this instance is syncing.

replSetInitiate

`replSetInitiate`

The `replSetInitiate` (page 299) 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 175) 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, http://docs.mongodb.org/manual/administration/replica-sets, and *Replica Set Reconfiguration* (page 177).

replSetMaintenance

Definition

replSetMaintenance

The replSetMaintenance (page 300) 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> }
```

Behavior Consider the following behavior when running the replSetMaintenance (page 300) command:

- You cannot run the command on the Primary.
- You must run the command against the admin database.
- When enabled replSetMaintenance: true, 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.
- On secondaries, the compact (page 322) command forces the secondary to enter RECOVERING state. Read operations issued to an instance in the RECOVERING state will fail. This prevents clients from reading during the operation. When the operation completes, the secondary returns to:replstate:*SECONDARY* state.
- See http://docs.mongodb.org/manual/reference/replica-states/ for more information about replica set member states.

See http://docs.mongodb.org/manual/tutorial/perform-maintence-on-replica-set-members for an example replica set maintenance procedure to maximize availability during maintenance operations.

replSetReconfig

replSetReconfig

The replSetReconfig (page 300) 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 replSetReconfig (page 300) with the shell's rs.reconfig() (page 176) method.

Behaviors Be aware of the following replSetReconfig (page 300) 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 300) 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 300) 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.

`replSetReconfig` (page 300) obtains a special mutually exclusive lock to prevent more than one `replSetReconfig` (page 300) operation from occurring at the same time.

Additional Information `http://docs.mongodb.org/manual/reference/replica-configuration`, `rs.reconfig()` (page 176), and `rs.conf()` (page 174).

replSetStepDown

Description

`replSetStepDown`

Forces the *primary* of the replica set to become a *secondary*. This initiates an *election for primary*.

`replSetStepDown` (page 301) has the following prototype form:

```
db.runCommand( { replSetStepDown: <seconds> , force: <true|false> } )
```

`replSetStepDown` (page 301) has the following fields:

field number replSetStepDown A number of seconds for the member to avoid election to primary. If you do not specify a value for `<seconds>`, `replSetStepDown` (page 301) will attempt to avoid reelection to primary for 60 seconds from the time that the `mongod` (page 583) received the `replSetStepDown` (page 301) command.

field Boolean force New in version 2.0: Forces the *primary* to step down even if there are no secondary members that could become primary.

field number secondaryCatchupPeriodSecs The amount of time that the `mongod` (page 583) will wait for another secondary in the replica set to catch up to the primary. If no secondary catches up before this period ends, then the command will fail and the member will not step down, unless you specify `{ force: true }`.

The default value is `10` seconds, unless you set `{ force: true }`, which changes the default to `0`.

Behavior

Client Impact `replSetStepDown` (page 301) forces all clients currently connected to the database to disconnect. This helps ensure that clients maintain an accurate view of the replica set.

Secondary Requirements Changed in version 2.8: To avoid rollbacks, `replSetStepDown` (page 301) will wait for one secondary to be totally caught up before allowing the primary to step down.

User Operations New in version 2.8.

Before stepping down, `replSetStepDown` (page 301) will attempt to terminate long running user operations that would block the primary from stepping down, such as an index build, a write operation or a map-reduce job.

Example The following example specifies that the former primary avoids reelection to primary for 120 seconds:

```
db.runCommand( { replSetStepDown: 120 } )
```

replSetSyncFrom

Description

`replSetSyncFrom`

New in version 2.2.

Explicitly configures which host the current `mongod` (page 583) pulls *oplog* entries from. This operation is useful for testing different patterns and in situations where a set member is not replicating from the desired host.

The `replSetSyncFrom` (page 302) command has the following form:

```
{ replSetSyncFrom: "hostname<:port>" }
```

The `replSetSyncFrom` (page 302) command has the following field:

field string replSetSyncFrom The name and port number of the replica set member that this member should replicate from. Use the `[hostname]:[port]` form.

For more information the use of `replSetSyncFrom` (page 302), see `http://docs.mongodb.org/manual/tutorial/configure-replica-set-secondary-sync-target`.

resync

`resync`

The `resync` (page 302) command forces an out-of-date slave `mongod` (page 583) 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.

See also:

`http://docs.mongodb.org/manual/replication` for more information regarding replication.

Sharding Commands

Sharding Commands

Name	Description
addShard (page 303)	Adds a *shard* to a *sharded cluster*.
checkShardingIndex (page 304)	Internal command that validates index on shard key.
cleanupOrphaned (page 305)	Removes orphaned data with shard key values outside of the ranges of the chunks owned by a shard.
enableSharding (page 308)	Enables sharding on a specific database.
flushRouterConfig (page 308)	Forces an update to the cluster metadata cached by a mongos (page 601).
getShardMap (page 308)	Internal command that reports on the state of a sharded cluster.
getShardVersion (page 308)	Internal command that returns the *config server* version.
isdbgrid (page 308)	Verifies that a process is a mongos (page 601).
listShards (page 309)	Returns a list of configured shards.
medianKey (page 309)	Deprecated internal command. See splitVector (page 316).
mergeChunks (page 309)	Provides the ability to combine chunks on a single shard.
moveChunk (page 310)	Internal command that migrates chunks between shards.
movePrimary (page 312)	Reassigns the *primary shard* when removing a shard from a sharded cluster.
removeShard (page 312)	Starts the process of removing a shard from a sharded cluster.
setShardVersion (page 314)	Internal command to sets the *config server* version.
shardCollection (page 314)	Enables the sharding functionality for a collection, allowing the collection to be sharded.
shardingState (page 315)	Reports whether the mongod (page 583) is a member of a sharded cluster.
splitChunk (page 315)	Internal command to split chunk. Instead use the methods sh.splitFind() (page 190) and sh.splitAt() (page 190).
splitVector (page 316)	Internal command that determines split points.
split (page 316)	Creates a new *chunk*.
unsetSharding (page 318)	Internal command that affects connections between instances in a MongoDB deployment.

addShard

Definition

addShard

Adds either a database instance or a *replica set* to a *sharded cluster*. The optimal configuration is to deploy shards across replica sets.

Run addShard (page 303) when connected to a mongos (page 601) instance. The command takes the following form when adding a single database instance as a shard:

```
{ addShard: "<hostname><:port>", maxSize: <size>, name: "<shard_name>" }
```

When adding a replica set as a shard, use the following form:

```
{ addShard: "<replica_set>/<hostname><:port>", maxSize: <size>, name: "<shard_name>" }
```

The command contains the following fields:

field string addShard The hostname and port of the mongod (page 583) instance to be added as a shard. To add a replica set as a shard, specify the name of the replica set and the hostname and port of a member of the replica set.

field integer maxSize The maximum size in megabytes of the shard. If you set maxSize to 0, MongoDB does not limit the size of the shard.

field string name A name for the shard. If this is not specified, MongoDB automatically provides a unique name.

The addShard (page 303) command stores shard configuration information in the *config database*. Always run addShard (page 303) when using the admin database.

Specify a maxSize when you have machines with different disk capacities, or if you want to limit the amount of data on some shards. The maxSize constraint prevents the *balancer* from migrating chunks to the shard when the value of mem.mapped (page 371) exceeds the value of maxSize.

Considerations

Balancing When you add a shard to a sharded cluster, you affect the balance of *chunks* among the shards of a cluster for all existing sharded collections. The balancer will begin migrating chunks so that the cluster will achieve balance. See http://docs.mongodb.org/manual/core/sharding-balancing for more information.

Hidden Members

Important: You cannot include a hidden member in the seed list provided to addShard (page 303).

Examples The following command adds the database instance running on port 27027 on the host mongodb0.example.net as a shard:

```
use admin
db.runCommand({addShard: "mongodb0.example.net:27027"})
```

Warning: Do not use localhost for the hostname unless your *configuration server* is also running on localhost.

The following command adds a replica set as a shard:

```
use admin
db.runCommand( { addShard: "repl0/mongodb3.example.net:27327"} )
```

You may specify all members in the replica set. All additional hostnames must be members of the same replica set.

checkShardingIndex

checkShardingIndex

checkShardingIndex (page 304) is an internal command that supports the sharding functionality.

cleanupOrphaned

Definition

cleanupOrphaned

New in version 2.6.

Deletes from a shard the *orphaned documents* whose shard key values fall into a single or a single contiguous range that do not belong to the shard. For example, if two contiguous ranges do not belong to the shard, the cleanupOrphaned (page 305) examines both ranges for orphaned documents.

cleanupOrphaned (page 305) has the following syntax:

```
db.runCommand( {
   cleanupOrphaned: "<database>.<collection>",
   startingAtKey: <minimumShardKeyValue>,
   secondaryThrottle: <boolean>,
   writeConcern: <document>
} )
```

cleanupOrphaned (page 305) has the following fields:

field string cleanupOrphaned The namespace, i.e. both the database and the collection name, of the sharded collection for which to clean the orphaned data.

field document startingFromKey The *shard key* value that determines the lower bound of the cleanup range. The default value is MinKey.

If the range that contains the specified startingFromKey value belongs to a chunk owned by the shard, cleanupOrphaned (page 305) continues to examine the next ranges until it finds a range not owned by the shard. See *Determine Range* (page 305) for details.

field boolean secondaryThrottle If true, each delete operation must be replicated to another secondary before the cleanup operation proceeds further. If false, do not wait for replication. Defaults to false.

Independent of the secondaryThrottle setting, after the final delete, cleanupOrphaned (page 305) waits for all deletes to replicate to a majority of replica set members before returning.

field document writeConcern A document that expresses the write concern that the secondaryThrottle will use to wait for the secondaries when removing orphaned data.

Any specified writeConcern implies _secondaryThrottle.

Behavior Run cleanupOrphaned (page 305) in the admin database directly on the mongod (page 583) instance that is the primary replica set member of the shard. Do not run cleanupOrphaned (page 305) on a mongos (page 601) instance.

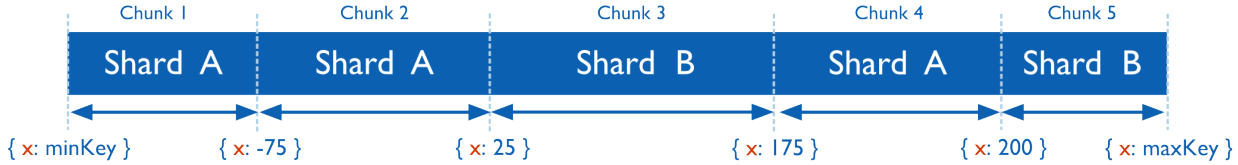
You do not need to disable the balancer before running cleanupOrphaned (page 305).

Determine Range The cleanupOrphaned (page 305) command uses the startingFromKey value, if specified, to determine the start of the range to examine for orphaned document:

- If the startingFromKey value falls into a range for a chunk not owned by the shard, cleanupOrphaned (page 305) begins examining at the start of this range, which may not necessarily be the startingFromKey.
- If the startingFromKey value falls into a range for a chunk owned by the shard, cleanupOrphaned (page 305) moves onto the next range until it finds a range for a chunk not owned by the shard.

The cleanupOrphaned (page 305) deletes orphaned documents from the start of the determined range and ends at the start of the chunk range that belongs to the shard.

Consider the following key space with documents distributed across Shard A and Shard B.



Shard A owns:

- Chunk 1 with the range { x: minKey } --> { x: -75 },
- Chunk 2 with the range { x: -75 } --> { x: 25 }, and
- Chunk 4 with the range { x: 175 } --> { x: 200 }.

Shard B owns:

- Chunk 3 with the range { x: 25 } --> { x: 175 } and
- Chunk 5 with the range { x: 200 } --> { x: maxKey }.

If on Shard A, the cleanupOrphaned (page 305) command runs with startingFromKey: { x: -70 } or any other value belonging to range for Chunk 1 or Chunk 2, the cleanupOrphaned (page 305) command examines the Chunk 3 range of { x: 25 } --> { x: 175 } to delete orphaned data.

If on Shard B, the cleanupOrphaned (page 305) command runs with the startingFromKey: { x: -70 } or any other value belonging to range for Chunk 1, the cleanupOrphaned (page 305) command examines the combined contiguous range for Chunk 1 and Chunk 2, namely { x: minKey } --> { x: 25 } to delete orphaned data.

Required Access On systems running with authorization, you must have clusterAdmin privileges to run cleanupOrphaned (page 305).

Output

Return Document Each cleanupOrphaned (page 305) command returns a document containing a subset of the following fields:

cleanupOrphaned.**ok**
: Equal to 1 on success.

  A value of 1 indicates that cleanupOrphaned (page 305) scanned the specified shard key range, deleted any orphaned documents found in that range, and confirmed that all deletes replicated to a majority of the members of that shard's replica set. If confirmation does not arrive within 1 hour, cleanupOrphaned (page 305) times out.

  A value of 0 could indicate either of two cases:

  - cleanupOrphaned (page 305) found orphaned documents on the shard but could not delete them.
  - cleanupOrphaned (page 305) found and deleted orphaned documents, but could not confirm replication before the 1 hour timeout. In this case, replication does occur, but only after cleanupOrphaned (page 305) returns.

cleanupOrphaned.**stoppedAtKey**
: The upper bound of the cleanup range of shard keys. If present, the value corresponds to the lower bound of the next chunk on the shard. The absence of the field signifies that the cleanup range was the uppermost range for the shard.

Log Files The cleanupOrphaned (page 305) command prints the number of deleted documents to the mongod (page 583) log. For example:

```
m30000| 2013-10-31T15:17:28.972-0400 [conn1] Deleter starting delete for: foo.bar from { _id: -35.0
m30000| 2013-10-31T15:17:28.972-0400 [conn1] rangeDeleter deleted 0 documents for foo.bar from { _id
```

Examples The following examples run the cleanupOrphaned (page 305) command directly on the primary of the shard.

Remove Orphaned Documents for a Specific Range For a sharded collection info in the test database, a shard owns a single chunk with the range: { x: MinKey } --> { x: 10 }.

The shard also contains documents whose shard keys values fall in a range for a chunk *not* owned by the shard: { x: 10 } --> { x: MaxKey }.

To remove orphaned documents within the { x: 10 } => { x: MaxKey } range, you can specify a startingFromKey with a value that falls into this range, as in the following example:

```
use admin
db.runCommand( {
   "cleanupOrphaned": "test.info",
   "startingFromKey": { x: 10 },
   "secondaryThrottle": true
} )
```

Or you can specify a startingFromKey with a value that falls into the previous range, as in the following:

```
use admin
db.runCommand( {
   "cleanupOrphaned": "test.info",
   "startingFromKey": { x: 2 },
   "secondaryThrottle": true
} )
```

Since { x: 2 } falls into a range that belongs to a chunk owned by the shard, cleanupOrphaned (page 305) examines the next range to find a range not owned by the shard, in this case { x: 10 } => { x: MaxKey }.

Remove All Orphaned Documents from a Shard cleanupOrphaned (page 305) examines documents from a single contiguous range of shard keys. To remove all orphaned documents from the shard, you can run cleanupOrphaned (page 305) in a loop, using the returned stoppedAtKey as the next startingFromKey, as in the following:

```
use admin
var nextKey = { };

while ( nextKey = db.runCommand( {
                      cleanupOrphaned: "test.user",
                      startingFromKey: nextKey
                  } ).stoppedAtKey ) {
```

```
    printjson(nextKey);
}
```

enableSharding

enableSharding

The enableSharding (page 308) 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 314) command to begin the process of distributing data among the shards.

flushRouterConfig

flushRouterConfig

flushRouterConfig (page 308) clears the current cluster information cached by a mongos (page 601) instance and reloads all *sharded cluster* metadata from the *config database*.

This forces an update when the configuration database holds data that is newer than the data cached in the mongos (page 601) process.

> **Warning:** Do not modify the config data, except as explicitly documented. A config database cannot typically tolerate manual manipulation.

flushRouterConfig (page 308) is an administrative command that is only available for mongos (page 601) instances.

New in version 1.8.2.

getShardMap

getShardMap

getShardMap (page 308) is an internal command that supports the sharding functionality.

getShardVersion

getShardVersion

getShardVersion (page 308) is a command that supports sharding functionality and is not part of the stable client facing API.

isdbgrid

isdbgrid

This command verifies that a process is a mongos (page 601).

If you issue the isdbgrid (page 308) command when connected to a mongos (page 601), the response document includes the isdbgrid field set to 1. The returned document is similar to the following:

```
{ "isdbgrid" : 1, "hostname" : "app.example.net", "ok" : 1 }
```

If you issue the isdbgrid (page 308) command when connected to a mongod (page 583), MongoDB returns an error document. The isdbgrid (page 308) command is not available to mongod (page 583). The error document, however, also includes a line that reads "isdbgrid" : 1, just as in the document returned for a mongos (page 601). The error document is similar to the following:

```
{
   "errmsg" : "no such cmd: isdbgrid",
   "bad cmd" : {
         "isdbgrid" : 1
   },
   "ok" : 0
}
```

You can instead use the `isMaster` (page 289) command to determine connection to a `mongos` (page 601). When connected to a `mongos` (page 601), the `isMaster` (page 289) command returns a document that contains the string `isdbgrid` in the `msg` field.

listShards

`listShards`

Use the `listShards` (page 309) command to return a list of configured shards. The command takes the following form:

```
{ listShards: 1 }
```

medianKey

`medianKey`

`medianKey` (page 309) is an internal command.

mergeChunks

Definition

`mergeChunks`

For a sharded collection, `mergeChunks` (page 309) combines contiguous *chunk* ranges on a shard into a single chunk. Issue the `mergeChunks` (page 309) command from a `mongos` (page 601) instance.

`mergeChunks` (page 309) has the following form:

```
db.runCommand( { mergeChunks : <namespace> ,
                 bounds : [ { <shardKeyField>: <minFieldValue> },
                            { <shardKeyField>: <maxFieldValue> } ] } )
```

For compound shard keys, you must include the full shard key in the `bounds` specification. If the shard key is `{ x: 1, y: 1 }`, `mergeChunks` (page 309) has the following form:

```
db.runCommand( { mergeChunks : <namespace> ,
                 bounds : [ { x: <minValue>, y: <minValue> },
                            { x: <maxValue>, y: <maxValue> } ] } )
```

The `mergeChunks` (page 309) command has the following fields:

field namespace mergeChunks The fully qualified *namespace* of the *collection* where both *chunks* exist. Namespaces take form of `<database>.<collection>`.

field array bounds An array that contains the minimum and maximum key values of the new chunk.

Behavior

Note: Use the `mergeChunks` (page 309) only in special circumstances. For instance, when cleaning up your *sharded cluster* after removing many documents.

In order to successfully merge chunks, the following *must* be true:

- In the bounds field, <minkey> and <maxkey> must correspond to the lower and upper bounds of the *chunks* to merge.
- The chunks must reside on the same shard.
- The chunks must be contiguous.

mergeChunks (page 309) returns an error if these conditions are not satisfied.

Return Messages On success, mergeChunks (page 309) returns to following document:

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

Another Operation in Progress mergeChunks (page 309) returns the following error message if another metadata operation is in progress on the chunks (page 683) collection:

```
errmsg: "The collection's metadata lock is already taken."
```

If another process, such as balancer process, changes metadata while mergeChunks (page 309) is running, you may see this error. You can retry the mergeChunks (page 309) operation without side effects.

Chunks on Different Shards If the input *chunks* are not on the same *shard*, mergeChunks (page 309) returns an error similar to the following:

```
{
   "ok" : 0,
   "errmsg" : "could not merge chunks, collection test.users does not contain a chunk ending at { use
}
```

Noncontiguous Chunks If the input *chunks* are not contiguous, mergeChunks (page 309) returns an error similar to the following:

```
{
   "ok" : 0,
   "errmsg" : "could not merge chunks, collection test.users has more than 2 chunks between [{ userna
}
```

moveChunk

Definition

moveChunk

Internal administrative command. Moves *chunks* between *shards*. Issue the moveChunk (page 310) command via a mongos (page 601) instance while using the *admin database*. Use the following forms:

```
db.runCommand( { moveChunk : <namespace> ,
                 find : <query> ,
                 to : <string>,
                 _secondaryThrottle : <boolean>,
                 writeConcern: <document>,
                 _waitForDelete : <boolean> } )
```

Alternately:

```
db.runCommand( { moveChunk : <namespace> ,
                 bounds : <array> ,
                 to : <string>,
                 _secondaryThrottle : <boolean>,
                 writeConcern: <document>,
                 _waitForDelete : <boolean> } )
```

The `moveChunk` (page 310) command has the following fields:

field string moveChunk The *namespace* of the *collection* where the *chunk* exists. Specify the collection's full namespace, including the database name.

field document find An equality match on the shard key that specifies the shard-key value of the chunk to move. Specify either the `bounds` field or the `find` field but not both. Do **not** use the `find` field to select chunks in collections that use a *hashed shard key*.

field array bounds The bounds of a specific chunk to move. The array must consist of two documents that specify the lower and upper shard key values of a chunk to move. Specify either the `bounds` field or the `find` field but not both. Use `bounds` to select chunks in collections that use a *hashed shard key*.

field string to The name of the destination shard for the chunk.

field Boolean secondaryThrottle Defaults to `true`. When `true`, the balancer waits for replication to *secondaries* when it copies and deletes data during chunk migrations. For details, see *sharded-cluster-config-secondary-throttle*.

field document writeConcern A document that expresses the `write concern` that the `_secondaryThrottle` will use to wait for secondaries during the chunk migration. Any specified `writeConcern` implies `_secondaryThrottle` and will take precedent over a contradictory `_secondaryThrottle` setting.

field Boolean _waitForDelete Internal option for testing purposes. The default is `false`. If set to `true`, the delete phase of a `moveChunk` (page 310) operation blocks.

The value of `bounds` takes the form:

```
[ { hashedField : <minValue> } ,
  { hashedField : <maxValue> } ]
```

The *chunk migration* section describes how chunks move between shards on MongoDB.

See also:

`split` (page 316), `sh.moveChunk()` (page 187), `sh.splitAt()` (page 190), and `sh.splitFind()` (page 190).

Considerations Only use the `moveChunk` (page 310) in special circumstances such as preparing your *sharded cluster* for an initial ingestion of data, or a large bulk import operation. In most cases allow the balancer to create and balance chunks in sharded clusters. See `http://docs.mongodb.org/manual/tutorial/create-chunks-in-sharded-cluster` for more information.

Behavior

Indexes Changed in version 2.8.0: In previous versions, `moveChunk` (page 310) would build indexes as part of the migrations.

moveChunk (page 310) requires that all indexes exist on the target (i.e. to) shard before migration and returns an error if a required index does not exist.

Meta Data Error moveChunk (page 310) returns the following error message if another metadata operation is in progress on the chunks (page 683) collection:

```
errmsg: "The collection's metadata lock is already taken."
```

If another process, such as a balancer process, changes meta data while moveChunk (page 310) is running, you may see this error. You may retry the moveChunk (page 310) operation without side effects.

movePrimary

movePrimary

In a *sharded cluster*, movePrimary (page 312) reassigns the *primary shard* which holds all un-sharded collections in the database. movePrimary (page 312) first changes the primary shard in the cluster metadata, and then migrates all un-sharded collections to the specified *shard*. Use the command with the following form:

```
{ movePrimary : "test", to : "shard0001" }
```

When the command returns, the database's primary location will shift to the designated shard. To fully decommission a shard, use the removeShard (page 312) command.

movePrimary (page 312) is an administrative command that is only available for mongos (page 601) instances.

Considerations

Behavior Avoid accessing an un-sharded collection during migration. movePrimary (page 312) does not prevent reading and writing during its operation, and such actions yield undefined behavior.

movePrimary (page 312) may take significant time to complete, and you should plan for this unavailability.

movePrimary (page 312) will fail if the destination shard contains a conflicting collection name. This may occur if documents are written to an un-sharded collection while the collection is moved away, and later the original primary shard is restored.

Use If you use the movePrimary (page 312) command to move un-sharded collections, you must either restart all mongos (page 601) instances, or use the flushRouterConfig (page 308) command on all mongos (page 601) instances before writing any data to the cluster. This action notifies the mongos (page 601) of the new shard for the database.

If you do not update the mongos (page 601) instances' metadata cache after using movePrimary (page 312), the mongos (page 601) may not write data to the correct shard. To recover, you must manually intervene.

Additional Information See http://docs.mongodb.org/manual/tutorial/remove-shards-from-cluster for a complete procedure.

removeShard

removeShard

Removes a shard from a *sharded cluster*. When you run removeShard (page 312), MongoDB first moves the shard's chunks to other shards in the cluster. Then MongoDB removes the shard.

Behavior

Access Requirements You *must* run `removeShard` (page 312) while connected to a `mongos` (page 601). Issue the command against the `admin` database or use the `sh._adminCommand()` (page 182) helper.

If you have `authorization` enabled, you must have the `clusterManager` role or any role that includes the `removeShard` action.

Database Migration Requirements 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 databases to a new shard after migrating all data from the shard. See the `movePrimary` (page 312) command and the `http://docs.mongodb.org/manual/tutorial/remove-shards-from-cluster` for more information.

Example From the `mongo` (page 610) shell, the `removeShard` (page 312) operation resembles the following:

```
use admin
db.runCommand( { removeShard : "bristol01" } )
```

Replace `bristol01` with the name of the shard to remove. When you run `removeShard` (page 312), the command returns immediately, with the following message:

```
{
    "msg" : "draining started successfully",
    "state" : "started",
    "shard" : "bristol01",
    "ok" : 1
}
```

The balancer begins migrating chunks from the shard named `bristol01` to other shards in the cluster. These migrations happens slowly to avoid placing undue load on the overall cluster.

If you run the command again, `removeShard` (page 312) returns the following progress output:

```
{
    "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : 23,
        "dbs" : 1
    },
    "ok" : 1
}
```

The `remaining` *document* specifies how many chunks and databases remain on the shard. Use `db.printShardingStatus()` (page 122) to list the databases that you must move from the shard. Use the `movePrimary` (page 312) to move databases.

After removing all chunks and databases from the shard, you can issue `removeShard` (page 312) again see the following:

```
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "bristol01",
    "ok" : 1
}
```

setShardVersion

setShardVersion

`setShardVersion` (page 314) is an internal command that supports sharding functionality.

shardCollection

Definition

shardCollection

Enables a collection for sharding and allows MongoDB to begin distributing data among shards. You must run `enableSharding` (page 308) on a database before running the `shardCollection` (page 314) command. `shardCollection` (page 314) has the following form:

```
{ shardCollection: "<database>.<collection>", key: <shardkey> }
```

`shardCollection` (page 314) has the following fields:

field string shardCollection The *namespace* of the collection to shard in the form `<database>.<collection>`.

field document key The index specification document to use as the shard key. The index must exist prior to the `shardCollection` (page 314) command, unless the collection is empty. If the collection is empty, in which case MongoDB creates the index prior to sharding the collection. New in version 2.4: The key may be in the form `{ field : "hashed" }`, which will use the specified field as a hashed shard key.

field Boolean unique When `true`, the `unique` option ensures that the underlying index enforces a unique constraint. Hashed shard keys do not support unique constraints.

field integer numInitialChunks Specifies the number of chunks to create initially when sharding an *empty* collection with a *hashed shard key*. MongoDB will then create and balance chunks across the cluster. The `numInitialChunks` must be less than `8192` per shard. If the collection is not empty, `numInitialChunks` has no effect.

Considerations

Use Do **not** run more than one `shardCollection` (page 314) command on the same collection at the same time.

MongoDB provides no method to deactivate sharding for a collection after calling `shardCollection` (page 314). Additionally, after `shardCollection` (page 314), you cannot change shard keys or modify the value of any field used in your shard key index.

Shard Keys Choosing the best shard key to effectively distribute load among your shards requires some planning. Review *sharding-shard-key* regarding choosing a shard key.

Hashed Shard Keys New in version 2.4.

Hashed shard keys use a hashed index of a single field as the shard key.

Note: If chunk migrations are in progress while creating a hashed shard key collection, the initial chunk distribution may be uneven until the balancer automatically balances the collection.

Example The following operation enables sharding for the `people` collection in the `records` database and uses the `zipcode` field as the *shard key*:

```
db.runCommand( { shardCollection: "records.people", key: { zipcode: 1 } } )
```

Additional Information `http://docs.mongodb.org/manual/sharding`, `http://docs.mongodb.org/manual/core/sharding`, and `http://docs.mongodb.org/manual/tutorial/dep`

shardingState

shardingState

`shardingState` (page 315) is an admin command that reports if `mongod` (page 583) is a member of a *sharded cluster*. `shardingState` (page 315) has the following prototype form:

```
{ shardingState: 1 }
```

For `shardingState` (page 315) to detect that a `mongod` (page 583) is a member of a sharded cluster, the `mongod` (page 583) must satisfy the following conditions:

1. the `mongod` (page 583) is a primary member of a replica set, and
2. the `mongod` (page 583) instance is a member of a sharded cluster.

If `shardingState` (page 315) detects that a `mongod` (page 583) is a member of a sharded cluster, `shardingState` (page 315) 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 315) will return the following document:

```
{ "note" : "from execCommand", "ok" : 0, "errmsg" : "not master" }
```

The response from `shardingState` (page 315) when used with a *config server* is:

```
{ "enabled": false, "ok": 1 }
```

Note: `mongos` (page 601) instances do not provide the `shardingState` (page 315).

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.

splitChunk

Definition

splitChunk

An internal administrative command. To split chunks, use the `sh.splitFind()` (page 190) and `sh.splitAt()` (page 190) functions in the `mongo` (page 610) shell.

> **Warning:** Be careful when splitting data in a sharded collection to create new chunks. When you shard a collection that has existing data, MongoDB automatically creates chunks to evenly distribute the collection. To split data effectively in a sharded cluster you must consider the number of documents in a chunk and the average document size to create a uniform chunk size. When chunks have irregular sizes, shards may have an equal number of chunks but have very different data sizes. Avoid creating splits that lead to a collection with differently sized chunks.

See also:

`moveChunk` (page 310) and `sh.moveChunk()` (page 187).

The `splitChunk` (page 315) command takes a document with the following fields:

field string ns The complete *namespace* of the *chunk* to split.

field document keyPattern The *shard key*.

field document min The lower bound of the shard key for the chunk to split.

field document max The upper bound of the shard key for the chunk to split.

field string from The *shard* that owns the chunk to split.

field document splitKeys The split point for the chunk.

field document shardId The shard.

splitVector

splitVector

Is an internal command that supports meta-data operations in sharded clusters.

split

Definition

split

Splits a *chunk* in a *sharded cluster* into two chunks. The `mongos` (page 601) instance splits and manages chunks automatically, but for exceptional circumstances the `split` (page 316) command does allow administrators to manually create splits. See `http://docs.mongodb.org/manual/tutorial/split-chunks-in-sharded-cluster` for information on these circumstances, and on the MongoDB shell commands that wrap `split` (page 316).

The `split` (page 316) command uses the following form:

```
db.adminCommand( { split: <database>.<collection>,
                   <find|middle|bounds> } )
```

The `split` (page 316) command takes a document with the following fields:

field string split The name of the *collection* where the *chunk* exists. Specify the collection's full *namespace*, including the database name.

field document find An query statement that specifies an equality match on the shard key. The match selects the chunk that contains the specified document. You must specify only one of the following: `find`, `bounds`, or `middle`.

You cannot use the `find` option on an empty collection.

field array bounds New in version 2.4: The bounds of a chunk to split. `bounds` applies to chunks in collections partitioned using a *hashed shard key*. The parameter's array must consist of two documents specifying the lower and upper shard-key values of the chunk. The values must match the minimum and maximum values of an existing chunk. Specify only one of the following: `find`, `bounds`, or `middle`.

You cannot use the `bounds` option on an empty collection.

field document middle The document to use as the split point to create two chunks. `split` (page 316) requires one of the following options: `find`, `bounds`, or `middle`.

Considerations When used with either the `find` or the `bounds` option, the `split` (page 316) command splits the chunk along the median. As such, the command cannot use the `find` or the `bounds` option to split an empty chunk since an empty chunk has no median.

To create splits in empty chunks, use either the `middle` option with the `split` (page 316) command or use the `splitAt` command.

Command Formats To create a chunk split, connect to a `mongos` (page 601) instance, and issue the following command to the `admin` database:

```
db.adminCommand( { split: <database>.<collection>,
                   find: <document> } )
```

Or:

```
db.adminCommand( { split: <database>.<collection>,
                   middle: <document> } )
```

Or:

```
db.adminCommand( { split: <database>.<collection>,
                   bounds: [ <lower>, <upper> ] } )
```

To create a split for a collection that uses a *hashed shard key*, use the `bounds` parameter. Do *not* use the `middle` parameter for this purpose.

Warning: Be careful when splitting data in a sharded collection to create new chunks. When you shard a collection that has existing data, MongoDB automatically creates chunks to evenly distribute the collection. To split data effectively in a sharded cluster you must consider the number of documents in a chunk and the average document size to create a uniform chunk size. When chunks have irregular sizes, shards may have an equal number of chunks but have very different data sizes. Avoid creating splits that lead to a collection with differently sized chunks.

See also:

`moveChunk` (page 310), `sh.moveChunk()` (page 187), `sh.splitAt()` (page 190), and `sh.splitFind()` (page 190), which wrap the functionality of `split` (page 316).

Examples The following sections provide examples of the `split` (page 316) command.

Split a Chunk in Half

```
db.runCommand( { split : "test.people", find : { _id : 99 } } )
```

The split (page 316) command identifies the chunk in the people collection of the test database, that holds documents that match { _id : 99 }. split (page 316) does not require that a match exist, in order to identify the appropriate chunk. Then the command splits it into two chunks of equal size.

Note: split (page 316) creates two equal chunks by range as opposed to size, and does not use the selected point as a boundary for the new chunks

---

Define an Arbitrary Split Point To define an arbitrary split point, use the following form:

```
db.runCommand( { split : "test.people", middle : { _id : 99 } } )
```

The split (page 316) command identifies the chunk in the people collection of the test database, that would hold documents matching the query { _id : 99 }. split (page 316) does not require that a match exist, in order to identify the appropriate chunk. Then the command splits it into two chunks, with the matching document as the lower bound of one of the split chunks.

This form is typically used when *pre-splitting* data in a collection.

Split a Chunk Using Values of a Hashed Shard Key This example uses the *hashed shard key* userid in a people collection of a test database. The following command uses an array holding two single-field documents to represent the minimum and maximum values of the hashed shard key to split the chunk:

```
db.runCommand( { split: "test.people",
                 bounds : [ { userid: NumberLong("-5838464104018346494") },
                            { userid: NumberLong("-5557153028469814163") }
             ] } )
```

Note: MongoDB uses the 64-bit *NumberLong* type to represent the hashed value.

Use sh.status() (page 191) to see the existing bounds of the shard keys.

Metadata Lock Error If another process in the mongos (page 601), such as a balancer process, changes metadata while split (page 316) is running, you may see a metadata lock error.

```
errmsg: "The collection's metadata lock is already taken."
```

This message indicates that the split has failed with no side effects. Retry the split (page 316) command.

unsetSharding

unsetSharding

unsetSharding (page 318) is an internal command that supports sharding functionality.

See also:

http://docs.mongodb.org/manual/sharding for more information about MongoDB's sharding functionality.

Instance Administration Commands

Administration Commands

Name	Description
clean (page 319)	Internal namespace administration command.
cloneCollectionAsCapped (page 319)	Copies a non-capped collection as a new *capped collection*.
cloneCollection (page 320)	Copies a collection from a remote host to the current host.
clone (page 320)	Copies a database from a remote host to the current host.
collMod (page 321)	Add flags to collection to modify the behavior of MongoDB.
compact (page 322)	Defragments a collection and rebuilds the indexes.
connPoolSync (page 325)	Internal command to flush connection pool.
connectionStatus (page 325)	Reports the authentication state for the current connection.
convertToCapped (page 326)	Converts a non-capped collection to a capped collection.
copydb (page 326)	Copies a database from a remote host to the current host.
createIndexes (page 330)	Builds one or more indexes for a collection.
create (page 333)	Creates a collection and sets collection parameters.
dropDatabase (page 334)	Removes the current database.
dropIndexes (page 335)	Removes indexes from a collection.
drop (page 335)	Removes the specified collection from the database.
filemd5 (page 335)	Returns the *md5* hash for files stored using *GridFS*.
fsync (page 336)	Flushes pending writes to the storage layer and locks the database to allow backups.
getParameter (page 337)	Retrieves configuration options.
listCollections (page 338)	Returns a list of collections in the current database.
listIndexes (page 338)	Lists all indexes for a collection.
logRotate (page 339)	Rotates the MongoDB logs to prevent a single file from taking too much space.
reIndex (page 340)	Rebuilds all indexes on a collection.
renameCollection (page 340)	Changes the name of an existing collection.
repairCursor (page 341)	Returns a cursor that iterates over all valid documents in a collection.
repairDatabase (page 341)	Repairs any errors and inconsistencies with the data storage.
setParameter (page 343)	Modifies configuration options.
shutdown (page 344)	Shuts down the mongod (page 583) or mongos (page 601) process.
touch (page 344)	Loads documents and indexes from data storage to memory.

clean

clean

clean (page 319) is an internal command.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

cloneCollectionAsCapped

cloneCollectionAsCapped

The cloneCollectionAsCapped (page 319) command creates a new *capped collection* from an existing, non-capped collection within the same database. The operation does not affect the original non-capped collection.

The command has the following syntax:

```
{ cloneCollectionAsCapped: <existing collection>, toCollection: <capped collection>, size: <capp
```

The command copies an `existing collection` and creates a new `capped collection` with a maximum size specified by the `capped size` in bytes. The name of the new capped collection must be distinct and cannot be the same as that of the original existing collection. To replace the original non-capped collection with a capped collection, use the `convertToCapped` (page 326) command.

During the cloning, the `cloneCollectionAsCapped` (page 319) command exhibit the following behavior:

- MongoDB will transverse the documents in the original collection in *natural order* as they're loaded.
- If the `capped size` specified for the new collection is smaller than the size of the original uncapped collection, then MongoDB will begin overwriting earlier documents in insertion order, which is *first in, first out* (e.g "FIFO").

cloneCollection

Definition

`cloneCollection`

Copies a collection from a remote `mongod` (page 583) instance to the current `mongod` (page 583) instance. `cloneCollection` (page 320) creates a collection in a database with the same name as the remote collection's database. `cloneCollection` (page 320) takes the following form:

```
{ cloneCollection: "<namespace>", from: "<hostname>", query: { <query> } }
```

Important: You cannot clone a collection through a `mongos` (page 601) but must connect directly to the `mongod` (page 583) instance.

`cloneCollection` (page 320) has the following fields:

field string cloneCollection The *namespace* of the collection to rename. The namespace is a combination of the database name and the name of the collection.

field string from Specify a resolvable hostname and optional port number of the remote server where the specified collection resides.

field document query A query that filters the documents in the remote collection that `cloneCollection` (page 320) will copy to the current database.

Example

```
{ cloneCollection: "users.profiles", from: "mongodb.example.net:27017", query: { active: true } }
```

This operation copies the `profiles` collection from the `users` database on the server at `mongodb.example.net`. The operation only copies documents that satisfy the query `{ active: true }`. `cloneCollection` (page 320) always copies indexes. The `query` arguments is optional.

If, in the above example, the `profiles` collection exists in the `users` database, then MongoDB appends documents from the remote collection to the destination collection.

clone

`clone`

The `clone` (page 320) command clones a database from a remote MongoDB instance to the current host. `clone` (page 320) copies the database on the remote instance with the same name as the current database. The command takes the following form:

```
{ clone: "db1.example.net:27017" }
```

Replace `db1.example.net:27017` above with the resolvable hostname for the MongoDB instance you wish to copy from. Note the following behaviors:

- `clone` (page 320) can copy from a non-*primary* member of a *replica set*.
- `clone` (page 320) does not snapshot the database. If any clients update the database you're copying at any point during the clone operation, the resulting database may be inconsistent.
- You must run `clone` (page 320) on the **destination server**.
- The destination database will be locked periodically during the `clone` (page 320) operation. In other words, `clone` (page 320) will occasionally yield to allow other operations on the database to complete.

See `copydb` (page 326) for similar functionality with greater flexibility.

collMod

Definition

`collMod`

New in version 2.2.

`collMod` (page 321) makes it possible to add flags to a collection to modify the behavior of MongoDB. Flags include `usePowerOf2Sizes` (page 322) and `index` (page 321). The command takes the following prototype form:

```
db.runCommand( {"collMod" : <collection> , "<flag>" : <value> } )
```

In this command substitute `<collection>` with the name of a collection in the current database, and `<flag>` and `<value>` with the flag and value you want to set.

Use the `userFlags` (page 349) field in the `db.collection.stats()` (page 71) output to check enabled collection flags.

Flags

TTL Collection Expiration Time

`index`

The `index` (page 321) flag changes the expiration time of a `TTL Collection`.

Specify the key and new expiration time with a document of the form:

```
{keyPattern: <index_spec>, expireAfterSeconds: <seconds> }
```

In this example, `<index_spec>` is an existing index in the collection and `seconds` is the number of seconds to subtract from the current time.

On success `collMod` (page 321) returns a document with fields `expireAfterSeconds_old` and `expireAfterSeconds_new` set to their respective values.

On failure, `collMod` (page 321) returns a document with `no expireAfterSeconds field to update` if there is no existing `expireAfterSeconds` field or `cannot find index { **key**: 1.0 } for ns **namespace**` if the specified `keyPattern` does not exist.

Record Allocation

Storage Engine Specific Feature

The power of two allocation strategy and `noPadding` (page 322) feature are only available with the `mmapv1` storage engine.

Disable All Record Padding

`noPadding`

New in version 2.8.0.

Disables all record padding. All records allocations are the size needed to hold the documents. All updates that require the document to grow will require a new allocation.

Only use `noPadding` (page 322) for collections where there are *no* delete operations or update operations that cause documents to grow.

Powers of Two Record Allocation

`usePowerOf2Sizes`

Deprecated since version 2.8: All collections have the `usePowerOf2Sizes` (page 322) allocation strategy by default unless you specify the `noPadding` option to `db.createCollection()` (page 104) or set `noPadding` (page 322).

The `usePowerOf2Sizes` (page 322) flag changes the method that MongoDB uses to allocate space on disk for documents in this collection. By setting `usePowerOf2Sizes` (page 322), yousthat are powers of 2 (e.g. 32, 64, 128, 256, 512...16777216.) The smallest allocation for a document is 32 bytes.

With `usePowerOf2Sizes` (page 322), MongoDB will be able to more effectively reuse space.

With `usePowerOf2Sizes` (page 322), MongoDB allocates records that have power of 2 sizes until record sizes equal 4 megabytes. For records larger than 4 megabytes with `usePowerOf2Sizes` (page 322) set, `mongod` (page 583) will allocate records in full megabytes by rounding up to the nearest megabyte.

Example: Change Expiration Value for Indexes To update the expiration value for a collection named `sessions` indexed on a `lastAccess` field from 30 minutes to 60 minutes, use the following operation:

```
db.runCommand({collMod: "sessions",
               index: {keyPattern: {lastAccess:1},
                       expireAfterSeconds: 3600}})
```

Which will return the document:

```
{ "expireAfterSeconds_old" : 1800, "expireAfterSeconds_new" : 3600, "ok" : 1 }
```

compact

Definition

`compact`

New in version 2.0.

Rewrites and defragments all data in a collection, as well as all of the indexes on that collection. `compact` (page 322) has the following form:

```
{ compact: <collection name> }
```

`compact` (page 322) has the following fields:

field string compact The name of the collection.

field boolean force If `true`, `compact` (page 322) can run on the *primary* in a *replica set*. If `false`, `compact` (page 322) returns an error when run on a primary, because the command blocks all other activity. Beginning with version 2.2, `compact` (page 322) blocks activity only for the database it is compacting.

field number paddingFactor Describes the *record size* allocated for each document as a factor of the document size for all records compacted during the `compact` (page 322) operation. The `paddingFactor` does not affect the padding of subsequent record allocations after `compact` (page 322) completes. For more information, see *paddingFactor* (page 323).

field integer paddingBytes Sets the padding as an absolute number of bytes for all records compacted during the `compact` (page 322) operation. After `compact` (page 322) completes, `paddingBytes` does not affect the padding of subsequent record allocations. For more information, see *paddingBytes* (page 323).

`compact` (page 322) is similar to `repairDatabase` (page 341); however, `repairDatabase` (page 341) operates on an entire database.

Warning: Always have an up-to-date backup before performing server maintenance such as the `compact` (page 322) operation.

paddingFactor New in version 2.2.

The `paddingFactor` field takes the following range of values:

- Default: `1.0`
- Minimum: `1.0` (no padding)
- Maximum: `4.0`

If your updates increase the size of the documents, padding will increase the amount of space allocated to each document and avoid expensive document relocation operations within the data files.

You can calculate the padding size by subtracting the document size from the record size or, in terms of the `paddingFactor`, by subtracting `1` from the `paddingFactor`:

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

For example, a `paddingFactor` of `1.0` specifies a padding size of `0` whereas a `paddingFactor` of `1.2` specifies a padding size of `0.2` or 20 percent (20%) of the document size.

With the following command, you can use the `paddingFactor` option of the `compact` (page 322) command to set the record size to `1.1` of the document size, or a padding factor of 10 percent (10%):

```
db.runCommand ( { compact: '<collection>', paddingFactor: 1.1 } )
```

`compact` (page 322) compacts existing documents but does not reset `paddingFactor` statistics for the collection. After the `compact` (page 322) MongoDB will use the existing `paddingFactor` when allocating new records for documents in this collection.

paddingBytes New in version 2.2.

Specifying `paddingBytes` can be useful if your documents start small but then increase in size significantly. For example, if your documents are initially 40 bytes long and you grow them by 1KB, using `paddingBytes: 1024` might be reasonable since using `paddingFactor: 4.0` would specify a record size of 160 bytes (`4.0` times the initial document size), which would only provide a padding of 120 bytes (i.e. record size of 160 bytes minus the document size).

With the following command, you can use the `paddingBytes` option of the `compact` (page 322) command to set the padding size to `100` bytes on the collection named by `<collection>`:

```
db.runCommand ( { compact: '<collection>', paddingBytes: 100 } )
```

Behaviors

Blocking In MongoDB 2.2, `compact` (page 322) blocks activities only for its database. Prior to 2.2, the command blocked all activities.

You may view the intermediate progress either by viewing the `mongod` (page 583) log file or by running the `db.currentOp()` (page 105) in another shell instance.

Operation Termination If you terminate the operation with the `db.killOp()` (page 119) method or restart the server before the `compact` (page 322) operation has finished:

- If you have journaling enabled, the data remains valid and usable, regardless of the state of the `compact` (page 322) operation. You may have to manually rebuild the indexes.
- If you do not have journaling enabled and the `mongod` (page 583) or `compact` (page 322) terminates during the operation, it is impossible to guarantee that the data is in a valid state.
- In either case, much of the existing free space in the collection may become un-reusable. In this scenario, you should rerun the compaction to completion to restore the use of this free space.

Disk Space `compact` (page 322) generally uses less disk space than `repairDatabase` (page 341) and is faster. However, the `compact` (page 322) command is still slow and blocks other database use. Only use `compact` (page 322) during scheduled maintenance periods.

`compact` (page 322) requires up to 2 gigabytes of additional disk space while running. Unlike `repairDatabase` (page 341), `compact` (page 322) does *not* free space on the file system.

To see how the storage space changes for the collection, run the `collStats` (page 348) command before and after compaction.

Size and Number of Data Files `compact` (page 322) may increase the total size and number of your data files, especially when run for the first time. However, this will not increase the total collection storage space since storage size is the amount of data allocated within the database files, and not the size/number of the files on the file system.

Replica Sets `compact` (page 322) commands do not replicate to secondaries in a *replica set*:

- Compact each member separately.
- Ideally run `compact` (page 322) on a secondary. See option `force:true` above for information regarding compacting the primary.
- On secondaries, the `compact` (page 322) command forces the secondary to enter `RECOVERING` state. Read operations issued to an instance in the `RECOVERING` state will fail. This prevents clients from reading during the operation. When the operation completes, the secondary returns to:replstate:*SECONDARY* state.
- See `http://docs.mongodb.org/manual/reference/replica-states/` for more information about replica set member states.

See `http://docs.mongodb.org/manual/tutorial/perform-maintence-on-replica-set-members` for an example replica set maintenance procedure to maximize availability during maintenance operations.

Sharded Clusters `compact` (page 322) is a command issued to a `mongod` (page 583). In a sharded environment, run `compact` (page 322) on each shard separately as a maintenance operation.

You cannot issue `compact` (page 322) against a `mongos` (page 601) instance.

Capped Collections It is not possible to compact *capped collections* because they don't have padding, and documents cannot grow in these collections. However, the documents of a *capped collection* are not subject to fragmentation.

Index Building New in version 2.6.

`mongod` (page 583) rebuilds all indexes in parallel following the `compact` (page 322) operation.

connPoolSync

`connPoolSync`

`connPoolSync` (page 325) is an internal command.

connectionStatus New in version 2.4.0.

Definition

`connectionStatus`

Returns information about the current connection, specifically the state of authenticated users and their available permissions.

Output

`connectionStatus.`**`authInfo`**

A document with data about the authentication state of the current collection, including users and available permissions.

`connectionStatus.authinfo.`**`authenticatedUsers`**

An array with documents for each authenticated user.

`connectionStatus.authInfo.authenticatedUsers[n].`**`user`**

The user's name.

`connectionStatus.authInfo.authenticatedUsers[n].`**`db`**

The database associated with the user's credentials.

`connectionStatus.authinfo.`**`authenticatedUserRoles`**

An array with documents for each role granted to the current connection:

`connectionStatus.authinfo.authenticatedUserRoles[n].`**`role`**

The definition of the current roles associated with the current authenticated users. See `http://docs.mongodb.org/manual/reference/built-in-roles` and `http://docs.mongodb.org/manual/reference/privilege-actions` for more information.

`connectionStatus.authinfo.authenticatedUserRoles[n].`**`db`**

The database to which `role` (page 325) applies.

`connectionStatus.`**`ok`**

The return value for the command. A value of `1` indicates success.

convertToCapped

convertToCapped

The convertToCapped (page 326) command converts an existing, non-capped collection to a *capped collection* within the same database.

The command has the following syntax:

```
{ convertToCapped: <collection>, size: <capped size> }
```

convertToCapped (page 326) takes an existing collection (<collection>) and transforms it into a capped collection with a maximum size in bytes, specified by the size argument (<capped size>).

During the conversion process, the convertToCapped (page 326) command exhibits the following behavior:

- MongoDB traverses the documents in the original collection in *natural order* and loads the documents into a new capped collection.
- If the capped size specified for the capped collection is smaller than the size of the original uncapped collection, then MongoDB will overwrite documents in the capped collection based on insertion order, or *first in, first out* order.
- Internally, to convert the collection, MongoDB uses the following procedure
  - cloneCollectionAsCapped (page 319) command creates the capped collection and imports the data.
  - MongoDB drops the original collection.
  - renameCollection (page 340) renames the new capped collection to the name of the original collection.

Note: MongoDB does not support the convertToCapped (page 326) command in a sharded cluster.

Warning: The convertToCapped (page 326) will not recreate indexes from the original collection on the new collection, other than the index on the _id field. If you need indexes on this collection you will need to create these indexes after the conversion is complete.

See also:

create (page 333)

copydb

Definition

copydb

Copies a database from a remote host to the current host or copies a database to another database within the current host. Run copydb (page 326) in the admin database of the destination server with the following syntax:

```
{ copydb: 1,
  fromhost: <hostname>,
  fromdb: <database>,
  todb: <database>,
  slaveOk: <bool>,
  username: <username>,
  nonce: <nonce>,
  key: <key> }
```

copydb (page 326) accepts the following options:

field string fromhost Hostname of the remote source mongod (page 583) instance. Omit fromhost to copy from one database to another on the same server.

field string fromdb Name of the source database.

field string todb Name of the target database.

field boolean slaveOk Set slaveOK to true to allow copydb (page 326) to copy data from secondary members as well as the primary. fromhost must also be set.

field string username The username credentials on the fromhost MongoDB deployment.

field string nonce A single use shared secret generated on the remote server, i.e. fromhost, using the copydbgetnonce (page 264) command. See *Authentication* (page 328) for details.

field string key A hash of the password used for authentication. See *Authentication* (page 328) for details.

The mongo (page 610) shell provides the db.copyDatabase() (page 102) wrapper for the copydb (page 326) command.

Behavior Be aware of the following properties of copydb (page 326):

- copydb (page 326) runs on the destination mongod (page 583) instance, i.e. the host receiving the copied data.
- If the destination mongod (page 583) has authorization enabled, copydb (page 326) *must* specify the credentials of a user present in the *source* database who has the privileges described in *Required Access* (page 102).
- copydb (page 326) creates the target database if it does not exist.
- copydb (page 326) requires enough free disk space on the host instance for the copied database. Use the db.stats() (page 126) operation to check the size of the database on the source mongod (page 583) instance.
- copydb (page 326) and clone (page 320) do not produce point-in-time snapshots of the source database. Write traffic to the source or destination database during the copy process will result in divergent data sets.
- copydb (page 326) does not lock the destination server during its operation, so the copy will occasionally yield to allow other operations to complete.

Required Access Changed in version 2.6.

On systems running with authorization, the copydb (page 326) command requires the following authorization on the target and source databases.

Source Database (fromdb)

Source is non-admin Database If the source database is a non-admin database, you must have privileges that specify find action on the source database, and find action on the system.js collection in the source database. For example:

```
{ resource: { db: "mySourceDB", collection: "" }, actions: [ "find" ] }
{ resource: { db: "mySourceDB", collection: "system.js" }, actions: [ "find" ] }
```

If the source database is on a remote server, you also need the find action on the system.indexes and system.namespaces collections in the source database; e.g.

```
{ resource: { db: "mySourceDB", collection: "system.indexes" }, actions: [ "find" ] }
{ resource: { db: "mySourceDB", collection: "system.namespaces" }, actions: [ "find" ] }
```

Source is `admin` Database If the source database is the `admin` database, you must have privileges that specify `find` action on the `admin` database, and `find` action on the `system.js`, `system.users`, `system.roles`, and `system.version` collections in the `admin` database. For example:

```
{ resource: { db: "admin", collection: "" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.js" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.users" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.roles" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.version" }, actions: [ "find" ] }
```

If the source database is on a remote server, the you also need the `find` action on the `system.indexes` and `system.namespaces` collections in the `admin` database; e.g.

```
{ resource: { db: "admin", collection: "system.indexes" }, actions: [ "find" ] }
{ resource: { db: "admin", collection: "system.namespaces" }, actions: [ "find" ] }
```

Source Database is on a Remote Server If copying from a remote server and the remote server has authentication enabled, you must authenticate to the remote host as a user with the proper authorization. See *Authentication* (page 328).

Target Database (`todb`)

Copy from non-`admin` Database If the source database is not the `admin` database, you must have privileges that specify `insert` and `createIndex` actions on the target database, and `insert` action on the `system.js` collection in the target database. For example:

```
{ resource: { db: "myTargetDB", collection: "" }, actions: [ "insert", "createIndex" ] }
{ resource: { db: "myTargetDB", collection: "system.js" }, actions: [ "insert" ] }
```

Copy from `admin` Database If the source database is the `admin` database, you must have privileges that specify `insert` and `createIndex` actions on the target database, and `insert` action on the `system.js`, `system.users`, `system.roles`, and `system.version` collections in the target database. For example:

```
{ resource: { db: "myTargetDB", collection: "" }, actions: [ "insert", "createIndex" ] },
{ resource: { db: "myTargetDB", collection: "system.js" }, actions: [ "insert" ] },
{ resource: { db: "myTargetDB", collection: "system.users" }, actions: [ "insert" ] },
{ resource: { db: "myTargetDB", collection: "system.roles" }, actions: [ "insert" ] },
{ resource: { db: "myTargetDB", collection: "system.version" }, actions: [ "insert" ] }
```

Authentication If copying from a remote server and the remote server has authentication enabled, then you must include a `username`, `nonce`, and `key`.

The `nonce` is a one-time password that you request from the remote server using the `copydbgetnonce` (page 264) command, as in the following:

```
use admin
mynonce = db.runCommand( { copydbgetnonce : 1, fromhost: <hostname> } ).nonce
```

If running the copydbgetnonce (page 264) command directly on the remote host, you can omit the fromhost field in the copydbgetnonce (page 264) command.

The key is a hash generated as follows:

```
hex_md5(mynonce + username + hex_md5(username + ":mongo:" + password))
```

Replica Sets With *read preference* configured to set the slaveOk option to true, you may run copydb (page 326) on a *secondary* member of a *replica set*.

Sharded Clusters

- Do not use copydb (page 326) from a mongos (page 601) instance.
- Do not use copydb (page 326) to copy databases that contain sharded collections.

Examples

Copy on the Same Host To copy from the same host, omit the fromhost field.

The following command copies the test database to a new records database on the current mongod (page 583) instance:

```
use admin
db.runCommand({
   copydb: 1,
   fromdb: "test",
   todb: "records"
})
```

Copy from a Remote Host to the Current Host To copy from a remote host, include the fromhost field.

The following command copies the test database from the remote host example.net to a new records database on the current mongod (page 583) instance:

```
use admin
db.runCommand({
   copydb: 1,
   fromdb: "test",
   todb: "records",
   fromhost: "example.net"
})
```

Copy Databases from Remote mongod Instances that Enforce Authentication To copy from a remote host that enforces authentication, include the fromhost, username, nonce and key fields.

The following command copies the test database from a remote host example.net that runs with authorization to a new records database on the local mongod (page 583) instance. Because the example.net has authorization enabled, the command includes the username, nonce and key fields:

```
use admin
db.runCommand({
   copydb: 1,
   fromdb: "test",
   todb: "records",
```

```
   fromhost: "example.net",
   username: "reportingAdmin",
   nonce: "<nonce>",
   key: "<passwordhash>"
})
```

See also:

- db.copyDatabase() (page 102)
- clone (page 320) and db.cloneDatabase() (page 101)
- http://docs.mongodb.org/manual/core/backups and http://docs.mongodb.org/manual/core/impo

createIndexes New in version 2.6.

Definition

createIndexes

Builds one or more indexes on a collection. The createIndexes (page 330) command takes the following form:

```
db.runCommand(
  {
    createIndexes: <collection>,
    indexes: [
        {
            key: {
                <key-value_pair>,
                <key-value_pair>,
                ...
            },
            name: <index_name>,
            <option1>,
            <option2>,
            ...
        },
        { ... },
        { ... }
    ]
  }
)
```

The createIndexes (page 330) command takes the following fields:

field string createIndexes The collection for which to create indexes.

field array indexes Specifies the indexes to create. Each document in the array specifies a separate index.

Each document in the indexes array can take the following fields:

field document key Specifies the index's fields. For each field, specify a key-value pair in which the key is the name of the field to index and the value is either the index direction or index type. If specifying direction, specify 1 for ascending or -1 for descending.

field string name A name that uniquely identifies the index.

field string ns The *namespace* (i.e. <database>.<collection>) of the collection for which to create the index. If you omit ns, MongoDB generates the namespace.

param Boolean background Builds the index in the background so that building an index does *not* block other database activities. Specify `true` to build in the background. The default value is `false`.

param Boolean unique Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify `true` to create a unique index. The default value is `false`.

The option is *unavailable* for `hashed` indexes.

param Boolean dropDups Creates a unique index on a field that *may* have duplicates. MongoDB indexes only the first occurrence of a key and **removes** all documents from the collection that contain subsequent occurrences of that key. Specify `true` to create unique index. The default value is `false`.

The option is *unavailable* for `hashed` indexes.

Deprecated since version 2.6.

> **Warning:** `dropDups` will delete data from your collection when building the index.

param Boolean sparse If `true`, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is `false`. See `http://docs.mongodb.org/manual/core/index-sparse` for more information.

Changed in version 2.6: `2dsphere` indexes are sparse by default and ignore this option. For a compound index that includes `2dsphere` index key(s) along with keys of other types, only the `2dsphere` index fields determine whether the index references a document.

`2d`, `geoHaystack`, and `text` indexes behave similarly to the `2dsphere` indexes.

param integer expireAfterSeconds Specifies a value, in seconds, as a *TTL* to control how long MongoDB retains documents in this collection. See `http://docs.mongodb.org/manual/tutorial/expire-data` for more information on this functionality. This applies only to *TTL* indexes.

param index version v The index version number. The default index version depends on the version of `mongod` (page 583) running when creating the index. Before version 2.0, the this value was 0; versions 2.0 and later use version 1, which provides a smaller and faster index format. Specify a different index version *only* in unusual situations.

param document storageEngine New in version 2.8.

Allows users to specify configuration to the storage engine on a per-index basis when creating an index. The value of the `storageEngine` option should take the following form:

```
{ <storage-engine-name>: <options> }
```

Storage engine configuration specified when creating indexes are validated and logged to the *oplog* during replication to support replica sets with members that use different storage engines.

param document weights For `text` indexes, a document that contains field and weight pairs. The weight is an integer ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score. You can specify weights for some or all the indexed fields. See `http://docs.mongodb.org/manual/tutorial/control-results-of-text-search` to adjust the scores. The default value is `1`.

param string default_language For `text` indexes, the language that determines the list of stop words and the rules for the stemmer and tokenizer. See *text-search-languages* for the available languages and `http://docs.mongodb.org/manual/tutorial/specify-language-for-text-index` for more information and examples. The default value is `english`.

param string language_override For `text` indexes, the name of the field, in the collection's documents, that contains the override language for the document. The default value is `language`. See *specify-language-field-text-index-example* for an example.

param integer textIndexVersion For `text` indexes, the `text` index version number. Version can be either `1` or `2`.

In MongoDB 2.6, the default version is `2`. MongoDB 2.4 can only support version `1`.

New in version 2.6.

param integer 2dsphereIndexVersion For `2dsphere` indexes, the `2dsphere` index version number. Version can be either `1` or `2`.

In MongoDB 2.6, the default version is `2`. MongoDB 2.4 can only support version `1`.

New in version 2.6.

param integer bits For `2d` indexes, the number of precision of the stored *geohash* value of the location data.

The `bits` value ranges from 1 to 32 inclusive. The default value is `26`.

param number min For `2d` indexes, the lower inclusive boundary for the longitude and latitude values. The default value is `-180.0`.

param number max For `2d` indexes, the upper inclusive boundary for the longitude and latitude values. The default value is `180.0`.

param number bucketSize For `geoHaystack` indexes, specify the number of units within which to group the location values; i.e. group in the same bucket those location values that are within the specified number of units to each other.

The value must be greater than 0.

Considerations An index name, including the *namespace*, cannot be longer than the *Index Name Length* (page 693) limit.

Behavior Non-background indexing operations block all other operations on a database. If you specify multiple indexes to `createIndexes` (page 330), MongoDB builds the indexes serially.

If you create an index with one set of options and then issue `createIndexes` (page 330) with the same index fields but different options, MongoDB will not change the options nor rebuild the index. To change index options, drop the existing index with `db.collection.dropIndex()` (page 29) before running the new `createIndexes` (page 330) with the new options.

Example The following command builds two indexes on the `inventory` collection of the `products` database:

```
db.getSiblingDB("products").runCommand(
  {
    createIndexes: "inventory",
    indexes: [
        {
            key: {
```

```
                item: 1,
                manufacturer: 1,
                model: 1
            },
            name: "item_manufacturer_model",
            unique: true
        },
        {
            key: {
                item: 1,
                supplier: 1,
                model: 1
            },
            name: "item_supplier_model",
            unique: true
        }
    ]
  }
)
```

When the indexes successfully finish building, MongoDB returns a results document that includes a status of `"ok" : 1`.

Output The `createIndexes` (page 330) command returns a document that indicates the success of the operation. The document contains some but not all of the following fields, depending on outcome:

`createIndexes.`**`createdCollectionAutomatically`**
: If `true`, then the collection didn't exist and was created in the process of creating the index.

`createIndexes.`**`numIndexesBefore`**
: The number of indexes at the start of the command.

`createIndexes.`**`numIndexesAfter`**
: The number of indexes at the end of the command.

`createIndexes.`**`ok`**
: A value of `1` indicates the indexes are in place. A value of `0` indicates an error.

`createIndexes.`**`note`**
: This `note` is returned if an existing index or indexes already exist. This indicates that the index was not created or changed.

`createIndexes.`**`errmsg`**
: Returns information about any errors.

`createIndexes.`**`code`**
: The error code representing the type of error.

create

Definition

`create`
: Explicitly creates a collection. `create` (page 333) has the following form:

```
{ create: <collection_name>,
  capped: <true|false>,
  autoIndexId: <true|false>,
```

```
  size: <max_size>,
  max: <max_documents>,
  flags: <0|1>
}
```

`create` (page 333) has the following fields:

field string create The name of the new collection.

field Boolean capped To create a *capped collection*. specify `true`. If you specify `true`, you must also set a maximum size in the `size` field.

field Boolean autoIndexId Specify `false` to disable the automatic creation of an index on the `_id` field. Before 2.2, the default value for `autoIndexId` was `false`.

field integer size The maximum size for the capped collection. Once a capped collection reaches its maximum size, MongoDB overwrites older old documents with new documents. The `size` field is required for capped collections.

field integer max The maximum number of documents to keep in the capped collection. The `size` limit takes precedence over this limit. If a capped collection reaches its maximum size before it reaches the maximum number of documents, MongoDB removes old documents. If you use this limit, ensure that the `size` limit is sufficient to contain the documents limit.

field integer flags New in version 2.6.

Set to `0` to disable the `usePowerOf2Sizes` (page 322) allocation strategy for this collection, or `1` to enable `usePowerOf2Sizes` (page 322). Defaults to `1` unless the `newCollectionsUsePowerOf2Sizes` parameter is set to `false`.

For more information on the `autoIndexId` field in versions before 2.2, see *_id Fields and Indexes on Capped Collections* (page 798).

The `db.createCollection()` (page 104) method wraps the `create` (page 333) command.

Considerations The `create` (page 333) command obtains a write lock on the affected database and will block other operations until it has completed. The write lock for this operation is typically short lived. However, allocations for large capped collections may take longer.

Example To create a *capped collection* limited to 64 kilobytes, issue the command in the following form:

```
db.runCommand( { create: "collection", capped: true, size: 64 * 1024 } )
```

dropDatabase

`dropDatabase`

The `dropDatabase` (page 334) command drops the current database, deleting the associated data files.

Changed in version 2.6: This command does not delete the *users* associated with the current database. To drop the associated users, run the `dropAllUsersFromDatabase` (page 266) command in the database you are deleting.

To run this command, issue the `use <database>` command in the shell, replacing `<database>` with the name of the database you wish to delete. Then use the following command form:

```
{ dropDatabase: 1 }
```

The `mongo` (page 610) shell also provides the following equivalent helper method `db.dropDatabase()` (page 111).

> **Warning:** This command obtains a global write lock and will block other operations until it has completed.

See also:

dropAllUsersFromDatabase (page 266)

dropIndexes

dropIndexes

The dropIndexes (page 335) command drops one or all indexes from the current collection. To drop all indexes, issue the command like so:

```
{ dropIndexes: "collection", index: "*" }
```

To drop a single, issue the command by specifying the name of the index you want to drop. For example, to drop the index named age_1, use the following command:

```
{ dropIndexes: "collection", index: "age_1" }
```

The shell provides a useful command helper. Here's the equivalent command:

```
db.collection.dropIndex("age_1");
```

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

drop

drop

The drop (page 335) command removes an entire collection from a database. The command has following syntax:

```
{ drop: <collection_name> }
```

The mongo (page 610) shell provides the equivalent helper method db.collection.drop() (page 28).

This command also removes any indexes associated with the dropped collection.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

filemd5

filemd5

The filemd5 (page 335) command returns the *md5* hashes for a single file stored using the *GridFS* specification. Client libraries use this command to verify that files are correctly written to MongoDB. The command takes the files_id of the file in question and the name of the GridFS root collection as arguments. For example:

```
{ filemd5: ObjectId("4f1f10e37671b50e4ecd2776"), root: "fs" }
```

MongoDB computes the filemd5 using all data in the GridFS file object pulled sequentially from each chunk in the chunks collection.

fsync

Definition

fsync

Forces the mongod (page 583) process to flush all pending writes from the storage layer to disk. Optionally, you can use fsync (page 336) to lock the mongod (page 583) instance and block write operations for the purpose of capturing backups.

As applications write data, MongoDB records the data in the storage layer and then writes the data to disk within the syncPeriodSecs interval, which is 60 seconds by default. Run fsync (page 336) when you want to flush writes to disk ahead of that interval.

The fsync (page 336) command has the following syntax:

```
{ fsync: 1, async: <Boolean>, lock: <Boolean> }
```

The fsync (page 336) command has the following fields:

field integer fsync Enter "1" to apply fsync (page 336).

field Boolean async Runs fsync (page 336) asynchronously. By default, the fsync (page 336) operation is synchronous.

field Boolean lock Locks mongod (page 583) instance and blocks all write operations.

Behavior An fsync (page 336) lock is only possible on *individual* mongod (page 583) instances of a sharded cluster, not on the entire cluster. To backup an entire sharded cluster, please see http://docs.mongodb.org/manual/administration/backup-sharded-clusters for more information.

If your mongod (page 583) has *journaling* enabled, consider using *another method* to create a back up of the data set.

After fsync (page 336), with lock, runs on a mongod (page 583), all write operations will block until a subsequent unlock. Read operations *may* also block. As a result, fsync (page 336), with lock, is not a reliable mechanism for making a mongod (page 583) instance operate in a read-only mode.

Important: Blocked read operations prevent verification of authentication. Such reads are necessary to establish new connections to a mongod (page 583) that enforces authorization checks.

Warning: When calling fsync (page 336) with lock, ensure that the connection remains open to allow a subsequent call to db.fsyncUnlock() (page 114).
Closing the connection may make it difficult to release the lock.

Examples

Run Asynchronously The fsync (page 336) operation is synchronous by default To run fsync (page 336) asynchronously, use the async field set to true:

```
{ fsync: 1, async: true }
```

The operation returns immediately. To view the status of the fsync (page 336) operation, check the output of db.currentOp() (page 105).

Lock mongod Instance The primary use of fsync (page 336) is to lock the mongod (page 583) instance in order to back up the files withing mongod (page 583)'s dbPath. The operation flushes all data to the storage layer and blocks all write operations until you unlock the mongod (page 583) instance.

To lock the database, use the `lock` field set to `true`:

```
{ fsync: 1, lock: true }
```

You may continue to perform read operations on a `mongod` (page 583) instance that has a `fsync` (page 336) lock. However, after the first write operation all subsequent read operations wait until you unlock the `mongod` (page 583) instance.

Unlock `mongod` Instance To unlock the `mongod` (page 583), use `db.fsyncUnlock()` (page 114):

```
db.fsyncUnlock();
```

Check Lock Status To check the state of the fsync lock, use `db.currentOp()` (page 105). Use the following JavaScript function in the shell to test if `mongod` (page 583) instance is currently locked:

```
serverIsLocked = function () {
                     var co = db.currentOp();
                     if (co && co.fsyncLock) {
                         return true;
                     }
                     return false;
                 }
```

After loading this function into your `mongo` (page 610) shell session call it, with the following syntax:

```
serverIsLocked()
```

This function will return `true` if the `mongod` (page 583) instance is currently locked and `false` if the `mongod` (page 583) is not locked.

getParameter

`getParameter`

`getParameter` (page 337) is an administrative command for retrieving the value of options normally set on the command line. Issue commands against the *admin database* as follows:

```
{ getParameter: 1, <option>: 1 }
```

The values specified for `getParameter` and `<option>` do not affect the output. The command works with the following options:

- **quiet**
- **notablescan**
- **logLevel**
- **syncdelay**

See also:

`setParameter` (page 343) for more about these parameters.

listCollections New in version 2.8.0.

Definition

listCollections

Returns an array with a document for each collection in the database.

field document filter A query expression that filters the contents of the collections (page 338) array in the listCollections (page 338) result document.

Output

listCollections.**collections**

An array of documents for each collection.

listCollections.collections[n].**name**

The name of the collection.

listCollections.collections[n].**options**

A document with a mapping of collection options and their values.

listCollections.**ok**

The return value for the command. A value of 1 indicates success.

Examples

List All Collections To return all collections in the records database use the following operation in the mongo (page 610) shell:

```
use records
db.runCommand( { listCollections: 1 } );
```

Consider the following output for a database that only has a users collection:

```
{
     "collections" : [
             {
                     "name" : "users",
                     "options" : {
                             "flags" : 1
                     }
             }
     ],
     "ok" : 1
}
```

List Matching Collections Use the filter argument to the listCollections (page 338) command to specify a query that will limit the collections returned in the list, as in the following examples:

```
use records
db.runCommand( { listCollections: 1, filter: { capped: true } } );
db.runCommand( { listCollections: 1, filter: { name: { $regex: "^users\..*" } } } );
```

listIndexes New in version 2.8.0.

Definition

listIndexes

Returns information about the indexes on the specified collection. The indexes (page 339) field contains an array with documents for every index that displays important information about the index's definition. Consider the following command document:

```
{ "listIndexes": "<collection-name>" }
```

field string collection The name of the collection.

Output

listIndexes.**indexes**

listIndexes.indexes[n].**v**
The index version.

listIndexes.indexes[n].**key**
A document that contains the index specification.

listIndexes.indexes[n].**name**
The name of the index.

listIndexes.indexes[n].**ns**
The *namespace* of the collection where the index exists.

listIndexes.indexes[n].**background**
A boolean that is true when the index is built in the background. Only appears in indexes (page 339) when true.

listIndexes.indexes[n].**sparse**
A boolean that is true when the index has a sparse filter. Only appears in indexes (page 339) when true.

listIndexes.indexes[n].**unique**
A boolean that is true when the index has a unique constraint. Only appears in indexes (page 339) when true.

listIndexes.indexes[n].**expireAfterSeconds**
The number of seconds. See http://docs.mongodb.org/manual/tutorial/expire-data for more information about data expiration. Only appears for indexes with that expire data.

listIndexes.indexes[n].**bucketSize**
The size of the buckets for geoHaystack indexes. See http://docs.mongodb.org/manual/core/geohaystack for more information. Only appears for geoHaystack indexes.

listIndexes.**ok**
The return value for the command. A value of 1 indicates success.

logRotate

logRotate

The logRotate (page 339) command is an administrative command that allows you to rotate the MongoDB logs to prevent a single logfile from consuming too much disk space. You must issue the logRotate (page 339) command against the *admin database* in the form:

```
{ logRotate: 1 }
```

Note: Your mongod (page 583) instance needs to be running with the *--logpath [file]* option.

You may also rotate the logs by sending a SIGUSR1 signal to the mongod (page 583) process. If your mongod (page 583) has a process ID of 2200, here's how to send the signal on Linux:

```
kill -SIGUSR1 2200
```

logRotate (page 339) renames the existing log file by appending the current timestamp to the filename. The appended timestamp has the following form:

```
<YYYY>-<mm>-<DD>T<HH>-<MM>-<SS>
```

Then logRotate (page 339) creates a new log file with the same name as originally specified by the systemLog.path setting to mongod (page 583) or mongos (page 601).

Note: New in version 2.0.3: The logRotate (page 339) command is available to mongod (page 583) instances running on Windows systems with MongoDB release 2.0.3 and higher.

---

reIndex

reIndex

The reIndex (page 340) command drops all indexes on a collection and recreates them. This operation may be expensive for collections that have a large amount of data and/or a large number of indexes. Use the following syntax:

```
{ reIndex: "collection" }
```

Normally, MongoDB compacts indexes during routine updates. For most users, the reIndex (page 340) command is unnecessary. However, it may be worth running if the collection size has changed significantly or if the indexes are consuming a disproportionate amount of disk space.

Call reIndex (page 340) using the following form:

```
db.collection.reIndex();
```

Note: For replica sets, reIndex (page 340) will not propagate from the *primary* to *secondaries*. reIndex (page 340) will only affect a single mongod (page 583) instance.

Important: reIndex (page 340) will rebuild indexes in the *background if the index was originally specified with this option*. However, reIndex (page 340) will rebuild the _id index in the foreground, which takes the database's write lock.

See

http://docs.mongodb.org/manual/core/index-creation for more information on the behavior of indexing operations in MongoDB.

---

renameCollection

Definition

renameCollection

Changes the name of an existing collection. Specify collections to renameCollection (page 340) in the form of a complete *namespace*, which includes the database name. Issue the renameCollection (page 340) command against the *admin database*. The command takes the following form:

```
{ renameCollection: "<source_namespace>", to: "<target_namespace>", dropTarget: <true|false> }
```

The command contains the following fields:

field string renameCollection The *namespace* of the collection to rename. The namespace is a combination of the database name and the name of the collection.

field string to The new namespace of the collection. If the new namespace specifies a different database, the `renameCollection` (page 340) command copies the collection to the new database and drops the source collection.

field boolean dropTarget If `true`, `mongod` (page 583) will drop the `target` of `renameCollection` (page 340) prior to renaming the collection.

`renameCollection` (page 340) is suitable for production environments; *however*:

- `renameCollection` (page 340) blocks all database activity for the duration of the operation.
- `renameCollection` (page 340) is **not** compatible with sharded collections.

> **Warning:** `renameCollection` (page 340) fails if `target` is the name of an existing collection *and* you do not specify `dropTarget: true`.
> If the `renameCollection` (page 340) operation does not complete the `target` collection and indexes will not be usable and will require manual intervention to clean up.

Exceptions

exception 10026 Raised if the `source` namespace does not exist.

exception 10027 Raised if the `target` namespace exists and `dropTarget` is either `false` or unspecified.

exception 15967 Raised if the `target` namespace is an invalid collection name.

Shell Helper The shell helper `db.collection.renameCollection()` (page 69) provides a simpler interface to using this command within a database. The following is equivalent to the previous example:

```
db.source-namespace.renameCollection( "target" )
```

> **Warning:** You cannot use `renameCollection` (page 340) with sharded collections.

> **Warning:** This command obtains a global write lock and will block other operations until it has completed.

repairCursor

New in version 2.8.0.

`repairCursor`

Returns a cursor that iterates through all documents in a collection, omitting those that are not valid BSON. Used by `mongodump` (page 622) to provide the underlying functionality for the *`--repair`* option.

For internal use.

repairDatabase

Definition

`repairDatabase`

Checks and repairs errors and inconsistencies in data storage. `repairDatabase` (page 341) is analogous to a `fsck` command for file systems. Run the `repairDatabase` (page 341) command to ensure data integrity after the system experiences an unexpected system restart or crash, if:

1. The mongod (page 583) instance is not running with *journaling* enabled.

   When using *journaling*, there is almost never any need to run repairDatabase (page 341). In the event of an unclean shutdown, the server will be able to restore the data files to a pristine state automatically.

2. There are *no* other intact *replica set* members with a complete data set.

> **Warning:** During normal operations, only use the repairDatabase (page 341) command and wrappers including db.repairDatabase() (page 123) in the mongo (page 610) shell and *mongod --repair*, to compact database files and/or reclaim disk space. Be aware that these operations remove and do not save any corrupt data during the repair process.
> If you are trying to repair a *replica set* member, and you have access to an intact copy of your data (e.g. a recent backup or an intact member of the *replica set*), you should restore from that intact copy, and **not** use repairDatabase (page 341).

repairDatabase (page 341) takes the following form:

```
{ repairDatabase: 1 }
```

repairDatabase (page 341) has the following fields:

field boolean preserveClonedFilesOnFailure When true, repairDatabase will not delete temporary files in the backup directory on error, and all new files are created with the "backup" instead of "_tmp" directory prefix. By default repairDatabase does not delete temporary files, and uses the "_tmp" naming prefix for new files.

field boolean backupOriginalFiles When true, repairDatabase moves old database files to the backup directory instead of deleting them before moving new files into place. New files are created with the "backup" instead of "_tmp" directory prefix. By default, repairDatabase leaves temporary files unchanged, and uses the "_tmp" naming prefix for new files.

You can explicitly set the options as follows:

```
{ repairDatabase: 1,
  preserveClonedFilesOnFailure: <boolean>,
  backupOriginalFiles: <boolean> }
```

> **Warning:** This command obtains a global write lock and will block other operations until it has completed.

Note: repairDatabase (page 341) requires free disk space equal to the size of your current data set plus 2 gigabytes. If the volume that holds dbpath lacks sufficient space, you can mount a separate volume and use that for the repair. When mounting a separate volume for repairDatabase (page 341) you must run repairDatabase (page 341) from the command line and use the *--repairpath* switch to specify the folder in which to store temporary repair files.

See *mongod --repair* and *mongodump --repair* for information on these related options.

Behavior Changed in version 2.6: The repairDatabase (page 341) command is now available for secondary as well as primary members of replica sets.

The repairDatabase (page 341) command compacts all collections in the database. It is identical to running the compact (page 322) command on each collection individually.

repairDatabase (page 341) reduces the total size of the data files on disk. It also recreates all indexes in the database.

The time requirement for repairDatabase (page 341) depends on the size of the data set.

You may invoke repairDatabase (page 341) from multiple contexts:

- Use the mongo (page 610) shell to run the command, as above.
- Use the db.repairDatabase() (page 123) in the mongo (page 610) shell.
- Run mongod (page 583) directly from your system's shell. Make sure that mongod (page 583) isn't already running, and that you invoke mongod (page 583) as a user that has access to MongoDB's data files. Run as:

  ```
  mongod --repair
  ```

  To add a repair path:

  ```
  mongod --repair --repairpath /opt/vol2/data
  ```

  See repairPath for more information.

  Note: *mongod --repair* will fail if your database is not a master or primary. In most cases, you should recover a corrupt secondary using the data from an existing intact node. To run repair on a secondary/slave restart the instance in standalone mode without the *--replSet* or *--slave* options.

Example

```
{ repairDatabase: 1 }
```

Using repairDatabase to Reclaim Disk Space You should not use repairDatabase (page 341) for data recovery unless you have no other option.

However, if you trust that there is no corruption and you have enough free space, then repairDatabase (page 341) is the appropriate and the only way to reclaim disk space.

setParameter

setParameter

setParameter (page 343) is an administrative command for modifying options normally set on the command line. You must issue the setParameter (page 343) command against the *admin database* in the form:

```
{ setParameter: 1, <option>: <value> }
```

Replace the <option> with one of the supported setParameter (page 343) options:

- journalCommitInterval
- logLevel
- logUserIds
- notablescan
- quiet
- replApplyBatchSize
- replIndexPrefetch
- syncdelay
- traceExceptions
- textSearchEnabled
- sslMode

- clusterAuthMode
- userCacheInvalidationIntervalSecs
- wiredTigerEngineRuntimeConfigSetting
- logComponentVerbosity

shutdown

shutdown

The shutdown (page 344) command cleans up all database resources and then terminates the process. You must issue the shutdown (page 344) command against the *admin database* in the form:

```
{ shutdown: 1 }
```

Note: Run the shutdown (page 344) against the *admin database*. When using shutdown (page 344), the connection must originate from localhost **or** use an authenticated connection.

If the node you're trying to shut down is a replica set primary, then the command will succeed only if there exists a secondary node whose oplog data is within 10 seconds of the primary. You can override this protection using the force option:

```
{ shutdown: 1, force: true }
```

Alternatively, the shutdown (page 344) command also supports a timeoutSecs argument which allows you to specify a number of seconds to wait for other members of the replica set to catch up:

```
{ shutdown: 1, timeoutSecs: 60 }
```

The equivalent mongo (page 610) shell helper syntax looks like this:

```
db.shutdownServer({timeoutSecs: 60});
```

touch

touch

New in version 2.2.

The touch (page 344) command loads data from the data storage layer into memory. touch (page 344) can load the data (i.e. documents) indexes or both documents and indexes. Use this command to ensure that a collection, and/or its indexes, are in memory before another operation. By loading the collection or indexes into memory, mongod (page 583) will ideally be able to perform subsequent operations more efficiently. The touch (page 344) command has the following prototypical form:

```
{ touch: [collection], data: [boolean], index: [boolean] }
```

By default, data and index are false, and touch (page 344) will perform no operation. For example, to load both the data and the index for a collection named records, you would use the following command in the mongo (page 610) shell:

```
db.runCommand({ touch: "records", data: true, index: true })
```

touch (page 344) will not block read and write operations on a mongod (page 583), and can run on *secondary* members of replica sets.

Considerations

Performance Impact Using `touch` (page 344) to control or tweak what a `mongod` (page 583) stores in memory may displace other records data in memory and hinder performance. Use with caution in production systems.

Replication and Secondaries If you run `touch` (page 344) on a secondary, the secondary will enter a `RECOVERING` state to prevent clients from sending read operations during the `touch` (page 344) operation. When `touch` (page 344) finishes the secondary will automatically return to `SECONDARY` state. See `state` (page 298) for more information on replica set member states.

Storage Engines Changed in version 2.8.0.

If the current storage engine does *not* support `touch` (page 344), the `touch` (page 344) command will return an error.

The `mmapv1` storage engine supports `touch` (page 344).

The `wiredTiger` storage engine does *not* support `touch` (page 344).

Diagnostic Commands

Diagnostic Commands

Name	Description
availableQueryOptions (page 346)	Internal command that reports on the capabilities of the current MongoDB instance.
buildInfo (page 346)	Displays statistics about the MongoDB build.
collStats (page 348)	Reports storage utilization statics for a specified collection.
connPoolStats (page 350)	Reports statistics on the outgoing connections from this MongoDB instance to other MongoDB instances in the deployment.
cursorInfo (page 352)	Deprecated. Reports statistics on active cursors.
dataSize (page 352)	Returns the data size for a range of data. For internal use.
dbHash (page 352)	Internal command to support sharding.
dbStats (page 352)	Reports storage utilization statistics for the specified database.
diagLogging (page 354)	Provides a diagnostic logging. For internal use.
driverOIDTest (page 354)	Internal command that converts an ObjectId to a string to support tests.
explain (page 354)	Returns information on the execution of various operations.
features (page 356)	Reports on features available in the current MongoDB instance.
getCmdLineOpts (page 357)	Returns a document with the run-time arguments to the MongoDB instance and their parsed options.
getLog (page 357)	Returns recent log messages.
hostInfo (page 358)	Returns data that reflects the underlying host system.
indexStats (page 360)	Experimental command that collects and aggregates statistics on all indexes.
isSelf	Internal command to support testing.
listCommands (page 365)	Lists all database commands provided by the current mongod (page 583) instance.
listDatabases (page 365)	Returns a document that lists all databases and returns basic database statistics.
netstat (page 365)	Internal command that reports on intra-deployment connectivity. Only available for mongos (page 601) instances.
ping (page 365)	Internal command that tests intra-deployment connectivity.
profile (page 366)	Interface for the *database profiler*.
serverStatus (page 366)	Returns a collection metrics on instance-wide resource utilization and status.
shardConnPoolStats (page 386)	Reports statistics on a mongos (page 601)'s connection pool for client operations against shards.
top (page 387)	Returns raw usage statistics for each database in the mongod (page 583) instance.
validate (page 389)	Internal command that scans for a collection's data and indexes for correctness.
whatsmyuri (page 392)	Internal command that returns information on the current client.

availableQueryOptions

availableQueryOptions

availableQueryOptions (page 346) is an internal command that is only available on mongos (page 601) instances.

buildInfo

buildInfo

The buildInfo (page 346) command is an administrative command which returns a build summary for the current mongod (page 583). buildInfo (page 346) has the following prototype form:

```
{ buildInfo: 1 }
```

In the mongo (page 610) shell, call buildInfo (page 346) in the following form:

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

Example

The output document of buildInfo (page 346) has the following form:

```
{
  "version" : "<string>",
  "gitVersion" : "<string>",
  "sysInfo" : "<string>",
  "loaderFlags" : "<string>",
  "compilerFlags" : "<string>",
  "allocator" : "<string>",
  "versionArray" : [ <num>, <num>, <...> ],
  "javascriptEngine" : "<string>",
  "bits" : <num>,
  "debug" : <boolean>,
  "maxBsonObjectSize" : <num>,
  "ok" : <num>
}
```

Consider the following documentation of the output of buildInfo (page 346):

buildInfo
The document returned by the buildInfo (page 346) command.

buildInfo.**gitVersion**
The commit identifier that identifies the state of the code used to build the mongod (page 583).

buildInfo.**sysInfo**
A string that holds information about the operating system, hostname, kernel, date, and Boost version used to compile the mongod (page 583).

buildInfo.**loaderFlags**
The flags passed to the loader that loads the mongod (page 583).

buildInfo.**compilerFlags**
The flags passed to the compiler that builds the mongod (page 583) binary.

buildInfo.**allocator**
Changed in version 2.2.

The memory allocator that mongod (page 583) uses. By default this is tcmalloc after version 2.2, and system before 2.2.

buildInfo.**versionArray**
An array that conveys version information about the mongod (page 583) instance. See version for a more readable version of this string.

buildInfo.**javascriptEngine**
Changed in version 2.4.

A string that reports the JavaScript engine used in the mongod (page 583) instance. By default, this is V8 after version 2.4, and SpiderMonkey before 2.4.

buildInfo.**bits**
A number that reflects the target processor architecture of the mongod (page 583) binary.

buildInfo.**debug**
A boolean. true when built with debugging options.

buildInfo.**maxBsonObjectSize**
A number that reports the Maximum BSON Document Size (page 692).

collStats

Definition

collStats

The collStats (page 348) command returns a variety of storage statistics for a given collection. Use the following syntax:

```
{ collStats: "collection" , scale : 1024, verbose: true }
```

Specify the collection you want statistics for, and use the scale argument to scale the output: a value of 1024 renders the results in kilobytes. The verbose: true option increases reporting for the mmapv1 storage engine.

Examine the following example output, which uses the db.collection.stats() (page 71) helper in the mongo (page 610) shell.

```
> db.users.stats()
{
        "ns" : "app.users",              // namespace
        "count" : 9,                     // number of documents
        "size" : 432,                    // collection size in bytes
        "avgObjSize" : 48,               // average object size in bytes
        "storageSize" : 3840,            // (pre)allocated space for the collection in bytes
        "nindexes" : 2,                  // number of indexes
        "totalIndexSize" : 16384,        // total index size in bytes
        "indexSizes" : {                 // size of specific indexes in bytes
                "_id_" : 8192,
                "username" : 8192
        },
        // ...
        "ok" : 1
}
```

Note: The scale factor rounds values to whole numbers. This can produce unexpected results in some situations.

If collStats (page 348) operates on a *capped collection*, then the following fields will also be present:

```
> db.users.stats()
{
        // ...
        "capped" : true,
        "max" : NumberLong("9223372036854775807"),
        "ok" : 1
}
```

Output

collStats.**ns**
The namespace of the current collection, which follows the format [database].[collection].

collStats.**count**
The number of objects or documents in this collection.

collStats.**size**
The total size of all records in a collection. This value does not include the record header, which is 16 bytes per record, but *does* include the record's *padding*. Additionally `size` (page 349) does not include the size of any indexes associated with the collection, which the `totalIndexSize` (page 349) field reports.

The `scale` argument affects this value.

collStats.**avgObjSize**
The average size of an object in the collection. The `scale` argument affects this value.

collStats.**storageSize**
The total amount of storage allocated to this collection for *document* storage. The `scale` argument affects this value.

For `mmapv1`, `storageSize` (page 349) will not decrease as you remove or shrink documents.

collStats.**numExtents**
The total number of contiguously allocated data file regions. Only present when using the `mmapv1` storage engine.

collStats.**nindexes**
The number of indexes on the collection. All collections have at least one index on the *_id* field.

Changed in version 2.2: Before 2.2, capped collections did not necessarily have an index on the `_id` field, and some capped collections created with pre-2.2 versions of `mongod` (page 583) may not have an `_id` index.

collStats.**lastExtentSize**
The size of the last extent allocated. The `scale` argument affects this value. Only present when using the `mmapv1` storage engine.

collStats.**paddingFactor**
Deprecated since version 2.8.0: `paddingFactor` (page 349) is no longer used in 2.8.0, and remains hard coded to 1.0 for compatibility only.

Changed in version 2.8.0: `paddingFactor` (page 349) only appears when using the `mmapv1` storage engine.

The amount of additional space added to the end of each document at insert time. Multiply the `paddingFactor` (page 349) by the size of the document to determine record size.

collStats.**userFlags**
Changed in version 2.8.0: `userFlags` (page 349) only appears when using the `mmapv1` storage engine.

New in version 2.2.

Reports the flags on this collection set by the user. In version 2.2 the only user flag is `usePowerOf2Sizes` (page 322). If `usePowerOf2Sizes` (page 322) is enabled, `userFlags` (page 349) will be set to `1`, otherwise it will be `0`.

See the `collMod` (page 321) command for more information on setting user flags and `usePowerOf2Sizes` (page 322).

collStats.**totalIndexSize**
The total size of all indexes. The `scale` argument affects this value.

collStats.**indexSizes**
This field specifies the key and size of every existing index on the collection. The `scale` argument affects this value.

collStats.**capped**
This field will be "true" if the collection is *capped*.

collStats.**max**
Shows the maximum number of documents that may be present in a *capped collection*.

collStats.**maxSize**
Shows the maximum size of a *capped collection*.

collStats.**wiredtiger**
New in version 2.8.0.

wiredtiger (page 350) only appears when using the wiredTiger storage engine and contains data reported directly by the WiredTiger engine.

connPoolStats

Definition

connPoolStats
The command connPoolStats (page 350) returns information regarding the number of open connections to the current database instance, including client connections and server-to-server connections for replication and clustering.

Note: connPoolStats (page 350) only returns meaningful results for mongos (page 601) instances and for mongod (page 583) instances in sharded clusters.

The command takes the following form:

```
{ connPoolStats: 1 }
```

The value of the argument (i.e. 1) does not affect the output of the command.

Output

connPoolStats.**hosts**
The sub-documents of the hosts (page 350) *document* report connections between the mongos (page 601) or mongod (page 583) instance and each component mongod (page 583) of the *sharded cluster*.

connPoolStats.hosts.[host].**available**
available (page 350) reports the total number of connections that the mongos (page 601) or mongod (page 583) could use to connect to this mongod (page 583).

connPoolStats.hosts.[host].**created**
created (page 350) reports the number of connections that this mongos (page 601) or mongod (page 583) has ever created for this host.

connPoolStats.**replicaSets**
replicaSets (page 350) is a *document* that contains *replica set* information for the *sharded cluster*.

connPoolStats.replicaSets.**shard**
The shard (page 350) *document* reports on each *shard* within the *sharded cluster*

connPoolStats.replicaSets.[shard].**host**
The host (page 350) field holds an array of *document* that reports on each host within the *shard* in the *replica set*.

These values derive from the *replica set status* (page 296) values.

connPoolStats.replicaSets.[shard].host[n].**addr**
addr (page 350) reports the address for the host in the *sharded cluster* in the format of "[hostname]:[port]".

connPoolStats.replicaSets.[shard].host[n].**ok**
ok (page 350) reports false when:
- the mongos (page 601) or mongod (page 583) cannot connect to instance.

•the mongos (page 601) or mongod (page 583) received a connection exception or error. This field is for internal use.

connPoolStats.replicaSets.[shard].host[n].**ismaster**
ismaster (page 351) reports true if this host (page 350) is the *primary* member of the *replica set*.

connPoolStats.replicaSets.[shard].host[n].**hidden**
hidden (page 351) reports true if this host (page 350) is a *hidden member* of the *replica set*.

connPoolStats.replicaSets.[shard].host[n].**secondary**
secondary (page 351) reports true if this host (page 350) is a *secondary* member of the *replica set*.

connPoolStats.replicaSets.[shard].host[n].**pingTimeMillis**
pingTimeMillis (page 351) reports the ping time in milliseconds from the mongos (page 601) or mongod (page 583) to this host (page 350).

connPoolStats.replicaSets.[shard].host[n].**tags**
New in version 2.2.

tags (page 351) reports the tags, if this member of the set has tags configured.

connPoolStats.replicaSets.[shard].**master**
master (page 351) reports the ordinal identifier of the host in the host (page 350) array that is the *primary* of the *replica set*.

connPoolStats.replicaSets.[shard].**nextSlave**
Deprecated since version 2.2.

nextSlave (page 351) reports the *secondary* member that the mongos (page 601) will use to service the next request for this *replica set*.

connPoolStats.**createdByType**
createdByType (page 351) *document* reports the number of each type of connection that mongos (page 601) or mongod (page 583) has created in all connection pools.

mongos (page 601) connect to mongod (page 583) instances using one of three types of connections. The following sub-document reports the total number of connections by type.

connPoolStats.createdByType.**master**
master (page 351) reports the total number of connections to the *primary* member in each *cluster*.

connPoolStats.createdByType.**set**
set (page 351) reports the total number of connections to a *replica set* member.

connPoolStats.createdByType.**sync**
sync (page 351) reports the total number of *config database* connections.

connPoolStats.**totalAvailable**
totalAvailable (page 351) reports the running total of connections from the mongos (page 601) or mongod (page 583) to all mongod (page 583) instances in the *sharded cluster* available for use.

connPoolStats.**totalCreated**
totalCreated (page 351) reports the total number of connections ever created from the mongos (page 601) or mongod (page 583) to all mongod (page 583) instances in the *sharded cluster*.

connPoolStats.**numDBClientConnection**
numDBClientConnection (page 351) reports the total number of connections from the mongos (page 601) or mongod (page 583) to all of the mongod (page 583) instances in the *sharded cluster*.

connPoolStats.**numAScopedConnection**
numAScopedConnection (page 351) reports the number of exception safe connections created from

mongos (page 601) or mongod (page 583) to all mongod (page 583) in the *sharded cluster*. The mongos (page 601) or mongod (page 583) releases these connections after receiving a socket exception from the mongod (page 583).

cursorInfo Deprecated since version 2.6: Use the serverStatus (page 366) command to return the serverStatus.metrics.cursor (page 385) information instead.

cursorInfo

The cursorInfo (page 352) command returns information about current cursor allotment and use. Use the following form:

```
{ cursorInfo: 1 }
```

The value (e.g. 1 above) does not affect the output of the command.

cursorInfo (page 352) returns the total number of open cursors (totalOpen), the size of client cursors in current use (clientCursors_size), and the number of timed out cursors since the last server restart (timedOut).

dataSize

dataSize

The dataSize (page 352) command returns the data size for a set of data within a certain range:

```
{ dataSize: "database.collection", keyPattern: { field: 1 }, min: { field: 10 }, max: { field: 1
```

This will return a document that contains the size of all matching documents. Replace database.collection value with database and collection from your deployment. The keyPattern, min, and max parameters are options.

The amount of time required to return dataSize (page 352) depends on the amount of data in the collection.

dbHash

dbHash

dbHash (page 352) is a command that supports *config servers* and is not part of the stable client facing API.

dbStats

Definition

dbStats

The dbStats (page 352) command returns storage statistics for a given database. The command takes the following syntax:

```
{ dbStats: 1, scale: 1 }
```

The values of the options above do not affect the output of the command. The scale option allows you to specify how to scale byte values. For example, a scale value of 1024 will display the results in kilobytes rather than in bytes:

```
{ dbStats: 1, scale: 1024 }
```

Note: Because scaling rounds values to whole numbers, scaling may return unlikely or unexpected results.

The time required to run the command depends on the total size of the database. Because the command must touch all data files, the command may take several seconds to run.

In the mongo (page 610) shell, the db.stats() (page 126) function provides a wrapper around dbStats (page 352).

Output

dbStats.**db**
Contains the name of the database.

dbStats.**collections**
Contains a count of the number of collections in that database.

dbStats.**objects**
Contains a count of the number of objects (i.e. *documents*) in the database across all collections.

dbStats.**avgObjSize**
The average size of each document in bytes. This is the dataSize (page 353) divided by the number of documents.

dbStats.**dataSize**
The total size in bytes of the data held in this database including the *padding factor*. The scale argument affects this value. The dataSize (page 353) will not decrease when *documents* shrink, but will decrease when you remove documents.

dbStats.**storageSize**
The total amount of space in bytes allocated to collections in this database for *document* storage. The scale argument affects this value. The storageSize (page 353) does not decrease as you remove or shrink documents.

dbStats.**numExtents**
Contains a count of the number of extents in the database across all collections.

dbStats.**indexes**
Contains a count of the total number of indexes across all collections in the database.

dbStats.**indexSize**
The total size in bytes of all indexes created on this database. The scale arguments affects this value.

dbStats.**fileSize**
The total size in bytes of the data files that hold the database. This value includes preallocated space and the *padding factor*. The value of fileSize (page 353) only reflects the size of the data files for the database and not the namespace file.

The scale argument affects this value. Only present when using the mmapv1 storage engine.

dbStats.**nsSizeMB**
The total size of the *namespace* files (i.e. that end with .ns) for this database. You cannot change the size of the namespace file after creating a database, but you can change the default size for all new namespace files with the nsSize runtime option.

Only present when using the mmapv1 storage engine.

See also:

The nsSize option, and *Maximum Namespace File Size* (page 692)

dbStats.**dataFileVersion**
New in version 2.4.

Document that contains information about the on-disk format of the data files for the database. Only present when using the mmapv1 storage engine.

dbStats.dataFileVersion.**major**
New in version 2.4.

The major version number for the on-disk format of the data files for the database. Only present when using the `mmapv1` storage engine.

`dbStats.dataFileVersion.`**`minor`**
New in version 2.4.

The minor version number for the on-disk format of the data files for the database. Only present when using the `mmapv1` storage engine.

`dbStats.`**`extentFreeList`**
New in version 2.8.0.

`dbStats.extentFreeList.`**`num`**
New in version 2.8.0.

Number of extents in the freelist. Only present when using the `mmapv1` storage engine.

`dbStats.extentFreeList.`**`size`**
New in version 2.8.0.

Total size of the extents on the freelist.

The `scale` argument affects this value. Only present when using the `mmapv1` storage engine.

diagLogging Deprecated since version 2.8.0.

`diagLogging`
`diagLogging` (page 354) is a command that captures additional data for diagnostic purposes and is not part of the stable client facing API.

`diaglogging` obtains a write lock on the affected database and will block other operations until it completes.

driverOIDTest
`driverOIDTest`
`driverOIDTest` (page 354) is an internal command.

explain

Definition
`explain`
New in version 2.8.

The `explain` (page 354) command provides information on the execution of the following commands: `count` (page 213), `group` (page 216), `delete` (page 234), and `update` (page 251).

Although MongoDB provides the `explain` (page 354) command, the preferred method for running `explain` (page 354) is to use the `db.collection.explain()` (page 33) helper.

Note: The shell helpers(), `db.collection.explain()` (page 33) and `cursor.explain()` (page 85), are the only available mechanisms for explaining `db.collection.find()` (page 36) operations.

The `explain` (page 354) command has the following syntax:

```
{
   explain: <command>,
   verbosity: <string>
}
```

The command takes the following fields:

field document explain A document specifying the command for which to return the execution information. For details on the specific command document, see `count` (page 213), `group` (page 216), `delete` (page 234), and `update` (page 251).

For `find()` operations, see `db.collection.explain()` (page 33).

field string verbosity A string specifying the mode in which to run `explain` (page 354). The mode affects the behavior of `explain` (page 354) and determines the amount of information to return.

Possible modes are: *"queryPlanner"* (page 355), *"executionStats"* (page 355), and *"allPlansExecution"* (page 355). For more information on the modes, see *explain behavior* (page 355).

By default, `explain` (page 354) runs in *"allPlansExecution"* (page 355) mode.

Behavior The behavior of `explain` (page 354) and the amount of information returned depend on the `verbosity` mode.

`queryPlanner` Mode MongoDB runs the `query optimizer` to choose the winning plan for the operation under evaluation. `explain` (page 354) returns the `queryPlanner` information for the evaluated `<command>`.

`executionStats` Mode MongoDB runs the `query optimizer` to choose the winning plan, executes the winning plan to completion, and returns statistics describing the execution of the winning plan.

For write operations, `explain` (page 354) returns information about the update or delete operations that *would* be performed, but does *not* apply the modifications to the database.

`explain` (page 354) returns the `queryPlanner` and `executionStats` information for the evaluated `<command>`. However, `executionStats` does not provide query execution information for the rejected plans.

`allPlansExecution` Mode By default, `explain` (page 354) runs in `"allPlansExecution"` verbosity mode.

MongoDB runs the `query optimizer` to choose the winning plan and executes the winning plan to completion. In `"allPlansExecution"` mode, MongoDB returns statistics describing the execution of the winning plan as well as statistics for the other candidate plans captured during *plan selection*.

For write operations, `explain` (page 354) returns information about the update or delete operations that *would* be performed, but does *not* apply the modifications to the database.

`explain` (page 354) returns the `queryPlanner` and `executionStats` information for the evaluated `<command>`. The `executionStats` includes the *completed* query execution information for the *winning plan*.

If the query optimizer considered more than one plan, `executionStats` information also includes the *partial* execution information captured during the *plan selection phase* for both the winning and rejected candidate plans.

Examples

`queryPlanner` Mode The following `explain` (page 354) command runs in *"queryPlanner"* (page 355) verbosity mode to return the query planning information for a `count` (page 213) command:

```
db.runCommand(
   {
     explain: { count: "products", query: { quantity: { $gt: 50 } } },
     verbosity: "queryPlanner"
   }
)
```

`executionStats` Mode The following `explain` (page 354) operation runs in *"executionStats"* (page 355) verbosity mode to return the query planning and execution information for a `count` (page 213) command:

```
db.runCommand(
   {
      explain: { count: "products", query: { quantity: { $gt: 50 } } },
      verbosity: "executionStats"
   }
)
```

`allPlansExecution` Mode By default, `explain` (page 354) runs in *"allPlansExecution"* (page 355) verbosity mode. The following `explain` (page 354) command returns the `queryPlanner` and `executionStats` for all considered plans for an `update` (page 251) command:

Note: The execution of this explain will *not* modify data but runs the query predicate of the update operation. For candidate plans, MongoDB returns the execution information captured during the *plan selection phase*.

```
db.runCommand(
   {
     explain: {
        update: "products",
        updates: [
           {
               q: { quantity: 1057, category: "apparel" },
               u: { $set: { reorder: true } }
           }
        ]
     }
   }
)
```

Output `explain` (page 354) operations can return information regarding:

- *queryPlanner*, which details the plan selected by the `query optimizer` and lists the rejected plans;
- *executionStats*, which details the execution of the winning plan and the rejected plans; and
- *serverInfo*, which provides information on the MongoDB instance.

The verbosity mode (i.e. `queryPlanner`, `executionStats`, `allPlansExecution`) determines whether the results include *executionStats* and whether *executionStats* includes data captured during *plan selection*.

For details on the output, see `http://docs.mongodb.org/manual/reference/explain-results`.

features

`features`

`features` (page 356) is an internal command that returns the build-level feature settings.

getCmdLineOpts

getCmdLineOpts

The getCmdLineOpts (page 357) command returns a document containing command line options used to start the given mongod (page 583) or mongos (page 601):

```
{ getCmdLineOpts: 1 }
```

This command returns a document with two fields, argv and parsed. The argv field contains an array with each item from the command string used to invoke mongod (page 583) or mongos (page 601). The document in the parsed field includes all runtime options, including those parsed from the command line and those specified in the configuration file, if specified.

Consider the following example output of getCmdLineOpts (page 357):

```
{
        "argv" : [
                "/usr/bin/mongod",
                "--config",
                "/etc/mongodb.conf",
                "--fork"
        ],
        "parsed" : {
                "bind_ip" : "127.0.0.1",
                "config" : "/etc/mongodb/mongodb.conf",
                "dbpath" : "/srv/mongodb",
                "fork" : true,
                "logappend" : "true",
                "logpath" : "/var/log/mongodb/mongod.log",
                "quiet" : "true"
        },
        "ok" : 1
}
```

getLog

getLog

The getLog (page 357) command returns a document with a log array that contains recent messages from the mongod (page 583) process log. The getLog (page 357) command has the following syntax:

```
{ getLog: <log> }
```

Replace <log> with one of the following values:

- global - returns the combined output of all recent log entries.
- rs - if the mongod (page 583) is part of a *replica set*, getLog (page 357) will return recent notices related to replica set activity.
- startupWarnings - will return logs that *may* contain errors or warnings from MongoDB's log from when the current process started. If mongod (page 583) started without warnings, this filter may return an empty array.

You may also specify an asterisk (e.g. *) as the <log> value to return a list of available log filters. The following interaction from the mongo (page 610) shell connected to a replica set:

```
db.adminCommand({getLog: "*" })
{ "names" : [ "global", "rs", "startupWarnings" ], "ok" : 1 }
```

getLog (page 357) returns events from a RAM cache of the mongod (page 583) events and *does not* read log data from the log file.

hostInfo

hostInfo

New in version 2.2.

Returns A document with information about the underlying system that the mongod (page 583) or mongos (page 601) runs on. Some of the returned fields are only included on some platforms.

You must run the hostInfo (page 358) command, which takes no arguments, against the admin database. Consider the following invocations of hostInfo (page 358):

```
db.hostInfo()
db.adminCommand( { "hostInfo" : 1 } )
```

In the mongo (page 610) shell you can use db.hostInfo() (page 119) as a helper to access hostInfo (page 358). The output of hostInfo (page 358) on a Linux system will resemble the following:

```
{
   "system" : {
          "currentTime" : ISODate("<timestamp>"),
          "hostname" : "<hostname>",
          "cpuAddrSize" : <number>,
          "memSizeMB" : <number>,
          "numCores" : <number>,
          "cpuArch" : "<identifier>",
          "numaEnabled" : <boolean>
   },
   "os" : {
          "type" : "<string>",
          "name" : "<string>",
          "version" : "<string>"
   },
   "extra" : {
          "versionString" : "<string>",
          "libcVersion" : "<string>",
          "kernelVersion" : "<string>",
          "cpuFrequencyMHz" : "<string>",
          "cpuFeatures" : "<string>",
          "pageSize" : <number>,
          "numPages" : <number>,
          "maxOpenFiles" : <number>
   },
   "ok" : <return>
}
```

Consider the following documentation of these fields:

hostInfo

The document returned by the hostInfo (page 358).

hostInfo.**system**

A sub-document about the underlying environment of the system running the mongod (page 583) or mongos (page 601)

hostInfo.system.**currentTime**

A time stamp of the current system time.

hostInfo.system.**hostname**

The system name, which should correspond to the output of hostname -f on Linux systems.

hostInfo.system.**cpuAddrSize**

A number reflecting the architecture of the system. Either 32 or 64.

hostInfo.system.**memSizeMB**
The total amount of system memory (RAM) in megabytes.

hostInfo.system.**numCores**
The total number of available logical processor cores.

hostInfo.system.**cpuArch**
A string that represents the system architecture. Either `x86` or `x86_64`.

hostInfo.system.**numaEnabled**
A boolean value. `false` if NUMA is interleaved (i.e. disabled), otherwise `true`.

hostInfo.**os**
A sub-document that contains information about the operating system running the `mongod` (page 583) and `mongos` (page 601).

hostInfo.os.**type**
A string representing the type of operating system, such as `Linux` or `Windows`.

hostInfo.os.**name**
If available, returns a display name for the operating system.

hostInfo.os.**version**
If available, returns the name of the distribution or operating system.

hostInfo.**extra**
A sub-document with extra information about the operating system and the underlying hardware. The content of the `extra` (page 359) sub-document depends on the operating system.

hostInfo.extra.**versionString**
A complete string of the operating system version and identification. On Linux and OS X systems, this contains output similar to `uname -a`.

hostInfo.extra.**libcVersion**
The release of the system `libc`.

`libcVersion` (page 359) only appears on Linux systems.

hostInfo.extra.**kernelVersion**
The release of the Linux kernel in current use.

`kernelVersion` (page 359) only appears on Linux systems.

hostInfo.extra.**alwaysFullSync**
`alwaysFullSync` (page 359) only appears on OS X systems.

hostInfo.extra.**nfsAsync**
`nfsAsync` (page 359) only appears on OS X systems.

hostInfo.extra.**cpuFrequencyMHz**
Reports the clock speed of the system's processor in megahertz.

hostInfo.extra.**cpuFeatures**
Reports the processor feature flags. On Linux systems this the same information that `/proc/cpuinfo` includes in the `flags` fields.

hostInfo.extra.**pageSize**
Reports the default system page size in bytes.

hostInfo.extra.**numPages**
`numPages` (page 359) only appears on Linux systems.

`hostInfo.extra.`**`maxOpenFiles`**
Reports the current system limits on open file handles. See `http://docs.mongodb.org/manual/reference/ulimit` for more information.

`maxOpenFiles` (page 359) only appears on Linux systems.

`hostInfo.extra.`**`scheduler`**
Reports the active I/O scheduler. `scheduler` (page 360) only appears on OS X systems.

indexStats

Definition

`indexStats`

The `indexStats` (page 360) command aggregates statistics for the B-tree data structure that stores data for a MongoDB index.

> **Warning:** This command is not intended for production deployments.

The command can be run *only* on a `mongod` (page 583) instance that uses the `--enableExperimentalIndexStatsCmd` option.

To aggregate statistics, issue the command like so:

```
db.runCommand( { indexStats: "<collection>", index: "<index name>" } )
```

Output The `db.collection.indexStats()` (page 54) method and equivalent `indexStats` (page 360) command aggregate statistics for the B-tree data structure that stores data for a MongoDB index. The commands aggregate statistics firstly for the entire B-tree and secondly for each individual level of the B-tree. The output displays the following values.

`indexStats.`**`index`**
The *index name*.

`indexStats.`**`version`**
The index version. For more information on index version numbers, see the `v` option in `db.collection.ensureIndex()` (page 30).

`indexStats.`**`isIdIndex`**
If `true`, the index is the default `_id` index for the collection.

`indexStats.`**`keyPattern`**
The indexed keys.

`indexStats.`**`storageNs`**
The namespace of the index's underlying storage.

`indexStats.`**`bucketBodyBytes`**
The fixed size, in bytes, of a B-tree bucket in the index, not including the record header. All indexes for a given version have the same value for this field. MongoDB allocates fixed size buckets on disk.

`indexStats.`**`depth`**
The number of levels in the B-tree, not including the root level.

`indexStats.`**`overall`**
This section of the output displays statistics for the entire B-tree.

`indexStats.overall.`**`numBuckets`**
The number of buckets in the entire B-tree, including all levels.

indexStats.overall.**keyCount**
: Statistics about the number of keys in a bucket, evaluated on a per-bucket level.

indexStats.overall.**usedKeyCount**
: Statistics about the number of used keys in a bucket, evaluated on a per-bucket level. Used keys are keys not marked as deleted.

indexStats.overall.**bsonRatio**
: Statistics about the percentage of the bucket body that is occupied by the key objects themselves, excluding associated metadata.

  For example, if you have the document `{ name: "Bob Smith" }` and an index on `{ name: 1 }`, the key object is the string `Bob Smith`.

indexStats.overall.**keyNodeRatio**
: Statistics about the percentage of the bucket body that is occupied by the key node objects (the metadata and links pertaining to the keys). This does not include the key itself. In the current implementation, a key node's objects consist of: the pointer to the key data (in the same bucket), the pointer to the record the key is for, and the pointer to a child bucket.

indexStats.overall.**fillRatio**
: The sum of the `bsonRatio` (page 361) and the `keyNodeRatio` (page 361). This shows how full the buckets are. This will be much higher for indexes with sequential inserts.

indexStats.**perLevel**
: This section of the output displays statistics for each level of the B-tree separately, starting with the root level. This section displays a different document for each B-tree level.

indexStats.perLevel.**numBuckets**
: The number of buckets at this level of the B-tree.

indexStats.perLevel.**keyCount**
: Statistics about the number of keys in a bucket, evaluated on a per-bucket level.

indexStats.perLevel.**usedKeyCount**
: Statistics about the number of used keys in a bucket, evaluated on a per-bucket level. Used keys are keys not marked as deleted.

indexStats.perLevel.**bsonRatio**
: Statistics about the percentage of the bucket body that is occupied by the key objects themselves, excluding associated metadata.

indexStats.perLevel.**keyNodeRatio**
: Statistics about the percentage of the bucket body that is occupied by the key node objects (the metadata and links pertaining to the keys).

indexStats.perLevel.**fillRatio**
: The sum of the `bsonRatio` (page 361) and the `keyNodeRatio` (page 361). This shows how full the buckets are. This will be much higher in the following cases:

  - For indexes with sequential inserts, such as the `_id` index when using ObjectId keys.
  - For indexes that were recently built in the foreground with existing data.
  - If you recently ran `compact` (page 322) or *--repair*.

Example The following is an example of `db.collection.indexStats()` (page 54) and `indexStats` (page 360) output.

```
{
    "index" : "type_1_traits_1",
    "version" : 1,
    "isIdIndex" : false,
    "keyPattern" : {
        "type" : 1,
        "traits" : 1
    },
    "storageNs" : "test.animals.$type_1_traits_1",
    "bucketBodyBytes" : 8154,
    "depth" : 2,
    "overall" : {
        "numBuckets" : 45513,
        "keyCount" : {
            "count" : NumberLong(45513),
            "mean" : 253.89602970579836,
            "stddev" : 21.784799875240708,
            "min" : 52,
            "max" : 290,
            "quantiles" : {
                "0.01" : 201.99785091648775,
                // ...
                "0.99" : 289.9999655156967
            }
        },
        "usedKeyCount" : {
            "count" : NumberLong(45513),
            // ...
            "quantiles" : {
                "0.01" : 201.99785091648775,
                // ...
                "0.99" : 289.9999655156967
            }
        },
        "bsonRatio" : {
            "count" : NumberLong(45513),
            // ...
            "quantiles" : {
                "0.01" : 0.4267797891997124,
                // ...
                "0.99" : 0.5945548174629648
            }
        },
        "keyNodeRatio" : {
            "count" : NumberLong(45513),
            // ...
            "quantiles" : {
                "0.01" : 0.3963656628236211,
                // ...
                "0.99" : 0.5690457993930765
            }
        },
        "fillRatio" : {
            "count" : NumberLong(45513),
            // ...
            "quantiles" : {
                "0.01" : 0.9909134214926929,
                // ...
```

```
                "0.99" : 0.9960755457453732
            }
        }
    },
    "perLevel" : [
        {
            "numBuckets" : 1,
            "keyCount" : {
                "count" : NumberLong(1),
                "mean" : 180,
                "stddev" : 0,
                "min" : 180,
                "max" : 180
            },
            "usedKeyCount" : {
                "count" : NumberLong(1),
                // ...
                "max" : 180
            },
            "bsonRatio" : {
                "count" : NumberLong(1),
                // ...
                "max" : 0.3619082658817758
            },
            "keyNodeRatio" : {
                "count" : NumberLong(1),
                // ...
                "max" : 0.35320088300220753
            },
            "fillRatio" : {
                "count" : NumberLong(1),
                // ...
                "max" : 0.7151091488839834
            }
        },
        {
            "numBuckets" : 180,
            "keyCount" : {
                "count" : NumberLong(180),
                "mean" : 250.84444444444443,
                "stddev" : 26.30057503009355,
                "min" : 52,
                "max" : 290
            },
            "usedKeyCount" : {
                "count" : NumberLong(180),
                // ...
                "max" : 290
            },
            "bsonRatio" : {
                "count" : NumberLong(180),
                // ...
                "max" : 0.5945548197203826
            },
            "keyNodeRatio" : {
                "count" : NumberLong(180),
                // ...
                "max" : 0.5690458670591121
```

```
        },
        "fillRatio" : {
            "count" : NumberLong(180),
            // ...
            "max" : 0.9963208241353937
        }
    },
    {
        "numBuckets" : 45332,
        "keyCount" : {
            "count" : NumberLong(45332),
            "mean" : 253.90977675813994,
            "stddev" : 21.761620836279018,
            "min" : 167,
            "max" : 290,
            "quantiles" : {
                "0.01" : 202.0000012563603,
                // ...
                "0.99" : 289.99996486571894
            }
        },
        "usedKeyCount" : {
            "count" : NumberLong(45332),
            // ...
            "quantiles" : {
                "0.01" : 202.0000012563603,
                // ...
                "0.99" : 289.99996486571894
            }
        },
        "bsonRatio" : {
            "count" : NumberLong(45332),
            // ...
            "quantiles" : {
                "0.01" : 0.42678446958950583,
                // ...
                "0.99" : 0.5945548175411283
            }
        },
        "keyNodeRatio" : {
            "count" : NumberLong(45332),
            // ...
            "quantiles" : {
                "0.01" : 0.39636988227885306,
                // ...
                "0.99" : 0.5690457981176729
            }
        },
        "fillRatio" : {
            "count" : NumberLong(45332),
            // ...
            "quantiles" : {
                "0.01" : 0.9909246995605362,
                // ...
                "0.99" : 0.996075546919481
            }
        }
    }
```

```
    ],
    "ok" : 1
}
```

Additional Resources For more information on the command's limits and output, see the following:

- The equivalent `db.collection.indexStats()` (page 54) method,
- *indexStats* (page 360), and
- https://github.com/10gen-labs/storage-viz#readme.

isSelf

_isSelf

`_isSelf` (page 365) is an internal command.

listCommands

listCommands

The `listCommands` (page 365) command generates a list of all database commands implemented for the current `mongod` (page 583) instance.

listDatabases

listDatabases

The `listDatabases` (page 365) command provides a list of existing databases along with basic statistics about them:

```
{ listDatabases: 1 }
```

The value (e.g. `1`) does not affect the output of the command. `listDatabases` (page 365) returns a document for each database. Each document contains a `name` field with the database name, a `sizeOnDisk` field with the total size of the database file on disk in bytes, and an `empty` field specifying whether the database has any data.

Example

The following operation returns a list of all databases:

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

See also:

`http://docs.mongodb.org/manual/tutorial/use-database-commands`.

netstat

netstat

`netstat` (page 365) is an internal command that is only available on `mongos` (page 601) instances.

ping

ping

The `ping` (page 365) command is a no-op used to test whether a server is responding to commands. This command will return immediately even if the server is write-locked:

```
{ ping: 1 }
```

The value (e.g. 1 above) does not impact the behavior of the command.

profile

profile

Use the profile (page 366) command to enable, disable, or change the query profiling level. This allows administrators to capture data regarding performance. The database profiling system can impact performance and can allow the server to write the contents of queries to the log. Your deployment should carefully consider the security implications of this. Consider the following prototype syntax:

```
{ profile: <level> }
```

The following profiling levels are available:

Level	Setting
-1	No change. Returns the current profile level.
0	Off. No profiling.
1	On. Only includes slow operations.
2	On. Includes all operations.

You may optionally set a threshold in milliseconds for profiling using the slowms option, as follows:

```
{ profile: 1, slowms: 200 }
```

mongod (page 583) writes the output of the database profiler to the system.profile collection.

mongod (page 583) records queries that take longer than the slowOpThresholdMs to the server log even when the database profiler is not active.

See also:

Additional documentation regarding *Database Profiling*.

See also:

"db.getProfilingStatus() (page 117)" and "db.setProfilingLevel() (page 126)" provide wrappers around this functionality in the mongo (page 610) shell.

Note: This command obtains a write lock on the affected database and will block other operations until it has completed. However, the write lock is only held while enabling or disabling the profiler. This is typically a short operation.

---

serverStatus

Definition

serverStatus

The serverStatus (page 366) command returns a document that provides an overview of the database process's state. Most monitoring applications run this command at a regular interval to collection statistics about the instance:

```
{ serverStatus: 1 }
```

The value (i.e. 1 above), does not affect the operation of the command.

Changed in version 2.4: In 2.4 you can dynamically suppress portions of the serverStatus (page 366) output, or include suppressed sections by adding fields to the command document as in the following examples:

```
db.runCommand( { serverStatus: 1, repl: 0, indexCounters: 0 } )
db.runCommand( { serverStatus: 1, workingSet: 1, metrics: 0, locks: 0 } )
```

serverStatus (page 366) includes all fields by default, except workingSet (page 381) rangeDeleter (page 377), and some content in the repl (page 374) document.

Note: You may only dynamically include top-level fields from the *serverStatus* (page 366) document that are not included by default. You can exclude any field that serverStatus (page 366) includes by default.

See also:

db.serverStatus() (page 124)

Output The serverStatus (page 366) command returns a collection of information that reflects the database's status. These data are useful for diagnosing and assessing the performance of your MongoDB instance. This reference catalogs each datum included in the output of this command and provides context for using this data to more effectively administer your database.

See also:

Much of the output of serverStatus (page 366) is also displayed dynamically by mongostat (page 657). See the *mongostat* (page 657) command for more information.

For examples of the serverStatus (page 366) output, see http://docs.mongodb.org/manual/reference/server-st

Instance Information For an example of the instance information, see the *Instance Information section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**host**
: The host (page 367) field contains the system's hostname. In Unix/Linux systems, this should be the same as the output of the hostname command.

serverStatus.**version**
: The version (page 367) field contains the version of MongoDB running on the current mongod (page 583) or mongos (page 601) instance.

serverStatus.**process**
: The process (page 367) field identifies which kind of MongoDB instance is running. Possible values are:

  - mongos (page 601)
  - mongod (page 583)

serverStatus.**uptime**
: The value of the uptime (page 367) field corresponds to the number of seconds that the mongos (page 601) or mongod (page 583) process has been active.

serverStatus.**uptimeEstimate**
: uptimeEstimate (page 367) provides the uptime as calculated from MongoDB's internal course-grained time keeping system.

serverStatus.**localTime**
: The localTime (page 367) value is the current time, according to the server, in UTC specified in an ISODate format.

locks New in version 2.1.2: All locks (page 368) statuses first appeared in the 2.1.2 development release for the 2.2 series.

For an example of the locks output, see the *locks section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**locks**
The locks (page 368) document contains sub-documents that provides a granular report on MongoDB database-level lock use. All values are of the NumberLong() type.

Generally, fields named:

- R refer to the global read lock,
- W refer to the global write lock,
- r refer to the database specific read lock, and
- w refer to the database specific write lock.

If a document does not have any fields, it means that no locks have existed with this context since the last time the mongod (page 583) started.

serverStatus.locks..
A field named . holds the first document in locks (page 368) that contains information about the global lock.

serverStatus.locks...**timeLockedMicros**
The timeLockedMicros (page 368) document reports the amount of time in microseconds that a lock has existed in all databases in this mongod (page 583) instance.

serverStatus.locks...timeLockedMicros.**R**
The R field reports the amount of time in microseconds that any database has held the global read lock.

serverStatus.locks...timeLockedMicros.**W**
The W field reports the amount of time in microseconds that any database has held the global write lock.

serverStatus.locks...timeLockedMicros.**r**
The r field reports the amount of time in microseconds that any database has held the local read lock.

serverStatus.locks...timeLockedMicros.**w**
The w field reports the amount of time in microseconds that any database has held the local write lock.

serverStatus.locks...**timeAcquiringMicros**
The timeAcquiringMicros (page 368) document reports the amount of time in microseconds that operations have spent waiting to acquire a lock in all databases in this mongod (page 583) instance.

serverStatus.locks...timeAcquiringMicros.**R**
The R field reports the amount of time in microseconds that any database has spent waiting for the global read lock.

serverStatus.locks...timeAcquiringMicros.**W**
The W field reports the amount of time in microseconds that any database has spent waiting for the global write lock.

serverStatus.locks.**admin**
The admin (page 368) document contains two sub-documents that report data regarding lock use in the *admin database*.

serverStatus.locks.admin.**timeLockedMicros**
The timeLockedMicros (page 368) document reports the amount of time in microseconds that locks have existed in the context of the *admin database*.

serverStatus.locks.admin.timeLockedMicros.**r**
The r field reports the amount of time in microseconds that the *admin database* has held the read lock.

serverStatus.locks.admin.timeLockedMicros.**w**
: The w field reports the amount of time in microseconds that the *admin database* has held the write lock.

serverStatus.locks.admin.**timeAcquiringMicros**
: The timeAcquiringMicros (page 369) document reports on the amount of field time in microseconds that operations have spent waiting to acquire a lock for the *admin database*.

serverStatus.locks.admin.timeAcquiringMicros.**r**
: The r field reports the amount of time in microseconds that operations have spent waiting to acquire a read lock on the *admin database*.

serverStatus.locks.admin.timeAcquiringMicros.**w**
: The w field reports the amount of time in microseconds that operations have spent waiting to acquire a write lock on the *admin database*.

serverStatus.locks.**local**
: The local (page 369) document contains two sub-documents that report data regarding lock use in the local database. The local database contains a number of instance specific data, including the *oplog* for replication.

serverStatus.locks.local.**timeLockedMicros**
: The timeLockedMicros (page 369) document reports on the amount of time in microseconds that locks have existed in the context of the local database.

serverStatus.locks.local.timeLockedMicros.**r**
: The r field reports the amount of time in microseconds that the local database has held the read lock.

serverStatus.locks.local.timeLockedMicros.**w**
: The w field reports the amount of time in microseconds that the local database has held the write lock.

serverStatus.locks.local.**timeAcquiringMicros**
: The timeAcquiringMicros (page 369) document reports on the amount of time in microseconds that operations have spent waiting to acquire a lock for the local database.

serverStatus.locks.local.timeAcquiringMicros.**r**
: The r field reports the amount of time in microseconds that operations have spent waiting to acquire a read lock on the local database.

serverStatus.locks.local.timeAcquiringMicros.**w**
: The w field reports the amount of time in microseconds that operations have spent waiting to acquire a write lock on the local database.

serverStatus.locks.**<database>**
: For each additional database locks (page 368) includes a document that reports on the lock use for this database. The names of these documents reflect the database name itself.

serverStatus.locks.<database>.**timeLockedMicros**
: The timeLockedMicros (page 369) document reports on the amount of time in microseconds that locks have existed in the context of the <database> database.

serverStatus.locks.<database>.timeLockedMicros.**r**
: The r field reports the amount of time in microseconds that the <database> database has held the read lock.

serverStatus.locks.<database>.timeLockedMicros.**w**
: The w field reports the amount of time in microseconds that the <database> database has held the write lock.

serverStatus.locks.<database>.**timeAcquiringMicros**
: The timeAcquiringMicros (page 369) document reports on the amount of time in microseconds that operations have spent waiting to acquire a lock for the <database> database.

serverStatus.locks.<database>.timeAcquiringMicros.**r**
: The r field reports the amount of time in microseconds that operations have spent waiting to acquire a read lock on the <database> database.

serverStatus.locks.<database>.timeAcquiringMicros.**w**
: The w field reports the amount of time in microseconds that operations have spent waiting to acquire a write lock on the <database> database.

globalLock For an example of the globalLock output, see the *globalLock section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**globalLock**
: The globalLock (page 370) data structure contains information regarding the database's current lock state, historical lock status, current operation queue, and the number of active clients.

serverStatus.globalLock.**totalTime**
: The value of totalTime (page 370) represents the time, in microseconds, since the database last started and creation of the globalLock (page 370). This is roughly equivalent to total server uptime.

serverStatus.globalLock.**lockTime**
: The value of lockTime (page 370) represents the time, in microseconds, since the database last started, that the globalLock (page 370) has been *held*.

  Consider this value in combination with the value of totalTime (page 370). MongoDB aggregates these values in the ratio (page 370) value. If the ratio (page 370) value is small but totalTime (page 370) is high the globalLock (page 370) has typically been held frequently for shorter periods of time, which may be indicative of a more normal use pattern. If the lockTime (page 370) is higher and the totalTime (page 370) is smaller (relatively) then fewer operations are responsible for a greater portion of server's use (relatively).

serverStatus.globalLock.**ratio**
: Changed in version 2.2: ratio (page 370) was removed. See locks (page 368).

  The value of ratio (page 370) displays the relationship between lockTime (page 370) and totalTime (page 370).

  Low values indicate that operations have held the globalLock (page 370) frequently for shorter periods of time. High values indicate that operations have held globalLock (page 370) infrequently for longer periods of time.

serverStatus.globalLock.**currentQueue**
: The currentQueue (page 370) data structure value provides more granular information concerning the number of operations queued because of a lock.

serverStatus.globalLock.currentQueue.**total**
: The value of total (page 370) provides a combined total of operations queued waiting for the lock.

  A consistently small queue, particularly of shorter operations should cause no concern. Also, consider this value in light of the size of queue waiting for the read lock (e.g. readers (page 370)) and write lock (e.g. writers (page 370)) individually.

serverStatus.globalLock.currentQueue.**readers**
: The value of readers (page 370) is the number of operations that are currently queued and waiting for the read lock. A consistently small read-queue, particularly of shorter operations should cause no concern.

serverStatus.globalLock.currentQueue.**writers**
: The value of writers (page 370) is the number of operations that are currently queued and waiting for the write lock. A consistently small write-queue, particularly of shorter operations is no cause for concern.

globalLock.activeClients

serverStatus.globalLock.**activeClients**
: The activeClients (page 370) data structure provides more granular information about the number of connected clients and the operation types (e.g. read or write) performed by these clients.

Use this data to provide context for the currentQueue (page 370) data.

serverStatus.globalLock.activeClients.**total**
The value of total (page 371) is the total number of active client connections to the database. This combines clients that are performing read operations (e.g. readers (page 371)) and clients that are performing write operations (e.g. writers (page 371)).

serverStatus.globalLock.activeClients.**readers**
The value of readers (page 371) contains a count of the active client connections performing read operations.

serverStatus.globalLock.activeClients.**writers**
The value of writers (page 371) contains a count of active client connections performing write operations.

mem For an example of the mem output, see the *mem section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**mem**
The mem (page 371) data structure holds information regarding the target system architecture of mongod (page 583) and current memory use.

serverStatus.mem.**bits**
The value of bits (page 371) is either 64 or 32, depending on which target architecture specified during the mongod (page 583) compilation process. In most instances this is 64, and this value does not change over time.

serverStatus.mem.**resident**
The value of resident (page 371) is roughly equivalent to the amount of RAM, in megabytes (MB), currently used by the database process. In normal use this value tends to grow. In dedicated database servers this number tends to approach the total amount of system memory.

serverStatus.mem.**virtual**
virtual (page 371) displays the quantity, in megabytes (MB), of virtual memory used by the mongod (page 583) process. With *journaling* enabled, the value of virtual (page 371) is at least twice the value of mapped (page 371).

If virtual (page 371) value is significantly larger than mapped (page 371) (e.g. 3 or more times), this may indicate a memory leak.

serverStatus.mem.**supported**
supported (page 371) is true when the underlying system supports extended memory information. If this value is false and the system does not support extended memory information, then other mem (page 371) values may not be accessible to the database server.

serverStatus.mem.**mapped**
The value of mapped (page 371) provides the amount of mapped memory, in megabytes (MB), by the database. Because MongoDB uses memory-mapped files, this value is likely to be to be roughly equivalent to the total size of your database or databases.

serverStatus.mem.**mappedWithJournal**
mappedWithJournal (page 371) provides the amount of mapped memory, in megabytes (MB), including the memory used for journaling. This value will always be twice the value of mapped (page 371). This field is only included if journaling is enabled.

connections For an example of the connections output, see the *connections section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**connections**
The connections (page 371) sub document data regarding the current status of incoming connections and availability of the database server. Use these values to assess the current load and capacity requirements of the server.

serverStatus.connections.**current**
: The value of current (page 371) corresponds to the number of connections to the database server from clients. This number includes the current shell session. Consider the value of available (page 372) to add more context to this datum.

  This figure will include all incoming connections including any shell connections or connections from other servers, such as *replica set* members or mongos (page 601) instances.

serverStatus.connections.**available**
: available (page 372) provides a count of the number of unused available incoming connections the database can provide. Consider this value in combination with the value of current (page 371) to understand the connection load on the database, and the http://docs.mongodb.org/manual/reference/ulimit document for more information about system thresholds on available connections.

serverStatus.connections.**totalCreated**
: totalCreated (page 372) provides a count of **all** incoming connections created to the server. This number includes connections that have since closed.

extra_info For an example of the extra_info output, see the *extra_info section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**extra_info**
: The extra_info (page 372) data structure holds data collected by the mongod (page 583) instance about the underlying system. Your system may only report a subset of these fields.

serverStatus.extra_info.**note**
: The field note (page 372) reports that the data in this structure depend on the underlying platform, and has the text: "fields vary by platform."

serverStatus.extra_info.**heap_usage_bytes**
: The heap_usage_bytes (page 372) field is only available on Unix/Linux systems, and reports the total size in bytes of heap space used by the database process.

serverStatus.extra_info.**page_faults**
: The page_faults (page 372) Reports the total number of page faults that require disk operations. Page faults refer to operations that require the database server to access data which isn't available in active memory. The page_faults (page 372) counter may increase dramatically during moments of poor performance and may correlate with limited memory environments and larger data sets. Limited and sporadic page faults do not necessarily indicate an issue.

indexCounters For an example of the indexCounters output, see the *indexCounters section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**indexCounters**
: Changed in version 2.2: Previously, data in the indexCounters (page 372) document reported sampled data, and were only useful in relative comparison to each other, because they could not reflect absolute index use. In 2.2 and later, these data reflect actual index use.

  Changed in version 2.4: Fields previously in the btree sub-document of indexCounters (page 372) are now fields in the indexCounters (page 372) document.

  The indexCounters (page 372) data structure reports information regarding the state and use of indexes in MongoDB.

serverStatus.indexCounters.**accesses**
: accesses (page 372) reports the number of times that operations have accessed indexes. This value is the combination of the hits (page 373) and misses (page 373). Higher values indicate that your database has

indexes and that queries are taking advantage of these indexes. If this number does not grow over time, this might indicate that your indexes do not effectively support your use.

serverStatus.indexCounters.**hits**
The hits (page 373) value reflects the number of times that an index has been accessed and mongod (page 583) is able to return the index from memory.

A higher value indicates effective index use. hits (page 373) values that represent a greater proportion of the accesses (page 372) value, tend to indicate more effective index configuration.

serverStatus.indexCounters.**misses**
The misses (page 373) value represents the number of times that an operation attempted to access an index that was not in memory. These "misses," do not indicate a failed query or operation, but rather an inefficient use of the index. Lower values in this field indicate better index use and likely overall performance as well.

serverStatus.indexCounters.**resets**
The resets (page 373) value reflects the number of times that the index counters have been reset since the database last restarted. Typically this value is 0, but use this value to provide context for the data specified by other indexCounters (page 372) values.

serverStatus.indexCounters.**missRatio**
The missRatio (page 373) value is the ratio of hits (page 373) to misses (page 373). This value is typically 0 or approaching 0.

backgroundFlushing For an example of the backgroundFlushing output, see the *backgroundFlushing section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**backgroundFlushing**
mongod (page 583) periodically flushes writes to disk. In the default configuration, this happens every 60 seconds. The backgroundFlushing (page 373) data structure contains data regarding these operations. Consider these values if you have concerns about write performance and *journaling* (page 379).

serverStatus.backgroundFlushing.**flushes**
flushes (page 373) is a counter that collects the number of times the database has flushed all writes to disk. This value will grow as database runs for longer periods of time.

serverStatus.backgroundFlushing.**total_ms**
The total_ms (page 373) value provides the total number of milliseconds (ms) that the mongod (page 583) processes have spent writing (i.e. flushing) data to disk. Because this is an absolute value, consider the value of flushes (page 373) and average_ms (page 373) to provide better context for this datum.

serverStatus.backgroundFlushing.**average_ms**
The average_ms (page 373) value describes the relationship between the number of flushes and the total amount of time that the database has spent writing data to disk. The larger flushes (page 373) is, the more likely this value is likely to represent a "normal," time; however, abnormal data can skew this value.

Use the last_ms (page 373) to ensure that a high average is not skewed by transient historical issue or a random write distribution.

serverStatus.backgroundFlushing.**last_ms**
The value of the last_ms (page 373) field is the amount of time, in milliseconds, that the last flush operation took to complete. Use this value to verify that the current performance of the server and is in line with the historical data provided by average_ms (page 373) and total_ms (page 373).

serverStatus.backgroundFlushing.**last_finished**
The last_finished (page 373) field provides a timestamp of the last completed flush operation in the *ISODate* format. If this value is more than a few minutes old relative to your server's current time and accounting for differences in time zone, restarting the database may result in some data loss.

Also consider ongoing operations that might skew this value by routinely block write operations.

cursors Deprecated since version 2.6: See the *serverStatus.metrics.cursor* (page 382) field instead.

For an example of the `cursors` output, see the *cursors section* of the `http://docs.mongodb.org/manual/reference/server-status` page.

`serverStatus.`**`cursors`**
The `cursors` (page 374) data structure contains data regarding cursor state and use.

`serverStatus.cursors.`**`note`**
A note specifying to use the `serverStatus.metrics.cursor` (page 385) field instead of `serverStatus.cursors` (page 374).

`serverStatus.cursors.`**`totalOpen`**
`totalOpen` (page 374) provides the number of cursors that MongoDB is maintaining for clients. Because MongoDB exhausts unused cursors, typically this value small or zero. However, if there is a queue, stale *tailable cursor*, or a large number of operations, this value may rise.

`serverStatus.cursors.`**`clientCursors_size`**
Deprecated since version 1.x: See `totalOpen` (page 374) for this datum.

`serverStatus.cursors.`**`timedOut`**
`timedOut` (page 374) provides a counter of the total number of cursors that have timed out since the server process started. If this number is large or growing at a regular rate, this may indicate an application error.

`serverStatus.cursors.`**`totalNoTimeout`**
`totalNoTimeout` (page 374) provides the number of open cursors with the option `DBQuery.Option.noTimeout` (page 82) set to prevent timeout after a period of inactivity.

`serverStatus.cursors.`**`pinned`**
`serverStatus.cursors.pinned` (page 374) provides the number of "pinned" open cursors.

network For an example of the `network` output, see the *network section* of the `http://docs.mongodb.org/manual/reference/server-status` page.

`serverStatus.`**`network`**
The `network` (page 374) data structure contains data regarding MongoDB's network use.

`serverStatus.network.`**`bytesIn`**
The value of the `bytesIn` (page 374) field reflects the amount of network traffic, in bytes, received *by* this database. Use this value to ensure that network traffic sent to the `mongod` (page 583) process is consistent with expectations and overall inter-application traffic.

`serverStatus.network.`**`bytesOut`**
The value of the `bytesOut` (page 374) field reflects the amount of network traffic, in bytes, sent *from* this database. Use this value to ensure that network traffic sent by the `mongod` (page 583) process is consistent with expectations and overall inter-application traffic.

`serverStatus.network.`**`numRequests`**
The `numRequests` (page 374) field is a counter of the total number of distinct requests that the server has received. Use this value to provide context for the `bytesIn` (page 374) and `bytesOut` (page 374) values to ensure that MongoDB's network utilization is consistent with expectations and application use.

repl For an example of the `repl` output, see the *repl section* of the `http://docs.mongodb.org/manual/reference/server-status` page.

`serverStatus.`**`repl`**
The `repl` (page 374) data structure contains status information for MongoDB's replication (i.e. "replica set") configuration. These values only appear when the current host has replication enabled.

See `http://docs.mongodb.org/manual/replication` for more information on replication.

serverStatus.repl.**setName**
The setName (page 375) field contains a string with the name of the current replica set. This value reflects the *--replSet* command line argument, or replSetName value in the configuration file.

See http://docs.mongodb.org/manual/replication for more information on replication.

serverStatus.repl.**ismaster**
The value of the ismaster (page 375) field is either true or false and reflects whether the current node is the master or primary node in the replica set.

See http://docs.mongodb.org/manual/replication for more information on replication.

serverStatus.repl.**secondary**
The value of the secondary (page 375) field is either true or false and reflects whether the current node is a secondary node in the replica set.

See http://docs.mongodb.org/manual/replication for more information on replication.

serverStatus.repl.**primary**
New in version 2.8.0.

A string in the format of "[hostname]:[port]" listing the current *primary* member of the replica set.

serverStatus.repl.**hosts**
hosts (page 375) is an array that lists the other nodes in the current replica set. Each member of the replica set appears in the form of hostname:port.

See http://docs.mongodb.org/manual/replication for more information on replication.

serverStatus.repl.**me**
New in version 2.8.0.

The [hostname]:[port] combination for the current member in the replica set.

serverStatus.repl.**rabid**
New in version 2.8.0.

Rollback identifier. Used to determine if a rollback has happened for this mongod (page 583) instance.

serverStatus.repl.**slaves**
New in version 2.8.0.

An array with one document for every member of the replica set that reports replication process to this member. Typically this is the primary, or secondaries if using chained replication.

To include this output you must pass the‘‘repl‘‘ option to the:dbcommand:*serverStatus*, as in the following:

```
db.serverStatus({ "repl": 1 })
db.runCommand({ "serverStatus": 1, "repl": 1 })
```

The content of the slaves (page 375) section depends on the source of each member's replication. This section supports internal operation and is for internal and diagnostic use only.

serverStatus.repl.slaves[n].**rid**
An ID in the form of an ObjectId of the member's‘ of the replica set. For internal use only.

serverStatus.repl.slaves[n].**host**
The name of the host in [hostname]:[port] format for the member of the replica set.

serverStatus.repl.slaves[n].**optime**
Information regarding the last operation from the *oplog* that the member applied, as reported from this member.

serverStatus.repl.slaves[n].**memberID**
The integer identifier for this member of the replica set.

opcountersRepl For an example of the opcountersRepl output, see the *opcountersRepl section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**opcountersRepl**
The opcountersRepl (page 376) data structure, similar to the opcounters (page 376) data structure, provides an overview of database replication operations by type and makes it possible to analyze the load on the replica in more granular manner. These values only appear when the current host has replication enabled.

These values will differ from the opcounters (page 376) values because of how MongoDB serializes operations during replication. See http://docs.mongodb.org/manual/replication for more information on replication.

These numbers will grow over time in response to database use. Analyze these values over time to track database utilization.

serverStatus.opcountersRepl.**insert**
insert (page 376) provides a counter of the total number of replicated insert operations since the mongod (page 583) instance last started.

serverStatus.opcountersRepl.**query**
query (page 376) provides a counter of the total number of replicated queries since the mongod (page 583) instance last started.

serverStatus.opcountersRepl.**update**
update (page 376) provides a counter of the total number of replicated update operations since the mongod (page 583) instance last started.

serverStatus.opcountersRepl.**delete**
delete (page 376) provides a counter of the total number of replicated delete operations since the mongod (page 583) instance last started.

serverStatus.opcountersRepl.**getmore**
getmore (page 376) provides a counter of the total number of "getmore" operations since the mongod (page 583) instance last started. This counter can be high even if the query count is low. Secondary nodes send getMore operations as part of the replication process.

serverStatus.opcountersRepl.**command**
command (page 376) provides a counter of the total number of replicated commands issued to the database since the mongod (page 583) instance last started.

opcounters For an example of the opcounters output, see the *opcounters section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**opcounters**
The opcounters (page 376) data structure provides an overview of database operations by type and makes it possible to analyze the load on the database in more granular manner.

These numbers will grow over time and in response to database use. Analyze these values over time to track database utilization.

Note: The data in opcounters (page 376) treats operations that affect multiple documents, such as bulk insert or multi-update operations, as a single operation. See document (page 382) for more granular document-level operation tracking.

---

serverStatus.opcounters.**insert**
insert (page 376) provides a counter of the total number of insert operations since the mongod (page 583) instance last started.

serverStatus.opcounters.**query**
: query (page 376) provides a counter of the total number of queries since the mongod (page 583) instance last started.

serverStatus.opcounters.**update**
: update (page 377) provides a counter of the total number of update operations since the mongod (page 583) instance last started.

serverStatus.opcounters.**delete**
: delete (page 377) provides a counter of the total number of delete operations since the mongod (page 583) instance last started.

serverStatus.opcounters.**getmore**
: getmore (page 377) provides a counter of the total number of "getmore" operations since the mongod (page 583) instance last started. This counter can be high even if the query count is low. Secondary nodes send getMore operations as part of the replication process.

serverStatus.opcounters.**command**
: command (page 377) provides a counter of the total number of commands issued to the database since the mongod (page 583) instance last started.

rangeDeleter New in version 2.8.0.

Note: The rangeDeleter (page 377) data is only included in the output of serverStatus (page 366) if explicitly enabled. To return the rangeDeleter (page 377), use one of the following commands:

```
db.serverStatus( { rangeDeleter: 1 } )
db.runCommand( { serverStatus: 1, rangeDeleter: 1 } )
```

---

serverStatus.**rangeDeleter**
: A document that reports on the work performed by the cleanupOrphaned (page 305) command and the cleanup phase of the moveChunk (page 310) command.

serverStatus.rangeDeleter.**lastDeleteStats**
: An array of documents that each report on the last operations of migration cleanup operations. At most lastDeleteStats (page 377) will report data for the last 10 operations.

serverStatus.rangeDeleter.lastDeleteStats[n].**deletedDocs**
: A counter with the number of documents deleted by migration cleanup operations.

serverStatus.rangeDeleter.lastDeleteStats[n].**queueStart**
: A timestamp that reflects when operations began entering the queue for the migration cleanup operation. Specifically, operations wait in the queue while the mongod (page 583) waits for open cursors to close on the namespace.

serverStatus.rangeDeleter.lastDeleteStats[n].**queueEnd**
: A timestamp that reflects when the migration cleanup operation begins.

serverStatus.rangeDeleter.lastDeleteStats[n].**deleteStart**
: A timestamp for the beginning of the delete process that is part of the migration cleanup operation.

serverStatus.rangeDeleter.lastDeleteStats[n].**deleteEnd**
: A timestamp for the end of the delete process that is part of the migration cleanup operation.

serverStatus.rangeDeleter.lastDeleteStats[n].**waitForReplStart**
: A timestamp that reflects when the migration cleanup operation began waiting for replication to process the delete operation.

serverStatus.rangeDeleter.lastDeleteStats[n].**waitForReplEnd**
A timestamp that reflects when the migration cleanup operation finished waiting for replication to process the delete operation.

security New in version 2.8.0.

For an example of the security output, see the *security section* of the http://docs.mongodb.org/manual/reference/server-status page.

severStatus.**security**
A document reporting security configuration and details. Only appears for mongod (page 583) instances compiled with support for SSL.

severStatus.security.**SSLServerSubjectName**
The subject name associated with the SSL certificate specified by net.ssl.PEMKeyPassword.

severStatus.security.**SSLServerHasCertificateAuthority**
A boolean that is true when the SSL certificate specified by net.ssl.PEMKeyPassword is associated with a certificate authority. false when the SSL certificate is self-signed.

severStatus.security.**SSLServerCertificateExpirationDate**
A *date object* object that represents the date when the SSL certificate specified by net.ssl.PEMKeyPassword expires.

storageEngine New in version 2.8.0.

For an example of the storageEngine output, see the *storageEngine section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**storageEngine**
A document with data about the current storage engine.

serverStatus.storageEngine.**name**
A string that represents the name of the current storage engine.

asserts For an example of the asserts output, see the *asserts section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**asserts**
The asserts (page 378) document reports the number of asserts on the database. While assert errors are typically uncommon, if there are non-zero values for the asserts (page 378), you should check the log file for the mongod (page 583) process for more information. In many cases these errors are trivial, but are worth investigating.

serverStatus.asserts.**regular**
The regular (page 378) counter tracks the number of regular assertions raised since the server process started. Check the log file for more information about these messages.

serverStatus.asserts.**warning**
The warning (page 378) counter tracks the number of warnings raised since the server process started. Check the log file for more information about these warnings.

serverStatus.asserts.**msg**
The msg (page 378) counter tracks the number of message assertions raised since the server process started. Check the log file for more information about these messages.

serverStatus.asserts.**user**
The user (page 378) counter reports the number of "user asserts" that have occurred since the last time the

server process started. These are errors that user may generate, such as out of disk space or duplicate key. You can prevent these assertions by fixing a problem with your application or deployment. Check the MongoDB log for more information.

`serverStatus.asserts.`**`rollovers`**
The `rollovers` (page 379) counter displays the number of times that the rollover counters have rolled over since the last time the server process started. The counters will rollover to zero after 2^{30} assertions. Use this value to provide context to the other values in the `asserts` (page 378) data structure.

writeBacksQueued For an example of the `writeBacksQueued` output, see the *writeBacksQueued section* of the `http://docs.mongodb.org/manual/reference/server-status` page.

`serverStatus.`**`writeBacksQueued`**
The value of `writeBacksQueued` (page 379) is `true` when there are operations from a `mongos` (page 601) instance queued for retrying. Typically this option is false.

See also:

writeBacks

Journaling (dur) New in version 1.8.

For an example of the `Journaling (dur)` output, see the *journaling section* of the `http://docs.mongodb.org/manual/reference/server-status` page.

`serverStatus.`**`dur`**
The `dur` (page 379) (for "durability") document contains data regarding the `mongod` (page 583)'s journaling-related operations and performance. `mongod` (page 583) must be running with journaling for these data to appear in the output of "`serverStatus` (page 366)".

MongoDB reports the data in `dur` (page 379) based on 3 second intervals of data, collected between 3 and 6 seconds in the past.

See also:

`http://docs.mongodb.org/manual/core/journaling` for more information about journaling operations.

`serverStatus.dur.`**`commits`**
The `commits` (page 379) provides the number of transactions written to the *journal* during the last *journal group commit interval*.

`serverStatus.dur.`**`journaledMB`**
The `journaledMB` (page 379) provides the amount of data in megabytes (MB) written to *journal* during the last *journal group commit interval*.

`serverStatus.dur.`**`writeToDataFilesMB`**
The `writeToDataFilesMB` (page 379) provides the amount of data in megabytes (MB) written from *journal* to the data files during the last *journal group commit interval*.

`serverStatus.dur.`**`compression`**
New in version 2.0.

The `compression` (page 379) represents the compression ratio of the data written to the *journal*:

```
( journaled_size_of_data / uncompressed_size_of_data )
```

`serverStatus.dur.`**`commitsInWriteLock`**
The `commitsInWriteLock` (page 379) provides a count of the commits that occurred while a write lock was held. Commits in a write lock indicate a MongoDB node under a heavy write load and call for further diagnosis.

serverStatus.dur.**earlyCommits**
: The earlyCommits (page 379) value reflects the number of times MongoDB requested a commit before the scheduled *journal group commit interval*. Use this value to ensure that your *journal group commit interval* is not too long for your deployment.

serverStatus.dur.**timeMS**
: The timeMS (page 380) document provides information about the performance of the mongod (page 583) instance during the various phases of journaling in the last *journal group commit interval*.

serverStatus.dur.timeMS.**dt**
: The dt (page 380) value provides, in milliseconds, the amount of time over which MongoDB collected the timeMS (page 380) data. Use this field to provide context to the other timeMS (page 380) field values.

serverStatus.dur.timeMS.**prepLogBuffer**
: The prepLogBuffer (page 380) value provides, in milliseconds, the amount of time spent preparing to write to the journal. Smaller values indicate better journal performance.

serverStatus.dur.timeMS.**writeToJournal**
: The writeToJournal (page 380) value provides, in milliseconds, the amount of time spent actually writing to the journal. File system speeds and device interfaces can affect performance.

serverStatus.dur.timeMS.**writeToDataFiles**
: The writeToDataFiles (page 380) value provides, in milliseconds, the amount of time spent writing to data files after journaling. File system speeds and device interfaces can affect performance.

serverStatus.dur.timeMS.**remapPrivateView**
: The remapPrivateView (page 380) value provides, in milliseconds, the amount of time spent remapping copy-on-write memory mapped views. Smaller values indicate better journal performance.

recordStats For an example of the recordStats output, see the *recordStats section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**recordStats**
: The recordStats (page 380) document provides fine grained reporting on page faults on a per database level.

  MongoDB uses a read lock on each database to return recordStats (page 380). To minimize this overhead, you can disable this section, as in the following operation:

  ```
  db.serverStatus( { recordStats: 0 } )
  ```

serverStatus.recordStats.**accessesNotInMemory**
: accessesNotInMemory (page 380) reflects the number of times mongod (page 583) needed to access a memory page that was *not* resident in memory for *all* databases managed by this mongod (page 583) instance.

serverStatus.recordStats.**pageFaultExceptionsThrown**
: pageFaultExceptionsThrown (page 380) reflects the number of page fault exceptions thrown by mongod (page 583) when accessing data for *all* databases managed by this mongod (page 583) instance.

serverStatus.recordStats.local.**accessesNotInMemory**
: accessesNotInMemory (page 380) reflects the number of times mongod (page 583) needed to access a memory page that was *not* resident in memory for the local database.

serverStatus.recordStats.local.**pageFaultExceptionsThrown**
: pageFaultExceptionsThrown (page 380) reflects the number of page fault exceptions thrown by mongod (page 583) when accessing data for the local database.

serverStatus.recordStats.admin.**accessesNotInMemory**
: accessesNotInMemory (page 380) reflects the number of times mongod (page 583) needed to access a memory page that was *not* resident in memory for the *admin database*.

serverStatus.recordStats.admin.**pageFaultExceptionsThrown**
: pageFaultExceptionsThrown (page 380) reflects the number of page fault exceptions thrown by mongod (page 583) when accessing data for the *admin database*.

serverStatus.recordStats.<database>.**accessesNotInMemory**
: accessesNotInMemory (page 381) reflects the number of times mongod (page 583) needed to access a memory page that was *not* resident in memory for the <database> database.

serverStatus.recordStats.<database>.**pageFaultExceptionsThrown**
: pageFaultExceptionsThrown (page 381) reflects the number of page fault exceptions thrown by mongod (page 583) when accessing data for the <database> database.

workingSet New in version 2.4.

Note: The workingSet (page 381) data is only included in the output of serverStatus (page 366) if explicitly enabled. To return the workingSet (page 381), use one of the following commands:

```
db.serverStatus( { workingSet: 1 } )
db.runCommand( { serverStatus: 1, workingSet: 1 } )
```

For an example of the workingSet output, see the *workingSet section* of the http://docs.mongodb.org/manual/reference/server-status page.

serverStatus.**workingSet**
: workingSet (page 381) is a document that contains values useful for estimating the size of the working set, which is the amount of data that MongoDB uses actively. workingSet (page 381) uses an internal data structure that tracks pages accessed by mongod (page 583).

serverStatus.workingSet.**note**
: note (page 381) is a field that holds a string warning that the workingSet (page 381) document is an estimate.

serverStatus.workingSet.**pagesInMemory**
: pagesInMemory (page 381) contains a count of the total number of pages accessed by mongod (page 583) over the period displayed in overSeconds (page 381). The default page size is 4 kilobytes: to convert this value to the amount of data in memory multiply this value by 4 kilobytes.

  If your total working set is less than the size of physical memory, over time the value of pagesInMemory (page 381) will reflect your data size.

  Use pagesInMemory (page 381) in conjunction with overSeconds (page 381) to help estimate the actual size of the working set.

serverStatus.workingSet.**computationTimeMicros**
: computationTimeMicros (page 381) reports the amount of time the mongod (page 583) instance used to compute the other fields in the workingSet (page 381) section.

  Reporting on workingSet (page 381) may impact the performance of other operations on the mongod (page 583) instance because MongoDB must collect some data within the context of a lock. Ensure that automated monitoring tools consider this metric when determining the frequency of collection for workingSet (page 381).

serverStatus.workingSet.**overSeconds**
: overSeconds (page 381) returns the amount of time elapsed between the newest and oldest pages tracked in the pagesInMemory (page 381) data point.

  If overSeconds (page 381) is decreasing, or if pagesInMemory (page 381) equals physical RAM *and* overSeconds (page 381) is very small, the working set may be much *larger* than physical RAM.

When overSeconds (page 381) is large, MongoDB's data set is equal to or *smaller* than physical RAM.

metrics For an example of the metrics output, see the *metrics section* of the http://docs.mongodb.org/manual/reference/server-status page.

New in version 2.4.0.

serverStatus.**metrics**
The metrics (page 382) document holds a number of statistics that reflect the current use and state of a running mongod (page 583) instance.

serverStatus.metrics.**commands**
New in version 2.8.0.

A document that reports on the use of database commands. The fields in commands (page 382) are the names of *database commands* (page 210) and each value is a document that reports the total number of commands executed as well as the number of failed executions.

serverStatus.metrics.commands.<command>.**failed**
The number of times <command> failed on this mongod (page 583).

serverStatus.metrics.commands.<command>.**total**
The number of times <command> executed on this mongod (page 583).

serverStatus.metrics.**document**
The document (page 382) holds a document of that reflect document access and modification patterns and data use. Compare these values to the data in the opcounters (page 376) document, which track total number of operations.

serverStatus.metrics.document.**deleted**
deleted (page 382) reports the total number of documents deleted.

serverStatus.metrics.document.**inserted**
inserted (page 382) reports the total number of documents inserted.

serverStatus.metrics.document.**returned**
returned (page 382) reports the total number of documents returned by queries.

serverStatus.metrics.document.**updated**
updated (page 382) reports the total number of documents updated.

serverStatus.metrics.**getLastError**
getLastError (page 382) is a document that reports on getLastError (page 245) use.

serverStatus.metrics.getLastError.**wtime**
wtime (page 382) is a sub-document that reports getLastError (page 245) operation counts with a w argument greater than 1.

serverStatus.metrics.getLastError.wtime.**num**
num (page 382) reports the total number of getLastError (page 245) operations with a specified write concern (i.e. w) that wait for one or more members of a replica set to acknowledge the write operation (i.e. a w value greater than 1.)

serverStatus.metrics.getLastError.wtime.**totalMillis**
totalMillis (page 382) reports the total amount of time in milliseconds that the mongod (page 583) has spent performing getLastError (page 245) operations with write concern (i.e. w) that wait for one or more members of a replica set to acknowledge the write operation (i.e. a w value greater than 1.)

serverStatus.metrics.getLastError.**wtimeouts**
wtimeouts (page 382) reports the number of times that *write concern* operations have timed out as a result of the wtimeout threshold to getLastError (page 245).

serverStatus.metrics.**operation**
operation (page 382) is a sub-document that holds counters for several types of update and query operations that MongoDB handles using special operation types.

serverStatus.metrics.operation.**fastmod**
fastmod (page 383) reports the number of update operations that neither cause documents to grow nor require updates to the index. For example, this counter would record an update operation that use the $inc (page 452) operator to increment the value of a field that is not indexed.

serverStatus.metrics.operation.**idhack**
idhack (page 383) reports the number of queries that contain the _id field. For these queries, MongoDB will use default index on the _id field and skip all query plan analysis.

serverStatus.metrics.operation.**scanAndOrder**
scanAndOrder (page 383) reports the total number of queries that return sorted numbers that cannot perform the sort operation using an index.

serverStatus.metrics.**queryExecutor**
queryExecutor (page 383) is a document that reports data from the query execution system.

serverStatus.metrics.queryExecutor.**scanned**
scanned (page 383) reports the total number of index items scanned during queries and query-plan evaluation. This counter is the same as nscanned (page **??**) in the output of explain() (page 85).

serverStatus.metrics.**record**
record (page 383) is a document that reports data related to record allocation in the on-disk memory files.

serverStatus.metrics.record.**moves**
moves (page 383) reports the total number of times documents move within the on-disk representation of the MongoDB data set. Documents move as a result of operations that increase the size of the document beyond their allocated record size.

serverStatus.metrics.**repl**
repl (page 383) holds a sub-document that reports metrics related to the replication process. repl (page 383) document appears on all mongod (page 583) instances, even those that aren't members of *replica sets*.

serverStatus.metrics.repl.**apply**
apply (page 383) holds a sub-document that reports on the application of operations from the replication *oplog*.

serverStatus.metrics.repl.apply.**batches**
batches (page 383) reports on the oplog application process on *secondaries* members of replica sets. See *replica-set-internals-multi-threaded-replication* for more information on the oplog application processes

serverStatus.metrics.repl.apply.batches.**num**
num (page 383) reports the total number of batches applied across all databases.

serverStatus.metrics.repl.apply.batches.**totalMillis**
totalMillis (page 383) reports the total amount of time the mongod (page 583) has spent applying operations from the oplog.

serverStatus.metrics.repl.apply.**ops**
ops (page 383) reports the total number of *oplog* operations applied.

serverStatus.metrics.repl.**buffer**
MongoDB buffers oplog operations from the replication sync source buffer before applying oplog entries in a batch. buffer (page 383) provides a way to track the oplog buffer. See *replica-set-internals-multi-threaded-replication* for more information on the oplog application process.

serverStatus.metrics.repl.buffer.**count**
count (page 383) reports the current number of operations in the oplog buffer.

serverStatus.metrics.repl.buffer.**maxSizeBytes**
maxSizeBytes (page 383) reports the maximum size of the buffer. This value is a constant setting in the mongod (page 583), and is not configurable.

serverStatus.metrics.repl.buffer.**sizeBytes**
sizeBytes (page 384) reports the current size of the contents of the oplog buffer.

serverStatus.metrics.repl.**network**
network (page 384) reports network use by the replication process.

serverStatus.metrics.repl.network.**bytes**
bytes (page 384) reports the total amount of data read from the replication sync source.

serverStatus.metrics.repl.network.**getmores**
getmores (page 384) reports on the getmore operations, which are requests for additional results from the oplog *cursor* as part of the oplog replication process.

serverStatus.metrics.repl.network.getmores.**num**
num (page 384) reports the total number of getmore operations, which are operations that request an additional set of operations from the replication sync source.

serverStatus.metrics.repl.network.getmores.**totalMillis**
totalMillis (page 384) reports the total amount of time required to collect data from getmore operations.

Note: This number can be quite large, as MongoDB will wait for more data even if the getmore operation does not initial return data.

---

serverStatus.metrics.repl.network.**ops**
ops (page 384) reports the total number of operations read from the replication source.

serverStatus.metrics.repl.network.**readersCreated**
readersCreated (page 384) reports the total number of oplog query processes created. MongoDB will create a new oplog query any time an error occurs in the connection, including a timeout, or a network operation. Furthermore, readersCreated (page 384) will increment every time MongoDB selects a new source fore replication.

serverStatus.metrics.repl.**oplog**
oplog (page 384) is a document that reports on the size and use of the *oplog* by this mongod (page 583) instance.

serverStatus.metrics.repl.oplog.**insert**
insert (page 384) is a document that reports insert operations into the *oplog*.

serverStatus.metrics.repl.oplog.insert.**num**
num (page 384) reports the total number of items inserted into the *oplog*.

serverStatus.metrics.repl.oplog.insert.**totalMillis**
totalMillis (page 384) reports the total amount of time spent for the mongod (page 583) to insert data into the *oplog*.

serverStatus.metrics.repl.oplog.**insertBytes**
insertBytes (page 384) the total size of documents inserted into the oplog.

serverStatus.metrics.repl.**preload**
preload (page 384) reports on the "pre-fetch" stage, where MongoDB loads documents and indexes into RAM to improve replication throughput.

See *replica-set-internals-multi-threaded-replication* for more information about the *pre-fetch* stage of the replication process.

serverStatus.metrics.repl.preload.**docs**
docs (page 384) is a sub-document that reports on the documents loaded into memory during the *pre-fetch* stage.

serverStatus.metrics.repl.preload.docs.**num**
num (page 385) reports the total number of documents loaded during the *pre-fetch* stage of replication.

serverStatus.metrics.repl.preload.docs.**totalMillis**
totalMillis (page 385) reports the total amount of time spent loading documents as part of the *pre-fetch* stage of replication.

serverStatus.metrics.repl.preload.**indexes**
indexes (page 385) is a sub-document that reports on the index items loaded into memory during the *pre-fetch* stage of replication.

See *replica-set-internals-multi-threaded-replication* for more information about the *pre-fetch* stage of replication.

serverStatus.metrics.repl.preload.indexes.**num**
num (page 385) reports the total number of index entries loaded by members before updating documents as part of the *pre-fetch* stage of replication.

serverStatus.metrics.repl.preload.indexes.**totalMillis**
totalMillis (page 385) reports the total amount of time spent loading index entries as part of the *pre-fetch* stage of replication.

serverStatus.storage.freelist.search.**bucketExhausted**
bucketExhausted (page 385) reports the number of times that mongod (page 583) has checked the free list without finding a suitably large record allocation.

serverStatus.storage.freelist.search.**requests**
requests (page 385) reports the number of times mongod (page 583) has searched for available record allocations.

serverStatus.storage.freelist.search.**scanned**
scanned (page 385) reports the number of available record allocations mongod (page 583) has searched.

serverStatus.metrics.**ttl**
ttl (page 385) is a sub-document that reports on the operation of the resource use of the `ttl index` process.

serverStatus.metrics.ttl.**deletedDocuments**
deletedDocuments (page 385) reports the total number of documents deleted from collections with a `ttl index`.

serverStatus.metrics.ttl.**passes**
passes (page 385) reports the number of times the background process removes documents from collections with a `ttl index`.

serverStatus.metrics.**cursor**
New in version 2.6.

The cursor (page 385) is a document that contains data regarding cursor state and use.

serverStatus.metrics.cursor.**timedOut**
New in version 2.6.

timedOut (page 385) provides the total number of cursors that have timed out since the server process started. If this number is large or growing at a regular rate, this may indicate an application error.

serverStatus.metrics.cursor.**open**
New in version 2.6.

The open (page 385) is an embedded document that contains data regarding open cursors.

serverStatus.metrics.cursor.open.**noTimeout**
New in version 2.6.

noTimeout (page 385) provides the number of open cursors with the option DBQuery.Option.noTimeout (page 82) set to prevent timeout after a period of inactivity.

serverStatus.metrics.cursor.open.**pinned**
New in version 2.6.

serverStatus.metrics.cursor.open.pinned (page 386) provides the number of "pinned" open cursors.

serverStatus.metrics.cursor.open.**total**
New in version 2.6.

total (page 386) provides the number of cursors that MongoDB is maintaining for clients. Because MongoDB exhausts unused cursors, typically this value small or zero. However, if there is a queue, stale tailable cursors, or a large number of operations this value may rise.

shardConnPoolStats

Definition

shardConnPoolStats
Returns information on the pooled and cached connections in the sharded connection pool. The command also returns information on the per-thread connection cache in the connection pool.

The shardConnPoolStats (page 386) command uses the following syntax:

```
{ shardConnPoolStats: 1 }
```

The sharded connection pool is specific to connections between members in a sharded cluster. The mongos (page 601) instances in a cluster use the connection pool to execute client reads and writes. The mongod (page 583) instances in a cluster use the pool when issuing mapReduce (page 220) to query temporary collections on other shards.

When the cluster requires a connection, MongoDB pulls a connection from the sharded connection pool into the per-thread connection cache. MongoDB returns the connection to the connection pool after every operation.

Output

shardConnPoolStats.**hosts**
Displays connection status for each *config server*, *replica set*, and *standalone instance* in the cluster.

shardConnPoolStats.hosts.<host>.**available**
The number of connections available for this host to connect to the mongos (page 601).

shardConnPoolStats.hosts.<host>.**created**
The number of connections the host has ever created to connect to the mongos (page 601).

shardConnPoolStats.**replicaSets**
Displays information specific to replica sets.

shardConnPoolStats.replicaSets.<name>.**host**
Holds an array of documents that report on each replica set member. These values derive from the *replica set status* (page 296) values.

shardConnPoolStats.replicaSets.<name>.host[n].**addr**
The host address in the format [hostname]:[port].

shardConnPoolStats.replicaSets.<name>.host[n].**ok**
This field is for internal use. Reports `false` when the `mongos` (page 601) either cannot connect to instance or received a connection exception or error.

shardConnPoolStats.replicaSets.<name>.host[n].**ismaster**
The host is the replica set's *primary* if this is set to `true`.

shardConnPoolStats.replicaSets.<name>.host[n].**hidden**
The host is a *hidden member* of the replica set if this is set to `true`.

shardConnPoolStats.replicaSets.<name>.host[n].**secondary**
The host is a *hidden member* of the replica set if this is set to `true`.

The host is a *secondary* member of the replica set if this is set to `true`.

shardConnPoolStats.replicaSets.<name>.host[n].**pingTimeMillis**
The latency, in milliseconds, from the `mongos` (page 601) to this member.

shardConnPoolStats.replicaSets.<name>.host[n].**tags**
The member has tags configured.

shardConnPoolStats.**createdByType**
The number connections in the cluster's connection pool.

shardConnPoolStats.createdByType.**master**
The number of connections to a shard.

shardConnPoolStats.createdByType.**set**
The number of connections to a replica set.

shardConnPoolStats.createdByType.**sync**
The number of connections to the config database.

shardConnPoolStats.**totalAvailable**
The number of connections available from the `mongos` (page 601) to the config servers, replica sets, and standalone `mongod` (page 583) instances in the cluster.

shardConnPoolStats.**totalCreated**
The number of connections the `mongos` (page 601) has ever created to other members of the cluster.

shardConnPoolStats.**threads**
Displays information on the per-thread connection cache.

shardConnPoolStats.threads.**hosts**
Displays each incoming client connection. For a `mongos` (page 601), this array field displays one document per incoming client thread. For a `mongod` (page 583), the array displays one entry per incoming sharded `mapReduce` (page 220) client thread.

shardConnPoolStats.threads.hosts.**host**
The host using the connection. The host can be a *config server*, *replica set*, or *standalone instance*.

shardConnPoolStats.threads.hosts.**created**
The number of times the host pulled a connection from the pool.

shardConnPoolStats.threads.hosts.**avail**
The thread's availability.

shardConnPoolStats.threads.**seenNS**
The namespaces used on this connection thus far.

top

top

top (page 387) is an administrative command that returns usage statistics for each collection. top (page 387) provides amount of time, in microseconds, used and a count of operations for the following event types:

- total
- readLock
- writeLock
- queries
- getmore
- insert
- update
- remove
- commands

Issue the top (page 387) command against the *admin database* in the form:

```
{ top: 1 }
```

Example At the mongo (page 610) shell prompt, use top (page 387) with the following evocation:

```
db.adminCommand("top")
```

Alternately you can use top (page 387) as follows:

```
use admin
db.runCommand( { top: 1 } )
```

The output of the top command would resemble the following output:

```
{
  "totals" : {
     "records.users" : {
                  "total" : {
                          "time" : 305277,
                          "count" : 2825
                  },
                  "readLock" : {
                          "time" : 305264,
                          "count" : 2824
                  },
                  "writeLock" : {
                          "time" : 13,
                          "count" : 1
                  },
                  "queries" : {
                          "time" : 305264,
                          "count" : 2824
                  },
                  "getmore" : {
                          "time" : 0,
                          "count" : 0
                  },
                  "insert" : {
                          "time" : 0,
```

```
                        "count" : 0
                },
                "update" : {
                        "time" : 0,
                        "count" : 0
                },
                "remove" : {
                        "time" : 0,
                        "count" : 0
                },
                "commands" : {
                        "time" : 0,
                        "count" : 0
                }
        }
}
```

validate

Definition

validate

The validate (page 389) command checks the structures within a namespace for correctness by scanning the collection's data and indexes. The command returns information regarding the on-disk representation of the collection.

The validate command can be slow, particularly on larger data sets.

The following example validates the contents of the collection named users.

```
{ validate: "users" }
```

You may also specify one of the following options:

- full: true provides a more thorough scan of the data.
- **scandata: false skips the scan of the base collection** without skipping the scan of the index.

The mongo (page 610) shell also provides a wrapper:

```
db.collection.validate();
```

Use one of the following forms to perform the full collection validation:

```
db.collection.validate(true)
db.runCommand( { validate: "collection", full: true } )
```

> **Warning:** This command is resource intensive and may have an impact on the performance of your MongoDB instance.

Output

validate.**ns**

The full namespace name of the collection. Namespaces include the database name and the collection name in the form database.collection.

validate.**firstExtent**

The disk location of the first extent in the collection. The value of this field also includes the namespace.

validate.**lastExtent**
The disk location of the last extent in the collection. The value of this field also includes the namespace.

validate.**extentCount**
The number of extents in the collection.

validate.**extents**
validate (page 389) returns one instance of this document for every extent in the collection. This sub-document is only returned when you specify the full option to the command.

validate.extents.**loc**
The disk location for the beginning of this extent.

validate.extents.**xnext**
The disk location for the extent following this one. "null" if this is the end of the linked list of extents.

validate.extents.**xprev**
The disk location for the extent preceding this one. "null" if this is the head of the linked list of extents.

validate.extents.**nsdiag**
The namespace this extent belongs to (should be the same as the namespace shown at the beginning of the validate listing).

validate.extents.**size**
The number of bytes in this extent.

validate.extents.**firstRecord**
The disk location of the first record in this extent.

validate.extents.**lastRecord**
The disk location of the last record in this extent.

validate.**datasize**
The number of bytes in all data records. This value does not include deleted records, nor does it include extent headers, nor record headers, nor space in a file unallocated to any extent. datasize (page 390) includes record *padding*.

validate.**nrecords**
The number of *documents* in the collection.

validate.**lastExtentSize**
The size of the last new extent created in this collection. This value determines the size of the *next* extent created.

validate.**padding**
A floating point value between 1 and 2.

When MongoDB creates a new record it uses the *padding factor* to determine how much additional space to add to the record. The padding factor is automatically adjusted by mongo when it notices that update operations are triggering record moves.

validate.**firstExtentDetails**
The size of the first extent created in this collection. This data is similar to the data provided by the extents (page 390) sub-document; however, the data reflects only the first extent in the collection and is always returned.

validate.firstExtentDetails.**loc**
The disk location for the beginning of this extent.

validate.firstExtentDetails.**xnext**
The disk location for the extent following this one. "null" if this is the end of the linked list of extents, which should only be the case if there is only one extent.

validate.firstExtentDetails.**xprev**
The disk location for the extent preceding this one. This should always be "null."

validate.firstExtentDetails.**nsdiag**
The namespace this extent belongs to (should be the same as the namespace shown at the beginning of the validate listing).

validate.firstExtentDetails.**size**
The number of bytes in this extent.

validate.firstExtentDetails.**firstRecord**
The disk location of the first record in this extent.

validate.firstExtentDetails.**lastRecord**
The disk location of the last record in this extent.

validate.**objectsFound**
The number of records actually encountered in a scan of the collection. This field should have the same value as the nrecords (page 390) field.

validate.**invalidObjects**
The number of records containing BSON documents that do not pass a validation check.

Note: This field is only included in the validation output when you specify the full option.

validate.**bytesWithHeaders**
This is similar to datasize, except that bytesWithHeaders (page 391) includes the record headers. In version 2.0, record headers are 16 bytes per document.

Note: This field is only included in the validation output when you specify the full option.

validate.**bytesWithoutHeaders**
bytesWithoutHeaders (page 391) returns data collected from a scan of all records. The value should be the same as datasize (page 390).

Note: This field is only included in the validation output when you specify the full option.

validate.**deletedCount**
The number of deleted or "free" records in the collection.

validate.**deletedSize**
The size of all deleted or "free" records in the collection.

validate.**nIndexes**
The number of indexes on the data in the collection.

validate.**keysPerIndex**
A document containing a field for each index, named after the index's name, that contains the number of keys, or documents referenced, included in the index.

validate.**valid**
Boolean. true, unless validate (page 389) determines that an aspect of the collection is not valid. When false, see the errors (page 391) field for more information.

validate.**errors**
Typically empty; however, if the collection is not valid (i.e valid (page 391) is false), this field will contain a message describing the validation error.

validate.**ok**
: Set to 1 when the command succeeds. If the command fails the ok (page 391) field has a value of 0.

whatsmyuri

whatsmyuri
: whatsmyuri (page 392) is an internal command.

2.2.3 Internal Commands

Internal Commands

Name	Description
_migrateClone (page 392)	Internal command that supports chunk migration. Do not call directly.
_recvChunkAbort (page 392)	Internal command that supports chunk migrations in sharded clusters. Do not call directly.
_recvChunkCommit (page 392)	Internal command that supports chunk migrations in sharded clusters. Do not call directly.
_recvChunkStart (page 393)	Internal command that facilitates chunk migrations in sharded clusters.. Do not call directly.
_recvChunkStatus (page 393)	Internal command that returns data to support chunk migrations in sharded clusters. Do not call directly.
_replSetFresh	Internal command that supports replica set election operations.
_transferMods (page 393)	Internal command that supports chunk migrations. Do not call directly.
handshake (page 393)	Internal command.
mapreduce.shardedfinish (page 393)	Internal command that supports *map-reduce* in *sharded cluster* environments.
replSetElect (page 393)	Internal command that supports replica set functionality.
replSetGetRBID (page 393)	Internal command that supports replica set operations.
replSetHeartbeat (page 394)	Internal command that supports replica set operations.
writeBacksQueued (page 394)	Internal command that supports chunk migrations in sharded clusters.
writebacklisten (page 395)	Internal command that supports chunk migrations in sharded clusters.

migrateClone

_migrateClone
: _migrateClone (page 392) is an internal command. Do not call directly.

recvChunkAbort

_recvChunkAbort
: _recvChunkAbort (page 392) is an internal command. Do not call directly.

recvChunkCommit

_recvChunkCommit
: _recvChunkCommit (page 392) is an internal command. Do not call directly.

recvChunkStart

_recvChunkStart
_recvChunkStart (page 393) is an internal command. Do not call directly.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

recvChunkStatus

_recvChunkStatus
_recvChunkStatus (page 393) is an internal command. Do not call directly.

replSetFresh

replSetFresh
replSetFresh (page 393) is an internal command that supports replica set functionality.

transferMods

_transferMods
_transferMods (page 393) is an internal command. Do not call directly.

handshake

handshake
handshake (page 393) is an internal command.

mapreduce.shardedfinish

mapreduce.**shardedfinish**
Provides internal functionality to support *map-reduce* in *sharded* environments.

See also:

"mapReduce (page 220)"

replSetElect

replSetElect
replSetElect (page 393) is an internal command that support replica set functionality.

replSetGetRBID

replSetGetRBID
replSetGetRBID (page 393) is an internal command that supports replica set functionality.

replSetHeartbeat

replSetHeartbeat

replSetHeartbeat (page 394) is an internal command that supports replica set functionality.

writeBacksQueued

writeBacksQueued

writeBacksQueued (page 394) is an internal command that returns a document reporting there are operations in the write back queue for the given mongos (page 601) and information about the queues.

writeBacksQueued.**hasOpsQueued**

Boolean.

hasOpsQueued (page 394) is `true` if there are `write Back` operations queued.

writeBacksQueued.**totalOpsQueued**

Integer.

totalOpsQueued (page 394) reflects the number of operations queued.

writeBacksQueued.**queues**

Document.

queues (page 394) holds a sub-document where the fields are all write back queues. These field hold a document with two fields that reports on the state of the queue. The fields in these documents are:

writeBacksQueued.queues.**n**

n (page 394) reflects the size, by number of items, in the queues.

writeBacksQueued.queues.**minutesSinceLastCall**

The number of minutes since the last time the mongos (page 601) touched this queue.

The command document has the following prototype form:

```
{writeBacksQueued: 1}
```

To call writeBacksQueued (page 394) from the mongo (page 610) shell, use the following db.runCommand() (page 123) form:

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

Consider the following example output:

```
{
  "hasOpsQueued" : true,
  "totalOpsQueued" : 7,
  "queues" : {
          "50b4f09f6671b11ff1944089" : { "n" : 0, "minutesSinceLastCall" : 1 },
          "50b4f09fc332bf1c5aeaaf59" : { "n" : 0, "minutesSinceLastCall" : 0 },
          "50b4f09f6671b1d51df98cb6" : { "n" : 0, "minutesSinceLastCall" : 0 },
          "50b4f0c67ccf1e5c6effb72e" : { "n" : 0, "minutesSinceLastCall" : 0 },
          "50b4faf12319f193cfdec0d1" : { "n" : 0, "minutesSinceLastCall" : 4 },
          "50b4f013d2c1f8d62453017e" : { "n" : 0, "minutesSinceLastCall" : 0 },
          "50b4f0f12319f193cfdec0d1" : { "n" : 0, "minutesSinceLastCall" : 1 }
  },
  "ok" : 1
}
```

writebacklisten

writebacklisten
writebacklisten (page 395) is an internal command.

2.2.4 Testing Commands

Testing Commands

Name	Description
_hashBSONElement (page 395)	Internal command. Computes the MD5 hash of a BSON element.
_journalLatencyTest	Tests the time required to write and perform a file system sync for a file in the journal directory.
captrunc (page 397)	Internal command. Truncates capped collections.
configureFailPoint (page 397)	Internal command for testing. Configures failure points.
emptycapped (page 398)	Internal command. Removes all documents from a capped collection.
forceerror (page 398)	Internal command for testing. Forces a user assertion exception.
godinsert (page 398)	Internal command for testing.
replSetTest (page 398)	Internal command for testing replica set functionality.
skewClockCommand	Internal command. Do not call this command directly.
sleep (page 399)	Internal command for testing. Forces MongoDB to block all operations.
testDistLockWithSkew	Internal command. Do not call this directly.
testDistLockWithSyncCluster	Internal command. Do not call this directly.

_hashBSONElement

Description
_hashBSONElement
New in version 2.4.

An internal command that computes the MD5 hash of a BSON element. The _hashBSONElement (page 395) command returns 8 bytes from the 16 byte MD5 hash.

The _hashBSONElement (page 395) command has the following form:

```
db.runCommand({ _hashBSONElement: <key> , seed: <seed> })
```

The _hashBSONElement (page 395) command has the following fields:

field BSONElement key The BSON element to hash.

field integer seed A seed used when computing the hash.

Note: _hashBSONElement (page 395) is an internal command that is not enabled by default. _hashBSONElement (page 395) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 583) command line. _hashBSONElement (page 395) cannot be enabled during run-time.

Output The _hashBSONElement (page 395) command returns a document that holds the following fields:

_hashBSONElement.**key**
: The original BSON element.

_hashBSONElement.**seed**
: The seed used for the hash, defaults to `0`.

_hashBSONElement.**out**
: The decimal result of the hash.

_hashBSONElement.**ok**
: Holds the `1` if the function returns successfully, and `0` if the operation encountered an error.

Example Invoke a `mongod` (page 583) instance with test commands enabled:

```
mongod --setParameter enableTestCommands=1
```

Run the following to compute the hash of an ISODate string:

```
db.runCommand({_hashBSONElement: ISODate("2013-02-12T22:12:57.211Z")})
```

The command returns the following document:

```
{
  "key" : ISODate("2013-02-12T22:12:57.211Z"),
  "seed" : 0,
  "out" : NumberLong("-4185544074338741873"),
  "ok" : 1
}
```

Run the following to hash the same ISODate string but this time to specify a seed value:

```
db.runCommand({_hashBSONElement: ISODate("2013-02-12T22:12:57.211Z"), seed:2013})
```

The command returns the following document:

```
{
  "key" : ISODate("2013-02-12T22:12:57.211Z"),
  "seed" : 2013,
  "out" : NumberLong("7845924651247493302"),
  "ok" : 1
}
```

journalLatencyTest

journalLatencyTest

`journalLatencyTest` (page 396) is an administrative command that tests the length of time required to write and perform a file system sync (e.g. *fsync*) for a file in the journal directory. You must issue the `journalLatencyTest` (page 396) command against the *admin database* in the form:

```
{ journalLatencyTest: 1 }
```

The value (i.e. `1` above), does not affect the operation of the command.

Note: `journalLatencyTest` (page 396) is an internal command that is not enabled by default. `journalLatencyTest` (page 396) must be enabled by using *`--setParameter enableTestCommands=1`* on the `mongod` (page 583) command line. `journalLatencyTest` (page 396) cannot be enabled during run-time.

---

captrunc

Definition

captrunc

captrunc (page 397) is a command that truncates capped collections for diagnostic and testing purposes and is not part of the stable client facing API. The command takes the following form:

```
{ captrunc: "<collection>", n: <integer>, inc: <true|false> }.
```

captrunc (page 397) has the following fields:

field string captrunc The name of the collection to truncate.

field integer n The number of documents to remove from the collection.

field boolean inc Specifies whether to truncate the nth document.

Note: captrunc (page 397) is an internal command that is not enabled by default. captrunc (page 397) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 583) command line. captrunc (page 397) cannot be enabled during run-time.

Examples The following command truncates 10 older documents from the collection records:

```
db.runCommand({captrunc: "records" , n: 10})
```

The following command truncates 100 documents and the 101st document:

```
db.runCommand({captrunc: "records", n: 100, inc: true})
```

configureFailPoint

Definition

configureFailPoint

Configures a failure point that you can turn on and off while MongoDB runs. configureFailPoint (page 397) is an internal command for testing purposes that takes the following form:

```
{configureFailPoint: "<failure_point>", mode: <behavior> }
```

You must issue configureFailPoint (page 397) against the *admin database*. configureFailPoint (page 397) has the following fields:

field string configureFailPoint The name of the failure point.

field document,string mode Controls the behavior of a failure point. The possible values are alwaysOn, off, or a document in the form of {times: n} that specifies the number of times the failure point remains on before it deactivates. The maximum value for the number is a 32-bit signed integer.

field document data Passes in additional data for use in configuring the fail point. For example, to imitate a slow connection pass in a document that contains a delay time.

Note: configureFailPoint (page 397) is an internal command that is not enabled by default. configureFailPoint (page 397) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 583) command line. configureFailPoint (page 397) cannot be enabled during run-time.

Example

```
db.adminCommand( { configureFailPoint: "blocking_thread", mode: {times: 21} } )
```

emptycapped

emptycapped

The `emptycapped` command removes all documents from a capped collection. Use the following syntax:

```
{ emptycapped: "events" }
```

This command removes all records from the capped collection named `events`.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

Note: `emptycapped` (page 398) is an internal command that is not enabled by default. `emptycapped` (page 398) must be enabled by using *`--setParameter enableTestCommands=1`* on the `mongod` (page 583) command line. `emptycapped` (page 398) cannot be enabled during run-time.

---

forceerror

forceerror

The `forceerror` (page 398) command is for testing purposes only. Use `forceerror` (page 398) to force a user assertion exception. This command always returns an `ok` value of 0.

godinsert

godinsert

`godinsert` (page 398) is an internal command for testing purposes only.

Note: This command obtains a write lock on the affected database and will block other operations until it has completed.

Note: `godinsert` (page 398) is an internal command that is not enabled by default. `godinsert` (page 398) must be enabled by using *`--setParameter enableTestCommands=1`* on the `mongod` (page 583) command line. `godinsert` (page 398) cannot be enabled during run-time.

---

replSetTest

replSetTest

`replSetTest` (page 398) is internal diagnostic command used for regression tests that supports replica set functionality.

Note: `replSetTest` (page 398) is an internal command that is not enabled by default. `replSetTest` (page 398) must be enabled by using *`--setParameter enableTestCommands=1`* on the `mongod` (page 583) command line. `replSetTest` (page 398) cannot be enabled during run-time.

---

skewClockCommand

_skewClockCommand

_skewClockCommand (page 399) is an internal command. Do not call directly.

Note: _skewClockCommand (page 399) is an internal command that is not enabled by default. _skewClockCommand (page 399) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 583) command line. _skewClockCommand (page 399) cannot be enabled during run-time.

---

sleep

Definition

sleep

Forces the database to block all operations. This is an internal command for testing purposes.

The sleep (page 399) command takes the following prototype form:

```
{ sleep: 1, w: <true|false>, secs: <seconds> }
```

The sleep (page 399) command has the following fields:

field boolean w If true, obtains a global write lock. Otherwise obtains a read lock.

field integer secs The number of seconds to sleep.

Behavior The command places the mongod (page 583) instance in a *write lock* state for 100 seconds. Without arguments, sleep (page 399) causes a "read lock" for 100 seconds.

Warning: sleep (page 399) claims the lock specified in the w argument and blocks *all* operations on the mongod (page 583) instance for the specified amount of time.

Note: sleep (page 399) is an internal command that is not enabled by default. sleep (page 399) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 583) command line. sleep (page 399) cannot be enabled during run-time.

---

testDistLockWithSkew

_testDistLockWithSkew

_testDistLockWithSkew (page 399) is an internal command. Do not call directly.

Note: _testDistLockWithSkew (page 399) is an internal command that is not enabled by default. _testDistLockWithSkew (page 399) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 583) command line. _testDistLockWithSkew (page 399) cannot be enabled during run-time.

---

testDistLockWithSyncCluster

_testDistLockWithSyncCluster

_testDistLockWithSyncCluster (page 399) is an internal command. Do not call directly.

Note: `_testDistLockWithSyncCluster` (page 399) is an internal command that is not enabled by default. `_testDistLockWithSyncCluster` (page 399) must be enabled by using `--setParameter enableTestCommands=1` on the `mongod` (page 583) command line. `_testDistLockWithSyncCluster` (page 399) cannot be enabled during run-time.

2.2.5 Auditing Commands

System Events Auditing Commands

Name	Description
`logApplicationMessage` (page 400)	Posts a custom message to the audit log.

logApplicationMessage

`logApplicationMessage`

Note: Available only in MongoDB Enterprise[15].

The `logApplicationMessage` (page 400) command allows users to post a custom message to the `audit` log. If running with authorization, users must have `clusterAdmin` role, or roles that inherit from `clusterAdmin`, to run the command.

The `logApplicationMessage` (page 400) has the following syntax:

```
{ logApplicationMessage: <string> }
```

MongoDB associates these custom messages with the *audit operation* `applicationMessage`, and the messages are subject to any *filtering*.

2.3 Operators

Query and Projection Operators **(page 400)** Query operators provide ways to locate data within the database and projection operators modify how data is presented.

Update Operators **(page 451)** Update operators are operators that enable you to modify the data in your database or add additional data.

Aggregation Pipeline Operators **(page 482)** Aggregation pipeline operations have a collection of operators available to define and manipulate documents in pipeline stages.

Query Modifiers **(page 555)** Query modifiers determine the way that queries will be executed.

2.3.1 Query and Projection Operators

Query Selectors

Comparison

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

Comparison Query Operators For comparison of different BSON type values, see the *specified BSON comparison order*.

Name	Description
$gt (page 401)	Matches values that are greater than the value specified in the query.
$gte (page 401)	Matches values that are greater than or equal to the value specified in the query.
$in (page 402)	Matches any of the values that exist in an array specified in the query.
$lt (page 403)	Matches values that are less than the value specified in the query.
$lte (page 403)	Matches values that are less than or equal to the value specified in the query.
$ne (page 404)	Matches all values that are not equal to the value specified in the query.
$nin (page 404)	Matches values that **do not** exist in an array specified to the query.

$gt

$gt

Syntax: `{field: {$gt: value} }`

$gt (page 401) selects those documents where the value of the `field` is greater than (i.e. >) the specified `value`.

For comparison of different BSON type values, see the *specified BSON comparison order*.

Consider the following example:

```
db.inventory.find( { qty: { $gt: 20 } } )
```

This query will select all documents in the `inventory` collection where the `qty` field value is greater than `20`.

Consider the following example that uses the $gt (page 401) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gt: 2 } }, { $set: { price: 9.99 } } )
```

This update() (page 72) operation will set the value of the `price` field in the first document found containing the embedded document `carrier` whose `fee` field value is greater than `2`.

To set the value of the `price` field in *all* documents containing the embedded document `carrier` whose `fee` field value is greater than `2`, specify the `multi:true` option in the update() (page 72) method:

```
db.inventory.update( { "carrier.fee": { $gt: 2 } },
                     { $set: { price: 9.99 },
                     { multi: true }
                   )
```

See also:

find() (page 36), update() (page 72), $set (page 459).

$gte

$gte

Syntax: `{field: {$gte: value} }`

$gte (page 401) selects the documents where the value of the `field` is greater than or equal to (i.e. >=) a specified value (e.g. `value`.)

For comparison of different BSON type values, see the *specified BSON comparison order*.

Consider the following example:

```
db.inventory.find( { qty: { $gte: 20 } } )
```

This query would select all documents in `inventory` where the `qty` field value is greater than or equal to `20`.

Consider the following example which uses the `$gte` (page 401) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gte: 2 } }, { $set: { price: 9.99 } } )
```

This `update()` (page 72) operation will set the value of the `price` field that contain the embedded document `carrier` whose `fee` field value is greater than or equal to `2`.

See also:

`find()` (page 36), `update()` (page 72), `$set` (page 459).

$in

$in

The `$in` (page 402) operator selects the documents where the value of a field equals any value in the specified array. To specify an `$in` (page 402) expression, use the following prototype:

For comparison of different BSON type values, see the *specified BSON comparison order*.

```
{ field: { $in: [<value1>, <value2>, ... <valueN> ] } }
```

If the `field` holds an array, then the `$in` (page 402) operator selects the documents whose `field` holds an array that contains at least one element that matches a value in the specified array (e.g. `<value1>`, `<value2>`, etc.)

Changed in version 2.6: MongoDB 2.6 removes the combinatorial limit for the `$in` (page 402) operator that exists for earlier versions[16] of the operator.

Examples

Use the `$in` Operator to Match Values Consider the following example:

```
db.inventory.find( { qty: { $in: [ 5, 15 ] } } )
```

This query selects all documents in the `inventory` collection where the `qty` field value is either `5` or `15`. Although you can express this query using the `$or` (page 408) operator, choose the `$in` (page 402) operator rather than the `$or` (page 408) operator when performing equality checks on the same field.

Use the `$in` Operator to Match Values in an Array The collection `inventory` contains documents that include the field `tags`, as in the following:

```
{ _id: 1, item: "abc", qty: 10, tags: [ "school", "clothing" ], sale: false }
```

Then, the following `update()` (page 72) operation will set the `sale` field value to `true` where the `tags` field holds an array with at least one element matching either `"appliances"` or `"school"`.

```
db.inventory.update(
                     { tags: { $in: ["appliances", "school"] } },
                     { $set: { sale:true } }
                   )
```

[16] http://docs.mongodb.org/v2.4/reference/operator/query/in

Use the `$in` Operator with a Regular Expression The `$in` (page 402) operator can specify matching values using regular expressions of the form `http://docs.mongodb.org/manual/pattern/`. You *cannot* use `$regex` (page 414) operator expressions inside an `$in` (page 402).

Consider the following example:

```
db.inventory.find( { tags: { $in: [ /^be/, /^st/ ] } } )
```

This query selects all documents in the `inventory` collection where the `tags` field holds an array that contains at least one element that starts with either `be` or `st`.

See also:

`find()` (page 36), `update()` (page 72), `$or` (page 408), `$set` (page 459).

$lt

`$lt`

Syntax: `{field: {$lt: value} }`

`$lt` (page 403) selects the documents where the value of the `field` is less than (i.e. <) the specified `value`.

For comparison of different BSON type values, see the *specified BSON comparison order*.

Consider the following example:

```
db.inventory.find( { qty: { $lt: 20 } } )
```

This query will select all documents in the `inventory` collection where the `qty` field value is less than `20`.

Consider the following example which uses the `$lt` (page 403) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lt: 20 } }, { $set: { price: 9.99 } } )
```

This `update()` (page 72) operation will set the `price` field value in the documents that contain the embedded document `carrier` whose `fee` field value is less than `20`.

See also:

`find()` (page 36), `update()` (page 72), `$set` (page 459).

$lte

`$lte`

Syntax: `{ field: { $lte: value} }`

`$lte` (page 403) selects the documents where the value of the `field` is less than or equal to (i.e. <=) the specified `value`.

For comparison of different BSON type values, see the *specified BSON comparison order*.

Consider the following example:

```
db.inventory.find( { qty: { $lte: 20 } } )
```

This query will select all documents in the `inventory` collection where the `qty` field value is less than or equal to `20`.

Consider the following example which uses the `$lt` (page 403) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lte: 5 } }, { $set: { price: 9.99 } } )
```

This update() (page 72) operation will set the price field value in the documents that contain the embedded document carrier whose fee field value is less than or equal to 5.

See also:

find() (page 36), update() (page 72), $set (page 459).

$ne

$ne

Syntax: {field: {$ne: value} }

$ne (page 404) selects the documents where the value of the field is not equal (i.e. !=) to the specified value. This includes documents that do not contain the field.

For comparison of different BSON type values, see the *specified BSON comparison order*.

Consider the following example:

```
db.inventory.find( { qty: { $ne: 20 } } )
```

This query will select all documents in the inventory collection where the qty field value does not equal 20, including those documents that do not contain the qty field.

Consider the following example which uses the $ne (page 404) operator with a field in an embedded document:

```
db.inventory.update( { "carrier.state": { $ne: "NY" } }, { $set: { qty: 20 } } )
```

This update() (page 72) operation will set the qty field value in the documents that contain the embedded document carrier whose state field value does not equal "NY", or where the state field or the carrier embedded document do not exist.

See also:

find() (page 36), update() (page 72), $set (page 459).

$nin

$nin

Syntax: { field: { $nin: [<value1>, <value2> ... <valueN>]} }

$nin (page 404) selects the documents where:

- the field value is not in the specified array **or**
- the field does not exist.

For comparison of different BSON type values, see the *specified BSON comparison order*.

Consider the following query:

```
db.inventory.find( { qty: { $nin: [ 5, 15 ] } } )
```

This query will select all documents in the inventory collection where the qty field value does **not** equal 5 nor 15. The selected documents will include those documents that do *not* contain the qty field.

If the field holds an array, then the $nin (page 404) operator selects the documents whose field holds an array with **no** element equal to a value in the specified array (e.g. <value1>, <value2>, etc.).

Consider the following query:

```
db.inventory.update( { tags: { $nin: [ "appliances", "school" ] } }, { $set: { sale: false } } )
```

This update() (page 72) operation will set the sale field value in the inventory collection where the tags field holds an array with **no** elements matching an element in the array ["appliances", "school"] or where a document does not contain the tags field.

See also:

find() (page 36), update() (page 72), $set (page 459).

Logical

Logical Query Operators

Name	Description
$and (page 405)	Joins query clauses with a logical AND returns all documents that match the condition clauses.
$nor (page 406)	Joins query clauses with a logical NOR returns all documents that fail to match both cl
$not (page 407)	Inverts the effect of a query expression and returns documents that do *not* match the q expression.
$or (page 408)	Joins query clauses with a logical OR returns all documents that match the conditions clause.

$and

$and

New in version 2.0.

Syntax: { $and: [{ <expression1> }, { <expression2> } , ... , { <expressionN> }] }

$and (page 405) performs a logical AND operation on an array of *two or more* expressions (e.g. <expression1>, <expression2>, etc.) and selects the documents that satisfy *all* the expressions in the array. The $and (page 405) operator uses *short-circuit evaluation*. If the first expression (e.g. <expression1>) evaluates to false, MongoDB will not evaluate the remaining expressions.

Note: MongoDB provides an implicit AND operation when specifying a comma separated list of expressions. Using an explicit AND with the $and (page 405) operator is necessary when the same field or operator has to be specified in multiple expressions.

Examples

AND Queries With Multiple Expressions Specifying the Same Field Consider the following example:

```
db.inventory.find( { $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] } )
```

This query will select all documents in the inventory collection where:

- the price field value is not equal to 1.99 **and**
- the price field exists.

This query can be also be constructed with an implicit AND operation by combining the operator expressions for the price field. For example, this query can be written as:

```
db.inventory.find( { price: { $ne: 1.99, $exists: true } } )
```

AND Queries With Multiple Expressions Specifying the Same Operator Consider the following example:

```
db.inventory.find( {
    $and : [
        { $or : [ { price : 0.99 }, { price : 1.99 } ] },
        { $or : [ { sale : true }, { qty : { $lt : 20 } } ] }
    ]
} )
```

This query will return all select all documents where:

- the `price` field value equals `0.99` or `1.99`, **and**
- the `sale` field value is equal to `true` **or** the `qty` field value is less than `20`.

This query cannot be constructed using an implicit `AND` operation, because it uses the `$or` (page 408) operator more than once.

See also:

`find()` (page 36), `update()` (page 72), `$ne` (page 404), `$exists` (page 409), `$set` (page 459).

$nor

$nor

`$nor` (page 406) performs a logical `NOR` operation on an array of one or more query expression and selects the documents that **fail** all the query expressions in the array. The `$nor` (page 406) has the following syntax:

```
{ $nor: [ { <expression1> }, { <expression2> }, ...  { <expressionN> } ] }
```

See also:

`find()` (page 36), `update()` (page 72), `$or` (page 408), `$set` (page 459), and `$exists` (page 409).

Examples

$nor Query with Two Expressions Consider the following query which uses only the `$nor` (page 406) operator:

```
db.inventory.find( { $nor: [ { price: 1.99 }, { sale: true } ]  } )
```

This query will return all documents that:

- contain the `price` field whose value is *not* equal to `1.99` and contain the `sale` field whose value *is not* equal to `true` **or**
- contain the `price` field whose value is *not* equal to `1.99` *but* do *not* contain the `sale` field **or**
- do *not* contain the `price` field *but* contain the `sale` field whose value *is not* equal to `true` **or**
- do *not* contain the `price` field *and* do *not* contain the `sale` field

$nor and Additional Comparisons Consider the following query:

```
db.inventory.find( { $nor: [ { price: 1.99 }, { qty: { $lt: 20 } }, { sale: true } ] } )
```

This query will select all documents in the `inventory` collection where:

- the `price` field value does *not* equal `1.99` **and**
- the `qty` field value is *not* less than `20` **and**
- the `sale` field value is *not* equal to `true`

including those documents that do not contain these field(s).

The exception in returning documents that do not contain the field in the $nor (page 406) expression is when the $nor (page 406) operator is used with the $exists (page 409) operator.

$nor and $exists Compare that with the following query which uses the $nor (page 406) operator with the $exists (page 409) operator:

```
db.inventory.find( { $nor: [ { price: 1.99 }, { price: { $exists: false } },
                             { sale: true }, { sale: { $exists: false } } ] } )
```

This query will return all documents that:

- contain the price field whose value is *not* equal to 1.99 and contain the sale field whose value *is not* equal to true

$not

$not

Syntax: { field: { $not: { <operator-expression> } } }

$not (page 407) performs a logical NOT operation on the specified <operator-expression> and selects the documents that do *not* match the <operator-expression>. This includes documents that do not contain the field.

Consider the following query:

```
db.inventory.find( { price: { $not: { $gt: 1.99 } } } )
```

This query will select all documents in the inventory collection where:

- the price field value is less than or equal to 1.99 **or**
- the price field does not exist

{ $not: { $gt: 1.99 } } is different from the $lte (page 403) operator. { $lte: 1.99 } returns *only* the documents where price field exists and its value is less than or equal to 1.99.

Remember that the $not (page 407) operator only affects *other operators* and cannot check fields and documents independently. So, use the $not (page 407) operator for logical disjunctions and the $ne (page 404) operator to test the contents of fields directly.

Consider the following behaviors when using the $not (page 407) operator:

- The operation of the $not (page 407) operator is consistent with the behavior of other operators but may yield unexpected results with some data types like arrays.
- The $not (page 407) operator does **not** support operations with the $regex (page 414) operator. Instead use http://docs.mongodb.org/manual// or in your driver interfaces, use your language's regular expression capability to create regular expression objects.

  Consider the following example which uses the pattern match expression http://docs.mongodb.org/manual//:

  ```
  db.inventory.find( { item: { $not: /^p.*/ } } )
  ```

  The query will select all documents in the inventory collection where the item field value does *not* start with the letter p.

  If you are using Python, you can write the above query with the PyMongo driver and Python's python:re.compile() method to compile a regular expression, as follows:

```
import re
for noMatch in db.inventory.find( { "item": { "$not": re.compile("^p.*") } } ):
    print noMatch
```

See also:

`find()` (page 36), `update()` (page 72), `$set` (page 459), `$gt` (page 401), `$regex` (page 414), PyMongo[17], *driver*.

$or

$or

The `$or` (page 408) operator performs a logical `OR` operation on an array of *two or more* `<expressions>` and selects the documents that satisfy *at least* one of the `<expressions>`. The `$or` (page 408) has the following syntax:

```
{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }
```

Consider the following example:

```
db.inventory.find( { $or: [ { quantity: { $lt: 20 } }, { price: 10 } ] } )
```

This query will select all documents in the `inventory` collection where either the `quantity` field value is less than `20` **or** the `price` field value equals `10`.

Behaviors

`$or` Clauses and Indexes When evaluating the clauses in the `$or` (page 408) expression, MongoDB either performs a collection scan or, if all the clauses are supported by indexes, MongoDB performs index scans. That is, for MongoDB to use indexes to evaluate an `$or` (page 408) expression, all the clauses in the `$or` (page 408) expression must be supported by indexes. Otherwise, MongoDB will perform a collection scan.

When using indexes with `$or` (page 408) queries, each clause of an `$or` (page 408) can use its own index. Consider the following query:

```
db.inventory.find( { $or: [ { quantity: { $lt: 20 } }, { price: 10 } ] } )
```

To support this query, rather than a compound index, you would create one index on `quantity` and another index on `price`:

```
db.inventory.ensureIndex( { quantity: 1 } )
db.inventory.ensureIndex( { price: 1 } )
```

MongoDB can use all but the `geoHaystack` index to support `$or` (page 408) clauses.

`$or` and `text` Queries Changed in version 2.6.

If `$or` (page 408) includes a `$text` (page 417) query, all clauses in the `$or` (page 408) array must be supported by an index. This is because a `$text` (page 417) query *must* use an index, and `$or` (page 408) can only use indexes if all its clauses are supported by indexes. If the `$text` (page 417) query cannot use an index, the query will return an error.

[17] http://api.mongodb.org/pythoncurrent

$or and GeoSpatial Queries Changed in version 2.6.

$or supports *geospatial clauses* (page 423) with the following exception for the near clause (near clause includes $nearSphere (page 428) and $near (page 429)). $or cannot contain a near clause with any other clause.

$or and Sort Operations Changed in version 2.6.

When executing $or (page 408) queries with a sort() (page 95), MongoDB can now use indexes that support the $or (page 408) clauses. Previous versions did not use the indexes.

$or versus $in When using $or (page 408) with <expressions> that are equality checks for the value of the same field, use the $in (page 402) operator instead of the $or (page 408) operator.

For example, to select all documents in the inventory collection where the quantity field value equals either 20 *or* 50, use the $in (page 402) operator:

```
db.inventory.find ( { quantity: { $in: [20, 50] } } )
```

Nested $or Clauses You may nest $or (page 408) operations.

See also:

$and (page 405), find() (page 36), sort() (page 95), $in (page 402)

Element

Element Query Operators

Name	Description
$exists (page 409)	Matches documents that have the specified field.
$type (page 411)	Selects documents if a field is of the specified type.

$exists

Definition

$exists

Syntax: { field: { $exists: <boolean> } }

When <boolean> is true, $exists (page 409) matches the documents that contain the field, including documents where the field value is null. If <boolean> is false, the query returns only the documents that do not contain the field.

MongoDB *$exists* does **not** correspond to SQL operator exists. For SQL exists, refer to the $in (page 402) operator.

See also:

$nin (page 404), $in (page 402), and *faq-developers-query-for-nulls*.

Examples

Exists and Not Equal To Consider the following example:

```
db.inventory.find( { qty: { $exists: true, $nin: [ 5, 15 ] } } )
```

This query will select all documents in the `inventory` collection where the `qty` field exists *and* its value does not equal `5` or `15`.

Null Values Given a collection named `records` with the following documents:

```
{ a: 5, b: 5, c: null }
{ a: 3, b: null, c: 8 }
{ a: null, b: 3, c: 9 }
{ a: 1, b: 2, c: 3 }
{ a: 2, c: 5 }
{ a: 3, b: 2 }
{ a: 4 }
{ b: 2, c: 4 }
{ b: 2 }
{ c: 6 }
```

Consider the output of the following queries:

Query:

```
db.records.find( { a: { $exists: true } } )
```

Result:

```
{ a: 5, b: 5, c: null }
{ a: 3, b: null, c: 8 }
{ a: null, b: 3, c: 9 }
{ a: 1, b: 2, c: 3 }
{ a: 2, c: 5 }
{ a: 3, b: 2 }
{ a: 4 }
```

Query:

```
db.records.find( { b: { $exists: false } } )
```

Result:

```
{ a: 2, c: 5 }
{ a: 4 }
{ c: 6 }
```

Query:

```
db.records.find( { c: { $exists: false } } )
```

Result:

```
{ a: 3, b: 2 }
{ a: 4 }
{ b: 2 }
```

$type

Definition

$type

Syntax: `{ field: { $type: <BSON type> } }`

`$type` (page 411) selects the documents where the *value* of the `field` is the specified numeric *BSON* type. This is useful when dealing with highly unstructured data where data types are not predictable.

> **Warning:** Data models that associate a field name with different data types within a collection are *strongly* discouraged.
> Without internal consistency complicates application code, and can lead to unnecessary complexity for application developers.

Behavior

Available Types Refer to the following table for the available *BSON* types and their corresponding numbers.

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

Minimum and Maximum Values `MinKey` and `MaxKey` compare less than and greater than all other possible *BSON* element values, respectively, and exist primarily for internal use.

To query if a field value is a `MinKey`, you must use `$type` (page 411) with `-1` as in the following example:

```
db.collection.find( { field: { $type: -1 } } );
```

Arrays When applied to arrays, `$type` (page 411) matches any **inner** element that is of the specified type. Without *projection* this means that the entire array will match if **any** element has the right type. With projection, the results will include just those elements of the requested type.

Examples

Querying by Data Type Consider the following query:

```
db.inventory.find( { tags: { $type : 2 } } );
```

This will list all documents containing a `tags` field that is either a string or an array holding at least one string. If you only want to list documents where `tags` is an array, you could use `$where` (page 421):

```
db.inventory.find( { $where : "Array.isArray(this.tags)" } );
```

Queries that use `$where` (page 421) requires a complete collection scan and uses *Server-side JavaScript*.

`MinKey` and `MaxKey` The following operation sequence demonstrates both type comparison *and* the special `MinKey` and `MaxKey` values:

```
> db.test.insert( [ { x : 3 },
                    { x : 2.9 },
                    { x : new Date() },
                    { x : true },
                    { x : MaxKey },
                    { x : MinKey } ] );

> db.test.find().sort( { x : 1 } );
{ "_id" : ObjectId("4b04094b7c65b846e2090112"), "x" : { $minKey : 1 } }
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Jul 25 2012 18:42:03 GMT-0500 (EST)" }
{ "_id" : ObjectId("4b0409487c65b846e2090111"), "x" : { $maxKey : 1 } }
```

Minimum Shard Key To query for the minimum value of a *shard key* of a *sharded cluster*, use the following operation when connected to the `mongos` (page 601):

```
use config
db.chunks.find( { "min.shardKey": { $type: -1 } } )
```

Additional Information `find()` (page 36), `insert()` (page 55), `$where` (page 421), *BSON*, *shard key*, *sharded cluster* .

Evaluation

Evaluation Query Operators

Name	Description
`$mod` (page 412)	Performs a modulo operation on the value of a field and selects documents with a result.
`$regex` (page 414)	Selects documents where values match a specified regular expression.
`$text` (page 417)	Performs text search.
`$where` (page 421)	Matches documents that satisfy a JavaScript expression.

$mod

`$mod`

Select documents where the value of a field divided by a divisor has the specified remainder (i.e. perform a modulo operation to select documents). To specify a `$mod` (page 412) expression, use the following syntax:

```
{ field: { $mod: [ divisor, remainder ] } }
```

Changed in version 2.6: The $mod (page 412) operator errors when passed an array with fewer or more elements. In previous versions, if passed an array with one element, the $mod (page 412) operator uses 0 as the remainder value, and if passed an array with more than two elements, the $mod (page 412) ignores all but the first two elements. Previous versions do return an error when passed an empty array. See *Not Enough Elements Error* (page 413) and *Too Many Elements Error* (page 414) for details.

Examples

Use $mod to Select Documents Consider a collection inventory with the following documents:

```
{ "_id" : 1, "item" : "abc123", "qty" : 0 }
{ "_id" : 2, "item" : "xyz123", "qty" : 5 }
{ "_id" : 3, "item" : "ijk123", "qty" : 12 }
```

Then, the following query selects those documents in the inventory collection where value of the qty field modulo 4 equals 0:

```
db.inventory.find( { qty: { $mod: [ 4, 0 ] } } )
```

The query returns the following documents:

```
{ "_id" : 1, "item" : "abc123", "qty" : 0 }
{ "_id" : 3, "item" : "ijk123", "qty" : 12 }
```

Not Enough Elements Error The $mod (page 412) operator errors when passed an array with fewer than two elements.

Array with Single Element The following operation incorrectly passes the $mod (page 412) operator an array that contains a single element:

```
db.inventory.find( { qty: { $mod: [ 4 ] } } )
```

The statement results in the following error:

```
error: {
     "$err" : "bad query: BadValue malformed mod, not enough elements",
     "code" : 16810
}
```

Changed in version 2.6: In previous versions, if passed an array with one element, the $mod (page 412) operator uses the specified element as the divisor and 0 as the remainder value.

Empty Array The following operation incorrectly passes the $mod (page 412) operator an empty array:

```
db.inventory.find( { qty: { $mod: [ ] } } )
```

The statement results in the following error:

```
error: {
     "$err" : "bad query: BadValue malformed mod, not enough elements",
     "code" : 16810
}
```

Changed in version 2.6: Previous versions returned the following error:

```
error: { "$err" : "mod can't be 0", "code" : 10073 }
```

Too Many Elements Error The $mod (page 412) operator errors when passed an array with more than two elements.

For example, the following operation attempts to use the $mod (page 412) operator with an array that contains four elements:

```
error: {
     "$err" : "bad query: BadValue malformed mod, too many elements",
     "code" : 16810
}
```

Changed in version 2.6: In previous versions, if passed an array with more than two elements, the $mod (page 412) ignores all but the first two elements.

$regex

Definition

$regex

Provides regular expression capabilities for pattern matching *strings* in queries. MongoDB uses Perl compatible regular expressions (i.e. "PCRE") version 8.30 with UTF-8 support.

To use $regex (page 414), use one of the following syntax:

```
{ <field>: { $regex: /pattern/, $options: '<options>' } }
{ <field>: { $regex: 'pattern', $options: '<options>' } }
{ <field>: { $regex: /pattern/<options> } }
```

In MongoDB, you can also use regular expression objects (i.e. `http://docs.mongodb.org/manual/pattern/`) to specify regular expressions:

```
{ <field>: /pattern/<options> }
```

For restrictions on particular syntax use, see *$regex vs. /pattern/ Syntax* (page 415).

$options

The following <options> are available for use with regular expression.

Option	Description	Syntax Restrictions
`i`	Case insensitivity to match upper and lower cases. For an example, see *Perform Case-Insensitive Regular Expression Match* (page 416).	
`m`	For patterns that include anchors (i.e. `^` for the start, `$` for the end), match at the beginning or end of each line for strings with multiline values. Without this option, these anchors match at beginning or end of the string. For an example, see *Multiline Match for Lines Starting with Specified Pattern* (page 416). If the pattern contains no anchors or if the string value has no newline characters (e.g. `\n`), the `m` option has no effect.	
`x`	"Extended" capability to ignore all white space characters in the `$regex` (page 414) pattern unless escaped or included in a character class. Additionally, it ignores characters in-between and including an un-escaped hash/pound (`#`) character and the next new line, so that you may include comments in complicated patterns. This only applies to data characters; white space characters may never appear within special character sequences in a pattern. The `x` option does not affect the handling of the VT character (i.e. code 11).	Requires `$regex` with `$options` syntax
`s`	Allows the dot character (i.e. `.`) to match all characters *including* newline characters. For an example, see *Use the . Dot Character to Match New Line* (page 417).	Requires `$regex` with `$options` syntax

Behavior

$regex vs. /pattern/ Syntax

`$in` Expressions To include a regular expression in an `$in` query expression, you can only use JavaScript regular expression objects (i.e. `http://docs.mongodb.org/manual/pattern/`). For example:

```
{ name: { $in: [ /^acme/i, /^ack/ ] } }
```

You *cannot* use `$regex` (page 414) operator expressions inside an `$in` (page 402).

Implicit `AND` Conditions for the Field To include a regular expression in a comma-separated list of query conditions for the field, use the `$regex` (page 414) operator. For example:

```
{ name: { $regex: /acme.*corp/i, $nin: [ 'acmeblahcorp' ] } }
{ name: { $regex: /acme.*corp/, $options: 'i', $nin: [ 'acmeblahcorp' ] } }
{ name: { $regex: 'acme.*corp', $options: 'i', $nin: [ 'acmeblahcorp' ] } }
```

`x` and `s` Options To use either the `x` option or `s` options, you must use the `$regex` (page 414) operator expression *with* the `$options` (page 414) operator. For example, to specify the `i` and the `s` options, you must use `$options` (page 414) for both:

```
{ name: { $regex: /acme.*corp/, $options: "si" } }
{ name: { $regex: 'acme.*corp', $options: "si" } }
```

PCRE vs JavaScript To use PCRE supported features in the regex pattern that are unsupported in JavaScript, you must use the `$regex` (page 414) operator expression with the pattern as a string. For example, to use `(?i)` in the

pattern to turn case-insensitivity on for the remaining pattern and `(?-i)` to turn case-sensitivity on for the remaining pattern, you must use the `$regex` (page 414) operator with the pattern as a string:

```
{ name: { $regex: '(?i)a(?-i)cme' } }
```

Index Use If an index exists for the field, then MongoDB matches the regular expression against the values in the index, which can be faster than a collection scan. Further optimization can occur if the regular expression is a "prefix expression", which means that all potential matches start with the same string. This allows MongoDB to construct a "range" from that prefix and only match against those values from the index that fall within that range.

A regular expression is a "prefix expression" if it starts with a caret (`^`) or a left anchor (`\A`), followed by a string of simple symbols. For example, the regex `http://docs.mongodb.org/manual/^abc.*/` will be optimized by matching only against the values from the index that start with `abc`.

Additionally, while `http://docs.mongodb.org/manual/^a/`, `http://docs.mongodb.org/manual/^a.*/`, and `http://docs.mongodb.org/manual/^a.*$/` match equivalent strings, they have different performance characteristics. All of these expressions use an index if an appropriate index exists; however, `http://docs.mongodb.org/manual/^a.*/`, and `http://docs.mongodb.org/manual/^a.*$/` are slower. `http://docs.mongodb.org/manual/^a/` can stop scanning after matching the prefix.

Examples The following examples use a collection `products` with the following documents:

```
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }
{ "_id" : 101, "sku" : "abc789", "description" : "First line\nSecond line" }
{ "_id" : 102, "sku" : "xyz456", "description" : "Many spaces before     line" }
{ "_id" : 103, "sku" : "xyz789", "description" : "Multiple\nline description" }
```

Perform Case-Insensitive Regular Expression Match The following example uses the `i` option perform a *case-insensitive* match for documents with `sku` value that starts with `ABC`.

```
db.products.find( { sku: { $regex: /^ABC/i } } )
```

The query matches the following documents:

```
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }
{ "_id" : 101, "sku" : "abc789", "description" : "First line\nSecond line" }
```

Multiline Match for Lines Starting with Specified Pattern The following example uses the `m` option to match lines starting with the letter `S` for multiline strings:

```
db.products.find( { description: { $regex: /^S/, $options: 'm' } } )
```

The query matches the following documents:

```
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }
{ "_id" : 101, "sku" : "abc789", "description" : "First line\nSecond line" }
```

Without the `m` option, the query would match just the following document:

```
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }
```

If the `$regex` (page 414) pattern does not contain an anchor, the pattern matches against the string as a whole, as in the following example:

```
db.products.find( { description: { $regex: /S/ } } )
```

Then, the $regex (page 414) would match both documents:

```
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }
{ "_id" : 101, "sku" : "abc789", "description" : "First line\nSecond line" }
```

Use the . Dot Character to Match New Line The following example uses the s option to allow the dot character (i.e. .) to match all characters *including* new line as well as the i option to perform a case-insensitive match:

```
db.products.find( { description: { $regex: /m.*line/, $options: 'si' } } )
```

The query matches the following documents:

```
{ "_id" : 102, "sku" : "xyz456", "description" : "Many spaces before     line" }
{ "_id" : 103, "sku" : "xyz789", "description" : "Multiple\nline description" }
```

Without the s option, the query would have matched only the following document:

```
{ "_id" : 102, "sku" : "xyz456", "description" : "Many spaces before     line" }
```

Ignore White Spaces in Pattern The following example uses the x option ignore white spaces and the comments, denoted by the # and ending with the \n in the matching pattern:

```
var pattern = "abc #category code\n123 #item number"
db.products.find( { sku: { $regex: pattern, $options: "x" } } )
```

The query matches the following document:

```
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }
```

$text

$text

New in version 2.6.

$text (page 417) performs a text search on the content of the fields indexed with a text index. A $text (page 417) expression has the following syntax:

```
{ $text: { $search: <string>, $language: <string> } }
```

The $text (page 417) operator accepts a text query document with the following fields:

field string $search A string of terms that MongoDB parses and uses to query the text index. MongoDB performs a logical OR search of the terms unless specified as a phrase. See *Behavior* (page 417) for more information on the field.

field string $language The language that determines the list of stop words for the search and the rules for the stemmer and tokenizer. If not specified, the search uses the default language of the index. For supported languages, see *text-search-languages*.

If you specify a language value of "none", then the text search uses simple tokenization with no list of stop words and no stemming.

The $text (page 417) operator, by default, does *not* return results sorted in terms of the results' score. For more information, see the *Text Score* (page 419) documentation.

Behavior

Restrictions

- A query can specify, at most, one $text (page 417) expression.
- The $text (page 417) query can not appear in $nor (page 406) expressions.
- To use a $text (page 417) query in an $or (page 408) expression, all clauses in the $or (page 408) array must be indexed.
- You cannot use hint() (page 87) if the query includes a $text (page 417) query expression.
- You cannot specify $natural (page 563) sort order if the query includes a $text (page 417) expression.
- You cannot combine the $text (page 417) expression, which requires a special *text index*, with a query operator that requires a different type of special index. For example you cannot combine $text (page 417) expression with the $near (page 429) operator.

$search Field In the $search field, specify a string of words that the text operator parses and uses to query the text index. The text operator treats most punctuation in the string as delimiters, except a hyphen - that negates term or an escaped double quotes \" that specifies a phrase.

Phrases To match on a phrase, as opposed to individual terms, enclose the phrase in escaped double quotes (\"), as in:

```
"\"ssl certificate\""
```

If the $search string includes a phrase and individual terms, text search will only match the documents that include the phrase. More specifically, the search performs a logical AND of the phrase with the individual terms in the search string.

For example, passed a $search string:

```
"\"ssl certificate\" authority key"
```

The $text (page 417) operator searches for the phrase "ssl certificate" **and** ("authority" **or** "key" **or** "ssl" **or** "certificate").

Negations Prefixing a word with a hyphen sign (-) negates a word:

- The negated word excludes documents that contain the negated word from the result set.
- When passed a search string that only contains negated words, text search will not match any documents.
- A hyphenated word, such as pre-market, is not a negation. The $text (page 417) operator treats the hyphen as a delimiter.

The $text (page 417) operator adds all negations to the query with the logical AND operator.

Match Operation The $text (page 417) operator ignores language-specific stop words, such as the and and in English.

The $text (page 417) operator matches on the complete *stemmed* word. So if a document field contains the word blueberry, a search on the term blue will not match. However, blueberry or blueberries will match.

For non-diacritics, text search is case insensitive; i.e. case insensitive for [A-z].

Text Score The $text (page 417) operator assigns a score to each document that contains the search term in the indexed fields. The score represents the relevance of a document to a given text search query. The score can be part of a sort() (page 95) method specification as well as part of the projection expression. The { $meta: "textScore" } expression provides information on the processing of the $text (page 417) operation. See $meta (page 449) projection operator for details on accessing the score for projection or sort.

Examples The following examples assume a collection articles that has a text index on the field subject:

```
db.articles.ensureIndex( { subject: "text" } )
```

Search for a Single Word The following query searches for the term coffee:

```
db.articles.find( { $text: { $search: "coffee" } } )
```

This query returns documents that contain the term coffee in the indexed subject field.

Match Any of the Search Terms If the search string is a space-delimited string, $text (page 417) operator performs a logical OR search on each term and returns documents that contains any of the terms.

The following query searches specifies a $search string of three terms delimited by space, "bake coffee cake":

```
db.articles.find( { $text: { $search: "bake coffee cake" } } )
```

This query returns documents that contain either bake **or** coffee **or** cake in the indexed subject field.

Search for a Phrase To match the exact phrase as a single term, escape the quotes.

The following query searches for the phrase coffee cake:

```
db.articles.find( { $text: { $search: "\"coffee cake\"" } } )
```

This query returns documents that contain the phrase coffee cake.

See also:

Phrases (page 418)

Exclude Documents That Contain a Term A *negated* term is a term that is prefixed by a minus sign -. If you negate a term, the $text (page 417) operator will exclude the documents that contain those terms from the results.

The following example searches for documents that contain the words bake or coffee but do **not** contain the term cake:

```
db.articles.find( { $text: { $search: "bake coffee -cake" } } )
```

See also:

Negations (page 418)

Return the Text Search Score The following query searches for the term cake and returns the score assigned to each matching document:

```
db.articles.find(
   { $text: { $search: "cake" } },
   { score: { $meta: "textScore" } }
)
```

In the result set, the returned documents includes an *additional* field `score` that contains the document's score associated with the text search. [18]

See also:

Text Score (page 419)

Sort by Text Search Score To sort by the text score, include the **same** `$meta` (page 449) expression in **both** the projection document and the sort expression. [1] The following query searches for the term `cake` and sorts the results by the descending score:

```
db.articles.find(
   { $text: { $search: "cake" } },
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } )
```

In the result set, the returned documents includes an additional field `score` that contains the document's score associated with the text search.

See also:

Text Score (page 419)

Return Top 3 Matching Documents Use the `limit()` (page 88) method in conjunction with a `sort()` (page 95) to return the top three matching documents. The following query searches for the term `cake` and sorts the results by the descending score:

```
db.articles.find(
   { $text: { $search: "cake" } },
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } ).limit(3)
```

See also:

Text Score (page 419)

Text Search with Additional Query and Sort Expressions The following query searches for documents with status equal to `"A"` that contain the terms `coffee` or `cake` in the indexed field `subject` and specifies a sort order of ascending date, descending text score:

```
db.articles.find(
   { status: "A", $text: { $search: "coffee cake" } },
   { score: { $meta: "textScore" } }
).sort( { date: 1, score: { $meta: "textScore" } } )
```

Search a Different Language Use the optional `$language` field in the `$text` (page 417) expression to specify a language that determines the list of stop words and the rules for the stemmer and tokenizer for the search string.

[18] The behavior and requirements of the `$meta` (page 449) operator differs from that of the `$meta` (page 529) aggregation operator. See the `$meta` (page 529) aggregation operator for details.

If you specify a language value of "none", then the text search uses simple tokenization with no list of stop words and no stemming.

The following query specifies es for Spanish as the language that determines the tokenization, stemming, and stop words:

```
db.articles.find(
   { $text: { $search: "leche", $language: "es" } }
)
```

The $text (page 417) expression can also accept the language by name, spanish. See *text-search-languages* for the supported languages.

See also:

http://docs.mongodb.org/manual/tutorial/text-search-in-aggregation

$where

$where

Use the $where (page 421) operator to pass either a string containing a JavaScript expression or a full JavaScript function to the query system. The $where (page 421) provides greater flexibility, but requires that the database processes the JavaScript expression or function for *each* document in the collection. Reference the document in the JavaScript expression or function using either this or obj .

Behavior

Map Reduce Changed in version 2.4.

In MongoDB 2.4, map-reduce operations (page 220), the group (page 216) command, and $where (page 421) operator expressions **cannot** access certain global functions or properties, such as db, that are available in the mongo (page 610) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your map-reduce operations (page 220), group (page 216) commands, or $where (page 421) operator expressions include any global shell functions or properties that are no longer available, such as db.

The following JavaScript functions and properties **are available** to map-reduce operations (page 220), the group (page 216) command, and $where (page 421) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	
args MaxKey MinKey	assert() BinData() DBPointer() DBRef() doassert() emit() gc() HexData() hex_md5() isNumber() isObject() ISODate() isString()	Map() MD5() NumberInt() NumberLong() ObjectId() print() printjson() printjsononeline() sleep() Timestamp() tojson() tojsononeline() tojsonObject() UUID() version()

elemMatch Changed in version 2.6.

Only apply the $where (page 421) query operator to top-level documents. The $where (page 421) query operator will not work inside a nested document, for instance, in an $elemMatch (page 442) query.

Considerations

- Do not write to the database within the $where (page 421) JavaScript function.
- $where (page 421) evaluates JavaScript and cannot take advantage of indexes. Therefore, query performance improves when you express your query using the standard MongoDB operators (e.g., $gt (page 401), $in (page 402)).
- In general, you should use $where (page 421) only when you can't express your query using another operator. If you must use $where (page 421), try to include at least one other standard query operator to filter the result set. Using $where (page 421) alone requires a table scan.

Using normal non-$where (page 421) query statements provides the following performance advantages:

- MongoDB will evaluate non-$where (page 421) components of query before $where (page 421) statements. If the non-$where (page 421) statements match no documents, MongoDB will not perform any query evaluation using $where (page 421).
- The non-$where (page 421) query statements may use an *index*.

Examples Consider the following examples:

```
db.myCollection.find( { $where: "this.credits == this.debits" } );
db.myCollection.find( { $where: "obj.credits == obj.debits" } );

db.myCollection.find( { $where: function() { return (this.credits == this.debits) } } );
db.myCollection.find( { $where: function() { return obj.credits == obj.debits; } } );
```

Additionally, if the query consists only of the $where (page 421) operator, you can pass in just the JavaScript expression or JavaScript functions, as in the following examples:

```
db.myCollection.find( "this.credits == this.debits || this.credits > this.debits" );

db.myCollection.find( function() { return (this.credits == this.debits || this.credits > this.debits
```

You can include both the standard MongoDB operators and the $where (page 421) operator in your query, as in the following examples:

```
db.myCollection.find( { active: true, $where: "this.credits - this.debits < 0" } );
db.myCollection.find( { active: true, $where: function() { return obj.credits - obj.debits < 0; } } )
```

Geospatial

Geospatial Query Operators

Operators

Query Selectors

Name	Description
$geoIntersects (page 423)	Selects geometries that intersect with a *GeoJSON* geometry. The 2dsphere index supports $geoIntersects (page 423).
$geoWithin (page 425)	Selects geometries within a bounding *GeoJSON geometry*. The 2dsphere and 2d indexes support $geoWithin (page 425).
$nearSphere (page 428)	Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The 2dsphere and 2d indexes support $nearSphere (page 428).
$near (page 429)	Returns geospatial objects in proximity to a point. Requires a geospatial index. The 2dsphere and 2d indexes support $near (page 429).

$geoIntersects

Definition

$geoIntersects

New in version 2.4.

Selects documents whose geospatial data intersects with a specified *GeoJSON* object; i.e. where the intersection of the data and the specified object is non-empty. This includes cases where the data and the specified object share an edge.

The $geoIntersects (page 423) operator uses the $geometry (page 434) operator to specify the *GeoJSON* object. To specify a GeoJSON polygons or multipolygons using the default coordinate reference system (CRS), use the following syntax:

```
{
  <location field>: {
     $geoIntersects: {
        $geometry: {
           type: "<GeoJSON object type>" ,
           coordinates: [ <coordinates> ]
        }
     }
  }
}
```

For $geoIntersects (page 423) queries that specify GeoJSON geometries with areas greater than a single hemisphere, the use of the default CRS results in queries for the complementary geometries.

New in version 2.8: To specify a single-ringed GeoJSON *polygon* with a custom MongoDB CRS, use the following prototype that specifies the custom MongoDB CRS in the $geometry (page 434) expression:

```
{
  <location field>: {
     $geoIntersects: {
        $geometry: {
           type: "Polygon" ,
           coordinates: [ <coordinates> ],
           crs: {
              type: "name",
              properties: { name: "urn:x-mongodb:crs:strictwinding:EPSG:4326" }
           }
        }
     }
  }
}
```

The custom MongoDB CRS uses a counter-clockwise winding order and allows $geoIntersects (page 423) to support queries with a single-ringed GeoJSON *polygon* whose area is greater than or equal to a single hemisphere. If the specified polygon is smaller than a single hemisphere, the behavior of $geoIntersects (page 423) with the MongoDB CRS is the same as with the default CRS. See also *"Big" Polygons* (page 424).

Important: If you use longitude and latitude, specify coordinates in order of: **longitude, latitude.**

Behavior

Geospatial Indexes $geoIntersects (page 423) uses spherical geometry. $geoIntersects (page 423) does not require a geospatial index. However, a geospatial index will improve query performance. Only the 2dsphere geospatial index supports $geoIntersects (page 423).

"Big" Polygons For $geoIntersects (page 423), if you specify a single-ringed polygon that has an area greater than a single hemisphere, include the custom MongoDB coordinate reference system in the $geometry (page 434) expression; otherwise, $geoIntersects (page 423) queries for the complementary geometry. For all other GeoJSON polygons with areas greater than a hemisphere, $geoIntersects (page 423) queries for the complementary geometry.

Examples

Intersects a Polygon The following example uses $geoIntersects (page 423) to select all loc data that intersect with the *geojson-polygon* defined by the coordinates array. The area of the polygon is less than the area of a single hemisphere:

```
db.places.find(
   {
     loc: {
       $geoIntersects: {
          $geometry: {
             type: "Polygon" ,
             coordinates: [
```

```
                [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ], [ 0, 0 ] ]
             ]
          }
       }
    }
 }
)
```

For single-ringed polygons with areas greater than a single hemisphere, see *Intersects a "Big" Polygon* (page 425).

Intersects a "Big" Polygon To query with a single-ringed GeoJSON polygon whose area is greater than a single hemisphere, the `$geometry` (page 434) expression must specify the custom MongoDB coordinate reference system. For example:

```
db.places.find(
   {
     loc: {
       $geoIntersects: {
          $geometry: {
             type : "Polygon",
             coordinates: [
               [
                 [ -100, 60 ], [ -100, 0 ], [ -100, -60 ], [ 100, -60 ], [ 100, 60 ], [ -100, 60 ]
               ]
             ],
             crs: {
                type: "name",
                properties: { name: "urn:x-mongodb:crs:strictwinding:EPSG:4326" }
             }
          }
       }
     }
   }
)
```

$geoWithin

Definition

`$geoWithin`

New in version 2.4: `$geoWithin` (page 425) replaces `$within` (page 427) which is deprecated.

Selects documents with geospatial data that exists entirely within a specified shape. When determining inclusion, MongoDB considers the border of a shape to be part of the shape, subject to the precision of floating point numbers.

The specified shape can be either a GeoJSON *geojson-polygon* (either single-ringed or multi-ringed), a GeoJSON *geojson-multipolygon*, or a shape defined by legacy coordinate pairs. The `$geoWithin` (page 425) operator uses the `$geometry` (page 434) operator to specify the *GeoJSON* object.

To specify a GeoJSON polygons or multipolygons using the default coordinate reference system (CRS), use the following syntax:

```
{
   <location field>: {
      $geoWithin: {
         $geometry: {
```

```
            type: <"Polygon" or "MultiPolygon"> ,
            coordinates: [ <coordinates> ]
         }
      }
   }
}
```

For $geoWithin (page 425) queries that specify GeoJSON geometries with areas greater than a single hemisphere, the use of the default CRS results in queries for the complementary geometries.

New in version 2.8: To specify a single-ringed GeoJSON *polygon* with a custom MongoDB CRS, use the following prototype that specifies the custom MongoDB CRS in the $geometry (page 434) expression:

```
{
   <location field>: {
      $geoWithin: {
         $geometry: {
           type: "Polygon" ,
           coordinates: [ <coordinates> ],
           crs: {
              type: "name",
              properties: { name: "urn:x-mongodb:crs:strictwinding:EPSG:4326" }
           }
         }
      }
   }
}
```

The custom MongoDB CRS uses a counter-clockwise winding order and allows $geoWithin (page 425) to support queries with a single-ringed GeoJSON *polygon* whose area is greater than or equal to a single hemisphere. If the specified polygon is smaller than a single hemisphere, the behavior of $geoWithin (page 425) with the MongoDB CRS is the same as with the default CRS. See also *"Big" Polygons* (page 427).

If querying for inclusion in a shape defined by legacy coordinate pairs on a plane, use the following syntax:

```
{
   <location field>: {
      $geoWithin: { <shape operator>: <coordinates> }
   }
}
```

The available shape operators are:

- $box (page 432),
- $polygon (page 436),
- $center (page 433) (defines a circle), and
- $centerSphere (page 432) (defines a circle on a sphere).

Important: If you use longitude and latitude, specify coordinates in order of `longitude, latitude`.

Behavior

Geospatial Indexes $geoWithin (page 425) does not require a geospatial index. However, a geospatial index will improve query performance. Both 2dsphere and 2d geospatial indexes support $geoWithin (page 425).

Unsorted Results The $geoWithin (page 425) operator does not return sorted results. As such, MongoDB can return $geoWithin (page 425) queries more quickly than geospatial $near (page 429) or $nearSphere (page 428) queries, which sort results.

"Big" Polygons For $geoWithin (page 425), if you specify a single-ringed polygon that has an area greater than a single hemisphere, include the custom MongoDB coordinate reference system in the $geometry (page 434) expression; otherwise, $geoWithin (page 425) queries for the complementary geometry. For all other GeoJSON polygons with areas greater than a hemisphere, $geoWithin (page 425) queries for the complementary geometry.

Examples

Within a Polygon The following example selects all loc data that exist entirely within a GeoJSON *geojson-polygon*. The area of the polygon is less than the area of a single hemisphere:

```
db.places.find(
   {
     loc: {
       $geoWithin: {
          $geometry: {
             type : "Polygon" ,
             coordinates: [ [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ], [ 0, 0 ] ] ]
          }
       }
     }
   }
)
```

For single-ringed polygons with areas greater than a single hemisphere, see *Within a "Big" Polygon* (page 427).

Within a "Big" Polygon To query with a single-ringed GeoJSON polygon whose area is greater than a single hemisphere, the $geometry (page 434) expression must specify the custom MongoDB coordinate reference system. For example:

```
db.places.find(
   {
     loc: {
       $geoWithin: {
          $geometry: {
             type : "Polygon" ,
             coordinates: [
               [
                 [ -100, 60 ], [ -100, 0 ], [ -100, -60 ], [ 100, -60 ], [ 100, 60 ], [ -100, 60 ]
               ]
             ],
             crs: {
                type: "name",
                properties: { name: "urn:x-mongodb:crs:strictwinding:EPSG:4326" }
             }
          }
       }
     }
   }
)
```

$within

Deprecated since version 2.4: $geoWithin (page 425) replaces $within (page 427) in MongoDB 2.4.

$nearSphere

Definition

$nearSphere

Specifies a point for which a *geospatial* query returns the documents from nearest to farthest. MongoDB calculates distances for $nearSphere (page 428) using spherical geometry.

$nearSphere (page 428) *requires* a geospatial index:

- 2dsphere index for location data defined as GeoJSON points
- 2d index for location data defined as legacy coordinate pairs. To use a 2d index on *GeoJSON points*, create the index on the coordinates field of the GeoJSON object.

The $nearSphere (page 428) operator can specify either a *GeoJSON* point or legacy coordinate point.

To specify a *GeoJSON* point, use the following syntax:

```
{
  $nearSphere: {
     $geometry: {
        type : "Point",
        coordinates : [ <longitude>, <latitude> ]
     },
     $minDistance: <distance in meters>,
     $maxDistance: <distance in meters>
  }
}
```

- The *optional* $minDistance (page 435) is available only if the query uses the 2dsphere index. $minDistance (page 435) limits the results to those documents that are *at least* the specified distance from the center point.

  New in version 2.6.

- The *optional* $maxDistance (page 434) is available for either index.

To specify a point using legacy coordinates, use the following syntax:

```
{
  $nearSphere: [ <x>, <y> ],
  $minDistance: <distance in radians>,
  $maxDistance: <distance in radians>
}
```

- The *optional* $minDistance (page 435) is available only if the query uses the 2dsphere index. $minDistance (page 435) limits the results to those documents that are *at least* the specified distance from the center point.

  New in version 2.6.

- The *optional* $maxDistance (page 434) is available for either index.

If you use longitude and latitude for legacy coordinates, specify the longitude first, then latitude.

See also:

2d Indexes and Geospatial Near Queries (page 716)

Examples

Specify Center Point Using GeoJSON Consider a collection `places` that contains documents with a `location` field and has a `2dsphere` index.

Then, the following example returns whose `location` is at least `1000` meters from and at most `5000` meters from the specified point, ordered from nearest to farthest:

```
db.places.find(
   {
     location: {
        $nearSphere: {
           $geometry: {
              type : "Point",
              coordinates : [ -73.9667, 40.78 ]
           },
           $minDistance: 1000,
           $maxDistance: 5000
        }
     }
   }
)
```

Specify Center Point Using Legacy Coordinates

`2d` Index Consider a collection `legacyPlaces` that contains documents with legacy coordinates pairs in the `location` field and has a `2d` index.

Then, the following example returns those documents whose `location` is at most `0.10` radians from the specified point, ordered from nearest to farthest:

```
db.legacyPlaces.find(
   { location : { $nearSphere : [ -73.9667, 40.78 ], $maxDistance: 0.10 } }
)
```

`2dsphere` Index If the collection has a `2dsphere` index instead, you can also specify the optional `$minDistance` (page 435) specification. For example, the following example returns the documents whose `location` is at least `0.0004` radians from the specified point, ordered from nearest to farthest:

```
db.legacyPlaces.find(
   { location : { $nearSphere : [ -73.9667, 40.78 ], $minDistance: 0.0004 } }
)
```

$near

Definition

`$near`

Specifies a point for which a *geospatial* query returns the documents from nearest to farthest. The `$near` (page 429) operator can specify either a *GeoJSON* point or legacy coordinate point.

`$near` (page 429) requires a geospatial index:

- `2dsphere` index if specifying a *GeoJSON* point,
- `2d` index if specifying a point using legacy coordinates.

To specify a *GeoJSON* point, `$near` (page 429) operator requires a `2dsphere` index and has the following syntax:

```
{
  $near: {
     $geometry: {
        type: "Point" ,
        coordinates: [ <longitude> , <latitude> ]
     },
     $maxDistance: <distance in meters>,
     $minDistance: <distance in meters>
  }
}
```

When specifying a *GeoJSON* point, you can use the *optional* `$minDistance` (page 435) and `$maxDistance` (page 434) specifications to limit the `$near` (page 429) results by distance in *meters*:

- `$minDistance` (page 435) limits the results to those documents that are *at least* the specified distance from the center point. `$minDistance` (page 435) is only available for use with `2dsphere` index.

  New in version 2.6.

- `$maxDistance` (page 434) limits the results to those documents that are *at most* the specified distance from the center point.

To specify a point using legacy coordinates, `$near` (page 429) requires a `2d` index and has the following syntax:

```
{
  $near: [ <x>, <y> ],
  $maxDistance: <distance in radians>
}
```

If you use longitude and latitude for legacy coordinates, specify the longitude first, then latitude.

When specifying a legacy coordinate, you can use the *optional* `$maxDistance` (page 434) specification to limit the `$near` (page 429) results by distance in *radians*. `$maxDistance` (page 434) limits the results to those documents that are *at most* the specified distance from the center point.

Behavior

- You cannot combine the `$near` (page 429) operator, which requires a special *geospatial index*, with a query operator or command that uses a different type of special index. For example you cannot combine `$near` (page 429) with the `$text` (page 417) query.
- For sharded collections, queries using `$near` (page 429) are not supported. You can instead use either the `geoNear` (page 229) command or the `$geoNear` (page 484) aggregation stage.
- `$near` (page 429) always returns the documents sorted by distance. Any other sort order requires to sort the documents in memory, which can be inefficient. To return results in a different sort order, use the `$geoWithin` operator and the `sort()` method.

See also:

2d Indexes and Geospatial Near Queries (page 716)

Examples

Query on GeoJSON Data

Important: Specify coordinates in this order: **"longitude, latitude."**

Consider a collection `places` that has a `2dsphere` index.

The following example returns documents that are at least `1000` meters from and at most `5000` meters from the specified GeoJSON point, sorted from nearest to farthest:

```
db.places.find(
   {
     location:
       { $near :
          {
            $geometry: { type: "Point",  coordinates: [ -73.9667, 40.78 ] },
            $minDistance: 1000,
            $maxDistance: 5000
          }
       }
   }
)
```

Query on Legacy Coordinates

Important: Specify coordinates in this order: **"longitude, latitude."**

Consider a collection `legacy2d` that has a `2d` index.

The following example returns documents that are at most `0.10` radians from the specified legacy coordinate pair, sorted from nearest to farthest:

```
db.legacy2d.find(
   { location : { $near : [ -73.9667, 40.78 ], $maxDistance: 0.10 } }
)
```

Geometry Specifiers

Name	Description
`$box` (page 432)	Specifies a rectangular box using legacy coordinate pairs for `$geoWithin` (page 425) queries. The `2d` index supports `$box` (page 432).
`$centerSphere` (page 432)	Specifies a circle using either legacy coordinate pairs or *GeoJSON* format for `$geoWith` (page 425) queries when using spherical geometry. The `2dsphere` and `2d` indexes supp `$centerSphere` (page 432).
`$center` (page 433)	Specifies a circle using legacy coordinate pairs to `$geoWithin` (page 425) queries when using planar geometry. The `2d` index supports `$center` (page 433).
`$geometry` (page 434)	Specifies a geometry in *GeoJSON* format to geospatial query operators.
`$maxDistance` (page 434)	Specifies a maximum distance to limit the results of `$near` (page 429) and `$nearSphe` (page 428) queries. The `2dsphere` and `2d` indexes support `$centerSphere` (page 43
`$minDistance` (page 435)	Specifies a minimum distance to limit the results of `$near` (page 429) and `$nearSphe` (page 428) queries. For use with `2dsphere` index only.
`$polygon` (page 436)	Specifies a polygon to using legacy coordinate pairs for `$geoWithin` (page 425) queries The `2d` index supports `$center` (page 433).
`$uniqueDocs` (page 437)	Deprecated. Modifies a `$geoWithin` (page 425) and `$near` (page 429) queries to ensu that even if a document matches the query multiple times, the query returns the document once.

$box

Definition

$box

Specifies a rectangle for a *geospatial* $geoWithin (page 425) query to return documents that are within the bounds of the rectangle, according to their point-based location data. When used with the $box (page 432) operator, $geoWithin (page 425) returns documents based on *grid coordinates* and does *not* query for GeoJSON shapes.

To use the $box (page 432) operator, you must specify the bottom left and top right corners of the rectangle in an array object:

```
{
  <location field>: {
     $geoWithin: {
        $box: [
          [ <bottom left coordinates> ],
          [ <upper right coordinates> ]
        ]
     }
  }
}
```

Important: If you use longitude and latitude, specify **longitude first**.

Behavior The query calculates distances using flat (planar) geometry.

Changed in version 2.2.3: Applications can use $box (page 432) *without* having a geospatial index. However, geospatial indexes support much faster queries than the unindexed equivalents. Before 2.2.3, a geospatial index *must* exist on a field holding coordinates before using any of the geospatial query operators.

Only the 2d geospatial index supports $box (page 432).

Example The following example query returns all documents that are within the box having points at: [0 , 0], [0 , 100], [100 , 0], and [100 , 100].

```
db.places.find( {
   loc: { $geoWithin: { $box:  [ [ 0, 0 ], [ 100, 100 ] ] } }
} )
```

$centerSphere

Definition

$centerSphere

New in version 1.8.

Defines a circle for a *geospatial* query that uses spherical geometry. The query returns documents that are within the bounds of the circle. You can use the $centerSphere (page 432) operator on both *GeoJSON* objects and legacy coordinate pairs.

To use $centerSphere (page 432), specify an array that contains:

- The grid coordinates of the circle's center point, and
- The circle's radius measured in radians. To calculate radians, see http://docs.mongodb.org/manual/tutorial/calculate-distances-using-spherical-geometr

```
{
   <location field>: {
      $geoWithin: { $centerSphere: [ [ <x>, <y> ], <radius> ] }
   }
}
```

Important: If you use longitude and latitude, specify **longitude first**.

Behavior Changed in version 2.2.3: Applications can use `$centerSphere` (page 432) *without* having a geospatial index. However, geospatial indexes support much faster queries than the unindexed equivalents. Before 2.2.3, a geospatial index *must* exist on a field holding coordinates before using any of the geospatial query operators.

Both `2dsphere` and `2d` geospatial indexes support `$centerSphere` (page 432).

Example The following example queries grid coordinates and returns all documents within a 10 mile radius of longitude `88 W` and latitude `30 N`. The query converts the distance to radians by dividing by the approximate radius of the earth, 3959 miles:

```
db.places.find( {
  loc: { $geoWithin: { $centerSphere: [ [ -88, 30 ], 10/3959 ] } }
} )
```

$center

Definition

`$center`

New in version 1.4.

The `$center` (page 433) operator specifies a circle for a `$geoWithin` (page 425) query. The query returns legacy coordinate pairs that are within the bounds of the circle. The operator does *not* return GeoJSON objects.

To use the `$center` (page 433) operator, specify an array that contains:

- The grid coordinates of the circle's center point, and
- The circle's radius, as measured in the units used by the coordinate system.

```
{
   <location field>: {
      $geoWithin: { $center: [ [ <x>, <y> ] , <radius> ] }
   }
}
```

Important: If you use longitude and latitude, specify **longitude first**.

Behavior The query calculates distances using flat (planar) geometry.

Changed in version 2.2.3: Applications can use `$center` (page 433) *without* having a geospatial index. However, geospatial indexes support much faster queries than the unindexed equivalents. Before 2.2.3, a geospatial index *must* exist on a field holding coordinates before using any of the geospatial query operators.

Only the `2d` geospatial index supports `$center` (page 433).

Example The following example query returns all documents that have coordinates that exist within the circle centered on `[ -74, 40.74 ]` and with a radius of `10`:

```
db.places.find(
   { loc: { $geoWithin: { $center: [ [-74, 40.74], 10 ] } } }
)
```

$geometry

`$geometry`

New in version 2.4.

Changed in version 2.8: Add support to specify single-ringed GeoJSON *polygons* with areas greater than a single hemisphere.

The `$geometry` (page 434) operator specifies a *GeoJSON* geometry for use with the following geospatial query operators: `$geoWithin` (page 425), `$geoIntersects` (page 423), `$near` (page 429), and `$nearSphere` (page 428). `$geometry` (page 434) uses `EPSG:4326` as the default coordinate reference system (CRS).

To specify GeoJSON objects with the default CRS, use the following prototype for `$geometry` (page 434):

```
$geometry: {
   type: "<GeoJSON object type>",
   coordinates: [ <coordinates> ]
}
```

New in version 2.8: To specify a single-ringed GeoJSON *polygon* with a custom MongoDB CRS, use the following prototype (available only for `$geoWithin` (page 425) and `$geoIntersects` (page 423)):

```
$geometry: {
   type: "Polygon",
   coordinates: [ <coordinates> ],
   crs: {
      type: "name",
      properties: { name: "urn:x-mongodb:crs:strictwinding:EPSG:4326" }
   }
}
```

The custom MongoDB coordinate reference system has a strict counter-clockwise winding order.

Important: If you use longitude and latitude, specify coordinates in order of: **longitude, latitude.**

$maxDistance

Definition

`$maxDistance`

The `$maxDistance` (page 434) operator constrains the results of a geospatial `$near` (page 429) or `$nearSphere` (page 428) query to the specified distance. The measuring units for the maximum distance are determined by the coordinate system in use. For *GeoJSON* point object, specify the distance in meters, not radians.

Changed in version 2.6: Specify a non-negative number for `$maxDistance` (page 434).

The `2dsphere` and `2d` geospatial indexes both support `$maxDistance` (page 434): .

Example The following example query returns documents with location values that are `10` or fewer units from the point `[ 100 , 100 ]`.

```
db.places.find( {
   loc: { $near: [ 100 , 100 ],  $maxDistance: 10 }
} )
```

MongoDB orders the results by their distance from `[ 100 , 100 ]`. The operation returns the first 100 results, unless you modify the query with the `cursor.limit()` (page 88) method.

$minDistance

Definition

`$minDistance`

New in version 2.6.

Filters the results of a geospatial `$near` (page 429) or `$nearSphere` (page 428) query to those documents that are *at least* the specified distance from the center point.

`$minDistance` (page 435) is available for use with `2dsphere` index only.

If `$near` (page 429) or `$nearSphere` (page 428) query specifies the center point as a *GeoJSON point*, specify the distance as a non-negative number in *meters*.

If `$nearSphere` (page 428) query specifies the center point as *legacy coordinate pair*, specify the distance as a non-negative number in *radians*. `$near` (page 429) can only use the `2dsphere` index if the query specifies the center point as a *GeoJSON point*.

Examples

Use with `$near`

Important: Specify coordinates in this order: **"longitude, latitude."**

Consider a collection `places` that has a `2dsphere` index.

The following example returns documents that are at least `1000` meters from and at most `5000` meters from the specified GeoJSON point, sorted from nearest to farthest:

```
db.places.find(
   {
     location:
       { $near :
          {
            $geometry: { type: "Point",  coordinates: [ -73.9667, 40.78 ] },
            $minDistance: 1000,
            $maxDistance: 5000
          }
       }
   }
)
```

Use with `$nearSphere` Consider a collection `places` that contains documents with a `location` field and has a `2dsphere` index.

Then, the following example returns whose `location` is at least `1000` meters from and at most `5000` meters from the specified point, ordered from nearest to farthest:

```
db.places.find(
   {
     location: {
        $nearSphere: {
           $geometry: {
              type : "Point",
              coordinates : [ -73.9667, 40.78 ]
           },
           $minDistance: 1000,
           $maxDistance: 5000
        }
     }
   }
)
```

For an example that specifies the center point as legacy coordinate pair, see `$nearSphere` (page 428)

$polygon

Definition

`$polygon`

New in version 1.9.

Specifies a polygon for a *geospatial* `$geoWithin` (page 425) query on legacy coordinate pairs. The query returns pairs that are within the bounds of the polygon. The operator does *not* query for GeoJSON objects.

To define the polygon, specify an array of coordinate points:

```
{
   <location field>: {
      $geoWithin: {
         $polygon: [ [ <x1> , <y1> ], [ <x2> , <y2> ], [ <x3> , <y3> ], ... ]
      }
   }
}
```

The last point is always implicitly connected to the first. You can specify as many points, i.e. sides, as you like.

Important: If you use longitude and latitude, specify **longitude first**.

Behavior The `$polygon` (page 436) operator calculates distances using flat (planar) geometry.

Changed in version 2.2.3: Applications can use `$polygon` (page 436) *without* having a geospatial index. However, geospatial indexes support much faster queries than the unindexed equivalents. Before 2.2.3, a geospatial index *must* exist on a field holding coordinates before using any of the geospatial query operators.

Only the `2d` geospatial index supports the `$polygon` (page 436) operator.

Example The following query returns all documents that have coordinates that exist within the polygon defined by `[ 0 , 0 ]`, `[ 3 , 6 ]`, and `[ 6 , 0 ]`:

```
db.places.find(
  {
     loc: {
       $geoWithin: { $polygon: [ [ 0 , 0 ], [ 3 , 6 ], [ 6 , 0 ] ] }
```

```
        }
    }
)
```

$uniqueDocs

Definition

$uniqueDocs

Deprecated since version 2.6: Geospatial queries no longer return duplicate results. The $uniqueDocs (page 437) operator has no impact on results.

Returns a document only once for a geospatial query even if the document matches the query multiple times.

Geospatial Query Compatibility While numerous combinations of query operators are possible, the following table shows the recommended operators for different types of queries. The table uses the $geoWithin (page 425), $geoIntersects (page 423) and $near (page 429) operators.

Query Document	Geometry of the Query Condition	Surface Type for Query Calculation	Units for Query Calculation	Supported by this Index
Returns points, lines and polygons				
`{ $geoWithin : { $geometry : <GeoJSON Polygon> } }`	polygon	sphere	meters	2dsphere
`{ $geoIntersects : { $geometry : <GeoJSON> } }`	point, line or polygon	sphere	meters	2dsphere
`{ $near : { $geometry : <GeoJSON Point>, $maxDistance : d } }`	point	sphere	meters	2dsphere The index is required.
Returns points only				
`{ $geoWithin : { $box : [[x1, y1], [x2, y2]] } }`	rectangle	flat	flat units	2d
`{ $geoWithin : { $polygon : [[x1, y1], [x1, y2], [x2, y2], [x2, y1]] } }`	polygon	flat	flat units	2d
`{ $geoWithin : { $center : [[x1, y1], r], } }`	circular region	flat	flat units	2d
`{ $geoWithin : { $centerSphere : [[x, y], radius] } }`	circular region	sphere	radians	2d 2dsphere
`{ $near : [x1, y1], $maxDistance : d }`	point	flat / flat units	flat units	2d The index is required.

Array

Query Operator Array

Name	Description
$all (page 439)	Matches arrays that contain all elements specified in the query.
$elemMatch (page 442)	Selects documents if element in the array field matches all the specified $elemMa (page 442) conditions.
$size (page 443)	Selects documents if the array field is a specified size.

$all

$all

The $all (page 439) operator selects the documents where the value of a field is an array that contains all the specified elements. To specify an $all (page 439) expression, use the following prototype:

```
{ <field>: { $all: [ <value1> , <value2> ... ] } }
```

Behavior

Equivalent to $and Operation Changed in version 2.6.

The $all (page 439) is equivalent to an $and (page 405) operation of the specified values; i.e. the following statement:

```
{ tags: { $all: [ "ssl" , "security" ] } }
```

is equivalent to:

```
{ $and: [ { tags: "ssl" }, { tags: "security" } ] }
```

Nested Array Changed in version 2.6.

When passed an array of a nested array (e.g. `[ [ "A" ] ]`), $all (page 439) can now match documents where the field contains the nested array as an element (e.g. `field: [ [ "A" ], ... ]`), *or* the field equals the nested array (e.g. `field: [ "A" ]`).

For example, consider the following query [19]:

```
db.articles.find( { tags: { $all: [ [ "ssl", "security" ] ] } } )
```

The query is equivalent to:

```
db.articles.find( { $and: [ { tags: [ "ssl", "security" ] } ] } )
```

which is equivalent to:

```
db.articles.find( { tags: [ "ssl", "security" ] } )
```

As such, the $all (page 439) expression can match documents where the `tags` field is an array that contains the nested array `[ "ssl", "security" ]` or is an array that equals the nested array:

```
tags: [ [ "ssl", "security" ], ... ]
tags: [ "ssl", "security" ]
```

This behavior for $all (page 439) allows for more matches than previous versions of MongoDB. Earlier versions could only match documents where the field contains the nested array.

[19] The $all (page 439) expression with a *single* element is for illustrative purposes since the $all (page 439) expression is unnecessary if matching only a single element. Instead, when matching a single element, a "contains" expression (i.e. `arrayField: element`) is more suitable.

Performance Queries that use the $all (page 439) operator must scan all the documents that match the first element in the $all (page 439) expression. As a result, even with an index to support the query, the operation may be long running, particularly when the first element in the $all (page 439) expression is not very selective.

Examples The following examples use the inventory collection that contains the documents:

```
{
   _id: ObjectId("5234cc89687ea597eabee675"),
   code: "xyz",
   tags: [ "school", "book", "bag", "headphone", "appliance" ],
   qty: [
          { size: "S", num: 10, color: "blue" },
          { size: "M", num: 45, color: "blue" },
          { size: "L", num: 100, color: "green" }
        ]
}

{
   _id: ObjectId("5234cc8a687ea597eabee676"),
   code: "abc",
   tags: [ "appliance", "school", "book" ],
   qty: [
          { size: "6", num: 100, color: "green" },
          { size: "6", num: 50, color: "blue" },
          { size: "8", num: 100, color: "brown" }
        ]
}

{
   _id: ObjectId("5234ccb7687ea597eabee677"),
   code: "efg",
   tags: [ "school", "book" ],
   qty: [
          { size: "S", num: 10, color: "blue" },
          { size: "M", num: 100, color: "blue" },
          { size: "L", num: 100, color: "green" }
        ]
}

{
   _id: ObjectId("52350353b2eff1353b349de9"),
   code: "ijk",
   tags: [ "electronics", "school" ],
   qty: [
          { size: "M", num: 100, color: "green" }
        ]
}
```

Use $all to Match Values The following operation uses the $all (page 439) operator to query the inventory collection for documents where the value of the tags field is an array whose elements include appliance, school, and book:

```
db.inventory.find( { tags: { $all: [ "appliance", "school", "book" ] } } )
```

The above query returns the following documents:

```
{
   _id: ObjectId("5234cc89687ea597eabee675"),
   code: "xyz",
   tags: [ "school", "book", "bag", "headphone", "appliance" ],
   qty: [
          { size: "S", num: 10, color: "blue" },
          { size: "M", num: 45, color: "blue" },
          { size: "L", num: 100, color: "green" }
        ]
}

{
   _id: ObjectId("5234cc8a687ea597eabee676"),
   code: "abc",
   tags: [ "appliance", "school", "book" ],
   qty: [
          { size: "6", num: 100, color: "green" },
          { size: "6", num: 50, color: "blue" },
          { size: "8", num: 100, color: "brown" }
        ]
}
```

Use `$all` with `$elemMatch` If the field contains an array of documents, you can use the `$all` (page 439) with the `$elemMatch` (page 442) operator.

The following operation queries the `inventory` collection for documents where the value of the `qty` field is an array whose elements match the `$elemMatch` (page 442) criteria:

```
db.inventory.find( {
                     qty: { $all: [
                                    { "$elemMatch" : { size: "M", num: { $gt: 50} } },
                                    { "$elemMatch" : { num : 100, color: "green" } }
                                  ] }
                   } )
```

The query returns the following documents:

```
{
   "_id" : ObjectId("5234ccb7687ea597eabee677"),
   "code" : "efg",
   "tags" : [ "school", "book"],
   "qty" : [
             { "size" : "S", "num" : 10, "color" : "blue" },
             { "size" : "M", "num" : 100, "color" : "blue" },
             { "size" : "L", "num" : 100, "color" : "green" }
           ]
}

{
   "_id" : ObjectId("52350353b2eff1353b349de9"),
   "code" : "ijk",
   "tags" : [ "electronics", "school" ],
   "qty" : [
             { "size" : "M", "num" : 100, "color" : "green" }
           ]
}
```

The `$all` (page 439) operator exists to support queries on arrays. But you may use the `$all` (page 439) operator to

select against a non-array `field`, as in the following example:

```
db.inventory.find( { "qty.num": { $all: [ 50 ] } } )
```

However, use the following form to express the same query:

```
db.inventory.find( { "qty.num" : 50 } )
```

Both queries will select all documents in the `inventory` collection where the value of the `num` field equals `50`.

Note: In most cases, MongoDB does not treat arrays as sets. This operator provides a notable exception to this approach.

See also:

`find()` (page 36), `update()` (page 72), and `$set` (page 459).

$elemMatch (query)

See also:

$elemMatch (projection) (page 446)

Definition

`$elemMatch`

The `$elemMatch` (page 442) operator matches documents in a collection that contain an array field with at least one element that matches all the specified query criteria.

```
{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }
```

Behavior You cannot specify a `$where` (page 421) expression as a query criterion for `$elemMatch` (page 442).

Examples

Element Match Given the following documents in the `scores` collection:

```
{ _id: 1, results: [ 82, 85, 88 ] }
{ _id: 2, results: [ 75, 88, 89 ] }
```

The following query matches only those documents where the `results` array contains at least one element that is both greater than or equal to `80` and is less than `85`.

```
db.scores.find(
   { results: { $elemMatch: { $gte: 80, $lt: 85 } } }
)
```

The query returns the following document since the element `82` is both greater than or equal to `80` and is less than `85`

```
{ "_id" : 1, "results" : [ 82, 85, 88 ] }
```

For more information on specifying multiple criteria on array elements, see *specify-multiple-criteria-for-array-elements*.

Array of Embedded Documents Given the following documents in the `survey` collection:

```
{ _id: 1, results: [ { product: "abc", score: 10 }, { product: "xyz", score: 5 } ] }
{ _id: 2, results: [ { product: "abc", score: 8 }, { product: "xyz", score: 7 } ] }
{ _id: 3, results: [ { product: "abc", score: 7 }, { product: "xyz", score: 8 } ] }
```

The following query matches only those documents where the `results` array contains at least one element with both `product` equal to `"xyz"` and `score` greater than or equal to `8`.

```
db.survey.find(
   { results: { $elemMatch: { product: "xyz", score: { $gte: 8 } } } }
)
```

Specifically, the query matches the following document:

```
{ "_id" : 3, "results" : [ { "product" : "abc", "score" : 7 }, { "product" : "xyz", "score" : 8 } ] }
```

For more information on querying arrays, see *read-operations-arrays*, including *specify-multiple-criteria-for-array-elements* and *array-match-embedded-documents* sections.

$size

`$size`

The `$size` (page 443) operator matches any array with the number of elements specified by the argument. For example:

```
db.collection.find( { field: { $size: 2 } } );
```

returns all documents in `collection` where `field` is an array with 2 elements. For instance, the above expression will return `{ field: [ red, green ] }` and `{ field: [ apple, lime ] }` but *not* `{ field: fruit }` or `{ field: [ orange, lemon, grapefruit ] }`. To match fields with only one element within an array use `$size` (page 443) with a value of 1, as follows:

```
db.collection.find( { field: { $size: 1 } } );
```

`$size` (page 443) does not accept ranges of values. To select documents based on fields with different numbers of elements, create a counter field that you increment when you add elements to a field.

Queries cannot use indexes for the `$size` (page 443) portion of a query, although the other portions of a query can use indexes if applicable.

Comments

$comment

Definition

`$comment`

The `$comment` (page 443) query operator associates a comment to any expression taking a query predicate.

Because comments propagate to the `profile` (page 366) log, adding a comment can make your profile data easier to interpret and trace.

The `$comment` (page 443) operator has the form:

```
db.collection.find( { <query>, $comment: <comment> } )
```

Behavior You can use the `$comment` (page 443) with any expression taking a query predicate, such as the query predicate in `db.collection.update()` (page 72) or in the `$match` (page 490) stage of the *aggregation pipeline* (page 564). For an example, see *Attach a Comment to an Aggregation Expression* (page 444).

Examples

Attach a Comment to `find` The following example adds a `$comment` (page 443) to a `find()` (page 36) operation :

```
db.records.find(
   {
     x: { $mod: [ 2, 0 ] },
     $comment: "Find even values."
   }
)
```

Attach a Comment to an Aggregation Expression You can use the `$comment` (page 443) with any expression taking a query predicate.

The following examples uses the `$comment` (page 443) operator in the `$match` (page 490) stage to clarify the operation:

```
db.records.aggregate( [
   { $match: { x: { $gt: 0 }, $comment: "Don't allow negative inputs." } },
   { $group : { _id: { $mod: [ "$x", 2 ] }, total: { $sum: "$x" } } }
] )
```

See also:

`$comment` (page 555)

Projection Operators

Projection Operators

Name	Description
`$` (page 444)	Projects the first element in an array that matches the query condition.
`$elemMatch` (page 446)	Projects the first element in an array that matches the specified `$elemMatch` (page 446) condition.
`$meta` (page 449)	Projects the document's score assigned during `$text` (page 417) operation.
`$slice` (page 450)	Limits the number of elements projected from an array. Supports skip and limit slices.

$ (projection)

Definition

$

The positional `$` (page 444) operator limits the contents of an `<array>` from the query results to contain only the **first** element matching the query document. To specify an array element to update, see the *positional $ operator for updates* (page 461).

Use `$` (page 444) in the *projection* document of the `find()` (page 36) method or the `findOne()` (page 46) method when you only need one particular array element in selected documents.

Usage Considerations Both the $ (page 444) operator and the $elemMatch (page 446) operator project a subset of elements from an array based on a condition.

The $ (page 444) operator projects the array elements based on some condition from the query statement.

The $elemMatch (page 446) projection operator takes an explicit condition argument. This allows you to project based on a condition not in the query, or if you need to project based on multiple fields in the array's subdocuments. See *Array Field Limitations* (page 445) for an example.

Behavior

Usage Requirements Given the form:

```
db.collection.find( { <array>: <value> ... },
                    { "<array>.$": 1 } )
db.collection.find( { <array.field>: <value> ...},
                    { "<array>.$": 1 } )
```

The <array> field being limited **must** appear in the *query document*, and the <value> can be documents that contain *query operator expressions* (page 400).

Array Field Limitations MongoDB requires the following when dealing with projection over arrays:

- Only one positional $ (page 444) operator may appear in the projection document.
- Only one array field may appear in the *query document*.
- The *query document* should only contain a single condition on the array field being projected. Multiple conditions may override each other internally and lead to undefined behavior.

Under these requirements, the following query is **incorrect**:

```
db.collection.find( { <array>: <value>, <someOtherArray>: <value2> },
                    { "<array>.$": 1 } )
```

To specify criteria on multiple fields of documents inside that array, use the $elemMatch (page 442) query operator. The following query will return any subdocuments inside a grades array that have a mean of greater than 70 and a grade of greater than 90.

```
db.students.find( { grades: { $elemMatch: {
                                            mean: { $gt: 70 },
                                            grade: { $gt:90 }
                                          } } },
                  { "grades.$": 1 } )
```

You must use the $elemMatch (page 446) operator if you need separate conditions for selecting documents and for choosing fields within those documents.

Sorts and the Positional Operator When the find() (page 36) method includes a sort() (page 95), the find() (page 36) method applies the sort() (page 95) to order the matching documents **before** it applies the positional $ (page 444) projection operator.

If an array field contains multiple documents with the same field name and the find() (page 36) method includes a sort() (page 95) on that repeating field, the returned documents may not reflect the sort order because the sort was applied to the elements of the array before the $ (page 444) projection operator.

Examples

Project Array Values A collection `students` contains the following documents:

```
{ "_id" : 1, "semester" : 1, "grades" : [ 70, 87, 90 ] }
{ "_id" : 2, "semester" : 1, "grades" : [ 90, 88, 92 ] }
{ "_id" : 3, "semester" : 1, "grades" : [ 85, 100, 90 ] }
{ "_id" : 4, "semester" : 2, "grades" : [ 79, 85, 80 ] }
{ "_id" : 5, "semester" : 2, "grades" : [ 88, 88, 92 ] }
{ "_id" : 6, "semester" : 2, "grades" : [ 95, 90, 96 ] }
```

In the following query, the projection `{ "grades.$": 1 }` returns only the first element greater than or equal to `85` for the `grades` field.

```
db.students.find( { semester: 1, grades: { $gte: 85 } },
                  { "grades.$": 1 } )
```

The operation returns the following documents:

```
{ "_id" : 1, "grades" : [ 87 ] }
{ "_id" : 2, "grades" : [ 90 ] }
{ "_id" : 3, "grades" : [ 85 ] }
```

Although the array field `grades` may contain multiple elements that are greater than or equal to `85`, the `$` (page 444) projection operator returns only the first matching element from the array.

Project Array Documents A `students` collection contains the following documents where the `grades` field is an array of documents; each document contain the three field names `grade`, `mean`, and `std`:

```
{ "_id" : 7, semester: 3, "grades" : [ { grade: 80, mean: 75, std: 8 },
                                       { grade: 85, mean: 90, std: 5 },
                                       { grade: 90, mean: 85, std: 3 } ] }

{ "_id" : 8, semester: 3, "grades" : [ { grade: 92, mean: 88, std: 8 },
                                       { grade: 78, mean: 90, std: 5 },
                                       { grade: 88, mean: 85, std: 3 } ] }
```

In the following query, the projection `{ "grades.$": 1 }` returns only the first element with the `mean` greater than `70` for the `grades` field:

```
db.students.find(
   { "grades.mean": { $gt: 70 } },
   { "grades.$": 1 }
)
```

The operation returns the following documents:

```
{ "_id" : 7, "grades" : [  {  "grade" : 80,  "mean" : 75,  "std" : 8 } ] }
{ "_id" : 8, "grades" : [  {  "grade" : 92,  "mean" : 88,  "std" : 8 } ] }
```

Further Reading `$elemMatch (projection)` (page 446)

$elemMatch (projection) **See also:**

$elemMatch (query) (page 442)

Definition

$elemMatch

New in version 2.2.

The $elemMatch (page 446) operator limits the contents of an <array> field from the query results to contain only the **first** element matching the $elemMatch (page 446) condition.

Usage Considerations Both the $ (page 444) operator and the $elemMatch (page 446) operator project a subset of elements from an array based on a condition.

The $ (page 444) operator projects the array elements based on some condition from the query statement.

The $elemMatch (page 446) projection operator takes an explicit condition argument. This allows you to project based on a condition not in the query, or if you need to project based on multiple fields in the array's subdocuments. See *Array Field Limitations* (page 445) for an example.

Examples The examples on the $elemMatch (page 446) projection operator assumes a collection school with the following documents:

```
{
 _id: 1,
 zipcode: "63109",
 students: [
              { name: "john", school: 102, age: 10 },
              { name: "jess", school: 102, age: 11 },
              { name: "jeff", school: 108, age: 15 }
            ]
}
{
 _id: 2,
 zipcode: "63110",
 students: [
              { name: "ajax", school: 100, age: 7 },
              { name: "achilles", school: 100, age: 8 },
            ]
}
{
 _id: 3,
 zipcode: "63109",
 students: [
              { name: "ajax", school: 100, age: 7 },
              { name: "achilles", school: 100, age: 8 },
            ]
}
{
 _id: 4,
 zipcode: "63109",
 students: [
              { name: "barney", school: 102, age: 7 },
              { name: "ruth", school: 102, age: 16 },
            ]
}
```

Zipcode Search The following find() (page 36) operation queries for all documents where the value of the zipcode field is 63109. The $elemMatch (page 446) projection returns only the **first** matching element of the students array where the school field has a value of 102:

```
db.schools.find( { zipcode: "63109" },
                 { students: { $elemMatch: { school: 102 } } } )
```

The operation returns the following documents:

```
{ "_id" : 1, "students" : [ { "name" : "john", "school" : 102, "age" : 10 } ] }
{ "_id" : 3 }
{ "_id" : 4, "students" : [ { "name" : "barney", "school" : 102, "age" : 7 } ] }
```

- For the document with `_id` equal to `1`, the `students` array contains multiple elements with the `school` field equal to `102`. However, the `$elemMatch` (page 446) projection returns only the first matching element from the array.
- The document with `_id` equal to `3` does not contain the `students` field in the result since no element in its `students` array matched the `$elemMatch` (page 446) condition.

`$elemMatch` with Multiple Fields The `$elemMatch` (page 446) projection can specify criteria on multiple fields:

The following `find()` (page 36) operation queries for all documents where the value of the `zipcode` field is `63109`. The projection includes the **first** matching element of the `students` array where the `school` field has a value of `102` **and** the `age` field is greater than `10`:

```
db.schools.find( { zipcode: "63109" },
                 { students: { $elemMatch: { school: 102, age: { $gt: 10} } } } )
```

The operation returns the three documents that have `zipcode` equal to `63109`:

```
{ "_id" : 1, "students" : [ { "name" : "jess", "school" : 102, "age" : 11 } ] }
{ "_id" : 3 }
{ "_id" : 4, "students" : [ { "name" : "ruth", "school" : 102, "age" : 16 } ] }
```

The document with `_id` equal to `3` does not contain the `students` field since no array element matched the `$elemMatch` (page 446) criteria.

`$elemMatch` with `sort()` When the `find()` (page 36) method includes a `sort()` (page 95), the `find()` (page 36) method applies the `sort()` (page 95) to order the matching documents **before** it applies the projection. This is a general rule when sorting and projecting, and is discussed in *Interaction with Projection* (page 97).

If an array field contains multiple documents with the same field name and the `find()` (page 36) method includes a `sort()` (page 95) on that repeating field, the returned documents may not reflect the sort order because the `sort()` (page 95) was applied to the elements of the array before the `$elemMatch` (page 446) projection.

An array's sorting value is taken from either its "minimum" or "maximum" value, depending on which way the sorting goes. The way that `sort()` (page 95) sorts documents containing arrays is described in *Ascending/Descending Sort* (page 96).

The following query includes a `sort()` (page 95) to order by descending `students.age` field:

```
db.schools.find(
                 { zipcode: "63109" },
                 { students: { $elemMatch: { school: 102 } } }
               ).sort( { "students.age": -1 } )
```

The operation applies the `sort()` (page 95) to order the documents that have the field `zipcode` equal to `63109` and then applies the projection. The operation returns the three documents in the following order:

```
{ "_id" : 4, "students" : [ { "name" : "barney", "school" : 102, "age" : 7 } ] }
{ "_id" : 1, "students" : [ { "name" : "john", "school" : 102, "age" : 10 } ] }
{ "_id" : 3 }
```

Even though the sort is descending, the younger student is listed first. This is because the sort occurred before the older students in Barney's document were projected out.

See also:

`$ (projection)` (page 444) operator

$meta

`$meta`

New in version 2.6.

The `$meta` (page 449) projection operator returns for each matching document the metadata (e.g. `"textScore"`) associated with the query.

A `$meta` (page 449) expression has the following syntax:

```
{ $meta: <metaDataKeyword> }
```

The `$meta` (page 449) expression can specify the following keyword as the `<metaDataKeyword>`:

Keyword	Description	Sort Order
`"textScore"`	Returns the score associated with the corresponding `$text` (page 417) query for each matching document. The text score signifies how well the document matched the stemmed term or terms. If not used in conjunction with a `$text` (page 417) query, returns a score of 0.	Descending

Behaviors The `$meta` (page 449) expression can be a part of the *projection* document as well as a `sort()` (page 95) expression as:

```
{ <projectedFieldName>: { $meta: "textScore" } }
```

Projected Field Name The `<projectedFieldName>` cannot include a dot (`.`) in the name.

If the specified `<projectedFieldName>` already exists in the matching documents, in the result set, the existing fields will return with the `$meta` (page 449) values instead of with the stored values.

Projection The `$meta` (page 449) expression can be used in the *projection* document, as in:

```
db.collection.find(
   <query>,
   { score: { $meta: "textScore" } }
)
```

The `$meta` (page 449) expression specifies the inclusion of the field to the result set and does *not* specify the exclusion of the other fields.

The `$meta` (page 449) expression can be a part of a projection document that specifies exclusions of other fields or that specifies inclusions of other fields.

The metadata returns information on the processing of the `<query>` operation. As such, the returned metadata, assigned to the `<projectedFieldName>`, has no meaning inside a `<query>` expression; i.e. specifying a condition

on the <projectedFieldName> as part of the <query> is similar to specifying a condition on a non-existing field if no field exists in the documents with the <projectedFieldName>.

Sort The $meta (page 449) expression can be part of a sort() (page 95) expression, as in:

```
db.collection.find(
   <query>,
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } )
```

To include a $meta (page 449) expression in a sort() (page 95) expression, the *same* $meta (page 449) expression, including the <projectedFieldName>, must appear in the projection document. The specified metadata determines the sort order. For example, the "textScore" metadata sorts in descending order.

For additional examples, see *Text Search with Additional Query and Sort Expressions* (page 420).

Examples For examples of "textScore" projections and sorts, see $text (page 417).

$slice (projection)

$slice

The $slice (page 450) operator controls the number of items of an array that a query returns. For information on limiting the size of an array during an update with $push (page 470), see the $slice (page 474) modifier instead.

Consider the following prototype query:

```
db.collection.find( { field: value }, { array: {$slice: count } } );
```

This operation selects the document collection identified by a field named field that holds value and returns the number of elements specified by the value of count from the array stored in the array field. If count has a value greater than the number of elements in array the query returns all elements of the array.

$slice (page 450) accepts arguments in a number of formats, including negative values and arrays. Consider the following examples:

```
db.posts.find( {}, { comments: { $slice: 5 } } )
```

Here, $slice (page 450) selects the first five items in an array in the comments field.

```
db.posts.find( {}, { comments: { $slice: -5 } } )
```

This operation returns the last five items in array.

The following examples specify an array as an argument to $slice (page 450). Arrays take the form of [skip , limit], where the first value indicates the number of items in the array to skip and the second value indicates the number of items to return.

```
db.posts.find( {}, { comments: { $slice: [ 20, 10 ] } } )
```

Here, the query will only return 10 items, after skipping the first 20 items of that array.

```
db.posts.find( {}, { comments: { $slice: [ -20, 10 ] } } )
```

This operation returns 10 items as well, beginning with the item that is 20th from the last item of the array.

2.3.2 Update Operators

The following modifiers are available for use in update operations; e.g. in `db.collection.update()` (page 72) and `db.collection.findAndModify()` (page 42).

Update Operators

Fields

Field Update Operators

Name	Description
`$currentDate` (page 451)	Sets the value of a field to current date, either as a Date or a Timestamp.
`$inc` (page 452)	Increments the value of the field by the specified amount.
`$max` (page 453)	Only updates the field if the specified value is greater than the existing field value.
`$min` (page 454)	Only updates the field if the specified value is less than the existing field value.
`$mul` (page 456)	Multiplies the value of the field by the specified amount.
`$rename` (page 457)	Renames a field.
`$setOnInsert` (page 458)	Sets the value of a field if an update results in an insert of a document. Has no effec update operations that modify existing documents.
`$set` (page 459)	Sets the value of a field in a document.
`$unset` (page 461)	Removes the specified field from a document.

$currentDate

Definition

`$currentDate`

The `$currentDate` (page 451) operator sets the value of a field to the current date, either as a *Date* or a *timestamp*. The default type is *Date*.

The `$currentDate` (page 451) operator has the form:

```
{ $currentDate: { <field1>: <typeSpecification1>, ... } }
```

`<typeSpecification>` can be either:

- a boolean `true` to set the field value to the current date as a Date, or
- a document `{ $type: "timestamp" }` or `{ $type: "date" }` which explicitly specifies the type. The operator is *case-sensitive* and accepts only the lowercase `"timestamp"` or the lowercase `"date"`.

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Behavior If the field does not exist, `$currentDate` (page 451) adds the field to a document.

Example Consider the following document in the `users` collection:

```
{ _id: 1, status: "a", lastModified: ISODate("2013-10-02T01:11:18.965Z") }
```

The following operation updates the `lastModified` field to the current date, the‘‘”cancellation.date”‘‘ field to the current timestamp as well as updating the `status` field to `"D"` and the `"cancellation.reason"` to `"user request"`.

```
db.users.update(
   { _id: 1 },
   {
     $currentDate: {
        lastModified: true,
        "cancellation.date": { $type: "timestamp" }
     },
     $set: {
        status: "D",
        "cancellation.reason": "user request"
     }
   }
)
```

The updated document would resemble:

```
{
   "_id" : 1,
   "status" : "D",
   "lastModified" : ISODate("2014-09-17T23:25:56.314Z"),
   "cancellation" : {
      "date" : Timestamp(1410996356, 1),
      "reason" : "user request"
   }
}
```

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42)

$inc

Definition

`$inc`

The `$inc` (page 452) operator increments a field by a specified value and has the following form:

```
{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }
```

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Behavior The `$inc` (page 452) operator accepts positive and negative values.

If the field does not exist, `$inc` (page 452) creates the field and sets the field to the specified value.

Use of the `$inc` (page 452) operator on a field with a null value will generate an error.

`$inc` (page 452) is an atomic operation within a single document.

Example Consider a collection `products` with the following document:

```
{
  _id: 1,
  sku: "abc123",
```

```
  quantity: 10,
  metrics: {
    orders: 2,
    ratings: 3.5
  }
}
```

The following `update()` (page 72) operation uses the `$inc` (page 452) operator to decrease the `quantity` field by `2` (i.e. increase by `-2`) and increase the `"metrics.orders"` field by `1`:

```
db.products.update(
   { sku: "abc123" },
   { $inc: { quantity: -2, "metrics.orders": 1 } }
)
```

The updated document would resemble:

```
{
   "_id" : 1,
   "sku" : "abc123",
   "quantity" : 8,
   "metrics" : {
      "orders" : 3,
      "ratings" : 3.5
   }
}
```

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42)

$max

Definition

`$max`

The `$max` (page 453) operator updates the value of the field to a specified value *if* the specified value is **greater than** the current value of the field. The `$max` (page 453) operator can compare values of different types, using the *BSON comparison order*.

The `$max` (page 453) operator expression has the form:

```
{ $max: { <field1>: <value1>, ... } }
```

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Behavior If the field does not exists, the `$max` (page 453) operator sets the field to the specified value.

Examples

Use `$max` to Compare Numbers Consider the following document in the collection `scores`:

```
{ _id: 1, highScore: 800, lowScore: 200 }
```

The `highScore` for the document currently has the value `800`. The following operation uses `$max` (page 558) to compare the `800` and the specified value `950` and updates the value of `highScore` to `950` since `950` is greater than `800`:

```
db.scores.update( { _id: 1 }, { $max: { highScore: 950 } } )
```

The `scores` collection now contains the following modified document:

```
{ _id: 1, highScore: 950, lowScore: 200 }
```

The next operation has no effect since the current value of the field `highScore`, i.e. `950`, is greater than `870`:

```
db.scores.update( { _id: 1 }, { $max: { highScore: 870 } } )
```

The document remains unchanged in the `scores` collection:

```
{ _id: 1, highScore: 950, lowScore: 200 }
```

Use `$max` to Compare Dates Consider the following document in the collection `tags`:

```
{
  _id: 1,
  desc: "crafts",
  dateEntered: ISODate("2013-10-01T05:00:00Z"),
  dateExpired: ISODate("2013-10-01T16:38:16.163Z")
}
```

The following operation compares the current value of the `dateExpired` field, i.e. `ISODate("2013-10-01T16:38:16.163Z")`, with the specified date `new Date("2013-09-30")` to determine whether to update the field:

```
db.tags.update(
   { _id: 1 },
   { $max: { dateExpired: new Date("2013-09-30") } }
)
```

The operation does *not* update the `dateExpired` field:

```
{
   _id: 1,
   desc: "decorative arts",
   dateEntered: ISODate("2013-10-01T05:00:00Z"),
   dateExpired: ISODate("2013-10-01T16:38:16.163Z")
}
```

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42)

$min

Definition

`$min`

The `$min` (page 454) updates the value of the field to a specified value *if* the specified value is **less than** the current value of the field. The `$min` (page 454) operator can compare values of different types, using the *BSON comparison order*.

```
{ $min: { <field1>: <value1>, ... } }
```

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Behavior If the field does not exists, the $min (page 454) operator sets the field to the specified value.

Examples

Use $min to Compare Numbers Consider the following document in the collection scores:

```
{ _id: 1, highScore: 800, lowScore: 200 }
```

The lowScore for the document currently has the value 200. The following operation uses $min (page 454) to compare 200 to the specified value 150 and updates the value of lowScore to 150 since 150 is less than 200:

```
db.scores.update( { _id: 1 }, { $min: { lowScore: 150 } } )
```

The scores collection now contains the following modified document:

```
{ _id: 1, highScore: 800, lowScore: 150 }
```

The next operation has no effect since the current value of the field lowScore, i.e 150, is less than 200:

```
db.scores.update( { _id: 1 }, { $min: { lowScore: 250 } } )
```

The document remains unchanged in the scores collection:

```
{ _id: 1, highScore: 800, lowScore: 150 }
```

Use $min to Compare Dates Consider the following document in the collection tags:

```
{
  _id: 1,
  desc: "crafts",
  dateEntered: ISODate("2013-10-01T05:00:00Z"),
  dateExpired: ISODate("2013-10-01T16:38:16Z")
}
```

The following operation compares the current value of the dateEntered field, i.e. ISODate("2013-10-01T05:00:00Z"), with the specified date new Date("2013-09-25") to determine whether to update the field:

```
db.tags.update(
   { _id: 1 },
   { $min: { dateEntered: new Date("2013-09-25") } }
)
```

The operation updates the dateEntered field:

```
{
  _id: 1,
  desc: "crafts",
  dateEntered: ISODate("2013-09-25T00:00:00Z"),
  dateExpired: ISODate("2013-10-01T16:38:16Z")
}
```

See also:

db.collection.update() (page 72), db.collection.findAndModify() (page 42)

$mul

Definition

$mul

New in version 2.6.

Multiply the value of a field by a number. To specify a $mul (page 456) expression, use the following prototype:

```
{ $mul: { field: <number> } }
```

The field to update must contain a numeric value.

To specify a <field> in an embedded document or in an array, use *dot notation*.

Behavior If the field does not exist in a document, $mul (page 456) creates the field and sets the value to zero of the same numeric type as the multiplier.

Multiplication with values of mixed numeric types (32-bit integer, 64-bit integer, float) may result in conversion of numeric type. See *Multiplication Type Conversion Rules* for details.

$mul (page 456) is an atomic operation within a single document.

Examples

Multiply the Value of a Field Consider a collection products with the following document:

```
{ _id: 1, item: "ABC", price: 10.99 }
```

The following db.collection.update() (page 72) operation updates the document, using the $mul (page 456) operator to multiply the value in the price field by 1.25:

```
db.products.update(
   { _id: 1 },
   { $mul: { price: 1.25 } }
)
```

The operation results in the following document, where the new value of the price field 13.7375 reflects the original value 10.99 multiplied by 1.25:

```
{ _id: 1, item: "ABC", price: 13.7375 }
```

Apply $mul Operator to a Non-existing Field Consider a collection products with the following document:

```
{ _id: 2,  item: "Unknown" }
```

The following db.collection.update() (page 72) operation updates the document, applying the $mul (page 456) operator to the field price that does not exist in the document:

```
db.products.update(
   { _id: 2 },
   { $mul: { price: NumberLong(100) } }
)
```

The operation results in the following document with a price field set to value 0 of numeric type *shell-type-long*, the same type as the multiplier:

```
{ "_id" : 2, "item" : "Unknown", "price" : NumberLong(0) }
```

Multiply Mixed Numeric Types Consider a collection `products` with the following document:

```
{ _id: 3,  item: "XYZ", price: NumberLong(10) }
```

The following `db.collection.update()` (page 72) operation uses the `$mul` (page 456) operator to multiply the value in the `price` field *NumberLong(10)* by *NumberInt(5)*:

```
db.products.update(
   { _id: 3 },
   { $mul: { price: NumberInt(5) } }
)
```

The operation results in the following document:

```
{ "_id" : 3, "item" : "XYZ", "price" : NumberLong(50) }
```

The value in the `price` field is of type *shell-type-long*. See *Multiplication Type Conversion Rules* for details.

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42)

$rename

Definition

`$rename`

The `$rename` (page 457) operator updates the name of a field and has the following form:

```
{$rename: { <field1>: <newName1>, <field2>: <newName2>, ... } }
```

The new field name must differ from the existing field name. To specify a `<field>` in an embedded document, use *dot notation*.

Consider the following example:

```
db.students.update( { _id: 1 }, { $rename: { 'nickname': 'alias', 'cell': 'mobile' } } )
```

This operation renames the field `nickname` to `alias`, and the field `cell` to `mobile`.

Behavior The `$rename` (page 457) operator logically performs an `$unset` (page 461) of both the old name and the new name, and then performs a `$set` (page 459) operation with the new name. As such, the operation may not preserve the order of the fields in the document; i.e. the renamed field may move within the document.

If the document already has a field with the `<newName>`, the `$rename` (page 457) operator removes that field and renames the specified `<field>` to `<newName>`.

If the field to rename does not exist in a document, `$rename` (page 457) does nothing (i.e. no operation).

For fields in embedded documents, the `$rename` (page 457) operator can rename these fields as well as move the fields in and out of embedded documents. `$rename` (page 457) does not work if these fields are in array elements.

Examples A collection `students` the following document where a field `nmae` appears misspelled, i.e. should be `name`:

```
{ "_id": 1,
  "alias": [ "The American Cincinnatus", "The American Fabius" ],
  "mobile": "555-555-5555",
```

```
  "nmae": { "first" : "george", "last" : "washington" }
}
```

The examples in this section successively updates this document.

Rename a Field To rename a field, call the `$rename` (page 457) operator with the current name of the field and the new name:

```
db.students.update( { _id: 1 }, { $rename: { "nmae": "name" } } )
```

This operation renames the field `nmae` to `name`:

```
{
  "_id": 1,
  "alias": [ "The American Cincinnatus", "The American Fabius" ],
  "mobile": "555-555-5555",
  "name": { "first" : "george", "last" : "washington" }
}
```

Rename a Field in an Embedded Document To rename a field in an embedded document, call the `$rename` (page 457) operator using the *dot notation* to refer to the field. If the field is to remain in the same embedded document, also use the dot notation in the new name, as in the following:

```
db.students.update( { _id: 1 }, { $rename: { "name.first": "name.fname" } } )
```

This operation renames the embedded field `first` to `fname`:

```
{
  "_id" : 1,
  "alias" : [ "The American Cincinnatus", "The American Fabius" ],
  "mobile" : "555-555-5555",
  "name" : { "fname" : "george", "last" : "washington" }
}
```

Rename a Field That Does Not Exist When renaming a field and the existing field name refers to a field that does not exist, the `$rename` (page 457) operator does nothing, as in the following:

```
db.students.update( { _id: 1 }, { $rename: { 'wife': 'spouse' } } )
```

This operation does nothing because there is no field named `wife`.

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42)

$setOnInsert

Definition

`$setOnInsert`

New in version 2.4.

If an update operation with *upsert: true* (page 74) results in an insert of a document, then `$setOnInsert` (page 458) assigns the specified values to the fields in the document. If the update operation does not result in an insert, `$setOnInsert` (page 458) does nothing.

You can specify the `upsert` option for either the `db.collection.update()` (page 72) or `db.collection.findAndModify()` (page 42) methods.

```
db.collection.update(
   <query>,
   { $setOnInsert: { <field1>: <value1>, ... } },
   { upsert: true }
)
```

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Example A collection named `products` contains no documents.

Then, the following `db.collection.update()` (page 72) with *upsert: true* (page 74) inserts a new document.

```
db.products.update(
  { _id: 1 },
  {
     $set: { item: "apple" },
     $setOnInsert: { defaultQty: 100 }
  },
  { upsert: true }
)
```

MongoDB creates a new document with `_id` equal to `1` from the `<query>` condition, and then applies the `$set` (page 459) and `$setOnInsert` (page 458) operations to this document.

The `products` collection contains the newly-inserted document:

```
{ "_id" : 1, "item" : "apple", "defaultQty" : 100 }
```

If the `db.collection.update()` (page 72) with *upsert: true* (page 74) had found a matching document, then MongoDB performs an update, applying the `$set` (page 459) operation but ignoring the `$setOnInsert` (page 458) operation.

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42)

$set

Definition

`$set`

The `$set` (page 459) operator replaces the value of a field with the specified value.

The `$set` (page 459) operator expression has the following form:

```
{ $set: { <field1>: <value1>, ... } }
```

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Behavior If the field does not exist, `$set` (page 459) will add a new field with the specified value, provided that the new field does not violate a type constraint. If you specify a dotted path for a non-existent field, `$set` (page 459) will create the embedded documents *as needed* to fulfill the dotted path to the field.

If you specify multiple field-value pairs, `$set` (page 459) will update or create each field.

Examples Consider a collection `products` with the following document:

```
{
  _id: 100,
  sku: "abc123",
  quantity: 250,
  instock: true,
  reorder: false,
  details: { model: "14Q2", make: "xyz" },
  tags: [ "apparel", "clothing" ],
  ratings: [ { by: "ijk", rating: 4 } ]
}
```

Set Top-Level Fields For the document matching the criteria `_id` equal to `100`, the following operation uses the `$set` (page 459) operator to update the value of the `quantity` field, `details` field, and the `tags` field.

```
db.products.update(
   { _id: 100 },
   { $set:
      {
        quantity: 500,
        details: { model: "14Q3", make: "xyz" },
        tags: [ "coats", "outerwear", "clothing" ]
      }
   }
)
```

The operation replaces the value of: `quantity` to `500`; the `details` field to a new embedded document, and the `tags` field to a new array.

Set Fields in Embedded Documents To specify a `<field>` in an embedded document or in an array, use *dot notation*.

For the document matching the criteria `_id` equal to `100`, the following operation updates the `make` field in the `details` document:

```
db.products.update(
   { _id: 100 },
   { $set: { "details.make": "zzz" } }
)
```

Set Elements in Arrays To specify a `<field>` in an embedded document or in an array, use *dot notation*.

For the document matching the criteria `_id` equal to `100`, the following operation update the value second element (array index of `1`) in the `tags` field and the `rating` field in the first element (array index of `0`) of the `ratings` array.

```
db.products.update(
   { _id: 100 },
   { $set:
      {
        "tags.1": "rain gear",
        "ratings.0.rating": 2
      }
   }
)
```

For additional update operators for arrays, see *Array Update Operators* (page 461).

See also:

db.collection.update() (page 72), db.collection.findAndModify() (page 42)

$unset

$unset

The $unset (page 461) operator deletes a particular field. Consider the following syntax:

```
{ $unset: { <field1>: "", ... } }
```

The specified value in the $unset (page 461) expression (i.e. "") does not impact the operation.

To specify a <field> in an embedded document or in an array, use *dot notation*.

Behavior If the field does not exist, then $unset (page 461) does nothing (i.e. no operation).

When used with $ (page 462) to match an array element, $unset (page 461) replaces the matching element with null rather than removing the matching element from the array. This behavior keeps consistent the array size and element positions.

Example The following update() (page 72) operation uses the $unset (page 461) operator to remove the fields quantity and instock from the *first* document in the products collection where the field sku has a value of unknown.

```
db.products.update(
   { sku: "unknown" },
   { $unset: { quantity: "", instock: "" } }
)
```

See also:

db.collection.update() (page 72), db.collection.findAndModify() (page 42)

Array

Array Update Operators

	Name	Description
Update Operators	$ (page 462)	Acts as a placeholder to update the first element that matches the query condition in an update.
	$addToSet (page 464)	Adds elements to an array only if they do not already exist in the set.
	$pop (page 465)	Removes the first or last item of an array.
	$pullAll (page 466)	Removes all matching values from an array.
	$pull (page 467)	Removes all array elements that match a specified query.
	$pushAll (page 469)	*Deprecated.* Adds several items to an array.
	$push (page 470)	Adds an item to an array.

$ (update)

Definition

$

The positional $ (page 462) operator identifies an element in an array to update without explicitly specifying the position of the element in the array. To project, or return, an array element from a read operation, see the $ (page 444) projection operator.

The positional $ (page 462) operator has the form:

```
{ "<array>.$" : value }
```

When used with update operations, e.g. db.collection.update() (page 72) and db.collection.findAndModify() (page 42),

- the positional $ (page 462) operator acts as a placeholder for the **first** element that matches the query document, and
- the array field **must** appear as part of the query document.

For example:

```
db.collection.update(
   { <array>: value ... },
   { <update operator>: { "<array>.$" : value } }
)
```

Behavior

upsert Do not use the positional operator $ (page 462) with *upsert* operations because inserts will use the $ as a field name in the inserted document.

Nested Arrays The positional $ (page 462) operator cannot be used for queries which traverse more than one array, such as queries that traverse arrays nested within other arrays, because the replacement for the $ (page 462) placeholder is a single value

Unsets When used with the $unset (page 461) operator, the positional $ (page 462) operator does not remove the matching element from the array but rather sets it to null.

Negations If the query matches the array using a negation operator, such as $ne (page 404), $not (page 407), or $nin (page 404), then you cannot use the positional operator to update values from this array.

However, if the negated portion of the query is inside of an $elemMatch (page 442) expression, then you *can* use the positional operator to update this field.

Examples

Update Values in an Array Consider a collection students with the following documents:

```
{ "_id" : 1, "grades" : [ 80, 85, 90 ] }
{ "_id" : 2, "grades" : [ 88, 90, 92 ] }
{ "_id" : 3, "grades" : [ 85, 100, 90 ] }
```

To update 80 to 82 in the grades array in the first document, use the positional $ (page 462) operator if you do not know the position of the element in the array:

```
db.students.update(
   { _id: 1, grades: 80 },
   { $set: { "grades.$" : 82 } }
)
```

Remember that the positional $ (page 462) operator acts as a placeholder for the **first match** of the update *query document*.

Update Documents in an Array The positional $ (page 462) operator facilitates updates to arrays that contain embedded documents. Use the positional $ (page 462) operator to access the fields in the embedded documents with the *dot notation* on the $ (page 462) operator.

```
db.collection.update(
   { <query selector> },
   { <update operator>: { "array.$.field" : value } }
)
```

Consider the following document in the `students` collection whose `grades` element value is an array of embedded documents:

```
{
  _id: 4,
  grades: [
     { grade: 80, mean: 75, std: 8 },
     { grade: 85, mean: 90, std: 5 },
     { grade: 90, mean: 85, std: 3 }
  ]
}
```

Use the positional $ (page 462) operator to update the value of the `std` field in the embedded document with the `grade` of `85`:

```
db.students.update(
   { _id: 4, "grades.grade": 85 },
   { $set: { "grades.$.std" : 6 } }
)
```

Update Embedded Documents Using Multiple Field Matches The $ (page 462) operator can update the first array element that matches multiple query criteria specified with the `$elemMatch()` (page 442) operator.

Consider the following document in the `students` collection whose `grades` field value is an array of embedded documents:

```
{
  _id: 4,
  grades: [
     { grade: 80, mean: 75, std: 8 },
     { grade: 85, mean: 90, std: 5 },
     { grade: 90, mean: 85, std: 3 }
  ]
}
```

In the example below, the $ (page 462) operator updates the value of the `std` field in the first embedded document that has `grade` field with a value less than or equal to `90` and a `mean` field with a value greater than `80`:

```
db.students.update(
   {
```

```
     _id: 4,
     grades: { $elemMatch: { grade: { $lte: 90 }, mean: { $gt: 80 } } }
   },
   { $set: { "grades.$.std" : 6 } }
)
```

This operation updates the first embedded document that matches the criteria, namely the second embedded document in the array:

```
{
  _id: 4,
  grades: [
    { grade: 80, mean: 75, std: 8 },
    { grade: 85, mean: 90, std: 6 },
    { grade: 90, mean: 85, std: 3 }
  ]
}
```

See also:

db.collection.update() (page 72), db.collection.findAndModify() (page 42), $elemMatch() (page 442)

$addToSet

Definition

$addToSet

The $addToSet (page 464) operator adds a value to an array only *if* the value is *not* already in the array. If the value *is* in the array, $addToSet (page 464) does not modify the array.

The $addToSet (page 464) operator has the form:

```
{ $addToSet: { <field1>: <value1>, ... } }
```

To specify a <field> in an embedded document or in an array, use *dot notation*.

Behavior $addToSet (page 464) only ensures that there are no duplicate items *added* to the set and does not affect existing duplicate elements. $addToSet (page 464) does not guarantee a particular ordering of elements in the modified set.

If the field is absent in the document to update, $addToSet (page 464) creates the array field with the specified value as its element.

If the field is **not** an array, the operation will fail.

If the value is an array, $addToSet (page 464) appends the whole array as a *single* element. To add each element of the value separately, use the $each (page 472) modifier with $addToSet (page 464). See *$each Modifier* (page 465) for details.

If the value is a document, MongoDB determines that the document is a duplicate if an existing document in the array matches the to-be-added document exactly; i.e. the existing document has the exact same fields and values *and* the fields are in the same order. As such, field order matters and you cannot specify that MongoDB compare only a subset of the fields in the document to determine whether the document is a duplicate of an existing array element.

Examples Consider a collection `inventory` with the following document:

```
{ _id: 1, item: "filter", tags: [ "electronics", "camera" ] }
```

Add to Array The following operation adds the element `"accessories"` to the `tags` array since `"accessories"` does not exist in the array:

```
db.inventory.update(
   { _id: 1 },
   { $addToSet: { tags: "accessories" } }
)
```

Value Already Exists The following `$addToSet` (page 464) operation has no effect as `"camera"` is already an element of the `tags` array:

```
db.inventory.update(
   { _id: 1 },
   { $addToSet: { tags: "camera"  } }
)
```

`$each` Modifier You can use the `$addToSet` (page 464) operator with the `$each` (page 472) modifier. The `$each` (page 472) modifier allows to `$addToSet` (page 464) operator to add multiple values to the array field.

A collection `inventory` has the following document:

```
{ _id: 2, item: "cable", tags: [ "electronics", "supplies" ] }
```

Then the following operation uses the `$addToSet` (page 464) operator with the `$each` (page 472) modifier to add multiple elements to the `tags` array:

```
db.inventory.update(
   { _id: 2 },
   { $addToSet: { tags: { $each: [ "camera", "electronics", "accessories" ] } } }
 )
```

The operation adds only `"camera"` and `"accessories"` to the `tags` array since `"electronics"` already exists in the array:

```
{
  _id: 2,
  item: "cable",
  tags: [ "electronics", "supplies", "camera", "accessories" ]
}
```

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42), `$push` (page 470)

$pop

Definition

`$pop`

The `$pop` (page 465) operator removes the first or last element of an array. Pass `$pop` (page 465) a value of `-1` to remove the first element of an array and `1` to remove the last element in an array.

The `$pop` (page 465) operator has the form:

```
{ $pop: { <field>: <-1 | 1>, ... } }
```

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Behavior The `$pop` (page 465) operation fails if the `<field>` is not an array.

If the `$pop` (page 465) operator removes the last item in the `<field>`, the `<field>` will then hold an empty array.

Examples

Remove the First Item of an Array Given the following document in a collection `students`:

```
{ _id: 1, scores: [ 8, 9, 10 ] }
```

The following example removes the *first* element (`8`) in the `scores` array:

```
db.students.update( { _id: 1 }, { $pop: { scores: -1 } } )
```

After the operation, the updated document has the first item `8` removed from its `scores` array:

```
{ _id: 1, scores: [ 9, 10 ] }
```

Remove the Last Item of an Array Given the following document in a collection `students`:

```
{ _id: 1, scores: [ 9, 10 ] }
```

The following example removes the *last* element (`10`) in the `scores` array by specifying `1` in the `$pop` (page 465) expression:

```
db.students.update( { _id: 1 }, { $pop: { scores: 1 } } )
```

After the operation, the updated document has the last item `10` removed from its `scores` array:

```
{ _id: 1, scores: [ 9 ] }
```

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42)

$pullAll

Definition

`$pullAll`

The `$pullAll` (page 466) operator removes all instances of the specified values from an existing array. Unlike the `$pull` (page 467) operator that removes elements by specifying a query, `$pullAll` (page 466) removes elements that match the listed values.

The `$pullAll` (page 466) operator has the form:

```
{ $pullAll: { <field1>: [ <value1>, <value2> ... ], ... } }
```

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Behavior If a `<value>` to remove is a document or an array, `$pullAll` (page 466) removes only the elements in the array that match the specified `<value>` exactly, including order.

Examples Given the following document in the `survey` collection:

```
{ _id: 1, scores: [ 0, 2, 5, 5, 1, 0 ] }
```

The following operation removes all instances of the value `0` and `5` from the `scores` array:

```
db.survey.update( { _id: 1 }, { $pullAll: { scores: [ 0, 5 ] } } )
```

After the operation, the updated document has all instances of `0` and `5` removed from the `scores` field:

```
{ "_id" : 1, "scores" : [ 2, 1 ] }
```

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42)

$pull

`$pull`

The `$pull` (page 467) operator removes from an existing array all instances of a value or values that match a specified query.

The `$pull` (page 467) operator has the form:

```
{ $pull: { <field1>: <value|query>, ... } }
```

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Behavior If you specify a `<query>` and the array elements are embedded documents, `$pull` (page 467) operator applies the `<query>` as if each array element were a document in a collection. See *Remove Items from an Array of Documents* (page 468) for an example.

If the specified `<value>` to remove is an array, `$pull` (page 467) removes only the elements in the array that match the specified `<value>` exactly, including order.

If the specified `<value>` to remove is a document, `$pull` (page 467) removes only the elements in the array that have the exact same fields and values. The ordering of the fields can differ.

Examples

Remove All Items That Equals a Specified Value Given the following documents in the `cpuinfo` collection:

```
{ _id: 1, flags: [ "vme", "de", "msr", "tsc", "pse", "msr" ] }
{ _id: 2, flags: [ "msr", "pse", "tsc" ] }
```

The following operation removes the value `"msr"` from the `flags` array:

```
db.cpuinfo.update(
                    { flags: "msr" },
                    { $pull: { flags: "msr" } },
                    { multi: true }
                  )
```

After the operation, the documents no longer contain any `"msr"` values in the `flags` array:

```
{ _id: 1, flags: [  "vme",  "de",  "tsc",  "pse" ] }
{ _id: 2, flags: [  "pse",  "tsc" ] }
```

Remove All Items That Match a Specified `$pull` Condition Given the following document in the `profiles` collection:

```
{ _id: 1, votes: [ 3, 5, 6, 7, 7, 8 ] }
```

The following operation will remove all items from the `votes` array that are greater than or equal to (`$gte` (page 401)) `6`:

```
db.profiles.update( { _id: 1 }, { $pull: { votes: { $gte: 6 } } } )
```

After the update operation, the document only has values less than 6:

```
{ _id: 1, votes: [  3,  5 ] }
```

Remove Items from an Array of Documents A `survey` collection contains the following documents:

```
{
   _id: 1,
   results: [
      { item: "A", score: 5 },
      { item: "B", score: 8, comment: "Strongly agree" }
   ]
}
{
   _id: 2,
   results: [
      { item: "C", score: 8, comment: "Strongly agree" },
      { item: "B", score: 4 }
   ]
}
```

The following operation will remove from the `results` array all elements that contain a `score` field equal to `8` and `item` field equal to `"B"`:

```
db.survey.update(
  { },
  { $pull: { results: { score: 8 , item: "B" } } },
  { multi: true }
)
```

The `$pull` (page 467) expression applies the condition to each element of the `results` array as though it were a top-level document.

After the operation, the `results` array contains no documents that contains `score` field equal to `8` and `item` field equal to `"B"`.

```
{
   "_id" : 1,
   "results" : [ { "item" : "A", "score" : 5 } ]
}
{
  "_id" : 2,
  "results" : [
      { "item" : "C", "score" : 8, "comment" : "Strongly agree" },
      { "item" : "B", "score" : 4 }
   ]
}
```

Because $pull (page 467) operator applies its query to each element as though it were a top-level object, the expression did not require the use of $elemMatch (page 442) to specify the condition of a score field equal to 8 and item field equal to "B". In fact, the following operation will not pull any element from the original collection.

```
db.survey.update(
  { },
  { $pull: { results: { $elemMatch: { score: 8 , item: "B" } } } },
  { multi: true }
)
```

However, if the survey collection contained the following documents, where the results array contains embedded documents that also contain arrays:

```
{
   _id: 1,
   results: [
      { item: "A", score: 5, answers: [ { q: 1, a: 4 }, { q: 2, a: 6 } ] },
      { item: "B", score: 8, answers: [ { q: 1, a: 8 }, { q: 2, a: 9 } ] }
   ]
}
{
   _id: 2,
   results: [
      { item: "C", score: 8, answers: [ { q: 1, a: 8 }, { q: 2, a: 7 } ] },
      { item: "B", score: 4, answers: [ { q: 1, a: 0 }, { q: 2, a: 8 } ] }
   ]
}
```

Then you can specify multiple conditions on the elements of the answers array with $elemMatch (page 442):

```
db.survey.update(
  { },
  { $pull: { results: { answers: { $elemMatch: { q: 2, a: { $gte: 8 } } } } } },
  { multi: true }
)
```

The operation removed from the results array those embedded documents with an answers field that contained at least one element with q equal to 2 and a greater than or equal to 8:

```
{
   "_id" : 1,
   "results" : [
      { "item" : "A", "score" : 5, "answers" : [ { "q" : 1, "a" : 4 }, { "q" : 2, "a" : 6 } ] }
   ]
}
{
   "_id" : 2,
   "results" : [
      { "item" : "C", "score" : 8, "answers" : [ { "q" : 1, "a" : 8 }, { "q" : 2, "a" : 7 } ] }
   ]
}
```

See also:

db.collection.update() (page 72), db.collection.findAndModify() (page 42)

$pushAll

$pushAll

Deprecated since version 2.4: Use the $push (page 470) operator with $each (page 472) instead.

The $pushAll (page 469) operator appends the specified values to an array.

The $pushAll (page 469) operator has the form:

```
{ $pushAll: { <field>: [ <value1>, <value2>, ... ] } }
```

If you specify a single value, $pushAll (page 469) will behave as $push (page 470).

$push

Definition

$push

The $push (page 470) operator appends a specified value to an array.

The $push (page 470) operator has the form:

```
{ $push: { <field1>: <value1>, ... }
```

To specify a <field> in an embedded document or in an array, use *dot notation*.

Behavior If the field is absent in the document to update, $push (page 470) adds the array field with the value as its element.

If the field is **not** an array, the operation will fail.

If the value is an array, $push (page 470) appends the whole array as a *single* element. To add each element of the value separately, use the $each (page 472) modifier with $push (page 470). For an example, see *Append Multiple Values to an Array* (page 471). For a list of modifiers available for $push (page 470), see *Modifiers* (page 470).

Changed in version 2.4: MongoDB adds support for the $each (page 472) modifier to the $push (page 470) operator. Before 2.4, use $pushAll (page 469) for similar functionality.

Modifiers New in version 2.4.

You can use the $push (page 470) operator with the following modifiers:

Modifier	Description
$each (page 472)	Appends multiple values to the array field. Changed in version 2.6: When used in conjunction with the other modifiers, the $each (page 472) modifier no longer needs to be first.
$slice (page 474)	Limits the number of array elements. Requires the use of the $each (page 472) modifier.
$sort (page 477)	Orders elements of the array. Requires the use of the $each (page 472) modifier. Changed in version 2.6: In previous versions, $sort (page 477) is only available when used with both $each (page 472) and $slice (page 474).
$position (page 473)	Specifies the location in the array at which to insert the new elements. Requires the use of the $each (page 472) modifier. Without the $position (page 473) modifier, the $push (page 470) appends the elements to the end of the array. New in version 2.6.

When used with modifiers, the $push (page 470) operator has the form:

```
{ $push: { <field1>: { <modifier1>: <value1>, ... }, ... } }
```

The processing of the push operation with modifiers occur in the following order, regardless of the order in which the modifiers appear:

1. Update array to add elements in the correct position.

2. Apply sort, if specified.
3. Slice the array, if specified.
4. Store the array.

Examples

Append a Value to an Array The following example appends `89` to the `scores` array:

```
db.students.update(
   { _id: 1 },
   { $push: { scores: 89 } }
)
```

Append Multiple Values to an Array Use `$push` (page 470) with the `$each` (page 472) modifier to append multiple values to the array field.

The following example appends each element of `[ 90, 92, 85 ]` to the `scores` array for the document where the `name` field equals `joe`:

```
db.students.update(
   { name: "joe" },
   { $push: { scores: { $each: [ 90, 92, 85 ] } } }
)
```

Use `$push` Operator with Multiple Modifiers A collection `students` has the following document:

```
{
   "_id" : 5,
   "quizzes" : [
      { "wk": 1, "score" : 10 },
      { "wk": 2, "score" : 8 },
      { "wk": 3, "score" : 5 },
      { "wk": 4, "score" : 6 }
   ]
}
```

The following `$push` (page 470) operation uses:

- the `$each` (page 472) modifier to add multiple documents to the `quizzes` array,
- the `$sort` (page 477) modifier to sort all the elements of the modified `quizzes` array by the `score` field in descending order, and
- the `$slice` (page 474) modifier to keep only the **first** three sorted elements of the `quizzes` array.

```
db.students.update(
   { _id: 5 },
   {
     $push: {
       quizzes: {
          $each: [ { wk: 5, score: 8 }, { wk: 6, score: 7 }, { wk: 7, score: 6 } ],
          $sort: { score: -1 },
          $slice: 3
       }
     }
```

```
    }
)
```

The result of the operation is keep only the three highest scoring quizzes:

```
{
  "_id" : 5,
  "quizzes" : [
     { "wk" : 1, "score" : 10 },
     { "wk" : 2, "score" : 8 },
     { "wk" : 5, "score" : 8 }
  ]
}
```

See also:

db.collection.update() (page 72), db.collection.findAndModify() (page 42)

Update Operator Modifiers

Name	Description
$each (page 472)	Modifies the $push (page 470) and $addToSet (page 464) operators to appe items for array updates.
$position (page 473)	Modifies the $push (page 470) operator to specify the position in the array to a elements.
$slice (page 474)	Modifies the $push (page 470) operator to limit the size of updated arrays.
$sort (page 477)	Modifies the $push (page 470) operator to reorder documents stored in an arra

$each

Definition

$each

The $each (page 472) modifier is available for use with the $addToSet (page 464) operator and the $push (page 470) operator.

Use with the $addToSet (page 464) operator to add multiple values to an array <field> if the values do not exist in the <field>.

```
{ $addToSet: { <field>: { $each: [ <value1>, <value2> ... ] } } }
```

Use with the $push (page 470) operator to append multiple values to an array <field>.

```
{ $push: { <field>: { $each: [ <value1>, <value2> ... ] } } }
```

Changed in version 2.4: MongoDB adds support for the $each (page 472) modifier to the $push (page 470) operator. The $push (page 470) operator can use $each (page 472) modifier with other modifiers. For a list of modifiers available for $push (page 470), see *Modifiers* (page 470).

Examples

Use $each with $push Operator The following example appends each element of [90, 92, 85] to the scores array for the document where the name field equals joe:

```
db.students.update(
   { name: "joe" },
   { $push: { scores: { $each: [ 90, 92, 85 ] } } }
)
```

Use `$each` with `$addToSet` Operator A collection `inventory` has the following document:

```
{ _id: 2, item: "cable", tags: [ "electronics", "supplies" ] }
```

Then the following operation uses the `$addToSet` (page 464) operator with the `$each` (page 472) modifier to add multiple elements to the `tags` array:

```
db.inventory.update(
   { _id: 2 },
   { $addToSet: { tags: { $each: [ "camera", "electronics", "accessories" ] } } }
 )
```

The operation adds only `"camera"` and `"accessories"` to the `tags` array since `"electronics"` already exists in the array:

```
{
  _id: 2,
  item: "cable",
  tags: [ "electronics", "supplies", "camera", "accessories" ]
}
```

$position

Definition

`$position`

New in version 2.6.

The `$position` (page 473) modifier specifies the location in the array at which the `$push` (page 470) operator insert elements. Without the `$position` (page 473) modifier, the `$push` (page 470) operator inserts elements to the end of the array. See *$push modifiers* (page 470) for more information.

To use the `$position` (page 473) modifier, it **must** appear with the `$each` (page 472) modifier.

```
{
  $push: {
    <field>: {
       $each: [ <value1>, <value2>, ... ],
       $position: <num>
    }
  }
}
```

The `<num>` is a non-negative number that corresponds to the position in the array, based on a zero-based index.

If the `<num>` is greater or equal to the length of the array, the `$position` (page 473) modifier has no effect and `$push` (page 470) adds elements to the end of the array.

Examples

Add Elements at the Start of the Array Consider a collection `students` that contains the following document:

```
{ "_id" : 1, "scores" : [ 100 ] }
```

The following operation updates the `scores` field to add the elements `50`, `60` and `70` to the beginning of the array:

```
db.students.update(
   { _id: 1 },
   {
     $push: {
        scores: {
           $each: [ 50, 60, 70 ],
           $position: 0
        }
     }
   }
)
```

The operation results in the following updated document:

```
{ "_id" : 1, "scores" : [  50,  60,  70,  100 ] }
```

Add Elements to the Middle of the Array Consider a collection `students` that contains the following document:

```
{ "_id" : 1, "scores" : [  50,  60,  70,  100 ] }
```

The following operation updates the `scores` field to add the elements `20` and `30` at the array index of `2`:

```
db.students.update(
   { _id: 1 },
   {
     $push: {
        scores: {
           $each: [ 20, 30 ],
           $position: 2
        }
     }
   }
)
```

The operation results in the following updated document:

```
{ "_id" : 1, "scores" : [  50,  60,  20,  30,  70,  100 ] }
```

$slice

`$slice`

New in version 2.4.

The `$slice` (page 474) modifier limits the number of array elements during a `$push` (page 470) operation. To project, or return, a specified number of array elements from a read operation, see the `$slice` (page 450) projection operator instead.

To use the `$slice` (page 474) modifier, it **must** appear with the `$each` (page 472) modifier. You can pass an empty array `[]` to the `$each` (page 472) modifier such that only the `$slice` (page 474) modifier has an effect.

```
{
  $push: {
```

```
      <field>: {
        $each: [ <value1>, <value2>, ... ],
        $slice: <num>
      }
    }
}
```

The <num> can be:

Value	Description
Zero	To update the array <field> to an empty array.
Negative	To update the array <field> to contain only the last <num> elements.
Positive	To update the array <field> contain only the first <num> elements.
	New in version 2.6.

Behavior Changed in version 2.6.

The order in which the modifiers appear is immaterial. Previous versions required the $each (page 472) modifier to appear as the first modifier if used in conjunction with $slice (page 474). For a list of modifiers available for $push (page 470), see *Modifiers* (page 470).

Trying to use the $slice (page 474) modifier without the $each (page 472) modifier results in an error.

Examples

Slice from the End of the Array A collection students contains the following document:

```
{ "_id" : 1, "scores" : [ 40, 50, 60 ] }
```

The following operation adds new elements to the scores array, and then uses the $slice (page 474) modifier to trim the array to the last five elements:

```
db.students.update(
   { _id: 1 },
   {
     $push: {
       scores: {
         $each: [ 80, 78, 86 ],
         $slice: -5
       }
     }
   }
)
```

The result of the operation is slice the elements of the updated scores array to the last five elements:

```
{ "_id" : 1, "scores" : [  50,  60,  80,  78,  86 ] }
```

Slice from the Front of the Array A collection students contains the following document:

```
{ "_id" : 2, "scores" : [ 89, 90 ] }
```

The following operation adds new elements to the scores array, and then uses the $slice (page 474) modifier to trim the array to the first three elements.

```
db.students.update(
   { _id: 2 },
   {
     $push: {
       scores: {
         $each: [ 100, 20 ],
         $slice: 3
       }
     }
   }
)
```

The result of the operation is to slice the elements of the updated `scores` array to the first three elements:

```
{ "_id" : 2, "scores" : [  89,  90,  100 ] }
```

Update Array Using Slice Only A collection `students` contains the following document:

```
{ "_id" : 3, "scores" : [  89,  70,  100,  20 ] }
```

To update the `scores` field with just the effects of the `$slice` (page 474) modifier, specify the number of elements to slice (e.g. `-3`) for the `$slice` (page 474) modifier and an empty array `[]` for the `$each` (page 472) modifier, as in the following:

```
db.students.update(
  { _id: 3 },
  {
    $push: {
      scores: {
         $each: [ ],
         $slice: -3
      }
    }
  }
)
```

The result of the operation is to slice the elements of the `scores` array to the last three elements:

```
{ "_id" : 3, "scores" : [  70,  100,  20 ] }
```

Use `$slice` with Other `$push` Modifiers A collection `students` has the following document:

```
{
   "_id" : 5,
   "quizzes" : [
      { "wk": 1, "score" : 10 },
      { "wk": 2, "score" : 8 },
      { "wk": 3, "score" : 5 },
      { "wk": 4, "score" : 6 }
   ]
}
```

The following `$push` (page 470) operation uses:

- the `$each` (page 472) modifier to add multiple documents to the `quizzes` array,
- the `$sort` (page 477) modifier to sort all the elements of the modified `quizzes` array by the `score` field in descending order, and

- the $slice (page 474) modifier to keep only the **first** three sorted elements of the quizzes array.

```
db.students.update(
   { _id: 5 },
   {
     $push: {
       quizzes: {
          $each: [ { wk: 5, score: 8 }, { wk: 6, score: 7 }, { wk: 7, score: 6 } ],
          $sort: { score: -1 },
          $slice: 3
       }
     }
   }
)
```

The result of the operation is keep only the three highest scoring quizzes:

```
{
  "_id" : 5,
  "quizzes" : [
     { "wk" : 1, "score" : 10 },
     { "wk" : 2, "score" : 8 },
     { "wk" : 5, "score" : 8 }
  ]
}
```

The order of the modifiers is immaterial to the order in which the modifiers are processed. See *Modifiers* (page 470) for details.

$sort

$sort

New in version 2.4.

The $sort (page 477) modifier orders the elements of an array during a $push (page 470) operation.

To use the $sort (page 477) modifier, it **must** appear with the $each (page 472) modifier. You can pass an empty array [] to the $each (page 472) modifier such that only the $sort (page 477) modifier has an effect.

```
{
  $push: {
     <field>: {
       $each: [ <value1>, <value2>, ... ],
       $sort: <sort specification>
     }
  }
}
```

For <sort specification>:

- To sort array elements that are not documents, or if the array elements are documents, to sort by the whole documents, specify 1 for ascending or -1 for descending.
- If the array elements are documents, to sort by a field in the documents, specify a sort document with the field and the direction, i.e. { field: 1 } or { field: -1 }. Do **not** reference the containing array field in the sort specification (e.g. { "arrayField.field": 1 } is incorrect).

Behavior Changed in version 2.6.

The $sort (page 477) modifier can sort array elements that are not documents. In previous versions, the $sort (page 477) modifier required the array elements be documents.

If the array elements are documents, the modifier can sort by either the whole document or by a specific field in the documents. In previous versions, the $sort (page 477) modifier can only sort by a specific field in the documents.

Trying to use the $sort (page 477) modifier without the $each (page 472) modifier results in an error. The $sort (page 477) no longer requires the $slice (page 474) modifier. For a list of modifiers available for $push (page 470), see *Modifiers* (page 470).

Examples

Sort Array of Documents by a Field in the Documents A collection students contains the following document:

```
{
  "_id": 1,
  "quizzes": [
    { "id" : 1, "score" : 6 },
    { "id" : 2, "score" : 9 }
  ]
}
```

The following update appends additional documents to the quizzes array and then sorts all the elements of the array by the ascending score field:

```
db.students.update(
   { _id: 1 },
   {
     $push: {
       quizzes: {
         $each: [ { id: 3, score: 8 }, { id: 4, score: 7 }, { id: 5, score: 6 } ],
         $sort: { score: 1 }
       }
     }
   }
)
```

Important: The sort document refers directly to the field in the documents and does not reference the containing array field quizzes; i.e. { score: 1 } and **not** { "quizzes.score": 1}

After the update, the array elements are in order of ascending score field.:

```
{
  "_id" : 1,
  "quizzes" : [
     { "id" : 1, "score" : 6 },
     { "id" : 5, "score" : 6 },
     { "id" : 4, "score" : 7 },
     { "id" : 3, "score" : 8 },
     { "id" : 2, "score" : 9 }
  ]
}
```

Sort Array Elements That Are Not Documents A collection students contains the following document:

```
{ "_id" : 2, "tests" : [  89,  70,  89,  50 ] }
```

The following operation adds two more elements to the `scores` array and sorts the elements:

```
db.students.update(
   { _id: 2 },
   { $push: { tests: { $each: [ 40, 60 ], $sort: 1 } } }
)
```

The updated document has the elements of the `scores` array in ascending order:

```
{ "_id" : 2, "tests" : [  40,  50,  60,  70,  89,  89 ] }
```

Update Array Using Sort Only A collection `students` contains the following document:

```
{ "_id" : 3, "tests" : [  89,  70,  100,  20 ] }
```

To update the `tests` field to sort its elements in descending order, specify the `{ $sort: -1 }` and specify an empty array `[]` for the `$each` (page 472) modifier, as in the following:

```
db.students.update(
   { _id: 3 },
   { $push: { tests: { $each: [ ], $sort: -1 } } }
)
```

The result of the operation is to update the `scores` field to sort its elements in descending order:

```
{ "_id" : 3, "tests" : [ 100,  89,  70,  20 ] }
```

Use `$sort` with Other `$push` Modifiers A collection `students` has the following document:

```
{
   "_id" : 5,
   "quizzes" : [
      { "wk": 1, "score" : 10 },
      { "wk": 2, "score" : 8 },
      { "wk": 3, "score" : 5 },
      { "wk": 4, "score" : 6 }
   ]
}
```

The following `$push` (page 470) operation uses:

- the `$each` (page 472) modifier to add multiple documents to the `quizzes` array,
- the `$sort` (page 477) modifier to sort all the elements of the modified `quizzes` array by the `score` field in descending order, and
- the `$slice` (page 474) modifier to keep only the **first** three sorted elements of the `quizzes` array.

```
db.students.update(
   { _id: 5 },
   {
     $push: {
       quizzes: {
          $each: [ { wk: 5, score: 8 }, { wk: 6, score: 7 }, { wk: 7, score: 6 } ],
          $sort: { score: -1 },
          $slice: 3
       }
```

```
      }
    }
)
```

The result of the operation is keep only the three highest scoring quizzes:

```
{
  "_id" : 5,
  "quizzes" : [
     { "wk" : 1, "score" : 10 },
     { "wk" : 2, "score" : 8 },
     { "wk" : 5, "score" : 8 }
  ]
}
```

The order of the modifiers is immaterial to the order in which the modifiers are processed. See *Modifiers* (page 470) for details.

Bitwise

Bitwise Update Operator

Name	Description
$bit (page 480)	Performs bitwise AND, OR, and XOR updates of integer values.

$bit

Definition

$bit

Changed in version 2.6: Added support for bitwise `xor` operation.

The `$bit` (page 480) operator performs a bitwise update of a field. The operator supports bitwise `and`, bitwise `or`, and bitwise `xor` (i.e. exclusive or) operations. To specify a `$bit` (page 480) operator expression, use the following prototype:

```
{ $bit: { <field>: { <and|or|xor>: <int> } } }
```

Only use this operator with integer fields (either 32-bit integer or 64-bit integer).

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Note: All numbers in the `mongo` (page 610) shell are doubles, not integers. Use the `NumberInt()` or the `NumberLong()` constructor to specify integers. See *shell-type-int* or *shell-type-long* for more information.

Examples

Bitwise AND Consider the following document inserted into the collection `switches`:

```
{ _id: 1, expdata: NumberInt(13) }
```

The following `update()` (page 72) operation updates the `expdata` field to the result of a bitwise `and` operation between the current value `NumberInt(13)` (i.e. `1101`) and `NumberInt(10)` (i.e. `1010`):

```
db.switches.update(
   { _id: 1 },
   { $bit: { expdata: { and: NumberInt(10) } } }
)
```

The bitwise `and` operation results in the integer 8 (i.e. `1000`):

```
1101
1010
----
1000
```

And the updated document has the following value for `expdata`:

```
{ "_id" : 1, "expdata" : 8 }
```

The `mongo` (page 610) shell displays `NumberInt(8)` as `8`.

Bitwise OR Consider the following document inserted into the collection `switches`:

```
{ _id: 2, expdata: NumberLong(3) }
```

The following `update()` (page 72) operation updates the `expdata` field to the result of a bitwise `or` operation between the current value `NumberLong(3)` (i.e. `0011`) and `NumberInt(5)` (i.e. `0101`):

```
db.switches.update(
   { _id: 2 },
   { $bit: { expdata: { or: NumberInt(5) } } }
)
```

The bitwise `or` operation results in the integer 7 (i.e. `0111`):

```
0011
0101
----
0111
```

And the updated document has the following value for `expdata`:

```
{ "_id" : 2, "expdata" : NumberLong(7) }
```

Bitwise XOR Consider the following document in the collection `switches`:

```
{ _id: 3, expdata: NumberLong(1) }
```

The following `update()` (page 72) operation updates the `expdata` field to the result of a bitwise `xor` operation between the current value `NumberLong(1)` (i.e. `0001`) and `NumberInt(5)` (i.e. `0101`):

```
db.switches.update(
   { _id: 3 },
   { $bit: { expdata: { xor: NumberInt(5) } } }
)
```

The bitwise `xor` operation results in the integer 4:

```
0001
0101
----
0100
```

And the updated document has the following value for `expdata`:

```
{ "_id" : 3, "expdata" : NumberLong(4) }
```

See also:

`db.collection.update()` (page 72), `db.collection.findAndModify()` (page 42)

Isolation

Isolation Update Operator

Name	Description
`$isolated` (page 482)	Modifies the behavior of a write operation to increase the isolation of the op

$isolated

Definition

`$isolated`

Prevents a write operation that affects multiple documents from yielding to other reads or writes once the first document is written. By using the `$isolated` (page 482) option, you can ensure that no client sees the changes until the operation completes or errors out.

This behavior can significantly affect the concurrency of the system as the operation holds the write lock much longer than normal.

Behavior The `$isolated` (page 482) isolation operator does **not** provide "all-or-nothing" atomicity for write operations.

`$isolated` (page 482) does **not** work with *sharded clusters*.

Example Consider the following example:

```
db.foo.update(
    { status : "A" , $isolated : 1 },
    { $inc : { count : 1 } },
    { multi: true }
)
```

Without the `$isolated` (page 482) operator, the `multi`-update operation will allow other operations to interleave with its update of the matched documents.

See also:

`db.collection.update()` (page 72) and `db.collection.remove()` (page 66)

`$atomic`

Deprecated since version 2.2: The `$isolated` (page 482) operator replaces `$atomic`.

2.3.3 Aggregation Pipeline Operators

Stage Operators

Pipeline stages appear in an array. Documents pass through the stages in sequence.

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

Name	Description
$geoNear (page 484)	Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of $match (page 490), $sort (page 499), and $limit (page 489) for geospatial data. The output documents include an additional distance field and can include a location identifier field.
$group (page 486)	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
$limit (page 489)	Passes the first *n* documents unmodified to the pipeline where *n* is the specified limit. For each input document, outputs either one document (for the first *n* documents) or zero documents (after the first *n* documents).
$match (page 490)	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match (page 490) uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$out (page 491)	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out (page 491) stage, it must be the last stage in the pipeline.
$project (page 492)	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
$redact (page 495)	Reshapes each document in the stream by restricting the content for each document based on information stored in the documents themselves. Incorporates the functionality of $project (page 492) and $match (page 490). Can be used to implement field level redaction. For each input document, outputs either one or zero document.
$skip (page 499)	Skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents).
$sort (page 499)	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
$unwind (page 501)	Deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array.

Pipeline Aggregation Stages

Name	Description
$geoNear (page 484)	Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of $match (page 490), $sort (page 499), and $limit (page 489) for geospatial data. The output documents include an additional distance field and can include a location identifier field.
$group (page 486)	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
$limit (page 489)	Passes the first *n* documents unmodified to the pipeline where *n* is the specified limit. For each input document, outputs either one document (for the first *n* documents) or zero documents (after the first *n* documents).
$match (page 490)	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match (page 490) uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$out (page 491)	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out (page 491) stage, it must be the last stage in the pipeline.
$project (page 492)	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
$redact (page 495)	Reshapes each document in the stream by restricting the content for each document based on information stored in the documents themselves. Incorporates the functionality of $project (page 492) and $match (page 490). Can be used to implement field level redaction. For each input document, outputs either one or zero document.
$skip (page 499)	Skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents).
$sort (page 499)	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
$unwind (page 501)	Deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array.

$geoNear (aggregation)

Definition

$geoNear

New in version 2.4.

Outputs documents in order of nearest to farthest from a specified point.

The $geoNear (page 484) stage has the following prototype form:

```
{ $geoNear: { <geoNear options> } }
```

The $geoNear (page 484) operator accepts a *document* that contains the following $geoNear (page 484) options. Specify all distances in the same units as those of the processed documents' coordinate system:

:field GeoJSON point,:term:*legacy coordinate pairs* <*legacy coordinate pairs*> near:

The point for which to find the closest documents.

field string distanceField The output field that contains the calculated distance. To specify a field within a subdocument, use *dot notation*.

field number limit The maximum number of documents to return. The default value is `100`. See also the `num` option.

field number num The `num` option provides the same function as the `limit` option. Both define the maximum number of documents to return. If both options are included, the `num` value overrides the `limit` value.

field number maxDistance The maximum distance from the center point that the documents *can* be. MongoDB limits the results to those documents that fall within the specified distance from the center point.

Specify the distance in meters for *GeoJSON* data and in radians for *legacy coordinate pairs*.

field document query Limits the results to the documents that match the query. The query syntax is the usual MongoDB *read operation query* syntax.

You cannot specify a `$near` (page 429) predicate in the `query` field of the `$geoNear` (page 484) stage.

field Boolean spherical Required *if* using a `2dsphere` index. For use with `2dsphere` indexes, `spherical` must be `true`.

The default value is `false`.

field number distanceMultiplier The factor to multiply all distances returned by the query. For example, use the `distanceMultiplier` to convert radians, as returned by a spherical query, to kilometers by multiplying by the radius of the Earth.

field string includeLocs This specifies the output field that identifies the location used to calculate the distance. This option is useful when a location field contains multiple locations. To specify a field within a subdocument, use *dot notation*.

field Boolean uniqueDocs If this value is `true`, the query returns a matching document once, even if more than one of the document's location fields match the query.

Deprecated since version 2.6: Geospatial queries no longer return duplicate results. The `$uniqueDocs` (page 437) operator has no impact on results.

Behavior When using `$geoNear` (page 484), consider that:

- You can only use `$geoNear` (page 484) as the first stage of a pipeline.
- You must include the `distanceField` option. The `distanceField` option specifies the field that will contain the calculated distance.
- The collection must have a `geospatial index`.
- The `$geoNear` (page 484) requires that a collection have *at most* only one `2d index` and/or only one `2dsphere index`.
- You do not need to specify which field in the documents hold the coordinate pair or point. Because `$geoNear` (page 484) requires that the collection have a single geospatial index, `$geoNear` (page 484) implicitly uses the indexed field.
- If using a `2dsphere index`, you must specify `spherical: true`.
- You cannot specify a `$near` (page 429) predicate in the `query` field of the `$geoNear` (page 484) stage.

Generally, the options for `$geoNear` (page 484) are similar to the `geoNear` (page 229) command with the following exceptions:

- `distanceField` is a mandatory field for the `$geoNear` (page 484) pipeline operator; the option does not exist in the `geoNear` (page 229) command.

- includeLocs accepts a string in the $geoNear (page 484) pipeline operator and a boolean in the geoNear (page 229) command.

Example Consider a collection places that has a 2dsphere index. The following aggregation finds at most 5 unique documents with a location at most 2 units from the center [-73.99279 , 40.719296] and have type equal to public:

```
db.places.aggregate([
   {
     $geoNear: {
        near: { type: "Point", coordinates: [ -73.99279 , 40.719296 ] },
        distanceField: "dist.calculated",
        maxDistance: 2,
        query: { type: "public" },
        includeLocs: "dist.location",
        num: 5,
        spherical: true
     }
   }
])
```

The aggregation returns the following:

```
{
   "_id" : 8,
   "name" : "Sara D. Roosevelt Park",
   "type" : "public",
   "location" : {
      "type" : "Point",
      "coordinates" : [ -73.9928, 40.7193 ]
   },
   "dist" : {
      "calculated" : 0.9539931676365992,
      "location" : {
         "type" : "Point",
         "coordinates" : [ -73.9928, 40.7193 ]
      }
   }
}
```

The matching document contains two new fields:

- dist.calculated field that contains the calculated distance, and
- dist.location field that contains the location used in the calculation.

$group (aggregation)

$group

Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The output documents contain an _id field which contains the distinct group by key. The output documents can also contain computed fields that hold the values of some accumulator expression grouped by the $group (page 486)'s _id field. $group (page 486) does *not* order its output documents.

The $group (page 486) stage has the following prototype form:

```
{ $group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... } }
```

The `_id` field is *mandatory*; however, you can specify an `_id` value of null to calculate accumulated values for all the input documents as a whole.

The remaining computed fields are *optional* and computed using the `<accumulator>` operators.

The `_id` and the `<accumulator>` expressions can accept any valid *expression* (page 565). For more information on expressions, see *Expressions* (page 565).

Accumulator Operator The `<accumulator>` operator must be one of the following accumulator operators:

Name	Description
`$addToSet` (page 548)	Returns an array of *unique* expression values for each group. Order of the array elements is undefined.
`$avg` (page 549)	Returns an average for each group. Ignores non-numeric values.
`$first` (page 549)	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.
`$last` (page 550)	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.
`$max` (page 551)	Returns the highest expression value for each group.
`$min` (page 552)	Returns the lowest expression value for each group.
`$push` (page 553)	Returns an array of expression values for each group.
`$sum` (page 554)	Returns a sum for each group. Ignores non-numeric values.

`$group` Operator and Memory The `$group` (page 486) stage has a limit of 100 megabytes of RAM. By default, if the stage exceeds this limit, `$group` (page 486) will produce an error. However, to allow for the handling of large datasets, set the `allowDiskUse` (page 22) option to `true` to enable `$group` (page 486) operations to write to temporary files. See `db.collection.aggregate()` (page 22) method and the `aggregate` (page 210) command for details.

Changed in version 2.6: MongoDB introduces a limit of 100 megabytes of RAM for the `$group` (page 486) stage as well as the `allowDiskUse` (page 22) option to handle operations for large datasets.

Examples

Calculate Count, Sum, and Average Given a collection `sales` with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-03-15T09:00:00Z") }
{ "_id" : 4, "item" : "xyz", "price" : 5, "quantity" : 20, "date" : ISODate("2014-04-04T11:21:39.736Z") }
{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.331Z") }
```

Group by Month, Day, and Year The following aggregation operation uses the `$group` (page 486) stage to group the documents by the month, day, and year and calculates the total price and the average quantity as well as counts the documents per each group:

```
db.sales.aggregate(
   [
      {
        $group : {
           _id : { month: { $month: "$date" }, day: { $dayOfMonth: "$date" }, year: { $year: "$date" } },
           totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
           averageQuantity: { $avg: "$quantity" },
           count: { $sum: 1 }
```

```
        }
      }
   ]
)
```

The operation returns the following results:

```
{ "_id" : { "month" : 3, "day" : 15, "year" : 2014 }, "totalPrice" : 50, "averageQuantity" : 10, "cou
{ "_id" : { "month" : 4, "day" : 4, "year" : 2014 }, "totalPrice" : 200, "averageQuantity" : 15, "cou
{ "_id" : { "month" : 3, "day" : 1, "year" : 2014 }, "totalPrice" : 40, "averageQuantity" : 1.5, "cou
```

Group by `null` The following aggregation operation specifies a group `_id` of `null`, calculating the total price and the average quantity as well as counts for all documents in the collection:

```
db.sales.aggregate(
   [
      {
        $group : {
           _id : null,
           totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
           averageQuantity: { $avg: "$quantity" },
           count: { $sum: 1 }
        }
      }
   ]
)
```

The operation returns the following result:

```
{ "_id" : null, "totalPrice" : 290, "averageQuantity" : 8.6, "count" : 5 }
```

Retrieve Distinct Values Given a collection `sales` with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-03-15T09:00:00Z") }
{ "_id" : 4, "item" : "xyz", "price" : 5, "quantity" : 20, "date" : ISODate("2014-04-04T11:21:39.736Z
{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.331
```

The following aggregation operation uses the `$group` (page 486) stage to group the documents by the item to retrieve the distinct item values:

```
db.sales.aggregate( [ { $group : { _id : "$item" } } ] )
```

The operation returns the following result:

```
{ "_id" : "xyz" }
{ "_id" : "jkl" }
{ "_id" : "abc" }
```

Pivot Data A collection `books` contains the following documents:

```
{ "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 }
{ "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 }
{ "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
{ "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 }
{ "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
```

Group `title` by `author` The following aggregation operation pivots the data in the `books` collection to have titles grouped by authors.

```
db.books.aggregate(
    [
      { $group : { _id : "$author", books: { $push: "$title" } } }
    ]
)
```

The operation returns the following documents:

```
{ "_id" : "Homer", "books" : [ "The Odyssey", "Iliad" ] }
{ "_id" : "Dante", "books" : [ "The Banquet", "Divine Comedy", "Eclogues" ] }
```

Group Documents by `author` The following aggregation operation uses the `$$ROOT` (page 574) system variable to group the documents by authors. The resulting documents must not exceed the `BSON Document Size` (page 692) limit.

```
db.books.aggregate(
    [
      { $group : { _id : "$author", books: { $push: "$$ROOT" } } }
    ]
)
```

The operation returns the following documents:

```
{
  "_id" : "Homer",
  "books" :
     [
       { "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 },
       { "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
     ]
}

{
  "_id" : "Dante",
  "books" :
     [
       { "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 },
       { "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 },
       { "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
     ]
}
```

See also:

The `http://docs.mongodb.org/manual/tutorial/aggregation-zip-code-data-set` tutorial provides an extensive example of the `$group` (page 486) operator in a common use case.

$limit (aggregation)

Definition

`$limit`

Limits the number of documents passed to the next stage in the *pipeline*.

The `$limit` (page 489) stage has the following prototype form:

```
{ $limit: <positive integer> }
```

$limit (page 489) takes a positive integer that specifies the maximum number of documents to pass along.

Example Consider the following example:

```
db.article.aggregate(
    { $limit : 5 }
);
```

This operation returns only the first 5 documents passed to it from by the pipeline. $limit (page 489) has no effect on the content of the documents it passes.

Note: When a $sort (page 499) immediately precedes a $limit (page 489) in the pipeline, the $sort (page 499) operation only maintains the top n results as it progresses, where n is the specified limit, and MongoDB only needs to store n items in memory. This optimization still applies when allowDiskUse is true and the n items exceed the *aggregation memory limit*.

Changed in version 2.4: Before MongoDB 2.4, $sort (page 499) would sort all the results in memory, and then limit the results to n results.

$match (aggregation)

Definition

$match

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

The $match (page 490) stage has the following prototype form:

```
{ $match: { <query> } }
```

$match (page 490) takes a document that specifies the query conditions. The query syntax is identical to the *read operation query* syntax.

Behavior

Pipeline Optimization

- Place the $match (page 490) as early in the aggregation *pipeline* as possible. Because $match (page 490) limits the total number of documents in the aggregation pipeline, earlier $match (page 490) operations minimize the amount of processing down the pipe.
- If you place a $match (page 490) at the very beginning of a pipeline, the query can take advantage of *indexes* like any other db.collection.find() (page 36) or db.collection.findOne() (page 46).

Restrictions

- You cannot use $where (page 421) in $match (page 490) queries as part of the aggregation pipeline.
- To use $text (page 417) in the $match (page 490) stage, the $match (page 490) stage has to be the first stage of the pipeline.

Examples

Equality Match The following operation uses $match (page 490) to perform a simple equality match:

```
db.articles.aggregate(
    [ { $match : { author : "dave" } } ]
);
```

The $match (page 490) selects the documents where the author field equals dave, and the aggregation returns the following:

```
{
  "result" : [
                {
                  "_id" : ObjectId("512bc95fe835e68f199c8686"),
                  "author": "dave",
                  "score" : 80
                },
                { "_id" : ObjectId("512bc962e835e68f199c8687"),
                  "author" : "dave",
                  "score" : 85
                }
              ],
  "ok" : 1
}
```

Perform a Count The following example selects documents to process using the $match (page 490) pipeline operator and then pipes the results to the $group (page 486) pipeline operator to compute a count of the documents:

```
db.articles.aggregate( [
                        { $match : { score : { $gt : 70, $lte : 90 } } },
                        { $group: { _id: null, count: { $sum: 1 } } }
                       ] );
```

In the aggregation pipeline, $match (page 490) selects the documents where the score is greater than 70 and less than or equal to 90. These documents are then piped to the $group (page 486) to perform a count. The aggregation returns the following:

```
{
  "result" : [
                {
                  "_id" : null,
                  "count" : 3
                }
              ],
  "ok" : 1
}
```

$out (aggregation) New in version 2.6.

Definition

$out

Takes the documents returned by the aggregation pipeline and writes them to a specified collection. The $out (page 491) operator must be *the last stage* in the pipeline. The $out (page 491) operator lets the aggregation framework return result sets of any size.

The $out (page 491) stage has the following prototype form:

```
{ $out: "<output-collection>" }
```

$out (page 491) takes a string that specifies the output collection name.

Important:

- You cannot specify a sharded collection as the output collection. The input collection for a pipeline can be sharded.
- The $out (page 491) operator cannot write results to a capped collection.

Behaviors

Create New Collection The $out (page 491) operation creates a new collection in the current database if one does not already exist. The collection is not visible until the aggregation completes. If the aggregation fails, MongoDB does not create the collection.

Replace Existing Collection If the collection specified by the $out (page 491) operation already exists, then upon completion of the aggregation, the $out (page 491) stage atomically replaces the existing collection with the new results collection. The $out (page 491) operation does not change any indexes that existed on the previous collection. If the aggregation fails, the $out (page 491) operation makes no changes to the pre-existing collection.

Index Constraints The pipeline will fail to complete if the documents produced by the pipeline would violate any unique indexes, including the index on the _id field of the original output collection.

Example A collection books contains the following documents:

```
{ "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 }
{ "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 }
{ "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
{ "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 }
{ "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
```

The following aggregation operation pivots the data in the books collection to have titles grouped by authors and then writes the results to the authors collection.

```
db.books.aggregate( [
                      { $group : { _id : "$author", books: { $push: "$title" } } },
                      { $out : "authors" }
                    ] )
```

After the operation, the authors collection contains the following documents:

```
{ "_id" : "Homer", "books" : [ "The Odyssey", "Iliad" ] }
{ "_id" : "Dante", "books" : [ "The Banquet", "Divine Comedy", "Eclogues" ] }
```

$project (aggregation)

$project

Passes along the documents with only the specified fields to the next stage in the pipeline. The specified fields can be existing fields from the input documents or newly computed fields.

The $project (page 492) stage has the following prototype form:

```
{ $project: { <specifications> } }
```

The $project (page 492) takes a document that can specify the inclusion of fields, the suppression of the _id field, the addition of new fields, and the resetting the values of existing fields. The specifications have the following forms:

Syntax	Description
`<field>: <1 or true>`	Specify the inclusion of a field.
`_id: <0 or false>`	Specify the suppression of the _id field.
`<field>: <expression>`	Add a new field or reset the value of an existing field.

Include Existing Fields

- The _id field is, by default, included in the output documents. To include the other fields from the input documents in the output documents, you must explicitly specify the inclusion in $project (page 492).
- If you specify an inclusion of a field that does not exist in the document, $project (page 492) ignores that field inclusion; i.e. $project (page 492) does not add the field to the document.

Suppress the _id Field The _id field is always included in the output documents by default. To exclude the _id field from the output documents, you must explicitly specify the suppression of the _id field in $project (page 492).

Add New Fields or Reset Existing Fields To add a new field or to reset the value of an existing field, specify the field name and set its value to some expression. For more information on expressions, see *Expressions* (page 565).

To set a field value directly to a numeric or boolean literal, as opposed to setting the field to an expression that resolves to a literal, use the $literal (page 533) operator. Otherwise, $project (page 492) treats the numeric or boolean literal as a flag for including or excluding the field.

By specifying a new field and setting its value to the field path of an existing field, you can effectively rename a field.

Embedded Document Fields When projecting or adding/resetting a field within an embedded document, you can either use *dot notation*, as in

```
"contact.address.country": <1 or 0 or expression>
```

Or you can nest the fields:

```
contact: { address: { country: <1 or 0 or expression> } }
```

When nesting the fields, you *cannot* use dot notation inside the embedded document to specify the field, e.g. `contact: { "address.country": <1 or 0 or expression> }` is *invalid*.

Examples

Include Specific Fields in Output Documents Consider a books collection with the following document:

```
{
  "_id" : 1,
  title: "abc123",
  isbn: "0001122223334",
  author: { last: "zzz", first: "aaa" },
  copies: 5
}
```

The following $project (page 492) stage includes only the _id, title, and the author fields in its output documents:

```
db.books.aggregate( [ { $project : { title : 1 , author : 1 } } ] )
```

The operation results in the following document:

```
{ "_id" : 1, "title" : "abc123", "author" : { "last" : "zzz", "first" : "aaa" } }
```

Suppress _id Field in the Output Documents The _id field is always included by default. To exclude the _id field from the output documents of the $project (page 492) stage, specify the exclusion of the _id field by setting it to 0 in the projection document.

Consider a books collection with the following document:

```
{
  "_id" : 1,
  title: "abc123",
  isbn: "0001122223334",
  author: { last: "zzz", first: "aaa" },
  copies: 5
}
```

The following $project (page 492) stage excludes the _id field but includes the title, and the author fields in its output documents:

```
db.books.aggregate( [ { $project : { _id: 0, title : 1 , author : 1 } } ] )
```

The operation results in the following document:

```
{ "title" : "abc123", "author" : { "last" : "zzz", "first" : "aaa" } }
```

Include Specific Fields from Embedded Documents Consider a bookmarks collection with the following documents:

```
{ _id: 1, user: "1234", stop: { title: "book1", author: "xyz", page: 32 } }
{ _id: 2, user: "7890", stop: [ { title: "book2", author: "abc", page: 5 }, { title: "b", author: "i
```

To include only the title field in the embedded document in the stop field, you can use the *dot notation*:

```
db.bookmarks.aggregate( [ { $project: { "stop.title": 1 } } ] )
```

Or, you can nest the inclusion specification in a document:

```
db.bookmarks.aggregate( [ { $project: { stop: { title: 1 } } } ] )
```

Both specifications result in the following documents:

```
{ "_id" : 1, "stop" : { "title" : "book1" } }
{ "_id" : 2, "stop" : [ { "title" : "book2" }, { "title" : "book3" } ] }
```

Include Computed Fields Consider a books collection with the following document:

```
{
  "_id" : 1,
  title: "abc123",
  isbn: "0001122223334",
  author: { last: "zzz", first: "aaa" },
```

```
  copies: 5
}
```

The following $project (page 492) stage adds the new fields isbn, lastName, and copiesSold:

```
db.books.aggregate(
   [
      {
         $project: {
            title: 1,
            isbn: {
               prefix: { $substr: [ "$isbn", 0, 3 ] },
               group: { $substr: [ "$isbn", 3, 2 ] },
               publisher: { $substr: [ "$isbn", 5, 4 ] },
               title: { $substr: [ "$isbn", 9, 3 ] },
               checkDigit: { $substr: [ "$isbn", 12, 1] }
            },
            lastName: "$author.last",
            copiesSold: "$copies"
         }
      }
   ]
)
```

The operation results in the following document:

```
{
   "_id" : 1,
   "title" : "abc123",
   "isbn" : {
      "prefix" : "000",
      "group" : "11",
      "publisher" : "2222",
      "title" : "333",
      "checkDigit" : "4"
   },
   "lastName" : "zzz",
   "copiesSold" : 5
}
```

$redact (aggregation)

Definition

$redact

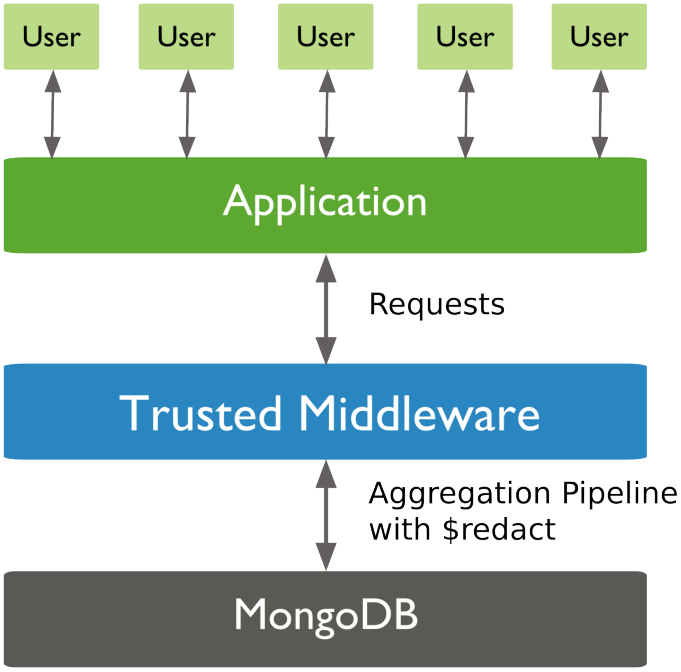
New in version 2.6.

Restricts the contents of the documents based on information stored in the documents themselves.

The $redact (page 495) stage has the following prototype form:

```
{ $redact: <expression> }
```

The argument can be any valid *expression* (page 565) as long as it resolves to $$DESCEND (page 496), $$PRUNE (page 496), or $$KEEP (page 496) system variables. For more information on expressions, see *Expressions* (page 565).



System Variable	Description
$$DESCEND	`$redact` (page 495) returns the *non-subdocument* fields at the current document/subdocument level. For subdocuments or subdocuments in arrays, apply the `$cond` (page 545) expression to the subdocuments to determine access for these subdocuments.
$$PRUNE	`$redact` (page 495) excludes all fields at this current document/subdocument level, **without** further inspection of any of the excluded fields. This applies even if the excluded field contains subdocuments that may have different access levels.
$$KEEP	`$redact` (page 495) returns or keeps all fields at this current document/subdocument level, **without** further inspection of the fields at this level. This applies even if the included field contains subdocuments that may have different access levels.

Examples The examples in this section use the `db.collection.aggregate()` (page 22) helper provided in the 2.6 version of the `mongo` (page 610) shell.

Evaluate Access at Every Document/Sub-document Level A `forecasts` collection contains documents of the following form where the `tags` field lists the different access values for that document/subdocument level; i.e. a value of `[ "G", "STLW" ]` specifies either `"G"` or `"STLW"` can access the data:

```
{
   _id: 1,
   title: "123 Department Report",
   tags: [ "G", "STLW" ],
   year: 2014,
   subsections: [
       {
           subtitle: "Section 1: Overview",
```

```
            tags: [ "SI", "G" ],
            content:  "Section 1: This is the content of section 1."
        },
        {
            subtitle: "Section 2: Analysis",
            tags: [ "STLW" ],
            content: "Section 2: This is the content of section 2."
        },
        {
            subtitle: "Section 3: Budgeting",
            tags: [ "TK" ],
            content: {
                text: "Section 3: This is the content of section3.",
                tags: [ "HCS" ]
            }
        }
    ]
}
```

A user has access to view information with either the tag `"STLW"` or `"G"`. To run a query on all documents with year `2014` for this user, include a `$redact` (page 495) stage as in the following:

```
var userAccess = [ "STLW", "G" ];
db.forecasts.aggregate(
   [
     { $match: { year: 2014 } },
     { $redact:
        {
           $cond:
              {
                if: { $gt: [ { $size: { $setIntersection: [ "$tags", userAccess ] } }, 0 ] },
                then: "$$DESCEND",
                else: "$$PRUNE"
              }
        }
     }
   ]
)
```

The aggregation operation returns the following "redacted" document:

```
{
   "_id" : 1,
   "title" : "123 Department Report",
   "tags" : [ "G", "STLW" ],
   "year" : 2014,
   "subsections" : [
      {
         "subtitle" : "Section 1: Overview",
         "tags" : [ "SI", "G" ],
         "content" : "Section 1: This is the content of section 1."
      },
      {
         "subtitle" : "Section 2: Analysis",
         "tags" : [ "STLW" ],
         "content" : "Section 2: This is the content of section 2."
      }
   ]
}
```

See also:

$size (page 530), $setIntersection (page 511)

Exclude All Fields at a Given Level A collection accounts contains the following document:

```
{
  _id: 1,
  level: 1,
  acct_id: "xyz123",
  cc: {
        level: 5,
        type: "yy",
        num: 000000000000,
        exp_date: ISODate("2015-11-01T00:00:00.000Z"),
        billing_addr: {
                         level: 5,
                         addr1: "123 ABC Street",
                         city: "Some City"
                       },
        shipping_addr: [
                           {
                             level: 3,
                             addr1: "987 XYZ Ave",
                             city: "Some City"
                           },
                           {
                             level: 3,
                             addr1: "PO Box 0123",
                             city: "Some City"
                           }
                         ]
      },
  status: "A"
}
```

In this example document, the level field determines the access level required to view the data.

To run a query on all documents with status A and exclude *all* fields contained in a document/subdocument at level 5, include a $redact (page 495) stage that specifies the system variable "$$PRUNE" in the then field:

```
db.accounts.aggregate(
   [
     { $match: { status: "A" } },
     { $redact:
        {
            $cond: {
                      if: { $eq: [ "$level", 5 ] },
                      then: "$$PRUNE",
                      else: "$$DESCEND"
                    }
        }
     }
   ]
)
```

The $redact (page 495) stage evaluates the level field to determine access. If the level field equals 5, then exclude all fields at that level, even if the excluded field contains subdocuments that may have different level values, such as the shipping_addr field.

The aggregation operation returns the following "redacted" document:

```
{
  "_id" : 1,
  "level" : 1,
  "acct_id" : "xyz123",
  "status" : "A"
}
```

The result set shows that the $redact (page 495) stage excluded the field cc as a whole, including the shipping_addr field which contained subdocuments that had level field values equal to 3 and not 5.

See also:

http://docs.mongodb.org/manual/tutorial/implement-field-level-redaction for steps to set up multiple combinations of access for the same data.

$skip (aggregation)

Definition

$skip

Skips over the specified number of *documents* that pass into the stage and passes the remaining documents to the next stage in the *pipeline*.

The $skip (page 499) stage has the following prototype form:

```
{ $skip: <positive integer> }
```

$skip (page 499) takes a positive integer that specifies the maximum number of documents to skip.

Example Consider the following example:

```
db.article.aggregate(
    { $skip : 5 }
);
```

This operation skips the first 5 documents passed to it by the pipeline. $skip (page 499) has no effect on the content of the documents it passes along the pipeline.

$sort (aggregation)

$sort

Sorts all input documents and returns them to the pipeline in sorted order.

The $sort (page 499) stage has the following prototype form:

```
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
```

$sort (page 499) takes a document that specifies the field(s) to sort by and the respective sort order. <sort order> can have one of the following values:

- 1 to specify ascending order.
- -1 to specify descending order.
- { $meta: "textScore" } to sort by the computed textScore metadata in descending order. See *Metadata Sort* (page 501) for an example.

Examples

Ascending/Descending Sort To ascending order for a field or fields 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)

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

The comparison treats a non-existent field as it would an empty BSON Object. As such, a sort on the `a` field in documents `{ }` and `{ a: null }` would treat the documents as equivalent in sort order.

With arrays, a less-than comparison or an ascending sort compares the smallest element of arrays, and a greater-than comparison or a descending sort compares the largest element of the arrays. As such, when comparing a field whose value is a single-element array (e.g. `[ 1 ]`) with non-array fields (e.g. `2`), the comparison is between `1` and `2`. A comparison of an empty array (e.g. `[ ]`) treats the empty array as less than `null` or a missing field.

MongoDB sorts `BinData` in the following order:

1. First, the length or size of the data.
2. Then, by the BSON one-byte subtype.
3. Finally, by the data, performing a byte-by-byte comparison.

Metadata Sort Specify in the `{ <sort-key> }` document, a new field name for the computed metadata and specify the `$meta` (page 529) expression as its value, as in the following example:

```
db.users.aggregate(
   [
     { $match: { $text: { $search: "operating" } } },
     { $sort: { score: { $meta: "textScore" }, posts: -1 } }
   ]
)
```

This operation uses the `$text` (page 417) operator to match the documents, and then sorts first by the `"textScore"` metadata and then by descending order of the `posts` field. The specified metadata determines the sort order. For example, the `"textScore"` metadata sorts in descending order. See `$meta` (page 529) for more information on metadata.

`$sort` Operator and Memory

`$sort` + `$limit` Memory Optimization When a `$sort` (page 499) immediately precedes a `$limit` (page 489) in the pipeline, the `$sort` (page 499) operation only maintains the top `n` results as it progresses, where `n` is the specified limit, and MongoDB only needs to store `n` items in memory. This optimization still applies when `allowDiskUse` is `true` and the `n` items exceed the *aggregation memory limit*.

Changed in version 2.4: Before MongoDB 2.4, `$sort` (page 499) would sort all the results in memory, and then limit the results to n results.

Optimizations are subject to change between releases.

`$sort` and Memory Restrictions The `$sort` (page 499) stage has a limit of 100 megabytes of RAM. By default, if the stage exceeds this limit, `$sort` (page 499) will produce an error. To allow for the handling of large datasets, set the `allowDiskUse` option to `true` to enable `$sort` (page 499) operations to write to temporary files. See the `allowDiskUse` option in `db.collection.aggregate()` (page 22) method and the `aggregate` (page 210) command for details.

Changed in version 2.6: The memory limit for `$sort` (page 499) changed from 10 percent of RAM to 100 megabytes of RAM.

`$sort` Operator and Performance `$sort` (page 499) operator can take advantage of an index when placed at the **beginning** of the pipeline or placed **before** the following aggregation operators: `$project` (page 492), `$unwind` (page 501), and `$group` (page 486).

$unwind (aggregation)

Definition

`$unwind`

Deconstructs an array field from the input documents to output a document for *each* element. Each output document is the input document with the value of the array field replaced by the element.

The `$unwind` (page 501) stage has the following prototype form:

```
{ $unwind: <field path> }
```

To specify a field path, prefix the field name with a dollar sign `$` and enclose in quotes.

Behaviors $unwind (page 501) has the following behaviors:

- If a value in the field specified by the field path is *not* an array, db.collection.aggregate() (page 22) generates an error.
- If you specify a path for a field that does not exist in an input document, the pipeline ignores the input document and will not output documents for that input document.
- If the array holds an empty array ([]) in an input document, the pipeline ignores the input document and will not output documents for that input document.

Examples Consider an inventory with the following document:

```
{ "_id" : 1, "item" : "ABC1", sizes: [ "S", "M", "L"] }
```

The following aggregation uses the $unwind (page 501) stage to output a document for each element in the sizes array:

```
db.inventory.aggregate( [ { $unwind : "$sizes" } ] )
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }
```

Each document is identical to the input document except for the value of the sizes field which now holds a value from the original sizes array.

Expression Operators

These expression operators are available to construct *expressions* (page 565) for use in the aggregation pipeline.

Operator expressions are similar to functions that take arguments. In general, these expressions take an array of arguments and have the following form:

```
{ <operator>: [ <argument1>, <argument2> ... ] }
```

If operator accepts a single argument, you can omit the outer array designating the argument list:

```
{ <operator>: <argument> }
```

To avoid parsing ambiguity if the argument is a literal array, you must wrap the literal array in a $literal (page 533) expression or keep the outer array that designates the argument list.

Boolean Operators

Boolean expressions evaluates its argument expressions as booleans and return a boolean as the result.

In addition to the false boolean value, Boolean expression evaluates as false the following: null, 0, and undefined values. The Boolean expression evaluates all other values as true, including non-zero numeric values and arrays.

Boolean Aggregation Operators Boolean expressions evaluates its argument expressions as booleans and return a boolean as the result.

In addition to the `false` boolean value, Boolean expression evaluates as `false` the following: `null`, `0`, and `undefined` values. The Boolean expression evaluates all other values as `true`, including non-zero numeric values and arrays.

Name	Description
`$and` (page 503)	Returns `true` only when *all* its expressions evaluate to `true`. Accepts any number of argument expressions.
`$not` (page 504)	Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression.
`$or` (page 505)	Returns `true` when *any* of its expressions evaluates to `true`. Accepts any number of argument expressions.

$and (aggregation)

`$and`

Evaluates one or more expressions and returns `true` if *all* of the expressions are `true` or if evoked with no argument expressions. Otherwise, `$and` (page 503) returns `false`.

`$and` (page 503) has the following syntax:

```
{ $and: [ <expression1>, <expression2>, ... ] }
```

For more information on expressions, see *Expressions* (page 565).

Behavior `$and` (page 503) uses short-circuit logic: the operation stops evaluation after encountering the first `false` expression.

In addition to the `false` boolean value, `$and` (page 503) evaluates as `false` the following: `null`, `0`, and `undefined` values. The `$and` (page 503) evaluates all other values as `true`, including non-zero numeric values and arrays.

Example	Result
`{ $and: [ 1, "green" ] }`	`true`
`{ $and: [ ] }`	`true`
`{ $and: [ [ null ], [ false ], [ 0 ] ] }`	`true`
`{ $and: [ null, true ] }`	`false`
`{ $and: [ 0, true ] }`	`false`

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the `$and` (page 503) operator to determine if `qty` is greater than 100 *and* less than `250`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            qty: 1,
```

```
            result: { $and: [ { $gt: [ "$qty", 100 ] }, { $lt: [ "$qty", 250 ] } ] }
          }
      }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc1", "result" : false }
{ "_id" : 2, "item" : "abc2", "result" : true }
{ "_id" : 3, "item" : "xyz1", "result" : false }
{ "_id" : 4, "item" : "VWZ1", "result" : false }
{ "_id" : 5, "item" : "VWZ2", "result" : true }
```

$not (aggregation)

$not

Evaluates a boolean and returns the opposite boolean value; i.e. when passed an expression that evaluates to `true`, `$not` (page 504) returns `false`; when passed an expression that evaluates to `false`, `$not` (page 504) returns `true`.

`$not` (page 504) has the following syntax:

```
{ $not: [ <expression> ] }
```

For more information on expressions, see *Expressions* (page 565).

Behavior In addition to the `false` boolean value, `$not` (page 504) evaluates as `false` the following: `null`, `0`, and `undefined` values. The `$not` (page 504) evaluates all other values as `true`, including non-zero numeric values and arrays.

Example		Result
`{ $not: [ true ] }`		`false`
`{ $not: [ [ false ] ] }`		`false`
`{ $not: [ false ] }`		`true`
`{ $not: [ null ] }`		`true`
`{ $not: [ 0 ] }`		`true`

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the `$or` (page 505) operator to determine if `qty` is greater than 250 *or* less than `200`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            result: { $not: [ { $gt: [ "$qty", 250 ] } ] }
          }
     }
```

```
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc1", "result" : false }
{ "_id" : 2, "item" : "abc2", "result" : true }
{ "_id" : 3, "item" : "xyz1", "result" : true }
{ "_id" : 4, "item" : "VWZ1", "result" : false }
{ "_id" : 5, "item" : "VWZ2", "result" : true }
```

$or (aggregation)

$or

Evaluates one or more expressions and returns `true` if *any* of the expressions are `true`. Otherwise, `$or` (page 505) returns `false`.

`$or` (page 505) has the following syntax:

```
{ $or: [ <expression1>, <expression2>, ... ] }
```

For more information on expressions, see *Expressions* (page 565).

Behavior `$or` (page 505) uses short-circuit logic: the operation stops evaluation after encountering the first `true` expression.

In addition to the `false` boolean value, `$or` (page 505) evaluates as `false` the following: `null`, `0`, and `undefined` values. The `$or` (page 505) evaluates all other values as `true`, including non-zero numeric values and arrays.

Example		Result
`{ $or: [ true, false ] }`		`true`
`{ $or: [ [ false ], false ] }`		`true`
`{ $or: [ null, 0, undefined ] }`		`false`
`{ $or: [ ] }`		`false`

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the `$or` (page 505) operator to determine if `qty` is greater than 250 *or* less than `200`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            result: { $or: [ { $gt: [ "$qty", 250 ] }, { $lt: [ "$qty", 200 ] } ] }
          }
     }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc1", "result" : true }
{ "_id" : 2, "item" : "abc2", "result" : false }
{ "_id" : 3, "item" : "xyz1", "result" : false }
{ "_id" : 4, "item" : "VWZ1", "result" : true }
{ "_id" : 5, "item" : "VWZ2", "result" : true }
```

Set Operators

Set expressions performs set operation on arrays, treating arrays as sets. Set expressions ignores the duplicate entries in each input array and the order of the elements.

If the set operation returns a set, the operation filters out duplicates in the result to output an array that contains only unique entries. The order of the elements in the output array is unspecified.

If a set contains a nested array element, the set expression does *not* descend into the nested array but evaluates the array at top-level.

Set Operators (Aggregation) Set expressions performs set operation on arrays, treating arrays as sets. Set expressions ignores the duplicate entries in each input array and the order of the elements.

If the set operation returns a set, the operation filters out duplicates in the result to output an array that contains only unique entries. The order of the elements in the output array is unspecified.

If a set contains a nested array element, the set expression does *not* descend into the nested array but evaluates the array at top-level.

Name	Description
`$allElementsTru` (page 506)	Returns `true` if *no* element of a set evaluates to `false`, otherwise, returns `false`. Accepts a single argument expression.
`$anyElementTrue` (page 507)	Returns `true` if *any* elements of a set evaluate to `true`; otherwise, returns `false`. Accepts a single argument expression.
`$setDifference` (page 509)	Returns a set with elements that appear in the first set but not in the second set; i.e. performs a relative complement[20] of the second set relative to the first. Accepts exactly two argument expressions.
`$setEquals` (page 510)	Returns `true` if the input sets have the same distinct elements. Accepts two or more argument expressions.
`$setIntersectio` (page 511)	Returns a set with elements that appear in *all* of the input sets. Accepts any number of argument expressions.
`$setIsSubset` (page 512)	Returns `true` if all elements of the first set appear in the second set, including when the first set equals the second set; i.e. not a strict subset[21]. Accepts exactly two argument expressions.
`$setUnion` (page 513)	Returns a set with elements that appear in *any* of the input sets. Accepts any number of argument expressions.

$allElementsTrue (aggregation)

`$allElementsTrue`

New in version 2.6.

Evaluates an array as a set and returns `true` if *no* element in the array is `false`. Otherwise, returns `false`. An empty array returns `true`.

[20] http://en.wikipedia.org/wiki/Complement_(set_theory)
[21] http://en.wikipedia.org/wiki/Subset

`$allElementsTrue` (page 506) has the following syntax:

```
{ $allElementsTrue: [ <expression> ] }
```

The `<expression>` itself must resolve to an array, separate from the outer array that denotes the argument list. For more information on expressions, see *Expressions* (page 565).

Behavior If a set contains a nested array element, `$allElementsTrue` (page 506) does *not* descend into the nested array but evaluates the array at top-level.

In addition to the `false` boolean value, `$allElementsTrue` (page 506) evaluates as `false` the following: `null`, `0`, and `undefined` values. The `$allElementsTrue` (page 506) evaluates all other values as `true`, including non-zero numeric values and arrays.

Example		Result
`{ $allElementsTrue: [ [ true, 1, "someString" ] ] }`		`true`
`{ $allElementsTrue: [ [ [ false ] ] ] }`		`true`
`{ $allElementsTrue: [ [ ] ] }`		`true`
`{ $allElementsTrue: [ [ null, false, 0 ] ] }`		`false`

Example Consider an `survey` collection with the following documents:

```
{ "_id" : 1, "responses" : [ true ] }
{ "_id" : 2, "responses" : [ true, false ] }
{ "_id" : 3, "responses" : [ ] }
{ "_id" : 4, "responses" : [ 1, true, "seven" ] }
{ "_id" : 5, "responses" : [ 0 ] }
{ "_id" : 6, "responses" : [ [ ] ] }
{ "_id" : 7, "responses" : [ [ 0 ] ] }
{ "_id" : 8, "responses" : [ [ false ] ] }
{ "_id" : 9, "responses" : [ null ] }
{ "_id" : 10, "responses" : [ undefined ] }
```

The following operation uses the `$allElementsTrue` (page 506) operator to determine if the `responses` array only contains values that evaluate to `true`:

```
db.survey.aggregate(
   [
     { $project: { responses: 1, isAllTrue: { $allElementsTrue: [ "$responses" ] }, _id: 0 } }
   ]
)
```

The operation returns the following results:

```
{ "responses" : [ true ], "isAllTrue" : true }
{ "responses" : [ true, false ], "isAllTrue" : false }
{ "responses" : [ ], "isAllTrue" : true }
{ "responses" : [ 1, true, "seven" ], "isAllTrue" : true }
{ "responses" : [ 0 ], "isAllTrue" : false }
{ "responses" : [ [ ] ], "isAllTrue" : true }
{ "responses" : [ [ 0 ] ], "isAllTrue" : true }
{ "responses" : [ [ false ] ], "isAllTrue" : true }
{ "responses" : [ null ], "isAllTrue" : false }
{ "responses" : [ null ], "isAllTrue" : false }
```

$anyElementTrue (aggregation)

$anyElementTrue

New in version 2.6.

Evaluates an array as a set and returns `true` if any of the elements are `true` and `false` otherwise. An empty array returns `false`.

`$anyElementTrue` (page 507) has the following syntax:

```
{ $anyElementTrue: [ <expression> ] }
```

The `<expression>` itself must resolve to an array, separate from the outer array that denotes the argument list. For more information on expressions, see *Expressions* (page 565).

Behavior If a set contains a nested array element, `$anyElementTrue` (page 507) does *not* descend into the nested array but evaluates the array at top-level.

In addition to the `false` boolean value, `$anyElementTrue` (page 507) evaluates as `false` the following: `null`, `0`, and `undefined` values. The `$anyElementTrue` (page 507) evaluates all other values as `true`, including non-zero numeric values and arrays.

Example		Result
`{ $anyElementTrue: [ [ true, false ] ] }`		`true`
`{ $anyElementTrue: [ [ [ false ] ] ] }`		`true`
`{ $anyElementTrue: [ [ null, false, 0 ] ] }`		`false`
`{ $anyElementTrue: [ [ ] ] }`		`false`

Example Consider an `survey` collection with the following documents:

```
{ "_id" : 1, "responses" : [ true ] }
{ "_id" : 2, "responses" : [ true, false ] }
{ "_id" : 3, "responses" : [ ] }
{ "_id" : 4, "responses" : [ 1, true, "seven" ] }
{ "_id" : 5, "responses" : [ 0 ] }
{ "_id" : 6, "responses" : [ [ ] ] }
{ "_id" : 7, "responses" : [ [ 0 ] ] }
{ "_id" : 8, "responses" : [ [ false ] ] }
{ "_id" : 9, "responses" : [ null ] }
{ "_id" : 10, "responses" : [ undefined ] }
```

The following operation uses the `$anyElementTrue` (page 507) operator to determine if the `responses` array contains any value that evaluates to `true`:

```
db.survey.aggregate(
   [
     { $project: { responses: 1, isAnyTrue: { $anyElementTrue: [ "$responses" ] }, _id: 0 } }
   ]
)
```

The operation returns the following results:

```
{ "responses" : [ true ], "isAnyTrue" : true }
{ "responses" : [ true, false ], "isAnyTrue" : true }
{ "responses" : [ ], "isAnyTrue" : false }
{ "responses" : [ 1, true, "seven" ], "isAnyTrue" : true }
{ "responses" : [ 0 ], "isAnyTrue" : false }
{ "responses" : [ [ ] ], "isAnyTrue" : true }
{ "responses" : [ [ 0 ] ], "isAnyTrue" : true }
{ "responses" : [ [ false ] ], "isAnyTrue" : true }
```

```
{ "responses" : [ null ], "isAnyTrue" : false }
{ "responses" : [ null ], "isAnyTrue" : false }
```

$setDifference (aggregation)

$setDifference

New in version 2.6.

Takes two sets and returns an array containing the elements that only exist in the first set; i.e. performs a relative complement[22] of the second set relative to the first.

`$setDifference` (page 509) has the following syntax:

```
{ $setDifference: [ <expression1>, <expression2> ] }
```

The arguments can be any valid *expression* (page 565) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 565).

Behavior `$setDifference` (page 509) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setDifference` (page 509) ignores the duplicate entries. `$setDifference` (page 509) ignores the order of the elements.

`$setDifference` (page 509) filters out duplicates in its result to output an array that contain only unique entries. The order of the elements in the output array is unspecified.

If a set contains a nested array element, `$setDifference` (page 509) does *not* descend into the nested array but evaluates the array at top-level.

Example		Result
`{ $setDifference: [ [ "a", "b", "a" ], [ "b", "a" ] ] }`		`[ ]`
`{ $setDifference: [ [ "a", "b" ], [ [ "a", "b" ] ] ] }`		`[ "a", "b" ]`

Example Consider an `experiments` collection with the following documents:

```
{ "_id" : 1, "A" : [ "red", "blue" ], "B" : [ "red", "blue" ] }
{ "_id" : 2, "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ] }
{ "_id" : 3, "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ] }
{ "_id" : 4, "A" : [ "red", "blue" ], "B" : [ "green", "red" ] }
{ "_id" : 5, "A" : [ "red", "blue" ], "B" : [ ] }
{ "_id" : 6, "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ] }
{ "_id" : 7, "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : [ "red" ] }
```

The following operation uses the `$setDifference` (page 509) operator to return an array of elements found in the `B` array but *not* in the `A` array:

```
db.experiments.aggregate(
   [
     { $project: { A: 1, B: 1, inBOnly: { $setDifference: [ "$B", "$A" ] }, _id: 0 } }
   ]
)
```

The operation returns the following results:

[22] http://en.wikipedia.org/wiki/Complement_(set_theory)

```
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue" ], "inBOnly" : [ ] }
{ "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ], "inBOnly" : [ ] }
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ], "inBOnly" : [ "green" ] }
{ "A" : [ "red", "blue" ], "B" : [ "green", "red" ], "inBOnly" : [ "green" ] }
{ "A" : [ "red", "blue" ], "B" : [ ], "inBOnly" : [ ] }
{ "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ], "inBOnly" : [ [ "red" ], [ "blue" ] ] }
{ "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ], "inBOnly" : [ [ "red", "blue" ] ] }
{ "A" : [ ], "B" : [ ], "inBOnly" : [ ] }
{ "A" : [ ], "B" : [ "red" ], "inBOnly" : [ "red" ] }
```

$setEquals (aggregation)

`$setEquals`

New in version 2.6.

Compares two or more arrays and returns `true` if they have the same distinct elements and `false` otherwise.

`$setEquals` (page 510) has the following syntax:

```
{ $setEquals: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 565) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 565).

Behavior `$setEquals` (page 510) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setEquals` (page 510) ignores the duplicate entries. `$setEquals` (page 510) ignores the order of the elements.

If a set contains a nested array element, `$setEquals` (page 510) does *not* descend into the nested array but evaluates the array at top-level.

Example		Result
`{ $setEquals: [ [ "a", "b", "a" ], [ "b", "a" ] ] }`		`true`
`{ $setEquals: [ [ "a", "b" ], [ [ "a", "b" ] ] ] }`		`false`

Example Consider an `experiments` collection with the following documents:

```
{ "_id" : 1, "A" : [ "red", "blue" ], "B" : [ "red", "blue" ] }
{ "_id" : 2, "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ] }
{ "_id" : 3, "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ] }
{ "_id" : 4, "A" : [ "red", "blue" ], "B" : [ "green", "red" ] }
{ "_id" : 5, "A" : [ "red", "blue" ], "B" : [ ] }
{ "_id" : 6, "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ] }
{ "_id" : 7, "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : [ "red" ] }
```

The following operation uses the `$setEquals` (page 510) operator to determine if the `A` array and the `B` array contain the same elements:

```
db.experiments.aggregate(
   [
     { $project: { A: 1, B: 1, sameElements: { $setEquals: [ "$A", "$B" ] }, _id: 0 } }
   ]
)
```

The operation returns the following results:

```
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue" ], "sameElements" : true }
{ "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ], "sameElements" : true }
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ], "sameElements" : false }
{ "A" : [ "red", "blue" ], "B" : [ "green", "red" ], "sameElements" : false }
{ "A" : [ "red", "blue" ], "B" : [ ], "sameElements" : false }
{ "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ], "sameElements" : false }
{ "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ], "sameElements" : false }
{ "A" : [ ], "B" : [ ], "sameElements" : true }
{ "A" : [ ], "B" : [ "red" ], "sameElements" : false }
```

$setIntersection (aggregation)

`$setIntersection`

New in version 2.6.

Takes two or more arrays and returns an array that contains the elements that appear in every input array.

`$setIntersection` (page 511) has the following syntax:

```
{ $setIntersection: [ <array1>, <array2>, ... ] }
```

The arguments can be any valid *expression* (page 565) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 565).

Behavior `$setIntersection` (page 511) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setIntersection` (page 511) ignores the duplicate entries. `$setIntersection` (page 511) ignores the order of the elements.

`$setIntersection` (page 511) filters out duplicates in its result to output an array that contain only unique entries. The order of the elements in the output array is unspecified.

If a set contains a nested array element, `$setIntersection` (page 511) does *not* descend into the nested array but evaluates the array at top-level.

Example		Result
`{ $setIntersection: [ [ "a", "b", "a" ], [ "b", "a" ] ] }`		`[ "b", "a" ]`
`{ $setIntersection: [ [ "a", "b" ], [ [ "a", "b" ] ] ] }`		`[ ]`

Example Consider an `experiments` collection with the following documents:

```
{ "_id" : 1, "A" : [ "red", "blue" ], "B" : [ "red", "blue" ] }
{ "_id" : 2, "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ] }
{ "_id" : 3, "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ] }
{ "_id" : 4, "A" : [ "red", "blue" ], "B" : [ "green", "red" ] }
{ "_id" : 5, "A" : [ "red", "blue" ], "B" : [ ] }
{ "_id" : 6, "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ] }
{ "_id" : 7, "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : [ "red" ] }
```

The following operation uses the `$setIntersection` (page 511) operator to return an array of elements common to both the `A` array and the `B` array:

```
db.experiments.aggregate(
   [
     { $project: { A: 1, B: 1, commonToBoth: { $setIntersection: [ "$A", "$B" ] }, _id: 0 } }
   ]
)
```

The operation returns the following results:

```
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue" ], "commonToBoth" : [ "blue", "red" ] }
{ "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ], "commonToBoth" : [ "blue", "red" ] }
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ], "commonToBoth" : [ "blue", "red" ] }
{ "A" : [ "red", "blue" ], "B" : [ "green", "red" ], "commonToBoth" : [ "red" ] }
{ "A" : [ "red", "blue" ], "B" : [ ], "commonToBoth" : [ ] }
{ "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ], "commonToBoth" : [ ] }
{ "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ], "commonToBoth" : [ ] }
{ "A" : [ ], "B" : [ ], "commonToBoth" : [ ] }
{ "A" : [ ], "B" : [ "red" ], "commonToBoth" : [ ] }
```

$setIsSubset (aggregation)

`$setIsSubset`

New in version 2.6.

Takes two arrays and returns `true` when the first array is a subset of the second, including when the first array equals the second array, and `false` otherwise.

`$setIsSubset` (page 512) has the following syntax:

```
{ $setIsSubset: [ <expression1>, <expression2> ] }
```

The arguments can be any valid *expression* (page 565) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 565).

Behavior `$setIsSubset` (page 512) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setIsSubset` (page 512) ignores the duplicate entries. `$setIsSubset` (page 512) ignores the order of the elements.

If a set contains a nested array element, `$setIsSubset` (page 512) does *not* descend into the nested array but evaluates the array at top-level.

Example		Result
`{ $setIsSubset: [ [ "a", "b", "a" ], [ "b", "a" ] ] }`		`true`
`{ $setIsSubset: [ [ "a", "b" ], [ [ "a", "b" ] ] ] }`		`false`

Example Consider an `experiments` collection with the following documents:

```
{ "_id" : 1, "A" : [ "red", "blue" ], "B" : [ "red", "blue" ] }
{ "_id" : 2, "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ] }
{ "_id" : 3, "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ] }
{ "_id" : 4, "A" : [ "red", "blue" ], "B" : [ "green", "red" ] }
{ "_id" : 5, "A" : [ "red", "blue" ], "B" : [ ] }
{ "_id" : 6, "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ] }
{ "_id" : 7, "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : [ "red" ] }
```

The following operation uses the `$setIsSubset` (page 512) operator to determine if the `A` array is a subset of the `B` array:

```
db.experiments.aggregate(
   [
     { $project: { A:1, B: 1, AisSubset: { $setIsSubset: [ "$A", "$B" ] }, _id:0 } }
   ]
)
```

The operation returns the following results:

```
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue" ], "AisSubset" : true }
{ "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ], "AisSubset" : true }
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ], "AisSubset" : true }
{ "A" : [ "red", "blue" ], "B" : [ "green", "red" ], "AisSubset" : false }
{ "A" : [ "red", "blue" ], "B" : [ ], "AisSubset" : false }
{ "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ], "AisSubset" : false }
{ "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ], "AisSubset" : false }
{ "A" : [ ], "B" : [ ], "AisSubset" : true }
{ "A" : [ ], "B" : [ "red" ], "AisSubset" : true }
```

$setUnion (aggregation)

`$setUnion`

New in version 2.6.

Takes two or more arrays and returns an array containing the elements that appear in any input array.

`$setUnion` (page 513) has the following syntax:

```
{ $setUnion: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 565) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 565).

Behavior `$setUnion` (page 513) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setUnion` (page 513) ignores the duplicate entries. `$setUnion` (page 513) ignores the order of the elements.

`$setUnion` (page 513) filters out duplicates in its result to output an array that contain only unique entries. The order of the elements in the output array is unspecified.

If a set contains a nested array element, `$setUnion` (page 513) does *not* descend into the nested array but evaluates the array at top-level.

Example		Result
`{ $setUnion: [ [ "a", "b", "a" ], [ "b", "a" ] ] }`		`[ "b", "a" ]`
`{ $setUnion: [ [ "a", "b" ], [ [ "a", "b" ] ] ] }`		`[ [ "a", "b" ], "b", "a" ]`

Example Consider an `experiments` collection with the following documents:

```
{ "_id" : 1, "A" : [ "red", "blue" ], "B" : [ "red", "blue" ] }
{ "_id" : 2, "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ] }
{ "_id" : 3, "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ] }
{ "_id" : 4, "A" : [ "red", "blue" ], "B" : [ "green", "red" ] }
{ "_id" : 5, "A" : [ "red", "blue" ], "B" : [ ] }
{ "_id" : 6, "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ] }
{ "_id" : 7, "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : [ "red" ] }
```

The following operation uses the `$setUnion` (page 513) operator to return an array of elements found in the `A` array or the `B` array or both:

```
db.experiments.aggregate(
   [
     { $project: { A:1, B: 1, allValues: { $setUnion: [ "$A", "$B" ] }, _id: 0 } }
   ]
)
```

The operation returns the following results:

```
{ "A": [ "red", "blue" ], "B": [ "red", "blue" ], "allValues": [ "blue", "red" ] }
{ "A": [ "red", "blue" ], "B": [ "blue", "red", "blue" ], "allValues": [ "blue", "red" ] }
{ "A": [ "red", "blue" ], "B": [ "red", "blue", "green" ], "allValues": [ "blue", "red", "green" ] }
{ "A": [ "red", "blue" ], "B": [ "green", "red" ], "allValues": [ "blue", "red", "green" ] }
{ "A": [ "red", "blue" ], "B": [ ], "allValues": [ "blue", "red" ] }
{ "A": [ "red", "blue" ], "B": [ [ "red" ], [ "blue" ] ], "allValues": [ "blue", "red", [ "red" ], [
{ "A": [ "red", "blue" ], "B": [ [ "red", "blue" ] ], "allValues": [ "blue", "red", [ "red", "blue"
{ "A": [ ], "B": [ ], "allValues": [ ] }
{ "A": [ ], "B": [ "red" ], "allValues": [ "red" ] }
```

Comparison Operators

Comparison expressions return a boolean except for `$cmp` (page 514) which returns a number.

The comparison expressions take two argument expressions and compare both value and type, using the *specified BSON comparison order* for values of different types.

Comparison Aggregation Operators Comparison expressions return a boolean except for `$cmp` (page 514) which returns a number.

The comparison expressions take two argument expressions and compare both value and type, using the *specified BSON comparison order* for values of different types.

Name	Description
`$cmp` (page 514)	Returns: `0` if the two values are equivalent, `1` if the first value is greater than the second, and `-1` if the first value is less than the second.
`$eq` (page 515)	Returns `true` if the values are equivalent.
`$gt` (page 516)	Returns `true` if the first value is greater than the second.
`$gte` (page 517)	Returns `true` if the first value is greater than or equal to the second.
`$lt` (page 518)	Returns `true` if the first value is less than the second.
`$lte` (page 519)	Returns `true` if the first value is less than or equal to the second.
`$ne` (page 519)	Returns `true` if the values are *not* equivalent.

$cmp (aggregation)

`$cmp`

Compares two values and returns:

- `-1` if the first value is less than the second.
- `1` if the first value is greater than the second.
- `0` if the two values are equivalent.

The $cmp (page 514) compares both value and type, using the *specified BSON comparison order* for values of different types.

$cmp (page 514) has the following syntax:

```
{ $cmp: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the $cmp (page 514) operator to compare the `qty` value with `250`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            qty: 1,
            cmpTo250: { $cmp: [ "$qty", 250 ] },
            _id: 0
          }
     }
   ]
)
```

The operation returns the following results:

```
{ "item" : "abc1", "qty" : 300, "cmpTo250" : 1 }
{ "item" : "abc2", "qty" : 200, "cmpTo250" : -1 }
{ "item" : "xyz1", "qty" : 250, "cmpTo250" : 0 }
{ "item" : "VWZ1", "qty" : 300, "cmpTo250" : 1 }
{ "item" : "VWZ2", "qty" : 180, "cmpTo250" : -1 }
```

$eq (aggregation)

$eq

Compares two values and returns:

- `true` when the values are equivalent.
- `false` when the values are **not** equivalent.

The $eq (page 515) compares both value and type, using the *specified BSON comparison order* for values of different types.

$eq (page 515) has the following syntax:

```
{ $eq: [ <expression1>, <expression2> ] }
```

The arguments can be any valid *expression* (page 565). For more information on expressions, see *Expressions* (page 565).

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the `$eq` (page 515) operator to determine if `qty` equals `250`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            qty: 1,
            qtyEq250: { $eq: [ "$qty", 250 ] },
            _id: 0
          }
     }
   ]
)
```

The operation returns the following results:

```
{ "item" : "abc1", "qty" : 300, "qtyEq250" : false }
{ "item" : "abc2", "qty" : 200, "qtyEq250" : false }
{ "item" : "xyz1", "qty" : 250, "qtyEq250" : true }
{ "item" : "VWZ1", "qty" : 300, "qtyEq250" : false }
{ "item" : "VWZ2", "qty" : 180, "qtyEq250" : false }
```

$gt (aggregation)

`$gt`

Compares two values and returns:

- `true` when the first value is *greater than* the second value.
- `false` when the first value is *less than or equivalent to* the second value.

The `$gt` (page 516) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$gt` (page 516) has the following syntax:

```
{ $gt: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the `$gt` (page 516) operator to determine if `qty` is greater than `250`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            qty: 1,
            qtyGt250: { $gt: [ "$qty", 250 ] },
            _id: 0
          }
     }
   ]
)
```

The operation returns the following results:

```
{ "item" : "abc1", "qty" : 300, "qtyGt250" : true }
{ "item" : "abc2", "qty" : 200, "qtyGt250" : false }
{ "item" : "xyz1", "qty" : 250, "qtyGt250" : false }
{ "item" : "VWZ1", "qty" : 300, "qtyGt250" : true }
{ "item" : "VWZ2", "qty" : 180, "qtyGt250" : false }
```

$gte (aggregation)

$gte

Compares two values and returns:

- `true` when the first value is *greater than or equivalent* to the second value.
- `false` when the first value is *less than* the second value.

The `$gte` (page 517) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$gte` (page 517) has the following syntax:

```
{ $gte: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the `$gte` (page 517) operator to determine if `qty` is greater than or equal to `250`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            qty: 1,
            qtyGte250: { $gte: [ "$qty", 250 ] },
            _id: 0
```

```
            }
      }
   ]
)
```

The operation returns the following results:

```
{ "item" : "abc1", "qty" : 300, "qtyGte250" : true }
{ "item" : "abc2", "qty" : 200, "qtyGte250" : false }
{ "item" : "xyz1", "qty" : 250, "qtyGte250" : true }
{ "item" : "VWZ1", "qty" : 300, "qtyGte250" : true }
{ "item" : "VWZ2", "qty" : 180, "qtyGte250" : false }
```

$lt (aggregation)

$lt

Compares two values and returns:

- `true` when the first value is *less than* the second value.
- `false` when the first value is *greater than or equivalent to* the second value.

The `$lt` (page 518) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$lt` (page 518) has the following syntax:

```
{ $lt: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the `$lt` (page 518) operator to determine if `qty` is less than `250`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            qty: 1,
            qtyLt250: { $lt: [ "$qty", 250 ] },
            _id: 0
          }
     }
   ]
)
```

The operation returns the following results:

```
{ "item" : "abc1", "qty" : 300, "qtyLt250" : false }
{ "item" : "abc2", "qty" : 200, "qtyLt250" : true }
{ "item" : "xyz1", "qty" : 250, "qtyLt250" : false }
```

```
{ "item" : "VWZ1", "qty" : 300, "qtyLt250" : false }
{ "item" : "VWZ2", "qty" : 180, "qtyLt250" : true }
```

$lte (aggregation)

$lte

Compares two values and returns:

- `true` when the first value is *less than or equivalent to* the second value.
- `false` when the first value is *greater than* the second value.

The `$lte` (page 519) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$lte` (page 519) has the following syntax:

```
{ $lte: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the `$lte` (page 519) operator to determine if `qty` is less than or equal to `250`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            qty: 1,
            qtyLte250: { $lte: [ "$qty", 250 ] },
            _id: 0
          }
     }
   ]
)
```

The operation returns the following results:

```
{ "item" : "abc1", "qty" : 300, "qtyLte250" : false }
{ "item" : "abc2", "qty" : 200, "qtyLte250" : true }
{ "item" : "xyz1", "qty" : 250, "qtyLte250" : true }
{ "item" : "VWZ1", "qty" : 300, "qtyLte250" : false }
{ "item" : "VWZ2", "qty" : 180, "qtyLte250" : true }
```

$ne (aggregation)

$ne

Compares two values and returns:

- `true` when the values are not equivalent.

- `false` when the values are equivalent.

The `$lte` (page 519) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$ne` (page 519) has the following syntax:

```
{ $ne: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider an `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

The following operation uses the `$ne` (page 519) operator to determine if `qty` does not equal `250`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            qty: 1,
            qtyNe250: { $ne: [ "$qty", 250 ] },
            _id: 0
          }
     }
   ]
)
```

The operation returns the following results:

```
{ "item" : "abc1", "qty" : 300, "qtyNe250" : true }
{ "item" : "abc2", "qty" : 200, "qtyNe250" : true }
{ "item" : "xyz1", "qty" : 250, "qtyNe250" : false }
{ "item" : "VWZ1", "qty" : 300, "qtyNe250" : true }
{ "item" : "VWZ2", "qty" : 180, "qtyNe250" : true }
```

Arithmetic Operators

Arithmetic expressions perform mathematic operations on numbers. Some arithmetic expressions can also support date arithmetic.

Arithmetic Aggregation Operators Arithmetic expressions perform mathematic operations on numbers. Some arithmetic expressions can also support date arithmetic.

Name	Description
$add (page 521)	Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.
$divide (page 522)	Returns the result of dividing the first number by the second. Accepts two argument expressions.
$mod (page 522)	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
$multiply (page 523)	Multiplies numbers to return the product. Accepts any number of argument expressions.
$subtract (page 523)	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds. If the two values are a date and a number in milliseconds, return the resulting date. Accepts two argument expressions. If the two values are a date and a number, specify the date argument first as it is not meaningful to subtract a date from a number.

$add (aggregation)

$add

Adds numbers together or adds numbers and a date. If one of the arguments is a date, $add (page 521) treats the other arguments as milliseconds to add to the date.

The $add (page 521) expression has the following syntax:

```
{ $add: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 565) as long as they resolve to either all numbers or to numbers and a date. For more information on expressions, see *Expressions* (page 565).

Examples The following examples use a sales collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "fee" : 2, date: ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "fee" : 1, date: ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5,  "fee" : 0, date: ISODate("2014-03-15T09:00:00Z") }
```

Add Numbers The following aggregation uses the $add (page 521) expression in the $project (page 492) pipeline to calculate the total cost:

```
db.sales.aggregate(
   [
     { $project: { item: 1, total: { $add: [ "$price", "$fee" ] } } }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc", "total" : 12 }
{ "_id" : 2, "item" : "jkl", "total" : 21 }
{ "_id" : 3, "item" : "xyz", "total" : 5 }
```

Perform Addition on a Date The following aggregation uses the $add (page 521) expression to compute the billing_date by adding 3*24*60*60000 milliseconds (i.e. 3 days) to the date field :

```
db.sales.aggregate(
   [
     { $project: { item: 1, billing_date: { $add: [ "$date", 3*24*60*60000 ] } } }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc", "billing_date" : ISODate("2014-03-04T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "billing_date" : ISODate("2014-03-04T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "billing_date" : ISODate("2014-03-18T09:00:00Z") }
```

$divide (aggregation)

$divide

Divides one number by another and returns the result. Pass the arguments to $divide (page 522) in an array.

The $divide (page 522) expression has the following syntax:

```
{ $divide: [ <expression1>, <expression2> ] }
```

The first argument is the dividend, and the second argument is the divisor; i.e. the first argument is divided by the second argument.

The arguments can be any valid *expression* (page 565) as long as the resolve to numbers. For more information on expressions, see *Expressions* (page 565).

Examples Consider a `planning` collection with the following documents:

```
{ "_id" : 1, "name" : "A", "hours" : 80, "resources" : 7 },
{ "_id" : 2, "name" : "B", "hours" : 40, "resources" : 4 }
```

The following aggregation uses the $divide (page 522) expression to divide the `hours` field by a literal `8` to compute the number of work days:

```
db.planning.aggregate(
   [
     { $project: { name: 1, workdays: { $divide: [ "$hours", 8 ] } } }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "name" : "A", "workdays" : 10 }
{ "_id" : 2, "name" : "B", "workdays" : 5 }
```

$mod (aggregation)

$mod

Divides one number by another and returns the *remainder*.

The $mod (page 522) expression has the following syntax:

```
{ $mod: [ <expression1>, <expression2> ] }
```

The first argument is the dividend, and the second argument is the divisor; i.e. first argument is divided by the second argument.

The arguments can be any valid *expression* (page 565) as long as they resolve to numbers. For more information on expressions, see *Expressions* (page 565).

Example Consider a `planning` collection with the following documents:

```
{ "_id" : 1, "project" : "A", "hours" : 80, "tasks" : 7 }
{ "_id" : 2, "project" : "B", "hours" : 40, "tasks" : 4 }
```

The following aggregation uses the `$mod` (page 522) expression to return the remainder of the `hours` field divided by the `tasks` field:

```
db.planning.aggregate(
   [
     { $project: { remainder: { $mod: [ "$hours", "$tasks" ] } } }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "remainder" : 3 }
{ "_id" : 2, "remainder" : 0 }
```

$multiply (aggregation)

`$multiply`

Multiplies numbers together and returns the result. Pass the arguments to `$multiply` (page 523) in an array.

The `$multiply` (page 523) expression has the following syntax:

```
{ $multiply: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 565) as long as they resolve to numbers. For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity": 2, date: ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity": 1, date: ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity": 10, date: ISODate("2014-03-15T09:00:00Z") }
```

The following aggregation uses the `$multiply` (page 523) expression in the `$project` (page 492) pipeline to multiply the `price` and the `quantity` fields:

```
db.sales.aggregate(
   [
     { $project: { date: 1, item: 1, total: { $multiply: [ "$price", "$quantity" ] } } }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc", "date" : ISODate("2014-03-01T08:00:00Z"), "total" : 20 }
{ "_id" : 2, "item" : "jkl", "date" : ISODate("2014-03-01T09:00:00Z"), "total" : 20 }
{ "_id" : 3, "item" : "xyz", "date" : ISODate("2014-03-15T09:00:00Z"), "total" : 50 }
```

$subtract (aggregation)

`$subtract`

Subtracts two numbers to return the difference, or two dates to return the difference in milliseconds, or a date and a number in milliseconds to return the resulting date.

The `$subtract` (page 523) expression has the following syntax:

```
{ $subtract: [ <expression1>, <expression2> ] }
```

The second argument is subtracted from the first argument.

The arguments can be any valid *expression* (page 565) as long as they resolve to numbers and/or dates. To subtract a number from a date, the date must be the first argument. For more information on expressions, see *Expressions* (page 565).

Examples Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "fee" : 2, "discount" : 5, "date" : ISODate("2014-03-01T08
{ "_id" : 2, "item" : "jkl", "price" : 20, "fee" : 1, "discount" : 2, "date" : ISODate("2014-03-01T09
```

Subtract Numbers The following aggregation uses the `$subtract` (page 523) expression to compute the `total` by subtracting the `discount` from the subtotal of `price` and `fee`.

```
db.sales.aggregate( [ { $project: { item: 1, total: { $subtract: [ { $add: [ "$price", "$fee" ] }, "$
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc", "total" : 7 }
{ "_id" : 2, "item" : "jkl", "total" : 19 }
```

Subtract Two Dates The following aggregation uses the `$subtract` (page 523) expression to subtract `$date` from the current date:

```
db.sales.aggregate( [ { $project: { item: 1, dateDifference: { $subtract: [ new Date(), "$date" ] } }
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc", "dateDifference" : NumberLong("11713985194") }
{ "_id" : 2, "item" : "jkl", "dateDifference" : NumberLong("11710385194") }
```

Subtract Milliseconds from a Date The following aggregation uses the `$subtract` (page 523) expression to subtract 5 * 60 * 1000 milliseconds (5 minutes) from the "$date" field:

```
db.sales.aggregate( [ { $project: { item: 1, dateDifference: { $subtract: [ "$date", 5 * 60 * 1000 ]
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc", "dateDifference" : ISODate("2014-03-01T07:55:00Z") }
{ "_id" : 2, "item" : "jkl", "dateDifference" : ISODate("2014-03-01T08:55:00Z") }
```

String Operators

String expressions, with the exception of `$concat` (page 525), only have a well-defined behavior for strings of ASCII characters.

`$concat` (page 525) behavior is well-defined regardless of the characters used.

String Aggregation Operators String expressions, with the exception of `$concat` (page 525), only have a well-defined behavior for strings of ASCII characters.

`$concat` (page 525) behavior is well-defined regardless of the characters used.

Name	Description
`$concat` (page 525)	Concatenates any number of strings.
`$strcasecmp` (page 525)	Performs case-insensitive string comparison and returns: `0` if two strings are equivalent, `1` if the first string is greater than the second, and `-1` if the first string is less than the second.
`$substr` (page 526)	Returns a substring of a string, starting at a specified index position up to a specified length. Accepts three expressions as arguments: the first argument must resolve to a string, and the second and third arguments must resolve to integers.
`$toLower` (page 527)	Converts a string to lowercase. Accepts a single argument expression.
`$toUpper` (page 528)	Converts a string to uppercase. Accepts a single argument expression.

$concat (aggregation)

`$concat`

New in version 2.4.

Concatenates strings and returns the concatenated string.

`$concat` (page 525) has the following syntax:

```
{ $concat: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 565) as long as they resolve to strings. For more information on expressions, see *Expressions* (page 565).

If the argument resolves to a value of `null` or refers to a field that is missing, `$concat` (page 525) returns `null`.

Examples Consider a `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2" }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
```

The following operation uses the `$concat` (page 525) operator to concatenate the `item` field and the `description` field with a ‘‘ - ‘‘ delimiter.

```
db.inventory.aggregate(
   [
      { $project: { itemDescription: { $concat: [ "$item", " - ", "$description" ] } } }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "itemDescription" : "ABC1 - product 1" }
{ "_id" : 2, "itemDescription" : "ABC2 - product 2" }
{ "_id" : 3, "itemDescription" : null }
```

$strcasecmp (aggregation)

$strcasecmp

Performs case-insensitive comparison of two strings. Returns

- 1 if first string is "greater than" the second string.
- 0 if the two strings are equal.
- -1 if the first string is "less than" the second string.

$strcasecmp (page 525) has the following syntax:

```
{ $strcasecmp: [ <expression1>, <expression2> ] }
```

The arguments can be any valid *expression* (page 565) as long as they resolve to strings. For more information on expressions, see *Expressions* (page 565).

Behavior $strcasecmp (page 525) only has a well-defined behavior for strings of ASCII characters.

For a case sensitive comparison, see $cmp (page 514).

Example Consider a `inventory` collection with the following documents:

```
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2" }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
```

The following operation uses the $strcasecmp (page 525) operator to perform case-insensitive comparison of the `quarter` field value to the string `"13q3"`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            comparisonResult: { $strcasecmp: [ "$quarter", "13q3" ] }
          }
      }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "ABC1", "comparisonResult" : -1 }
{ "_id" : 2, "item" : "ABC2", "comparisonResult" : 0 }
{ "_id" : 3, "item" : "XYZ1", "comparisonResult" : 1 }
```

$substr (aggregation)

$substr

Returns a substring of a string, starting at a specified index position and including the specified number of characters. The index is zero-based.

$substr (page 526) has the following syntax:

```
{ $substr: [ <string>, <start>, <length> ] }
```

The arguments can be any valid *expression* (page 565) as long as long as the first argument resolves to a string, and the second and third arguments resolve to integers. For more information on expressions, see *Expressions* (page 565).

Behavior If <start> is a negative number, $substr (page 526) returns an empty string "".

If <length> is a negative number, $substr (page 526) returns a substring that starts at the specified index and includes the rest of the string.

$substr (page 526) is not encoding aware and if used improperly may produce a result string containing an invalid UTF-8 character sequence.

$substr (page 526) only has a well-defined behavior for strings of ASCII characters.

Example Consider an inventory collection with the following documents:

```
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2" }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
```

The following operation uses the $substr (page 526) operator separate the quarter value into a yearSubstring and a quarterSubstring:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            yearSubstring: { $substr: [ "$quarter", 0, 2 ] },
            quarterSubtring: { $substr: [ "$quarter", 2, -1 ] }
          }
      }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "ABC1", "yearSubstring" : "13", "quarterSubtring" : "Q1" }
{ "_id" : 2, "item" : "ABC2", "yearSubstring" : "13", "quarterSubtring" : "Q4" }
{ "_id" : 3, "item" : "XYZ1", "yearSubstring" : "14", "quarterSubtring" : "Q2" }
```

$toLower (aggregation)

$toLower

Converts a string to lowercase, returning the result.

$toLower (page 527) has the following syntax:

```
{ $toLower: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a string. For more information on expressions, see *Expressions* (page 565).

If the argument resolves to null, $toLower (page 527) returns an empty string "".

Behavior $toLower (page 527) only has a well-defined behavior for strings of ASCII characters.

Example Consider a inventory collection with the following documents:

```
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "PRODUCT 1" }
{ "_id" : 2, "item" : "abc2", quarter: "13Q4", "description" : "Product 2" }
{ "_id" : 3, "item" : "xyz1", quarter: "14Q2", "description" : null }
```

The following operation uses the $toLower (page 527) operator return lowercase item and lowercase description value:

```
db.inventory.aggregate(
   [
     {
       $project:
         {
           item: { $toLower: "$item" },
           description: { $toLower: "$description" }
         }
     }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc1", "description" : "product 1" }
{ "_id" : 2, "item" : "abc2", "description" : "product 2" }
{ "_id" : 3, "item" : "xyz1", "description" : "" }
```

$toUpper (aggregation)

$toUpper

Converts a string to uppercase, returning the result.

$toUpper (page 528) has the following syntax:

```
{ $toUpper: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a string. For more information on expressions, see *Expressions* (page 565).

If the argument resolves to null, $toLower (page 527) returns an empty string "".

Behavior $toUpper (page 528) only has a well-defined behavior for strings of ASCII characters.

Example Consider a inventory collection with the following documents:

```
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "PRODUCT 1" }
{ "_id" : 2, "item" : "abc2", quarter: "13Q4", "description" : "Product 2" }
{ "_id" : 3, "item" : "xyz1", quarter: "14Q2", "description" : null }
```

The following operation uses the $toUpper (page 528) operator return uppercase item and uppercase description values:

```
db.inventory.aggregate(
   [
     {
       $project:
         {
           item: { $toUpper: "$item" },
           description: { $toUpper: "$description" }
         }
     }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "ABC1", "description" : "PRODUCT 1" }
{ "_id" : 2, "item" : "ABC2", "description" : "PRODUCT 2" }
{ "_id" : 3, "item" : "XYZ1", "description" : "" }
```

Text Search Operators

Text Search Aggregation Operators

Name	Description
$meta (page 529)	Access text search metadata.

$meta (aggregation)

$meta

New in version 2.6.

Returns the metadata associated with a document in a pipeline operations, e.g. "textScore" when performing text search.

A $meta (page 529) expression has the following syntax:

```
{ $meta: <metaDataKeyword> }
```

The $meta (page 529) expression can specify the following keyword as the <metaDataKeyword>:

Keyword	Description	Sort Order
"textScore"	Returns the score associated with the corresponding $text (page 417) query for each matching document. The text score signifies how well the document matched the stemmed term or terms. If not used in conjunction with a $text (page 417) query, returns a score of null.	Descending

Behavior The { $meta: "textScore" } expression is the only *expression* (page 565) that the $sort (page 499) stage accepts.

Although available for use everywhere expressions are accepted in the pipeline, the { $meta: "textScore" } expression is only meaningful in a pipeline that includes a $match (page 490) stage with a $text (page 417) query.

Examples Consider an articles collection with the following documents:

```
{ "_id" : 1, "title" : "cakes and ale" }
{ "_id" : 2, "title" : "more cakes" }
{ "_id" : 3, "title" : "bread" }
{ "_id" : 4, "title" : "some cakes" }
```

The following aggregation operation performs a text search and use the $meta (page 529) operator to group by the text search score:

```
db.articles.aggregate(
   [
     { $match: { $text: { $search: "cake" } } },
     { $group: { _id: { $meta: "textScore" }, count: { $sum: 1 } } }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 0.75, "count" : 1 }
{ "_id" : 1, "count" : 2 }
```

For more examples, see http://docs.mongodb.org/manual/tutorial/text-search-in-aggregation.

Array Operators

Array Aggregation Operators

Name	Description
$size (page 530)	Returns the number of elements in the array. Accepts a single expression as ar

$size (aggregation) New in version 2.6.

Definition

$size

Counts and returns the total the number of items in an array.

$size (page 530) has the following syntax:

```
{ $size: <expression> }
```

The argument for $size (page 530) can be any *expression* (page 565) as long as it resolves to an array. For more information on expressions, see *Expressions* (page 565).

Example Consider a inventory collection with the following documents:

```
{ "_id" : 1, "item" : "ABC1", "description" : "product 1", colors: [ "blue", "black", "red" ] }
{ "_id" : 2, "item" : "ABC2", "description" : "product 2", colors: [ "purple" ] }
{ "_id" : 3, "item" : "XYZ1", "description" : "product 3", colors: [ ] }
```

The following aggregation pipeline operation use the $size (page 530) to return the number of elements in the colors array:

```
db.inventory.aggregate(
   [
      {
         $project: {
            item: 1,
            numberOfColors: { $size: "$colors" }
         }
      }
   ]
)
```

The operation returns the following:

```
{ "_id" : 1, "item" : "ABC1", "numberOfColors" : 3 }
{ "_id" : 2, "item" : "ABC2", "numberOfColors" : 1 }
{ "_id" : 3, "item" : "XYZ1", "numberOfColors" : 0 }
```

Variable Operators

Aggregation Variable Operators

Name	Description
$let (page 531)	Defines variables for use within the scope of a subexpression and returns the resu subexpression. Accepts named parameters.
$map (page 532)	Applies a subexpression to each element of an array and returns the array of resul order. Accepts named parameters.

$let (aggregation)

Definition

$let

Binds *variables* (page 573) for use in the specified expression, and returns the result of the expression.

The $let (page 531) expression has the following syntax:

```
{
  $let:
     {
       vars: { <var1>: <expression>, ... },
       in: <expression>
     }
}
```

Field	Specification
vars	Assignment block for the *variables* (page 573) accessible in the in expression. To assign a variable, specify a string for the variable name and assign a valid expression for the value. The variable assignments have no meaning outside the in expression, not even within the vars block itself.
in	The *expression* (page 565) to evaluate.

To access variables in aggregation expressions, prefix the variable name with double dollar signs ($$) and enclosed in quotes. For more information on expressions, see *Expressions* (page 565). For information on use of variables in the aggregation pipeline, see *Variables in Aggregation Expressions* (page 573).

Behavior $let (page 531) can access variables defined outside its expression block, including *system variables* (page 573).

If you modify the values of externally defined variables in the vars block, the new values take effect only in the in expression. Outside of the in expression, the variables retain their previous values.

In the vars assignment block, the order of the assignment does **not** matter, and the variable assignments only have meaning inside the in expression. As such, accessing a variable's value in the vars assignment block refers to the value of the variable defined outside the vars block and **not** inside the same vars block.

For example, consider the following $let (page 531) expression:

```
{
  $let:
    {
      vars: { low: 1, high: "$$low" },
      in: { $gt: [ "$$low", "$$high" ] }
    }
}
```

In the vars assignment block, "$$low" refers to the value of an externally defined variable low and not the variable defined in the same vars block. If low is not defined outside this $let (page 531) expression block, the expression is invalid.

Example A `sales` collection has the following documents:

```
{ _id: 1, price: 10, tax: 0.50, applyDiscount: true }
{ _id: 2, price: 10, tax: 0.25, applyDiscount: false }
```

The following aggregation uses `$let` (page 531) in the `$project` (page 492) pipeline stage to calculate and return the `finalTotal` for each document:

```
db.sales.aggregate( [
   {
      $project: {
         finalTotal: {
            $let: {
               vars: {
                  total: { $add: [ '$price', '$tax' ] },
                  discounted: { $cond: { if: '$applyDiscount', then: 0.9, else: 1 } }
               },
               in: { $multiply: [ "$$total", "$$discounted" ] }
            }
         }
      }
   }
] )
```

The aggregation returns the following results:

```
{ "_id" : 1, "finalTotal" : 9.450000000000001 }
{ "_id" : 2, "finalTotal" : 10.25 }
```

See also:

`$map` (page 532)

$map (aggregation)

Definition

`$map`

Applies an *expression* (page 565) to each item in an array and returns an array with the applied results.

The `$map` (page 532) expression has the following syntax:

```
{ $map: { input: <expression>, as: <string>, in: <expression> } }
```

Field	Specification
`input`	An *expression* (page 565) that resolves to an array.
`as`	The variable name for the items in the `input` array. The `in` expression accesses each item in the `input` array by this *variable* (page 573).
`in`	The *expression* (page 565) to apply to each item in the `input` array. The expression accesses the item by its variable name.

For more information on expressions, see *Expressions* (page 565).

Example Consider a `grades` collection with the following documents:

```
{ _id: 1, quizzes: [ 5, 6, 7 ] }
{ _id: 2, quizzes: [ ] }
{ _id: 3, quizzes: [ 3, 8, 9 ] }
```

And the following $project (page 492) statement:

```
db.grades.aggregate(
   [
      { $project:
         { adjustedGrades:
            {
              $map:
                 {
                   input: "$quizzes",
                   as: "grade",
                   in: { $add: [ "$$grade", 2 ] }
                 }
            }
         }
      }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "adjustedGrades" : [ 7, 8, 9 ] }
{ "_id" : 2, "adjustedGrades" : [ ] }
{ "_id" : 3, "adjustedGrades" : [ 5, 10, 11 ] }
```

See also:

$let (page 531)

Literal Operators

Aggregation Literal Operators

Name	Description
$literal (page 533)	Return a value without parsing. Use for values that the aggregation pipeline may i expression. For example, use a $literal (page 533) expression to a string that s to avoid parsing as a field path.

$literal (aggregation)

Definition

$literal

Returns a value without parsing. Use for values that the aggregation pipeline may interpret as an expression.

The $literal (page 533) expression has the following syntax:

```
{ $literal: <value> }
```

Behavior If the <value> is an *expression* (page 565), $literal (page 533) does not evaluate the expression but instead returns the unparsed expression.

Example		Result
{ $literal: { $add: [2, 3] } }		{ "$add" : [2, 3] }
{ $literal: { $literal: 1 } }		{ "$literal" : 1 }

Examples

Treat $ as a Literal In *expression* (page 565), the dollar sign `$` evaluates to a field path; i.e. provides access to the field. For example, the `$eq` expression `$eq: [ "$price", "$1" ]` performs an equality check between the value in the field named `price` and the value in the field named `1` in the document.

The following example uses a `$literal` (page 533) expression to treat a string that contains a dollar sign `"$1"` as a constant value.

A collection `records` has the following documents:

```
{ "_id" : 1, "item" : "abc123", price: "$2.50" }
{ "_id" : 2, "item" : "xyz123", price: "1" }
{ "_id" : 3, "item" : "ijk123", price: "$1" }

db.records.aggregate( [
   { $project: { costsOneDollar: { $eq: [ "$price", { $literal: "$1" } ] } } }
] )
```

This operation projects a field named `costsOneDollar` that holds a boolean value, indicating whether the value of the `price` field is equal to the string `"$1"`:

```
{ "_id" : 1, "costsOneDollar" : false }
{ "_id" : 2, "costsOneDollar" : false }
{ "_id" : 3, "costsOneDollar" : true }
```

Project a New Field with Value 1 The `$project` (page 492) stage uses the expression `<field>: 1` to include the `<field>` in the output. The following example uses the `$literal` (page 533) to return a new field set to the value of `1`.

A collection `bids` has the following documents:

```
{ "_id" : 1, "item" : "abc123", condition: "new" }
{ "_id" : 2, "item" : "xyz123", condition: "new" }
```

The following aggregation evaluates the expression `item: 1` to mean return the existing field `item` in the output, but uses the `{ $literal: 1 }` (page 533) expression to return a new field `startAt` set to the value `1`:

```
db.bids.aggregate( [
   { $project: { item: 1, startAt: { $literal: 1 } } }
] )
```

The operation results in the following documents:

```
{ "_id" : 1, "item" : "abc123", "startAt" : 1 }
{ "_id" : 2, "item" : "xyz123", "startAt" : 1 }
```

Date Operators

Date Aggregation Operators

Name	Description
$dateToString (page 535)	Returns the date as a formatted string.
$dayOfMonth (page 536)	Returns the day of the month for a date as a number between 1 and 31.
$dayOfWeek (page 537)	Returns the day of the week for a date as a number between 1 (Sunday) and 7
$dayOfYear (page 538)	Returns the day of the year for a date as a number between 1 and 366 (leap ye
$hour (page 539)	Returns the hour for a date as a number between 0 and 23.
$millisecond (page 540)	Returns the milliseconds of a date as a number between 0 and 999.
$minute (page 541)	Returns the minute for a date as a number between 0 and 59.
$month (page 542)	Returns the month for a date as a number between 1 (January) and 12 (Decem
$second (page 542)	Returns the seconds for a date as a number between 0 and 60 (leap seconds).
$week (page 543)	Returns the week number for a date as a number between 0 (the partial week t the first Sunday of the year) and 53 (leap year).
$year (page 544)	Returns the year for a date as a number (e.g. 2014).

$dateToString (aggregation)

`$dateToString`

New in version 2.8.

Converts a date object to a string according to a user-specified format.

The $dateToString (page 535) expression has the following syntax:

```
{ $dateToString: { format: <formatString>, date: <dateExpression> } }
```

The <formatString> can be any string literal, containing 0 or more format specifiers. For a list of specifiers available, see *Format Specifiers* (page 535).

The <dateExpression> can be any *expression* (page 565) that evaluates to a date. For more information on expressions, see *Expressions* (page 565).

Format Specifiers The following format specifiers are available for use in the <formatString>:

Specifiers	Description	Possible Values
%Y	Year (4 digits, zero padded)	0000-9999
%m	Month (2 digits, zero padded)	01-12
%d	Day of Month (2 digits, zero padded)	01-31
%H	Hour (2 digits, zero padded, 24-hour clock)	00-23
%M	Minute (2 digits, zero padded)	00-59
%S	Second (2 digits, zero padded)	00-60
%L	Millisecond (3 digits, zero padded)	000-999
%j	Day of year (3 digits, zero padded)	001-366
%w	Day of week (1-Sunday, 7-Saturday)	1-7
%U	Week of year (2 digits, zero padded)	00-53
%%	Percent Character as a Literal	%

Example Consider a sales collection with the following document:

```
{
  "_id" : 1,
  "item" : "abc",
  "price" : 10,
  "quantity" : 2,
  "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

The following aggregation uses the `$dateToString` (page 535) to return the `date` field as formatted strings:

```
db.sales.aggregate(
   [
     {
       $project: {
          yearMonthDay: { $dateToString: { format: "%Y-%m-%d", date: "$date" } },
          time: { $dateToString: { format: "%H:%M:%S:%L", date: "$date" } }
       }
     }
   ]
)
```

The operation returns the following result:

```
{ "_id" : 1, "yearMonthDay" : "2014-01-01", "time" : "08:15:39:736" }
```

$dayOfMonth (aggregation)

`$dayOfMonth`

Returns the day of the month for a date as a number between 1 and 31.

The `$dayOfMonth` (page 536) expression has the following syntax:

```
{ $dayOfMonth: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the `$dayOfMonth` (page 536) and other date operators to break down the `date` field:

```
db.sales.aggregate(
   [
     {
       $project:
         {
           year: { $year: "$date" },
           month: { $month: "$date" },
           day: { $dayOfMonth: "$date" },
           hour: { $hour: "$date" },
           minutes: { $minute: "$date" },
           seconds: { $second: "$date" },
           milliseconds: { $millisecond: "$date" },
           dayOfYear: { $dayOfYear: "$date" },
           dayOfWeek: { $dayOfWeek: "$date" },
           week: { $week: "$date" }
```

```
          }
      }
   ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
  "dayOfWeek" : 4,
  "week" : 0
}
```

$dayOfWeek (aggregation)

$dayOfWeek

Returns the day of the week for a date as a number between 1 (Sunday) and 7 (Saturday).

The $dayOfWeek (page 537) expression has the following syntax:

```
{ $dayOfWeek: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the $dayOfWeek (page 537) and other date operators to break down the `date` field:

```
db.sales.aggregate(
   [
     {
       $project:
         {
           year: { $year: "$date" },
           month: { $month: "$date" },
           day: { $dayOfMonth: "$date" },
           hour: { $hour: "$date" },
           minutes: { $minute: "$date" },
           seconds: { $second: "$date" },
           milliseconds: { $millisecond: "$date" },
           dayOfYear: { $dayOfYear: "$date" },
           dayOfWeek: { $dayOfWeek: "$date" },
           week: { $week: "$date" }
         }
     }
   ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
  "dayOfWeek" : 4,
  "week" : 0
}
```

$dayOfYear (aggregation)

$dayOfYear

Returns the day of the year for a date as a number between 1 and 366.

The `$dayOfYear` (page 538) expression has the following syntax:

```
{ $dayOfYear: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the `$dayOfYear` (page 538) and other date expressions to break down the `date` field:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
            dayOfWeek: { $dayOfWeek: "$date" },
            week: { $week: "$date" }
          }
      }
    ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
```

```
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
  "dayOfWeek" : 4,
  "week" : 0
}
```

$hour (aggregation)

$hour

Returns the hour portion of a date as a number between 0 and 23.

The `$hour` (page 539) expression has the following syntax:

```
{ $hour: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the `$hour` (page 539) and other date expressions to break down the `date` field:

```
db.sales.aggregate(
   [
     {
       $project:
         {
           year: { $year: "$date" },
           month: { $month: "$date" },
           day: { $dayOfMonth: "$date" },
           hour: { $hour: "$date" },
           minutes: { $minute: "$date" },
           seconds: { $second: "$date" },
           milliseconds: { $millisecond: "$date" },
           dayOfYear: { $dayOfYear: "$date" },
           dayOfWeek: { $dayOfWeek: "$date" }
         }
     }
   ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
```

```
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
  "dayOfWeek" : 4,
  "week" : 0
}
```

$millisecond (aggregation)

$millisecond

Returns the millisecond portion of a date as an integer between 0 and 999.

The `$millisecond` (page 540) expression has the following syntax:

```
{ $millisecond: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the `$millisecond` (page 540) and other date operators to break down the `date` field:

```
db.sales.aggregate(
   [
     {
       $project:
         {
           year: { $year: "$date" },
           month: { $month: "$date" },
           day: { $dayOfMonth: "$date" },
           hour: { $hour: "$date" },
           minutes: { $minute: "$date" },
           seconds: { $second: "$date" },
           milliseconds: { $millisecond: "$date" },
           dayOfYear: { $dayOfYear: "$date" },
           dayOfWeek: { $dayOfWeek: "$date" },
           week: { $week: "$date" }
         }
     }
   ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
```

```
  "dayOfWeek" : 4,
  "week" : 0
}
```

$minute (aggregation)

$minute

Returns the minute portion of a date as a number between 0 and 59.

The $minute (page 541) expression has the following syntax:

```
{ $minute: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a sales collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the $minute (page 541) and other date expressions to break down the date field:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
            dayOfWeek: { $dayOfWeek: "$date" },
            week: { $week: "$date" }
          }
      }
    ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
  "dayOfWeek" : 4,
  "week" : 0
}
```

$month (aggregation)

$month

Returns the month of a date as a number between 1 and 12.

The $month (page 542) expression has the following syntax:

```
{ $month: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a sales collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the $month (page 542) and other date operators to break down the date field:

```
db.sales.aggregate(
   [
     {
       $project:
         {
           year: { $year: "$date" },
           month: { $month: "$date" },
           day: { $dayOfMonth: "$date" },
           hour: { $hour: "$date" },
           minutes: { $minute: "$date" },
           seconds: { $second: "$date" },
           milliseconds: { $millisecond: "$date" },
           dayOfYear: { $dayOfYear: "$date" },
           dayOfWeek: { $dayOfWeek: "$date" },
           week: { $week: "$date" }
         }
     }
   ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
  "dayOfWeek" : 4,
  "week" : 0
}
```

$second (aggregation)

$second

Returns the second portion of a date as a number between 0 and 59, but can be 60 to account for leap seconds.

The $second (page 542) expression has the following syntax:

```
{ $second: <expression> }
```

The argument can be any valid *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the `$second` (page 542) and other date expressions to break down the `date` field:

```
db.sales.aggregate(
   [
     {
       $project:
         {
           year: { $year: "$date" },
           month: { $month: "$date" },
           day: { $dayOfMonth: "$date" },
           hour: { $hour: "$date" },
           minutes: { $minute: "$date" },
           seconds: { $second: "$date" },
           milliseconds: { $millisecond: "$date" },
           dayOfYear: { $dayOfYear: "$date" },
           dayOfWeek: { $dayOfWeek: "$date" },
           week: { $week: "$date" }
         }
     }
   ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
  "dayOfWeek" : 4,
  "week" : 0
}
```

$week (aggregation)

`$week`

Returns the week of the year for a date 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.

The `$week` (page 543) expression has the following syntax:

```
{ $week: <expression> }
```

The argument can be any valid *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the `$week` (page 543) and other date operators to break down the `date` field:

```
db.sales.aggregate(
   [
     {
       $project:
         {
           year: { $year: "$date" },
           month: { $month: "$date" },
           day: { $dayOfMonth: "$date" },
           hour: { $hour: "$date" },
           minutes: { $minute: "$date" },
           seconds: { $second: "$date" },
           milliseconds: { $millisecond: "$date" },
           dayOfYear: { $dayOfYear: "$date" },
           dayOfWeek: { $dayOfWeek: "$date" },
           week: { $week: "$date" }
         }
     }
   ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
  "dayOfWeek" : 4,
  "week" : 0
}
```

$year (aggregation)

`$year`

Returns the year portion of a date.

The `$year` (page 544) expression has the following syntax:

```
{ $year: <expression> }
```

The argument can be any *expression* (page 565) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736Z
```

The following aggregation uses the `$year` (page 544) and other date operators to break down the `date` field:

```
db.sales.aggregate(
   [
     {
       $project:
         {
           year: { $year: "$date" },
           month: { $month: "$date" },
           day: { $dayOfMonth: "$date" },
           hour: { $hour: "$date" },
           minutes: { $minute: "$date" },
           seconds: { $second: "$date" },
           milliseconds: { $millisecond: "$date" },
           dayOfYear: { $dayOfYear: "$date" },
           dayOfWeek: { $dayOfWeek: "$date" },
           week: { $week: "$date" }
         }
     }
   ]
)
```

The operation returns the following result:

```
{
  "_id" : 1,
  "year" : 2014,
  "month" : 1,
  "day" : 1,
  "hour" : 8,
  "minutes" : 15,
  "seconds" : 39,
  "milliseconds" : 736,
  "dayOfYear" : 1,
  "dayOfWeek" : 4,
  "week" : 0
}
```

Conditional Expressions

Conditional Aggregation Operators

Name	Description
`$cond` (page 545)	A ternary operator that evaluates one expression, and depending on the result, one of the other two expressions. Accepts either three expressions in an ordere named parameters.
`$ifNull` (page 547)	Returns either the non-null result of the first expression or the result of the sec the first expression results in a null result. Null result encompasses instances o or missing fields. Accepts two expressions as arguments. The result of the sec be null.

$cond (aggregation)

`$cond`

Evaluates a boolean expression to return one of the two specified return expressions.

The $cond (page 545) expression has one of two syntaxes:

New in version 2.6.

```
{ $cond: { if: <boolean-expression>, then: <true-case>, else: <false-case> } }
```

Or:

```
{ $cond: [ <boolean-expression>, <true-case>, <false-case> ] }
```

If the <boolean-expression> evaluates to true, then $cond (page 545) evaluates and returns the value of the <true-case> expression. Otherwise, $cond (page 545) evaluates and returns the value of the <false-case> expression.

The arguments can be any valid *expression* (page 565). For more information on expressions, see *Expressions* (page 565).

Example The following example use a inventory collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", qty: 300 }
{ "_id" : 2, "item" : "abc2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", qty: 250 }
```

The following aggregation operation uses the $cond (page 545) expression to set the discount value to 30 if qty value is greater than or equal to 250 and to 20 if qty value is less than 250:

```
db.inventory.aggregate(
   [
      {
         $project:
           {
             item: 1,
             discount:
               {
                 $cond: { if: { $gte: [ "$qty", 250 ] }, then: 30, else: 20 }
               }
           }
      }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc1", "discount" : 30 }
{ "_id" : 2, "item" : "abc2", "discount" : 20 }
{ "_id" : 3, "item" : "xyz1", "discount" : 30 }
```

The following operation uses the array syntax of the $cond (page 545) expression and returns the same results:

```
db.inventory.aggregate(
   [
      {
         $project:
           {
             item: 1,
             discount:
               {
                 $cond: [ { $gte: [ "$qty", 250 ] }, 30, 20 ]
               }
           }
```

```
      }
   ]
)
```

$ifNull (aggregation)

$ifNull

Evaluates an expression and returns the value of the expression if the expression evaluates to a non-null value. If the expression evaluates to a null value, including instances of undefined values or missing fields, returns the value of the replacement expression.

The $ifNull (page 547) expression has the following syntax:

```
{ $ifNull: [ <expression>, <replacement-expression-if-null> ] }
```

The arguments can be any valid *expression* (page 565). For more information on expressions, see *Expressions* (page 565).

Example The following example use a inventory collection with the following documents:

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: null, qty: 200 }
{ "_id" : 3, "item" : "xyz1", qty: 250 }
```

The following operation uses the $ifNull (page 547) expression to return either the non-null description field value or the string "Unspecified" if the description field is null or does not exist:

```
db.inventory.aggregate(
   [
      {
         $project: {
            item: 1,
            description: { $ifNull: [ "$description", "Unspecified" ] }
         }
      }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "abc1", "description" : "product 1" }
{ "_id" : 2, "item" : "abc2", "description" : "Unspecified" }
{ "_id" : 3, "item" : "xyz1", "description" : "Unspecified" }
```

Accumulators

Accumulators, available only for the $group (page 486) stage, compute values by combining documents that share the same group key. Accumulators take as input a single expression, evaluating the expression once for each input document, and maintain their state for the group of documents.

Group Accumulator Operators

Accumulators, available only for the $group (page 486) stage, compute values by combining documents that share the same group key. Accumulators take as input a single expression, evaluating the expression once for each input document, and maintain their state for the group of documents.

Name	Description
$addToSet (page 548)	Returns an array of *unique* expression values for each group. Order of the array elements is undefined.
$avg (page 549)	Returns an average for each group. Ignores non-numeric values.
$first (page 549)	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.
$last (page 550)	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.
$max (page 551)	Returns the highest expression value for each group.
$min (page 552)	Returns the lowest expression value for each group.
$push (page 553)	Returns an array of expression values for each group.
$sum (page 554)	Returns a sum for each group. Ignores non-numeric values.

$addToSet (aggregation)

$addToSet

Returns an array of all *unique* values that results from applying an expression to each document in a group of documents that share the same group by key. Order of the elements in the output array is unspecified.

$addToSet (page 548) is an *accumulator operator* (page 547) available only in the $group (page 486) stage.

$addToSet has the following syntax:

```
{ $addToSet: <expression> }
```

For more information on expressions, see *Expressions* (page 565).

Behavior If the value of the expression is an array, $addToSet (page 548) appends the whole array as a *single* element.

If the value of the expression is a document, MongoDB determines that the document is a duplicate if another document in the array matches the to-be-added document exactly; i.e. the existing document has the exact same fields and values in the exact same order.

Example Consider a sales collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:12:00Z") }
```

Grouping the documents by the day and the year of the date field, the following operation uses the $addToSet (page 548) accumulator to compute the list of unique items sold for each group:

```
db.sales.aggregate(
   [
     {
       $group:
         {
           _id: { day: { $dayOfYear: "$date"}, year: { $year: "$date" } },
           itemsSold: { $addToSet: "$item" }
         }
     }
   ]
)
```

The operation returns the following results:

```
{ "_id" : { "day" : 46, "year" : 2014 }, "itemsSold" : [ "xyz", "abc" ] }
{ "_id" : { "day" : 34, "year" : 2014 }, "itemsSold" : [ "xyz", "jkl" ] }
{ "_id" : { "day" : 1, "year" : 2014 }, "itemsSold" : [ "abc" ] }
```

$avg (aggregation)

$avg

Returns the average value of the numeric values that result from applying a specified expression to each document in a group of documents that share the same group by key. $avg (page 549) ignores non-numeric values.

$avg (page 549) is an *accumulator operator* (page 547) available only in the $group (page 486) stage.

$avg has the following syntax:

```
{ $avg: <expression> }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider a sales collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:12:00Z") }
```

Grouping the documents by the item field, the following operation uses the $avg (page 549) accumulator to compute the average amount and average quantity for each grouping.

```
db.sales.aggregate(
   [
     {
       $group:
         {
           _id: "$item",
           avgAmount: { $avg: { $multiply: [ "$price", "$quantity" ] } },
           avgQuantity: { $avg: "$quantity" }
         }
     }
   ]
)
```

The operation returns the following results:

```
{ "_id" : "xyz", "avgAmount" : 37.5, "avgQuantity" : 7.5 }
{ "_id" : "jkl", "avgAmount" : 20, "avgQuantity" : 1 }
{ "_id" : "abc", "avgAmount" : 60, "avgQuantity" : 6 }
```

$first (aggregation)

$first

Returns the value that results from applying an expression to the first document in a group of documents that share the same group by key. Only meaningful when documents are in a defined order.

$first (page 549) is an *accumulator operator* (page 547) available only in the $group (page 486) stage.

$first has the following syntax:

```
{ $first: <expression> }
```

For more information on expressions, see *Expressions* (page 565).

Behavior When using `$first` (page 549) in a `$group` (page 486) stage, the `$group` (page 486) stage should follow a `$sort` (page 499) stage to have the input documents in a defined order.

Example Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
{ "_id" : 6, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-15T12:05:10Z") }
{ "_id" : 7, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T14:12:12Z") }
```

Grouping the documents by the `item` field, the following operation uses the `$first` (page 549) accumulator to compute the first sales date for each item:

```
db.sales.aggregate(
   [
     { $sort: { item: 1, date: 1 } },
     {
       $group:
         {
           _id: "$item",
           firstSalesDate: { $first: "$date" }
         }
     }
   ]
)
```

The operation returns the following results:

```
{ "_id" : "xyz", "firstSalesDate" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : "jkl", "firstSalesDate" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : "abc", "firstSalesDate" : ISODate("2014-01-01T08:00:00Z") }
```

$last (aggregation)

`$last`

Returns the value that results from applying an expression to the last document in a group of documents that share the same group by a field. Only meaningful when documents are in a defined order.

`$last` (page 550) is an *accumulator operator* (page 547) available only in the `$group` (page 486) stage.

`$last` has the following syntax:

```
{ $last: <expression> }
```

For more information on expressions, see *Expressions* (page 565).

Behavior When using `$last` (page 550) in a `$group` (page 486) stage, the `$group` (page 486) stage should follow a `$sort` (page 499) stage to have the input documents in a defined order.

Example Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
{ "_id" : 6, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-15T12:05:10Z") }
{ "_id" : 7, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T14:12:12Z") }
```

Grouping the documents by the `item` field, the following operation uses the `$last` (page 550) accumulator to compute the last sales date for each item:

```
db.sales.aggregate(
   [
     { $sort: { item: 1, date: 1 } },
     {
       $group:
         {
           _id: "$item",
           lastSalesDate: { $last: "$date" }
         }
     }
   ]
)
```

The operation returns the following results:

```
{ "_id" : "xyz", "lastSalesDate" : ISODate("2014-02-15T14:12:12Z") }
{ "_id" : "jkl", "lastSalesDate" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : "abc", "lastSalesDate" : ISODate("2014-02-15T08:00:00Z") }
```

$max (aggregation)

`$max`

Returns the highest value that results from applying an expression to each document in a group of documents that share the same group by key.

`$max` (page 551) is an *accumulator operator* (page 547) available only in the `$group` (page 486) stage.

`$max` has the following syntax:

```
{ $max: <expression> }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
```

Grouping the documents by the `item` field, the following operation uses the `$max` (page 551) accumulator to compute the maximum total amount and maximum quantity for each group of documents.

```
db.sales.aggregate(
   [
```

```
    {
      $group:
        {
          _id: "$item",
          maxTotalAmount: { $max: { $multiply: [ "$price", "$quantity" ] } },
          maxQuantity: { $max: "$quantity" }
        }
    }
  ]
)
```

The operation returns the following results:

```
{ "_id" : "xyz", "maxTotalAmount" : 50, "maxQuantity" : 10 }
{ "_id" : "jkl", "maxTotalAmount" : 20, "maxQuantity" : 1 }
{ "_id" : "abc", "maxTotalAmount" : 100, "maxQuantity" : 10 }
```

$min (aggregation)

$min

Returns the lowest value that results from applying an expression to each document in a group of documents that share the same group by key.

`$min` (page 552) is an *accumulator operator* (page 547) available only in the `$group` (page 486) stage.

`$min` has the following syntax:

```
{ $min: <expression> }
```

For more information on expressions, see *Expressions* (page 565).

Behavior Changed in version 2.4.

If some, **but not all**, documents for the `$min` (page 552) operation have either a `null` value for the field or are missing the field, the `$min` (page 552) operator only considers the non-null and the non-missing values for the field.

If **all** documents for the `$min` (page 552) operation have `null` value for the field or are missing the field, the `$min` (page 552) operator returns `null` for the minimum value.

Before 2.4, if any of the documents for the `$min` (page 552) operation were missing the field, the `$min` (page 552) operator would not return any value. If any of the documents for the `$min` (page 552) had the value `null`, the `$min` (page 552) operator would return a `null`.

Example Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
```

Grouping the documents by the `item` field, the following operation uses the `$min` (page 552) accumulator to compute the minimum amount and minimum quantity for each grouping.

```
db.sales.aggregate(
  [
    {
      $group:
```

```
          {
            _id: "$item",
            minQuantity: { $min: "$quantity" }
          }
      }
   ]
)
```

The operation returns the following results:

```
{ "_id" : "xyz", "minQuantity" : 5 }
{ "_id" : "jkl", "minQuantity" : 1 }
{ "_id" : "abc", "minQuantity" : 2 }
```

$push (aggregation)

$push

Returns an array of *all* values that result from applying an expression to each document in a group of documents that share the same group by key.

`$push` (page 553) is an *accumulator operator* (page 547) available only in the `$group` (page 486) stage.

`$push` has the following syntax:

```
{ $push: <expression> }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
{ "_id" : 6, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-15T12:05:10Z") }
{ "_id" : 7, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T14:12:12Z") }
```

Grouping the documents by the day and the year of the `date` field, the following operation uses the `$addToSet` (page 548) accumulator to compute the list of items and quantities sold for each group:

```
db.sales.aggregate(
   [
     {
       $group:
         {
           _id: { day: { $dayOfYear: "$date"}, year: { $year: "$date" } },
           itemsSold: { $push:  { item: "$item", quantity: "$quantity" } }
         }
     }
   ]
)
```

The operation returns the following results:

```
{
   "_id" : { "day" : 46, "year" : 2014 },
   "itemsSold" : [
      { "item" : "abc", "quantity" : 10 },
```

```
      { "item" : "xyz", "quantity" : 10 },
      { "item" : "xyz", "quantity" : 5 },
      { "item" : "xyz", "quantity" : 10 }
   ]
}
{
   "_id" : { "day" : 34, "year" : 2014 },
   "itemsSold" : [
      { "item" : "jkl", "quantity" : 1 },
      { "item" : "xyz", "quantity" : 5 }
   ]
}
{
   "_id" : { "day" : 1, "year" : 2014 },
   "itemsSold" : [ { "item" : "abc", "quantity" : 2 } ]
}
```

$sum (aggregation)

$sum

Calculates and returns the sum of all the numeric values that result from applying a specified expression to each document in a group of documents that share the same group by key. $sum (page 554) ignores non-numeric values.

$sum (page 554) is an *accumulator operator* (page 547) available only in the $group (page 486) stage.

$sum has the following syntax:

```
{ $sum: <expression> }
```

For more information on expressions, see *Expressions* (page 565).

Example Consider a sales collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
```

Grouping the documents by the day and the year of the date field, the following operation uses the $sum (page 554) accumulator to compute the total amount and the count for each group of documents.

```
db.sales.aggregate(
   [
     {
       $group:
         {
           _id: { day: { $dayOfYear: "$date"}, year: { $year: "$date" } },
           totalAmount: { $sum: { $multiply: [ "$price", "$quantity" ] } },
           count: { $sum: 1 }
         }
     }
   ]
)
```

The operation returns the following results:

```
{ "_id" : { "day" : 46, "year" : 2014 }, "totalAmount" : 150, "count" : 2 }
{ "_id" : { "day" : 34, "year" : 2014 }, "totalAmount" : 45, "count" : 2 }
{ "_id" : { "day" : 1, "year" : 2014 }, "totalAmount" : 20, "count" : 1 }
```

2.3.4 Query Modifiers

Introduction

In addition to the *MongoDB Query Operators* (page 400), there are a number of "meta" operators that let you modify the output or behavior of a query. On the server, MongoDB treats the query and the options as a single object. The `mongo` (page 610) shell and driver interfaces may provide *cursor methods* (page 81) that wrap these options. When possible, use these methods; otherwise, you can add these options using either of the following syntax:

```
db.collection.find( { <query> } )._addSpecial( <option> )
db.collection.find( { $query: { <query> }, <option> } )
```

Operators

Modifiers

Many of these operators have corresponding *methods in the shell* (page 81). These methods provide a straightforward and user-friendly interface and are the preferred way to add these options.

Name	Description
`$comment` (page 555)	Adds a comment to the query to identify queries in the *database profiler* output.
`$explain` (page 556)	Forces MongoDB to report on query execution plans. See `explain()` (page 85).
`$hint` (page 556)	Forces MongoDB to use a specific index. See `hint()` (page 87)
`$maxScan` (page 557)	Limits the number of documents scanned.
`$maxTimeMS` (page 557)	Specifies a cumulative time limit in milliseconds for processing operations on a cursor. See `maxTimeMS()` (page 90).
`$max` (page 558)	Specifies an *exclusive* upper limit for the index to use in a query. See `max()` (page 88).
`$min` (page 559)	Specifies an *inclusive* lower limit for the index to use in a query. See `min()` (page 91).
`$orderby`	Returns a cursor with documents sorted according to a sort specification. See `sort()` (page 95).
`$query` (page 560)	Wraps a query document.
`$returnKey` (page 561)	Forces the cursor to only return fields included in the index.
`$showDiskLoc` (page 561)	Modifies the documents returned to include references to the on-disk location of each document.
`$snapshot` (page 562)	Forces the query to use the index on the `_id` field. See `snapshot()` (page 95).

$comment

`$comment`

The `$comment` (page 555) meta-operator makes it possible to attach a comment to a query in any context that `$query` (page 560) may appear.

Because comments propagate to the `profile` (page 366) log, adding a comment can make your profile data easier to interpret and trace.

Use $comment (page 555) in one of the following ways:

```
db.collection.find( { <query> } )._addSpecial( "$comment", <comment> )
db.collection.find( { <query> } ).comment( <comment> )
db.collection.find( { $query: { <query> }, $comment: <comment> } )
```

To attach comments to query expressions in other contexts, such as db.collection.update() (page 72), use the $comment (page 443) query operator instead of the meta-operator.

See also:

$comment (page 443) query operator

$explain

$explain

Deprecated since version 2.8: Use db.collection.explain() (page 33) or cursor.explain() (page 85) instead

The $explain (page 556) operator provides information on the query plan. It returns a document that describes the process and indexes used to return the query. This may provide useful insight when attempting to optimize a query. For details on the output, see *cursor.explain()* (page 85).

You can specify the $explain (page 556) operator in either of the following forms:

```
db.collection.find()._addSpecial( "$explain", 1 )
db.collection.find( { $query: {}, $explain: 1 } )
```

In the mongo (page 610) shell, you also can retrieve query plan information through the explain() (page 85) method:

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

Behavior $explain (page 556) runs the actual query to determine the result. Although there are some differences between running the query with $explain (page 556) and running without, generally, the performance will be similar between the two. So, if the query is slow, the $explain (page 556) operation is also slow.

Additionally, the $explain (page 556) operation reevaluates a set of candidate query plans, which may cause the $explain (page 556) operation to perform differently than a normal query. As a result, these operations generally provide an accurate account of *how* MongoDB would perform the query, but do not reflect the length of these queries.

See also:

- explain() (page 85)
- http://docs.mongodb.org/manual/administration/optimization page for information regarding optimization strategies.
- http://docs.mongodb.org/manual/tutorial/manage-the-database-profiler tutorial for information regarding the database profile.
- *Current Operation Reporting* (page 105)

$hint

$hint

The $hint (page 556) operator forces the *query optimizer* to use a specific index to fulfill the query. Specify the index either by the index name or by document.

Use $hint (page 556) for testing query performance and indexing strategies. The mongo (page 610) shell provides a helper method hint() (page 87) for the $hint (page 556) operator.

Consider the following operation:

```
db.users.find().hint( { age: 1 } )
```

This operation returns all documents in the collection named `users` using the index on the `age` field.

You can also specify a hint using either of the following forms:

```
db.users.find()._addSpecial( "$hint", { age : 1 } )
db.users.find( { $query: {}, $hint: { age : 1 } } )
```

Note: When the query specifies the `$hint` (page 556) in the following form:

```
db.users.find( { $query: {}, $hint: { age : 1 } } )
```

Then, in order to include the `$explain` (page 556) option, you must add the `$explain` (page 556) option to the document, as in the following:

```
db.users.find( { $query: {}, $hint: { age : 1 }, $explain: 1 } )
```

When an *index filter* exists for the query shape, MongoDB ignores the `$hint` (page 556).

$maxScan

`$maxScan`

Constrains the query to only scan the specified number of documents when fulfilling the query. Use one of the following forms:

```
db.collection.find( { <query> } )._addSpecial( "$maxScan" , <number> )
db.collection.find( { $query: { <query> }, $maxScan: <number> } )
```

Use this modifier to prevent potentially long running queries from disrupting performance by scanning through too much data.

$maxTimeMS

`$maxTimeMS`

New in version 2.6: The `$maxTimeMS` (page 557) operator specifies a cumulative time limit in milliseconds for processing operations on the cursor. MongoDB interrupts the operation at the earliest following *interrupt point*.

The `mongo` (page 610) shell provides the `cursor.maxTimeMS()` (page 90) method

```
db.collection.find().maxTimeMS(100)
```

You can also specify the option in either of the following forms:

```
db.collection.find( { } , { $maxTimeMS: 100 } )
db.collection.find( { } )._addSpecial("$maxTimeMS", 100)
```

Interrupted operations return an error message similar to the following:

```
error: { "$err" : "operation exceeded time limit", "code" : 50 }
```

$max

Definition

$max

Specify a $max (page 558) value to specify the *exclusive* upper bound for a specific index in order to constrain the results of find() (page 36). The $max (page 558) specifies the upper bound for *all* keys of a specific index *in order*.

The mongo (page 610) shell provides the max() (page 88) wrapper method:

```
db.collection.find( { <query> } ).max( { field1: <max value>, ... fieldN: <max valueN> } )
```

You can also specify $max (page 558) with either of the two forms:

```
db.collection.find( { <query> } )._addSpecial( "$max", { field1: <max value1>, ... fieldN: <max
db.collection.find( { $query: { <query> }, $max: { field1: <max value1>, ... fieldN: <max valueN
```

Behavior

Interaction with Index Selection Because max() (page 88) requires an index on a field, and forces the query to use this index, you may prefer the $lt (page 403) operator for the query if possible. Consider the following example:

```
db.collection.find( { _id: 7 } ).max( { age: 25 } )
```

The query uses the index on the age field, even if the index on _id may be better.

Index Bounds If you use $max (page 558) with $min (page 559) to specify a range, the index bounds specified in $min (page 559) and $max (page 558) must both refer to the keys of the same index.

$max without $min The min and max operators indicate that the system should avoid normal query planning. Instead they construct an index scan where the index bounds are explicitly specified by the values given in min and max.

If one of the two boundaries is not specified, then the query plan will be an index scan that is unbounded on one side. This may degrade performance compared to a query containing neither operator, or one that uses both operators to more tightly constrain the index scan.

Examples The following examples use the mongo (page 610) shell wrappers.

Specify Exclusive Upper Bound Consider the following operations on a collection named collection that has an index { age: 1 }:

```
db.collection.find( { <query> } ).max( { age: 100 } )
```

This operation limits the query to those documents where the field age is less than 100 and forces a query plan which scans the { age: 1 } index from MinKey to 100.

Index Selection You can explicitly specify the corresponding index with hint() (page 87). Otherwise, MongoDB selects the index using the fields in the $max (page 558) and $min (page 559) bounds; however, if multiple indexes exist on same fields with different sort orders, the selection of the index may be ambiguous.

Consider a collection named collection that has the following two indexes:

```
{ age: 1, type: -1 }
{ age: 1, type: 1 }
```

Without explicitly using `hint()` (page 87), MongoDB may select either index for the following operation:

```
db.collection.find().max( { age: 50, type: 'B' } )
```

Use with $min Use `$max` (page 558) alone or in conjunction with `$min` (page 559) to limit results to a specific range for the *same* index, as in the following example:

```
db.collection.find().min( { age: 20 } ).max( { age: 25 } )
```

$min

Definition

$min

Specify a `$min` (page 559) value to specify the *inclusive* lower bound for a specific index in order to constrain the results of `find()` (page 36). The `$min` (page 559) specifies the lower bound for *all* keys of a specific index *in order*.

The `mongo` (page 610) shell provides the `min()` (page 91) wrapper method:

```
db.collection.find( { <query> } ).min( { field1: <min value>, ... fieldN: <min valueN>} )
```

You can also specify the option with either of the two forms:

```
db.collection.find( { <query> } )._addSpecial( "$min", { field1: <min value1>, ... fieldN: <min
db.collection.find( { $query: { <query> }, $min: { field1: <min value1>, ... fieldN: <min valueN
```

Behavior

Interaction with Index Selection Because `min()` (page 91) requires an index on a field, and forces the query to use this index, you may prefer the `$gte` (page 401) operator for the query if possible. Consider the following example:

```
db.collection.find( { _id: 7 } ).min( { age: 25 } )
```

The query will use the index on the `age` field, even if the index on `_id` may be better.

Index Bounds If you use `$max` (page 558) with `$min` (page 559) to specify a range, the index bounds specified in `$min` (page 559) and `$max` (page 558) must both refer to the keys of the same index.

$min without $max The `min` and `max` operators indicate that the system should avoid normal query planning. Instead they construct an index scan where the index bounds are explicitly specified by the values given in `min` and `max`.

If one of the two boundaries is not specified, then the query plan will be an index scan that is unbounded on one side. This may degrade performance compared to a query containing neither operator, or one that uses both operators to more tightly constrain the index scan.

Examples The following examples use the `mongo` (page 610) shell wrappers.

Specify Inclusive Lower Bound Consider the following operations on a collection named `collection` that has an index `{ age: 1 }`:

```
db.collection.find().min( { age: 20 } )
```

This operation limits the query to those documents where the field `age` is at least `20` and forces a query plan which scans the `{ age: 1 }` index from 20 to `MaxKey`.

Index Selection You can explicitly specify the corresponding index with `hint()` (page 87). Otherwise, MongoDB selects the index using the fields in the `$max` (page 558) and `$min` (page 559) bounds; however, if multiple indexes exist on same fields with different sort orders, the selection of the index may be ambiguous.

Consider a collection named `collection` that has the following two indexes:

```
{ age: 1, type: -1 }
{ age: 1, type: 1 }
```

Without explicitly using `hint()` (page 87), it is unclear which index the following operation will select:

```
db.collection.find().min( { age: 20, type: 'C' } )
```

Use with `$max` You can use `$min` (page 559) in conjunction with `$max` (page 558) to limit results to a specific range for the *same* index, as in the following example:

```
db.collection.find().min( { age: 20 } ).max( { age: 25 } )
```

$orderby

`$orderby`

The `$orderby` operator sorts the results of a query in ascending or descending order.

The `mongo` (page 610) shell provides the `cursor.sort()` (page 95) method:

```
db.collection.find().sort( { age: -1 } )
```

You can also specify the option in either of the following forms:

```
db.collection.find()._addSpecial( "$orderby", { age : -1 } )
db.collection.find( { $query: {}, $orderby: { age : -1 } } )
```

These examples return all documents in the collection named `collection` sorted by the `age` field in descending order. Specify a value to `$orderby` of negative one (e.g. `-1`, as above) to sort in descending order or a positive value (e.g. `1`) to sort in ascending order.

Behavior The sort function requires that the entire sort be able to complete within 32 megabytes. When the sort option consumes more than 32 megabytes, MongoDB will return an error.

To avoid this error, create an index to support the sort operation or use `$orderby` in conjunction with `$maxScan` (page 557) and/or `cursor.limit()` (page 88). The `cursor.limit()` (page 88) increases the speed and reduces the amount of memory required to return this query by way of an optimized algorithm. The specified limit must result in a number of documents that fall within the 32 megabyte limit.

$query

`$query`

The `$query` (page 560) operator provides an interface to describe queries. Consider the following operation:

```
db.collection.find( { $query: { age : 25 } } )
```

This is equivalent to the more familiar db.collection.find() (page 36) method:

```
db.collection.find( { age : 25 } )
```

These operations return only those documents in the collection named collection where the age field equals 25.

Note: Do not mix query forms. If you use the $query (page 560) format, do not append *cursor methods* (page 81) to the find() (page 36). To modify the query use the *meta-query operators* (page 555), such as $explain (page 556).

Therefore, the following two operations are equivalent:

```
db.collection.find( { $query: { age : 25 }, $explain: true } )
db.collection.find( { age : 25 } ).explain()
```

See also:

For more information about queries in MongoDB see http://docs.mongodb.org/manual/core/read-operations db.collection.find() (page 36), and http://docs.mongodb.org/manual/tutorial/getting-started.

$returnKey

$returnKey

Only return the index field or fields for the results of the query. If $returnKey (page 561) is set to true and the query does not use an index to perform the read operation, the returned documents will not contain any fields. Use one of the following forms:

```
db.collection.find( { <query> } )._addSpecial( "$returnKey", true )
db.collection.find( { $query: { <query> }, $returnKey: true } )
```

$showDiskLoc

$showDiskLoc

$showDiskLoc (page 561) option adds a field $diskLoc to the returned documents. The value of the added $diskLoc field is a document that contains the disk location information:

```
"$diskLoc": {
  "file": <int>,
  "offset": <int>
}
```

The mongo (page 610) shell provides the cursor.showDiskLoc() (page 93) method for $showDiskLoc (page 561):

```
db.collection.find().showDiskLoc()
```

You can also specify the $showDiskLoc (page 561) option in either of the following forms:

```
db.collection.find( { <query> } )._addSpecial("$showDiskLoc" , true)
db.collection.find( { $query: { <query> }, $showDiskLoc: true } )
```

Example The following operation appends the showDiskLoc() (page 93) method to the db.collection.find() (page 36) method in order to include in the matching documents the disk location information:

```
db.collection.find( { a: 1 } ).showDiskLoc()
```

The operation returns the following documents, which includes the `$diskLoc` field:

```
{
  "_id" : ObjectId("53908ccb18facd50a75bfbac"),
  "a" : 1,
  "b" : 1,
  "$diskLoc" : { "file" : 0, "offset" : 16195760 }
}
{
   "_id" : ObjectId("53908cd518facd50a75bfbad"),
   "a" : 1,
   "b" : 2,
   "$diskLoc" : { "file" : 0, "offset" : 16195824 }
}
```

The *projection* can also access the added field `$diskLoc`, as in the following example:

```
db.collection.find( { a: 1 }, { $diskLoc: 1 } ).showDiskLoc()
```

The operation returns just the `_id` field and the `$diskLoc` field in the matching documents:

```
{
  "_id" : ObjectId("53908ccb18facd50a75bfbac"),
  "$diskLoc" : { "file" : 0, "offset" : 16195760 }
}
{
   "_id" : ObjectId("53908cd518facd50a75bfbad"),
   "$diskLoc" : { "file" : 0, "offset" : 16195824 }
}
```

$snapshot

$snapshot

The `$snapshot` (page 562) operator prevents the cursor from returning a document more than once because an intervening write operation results in a move of the document.

Even in snapshot mode, objects inserted or deleted during the lifetime of the cursor may or may not be returned.

The `mongo` (page 610) shell provides the `cursor.snapshot()` (page 95) method:

```
db.collection.find().snapshot()
```

You can also specify the option in either of the following forms:

```
db.collection.find()._addSpecial( "$snapshot", true )
db.collection.find( { $query: {}, $snapshot: true } )
```

The `$snapshot` (page 562) operator traverses the index on the `_id` field [23].

Warning:

- You cannot use `$snapshot` (page 562) with *sharded collections*.
- Do **not** use `$snapshot` (page 562) with `$hint` (page 556) or `$orderby` (or the corresponding `cursor.hint()` (page 87) and `cursor.sort()` (page 95) methods.)

[23] You can achieve the `$snapshot` (page 562) isolation behavior using any *unique* index on invariable fields.

Sort Order

Name	Description
$natural (page 563)	A special sort order that orders documents using the order of documents on disk.

$natural

Definition

$natural

Use the $natural (page 563) operator to use *natural order* for the results of a sort operation. Natural order refers to the logical *ordering* (page 98) of documents internally within the database.

The $natural (page 563) operator uses the following syntax to return documents in the order they exist on disk:

```
db.collection.find().sort( { $natural: 1 } )
```

Behavior On a sharded collection the $natural (page 563) operator returns a collection scan sorted in *natural order* (page 98), the order the database inserts and stores documents on disk.

You cannot specify $natural (page 563) sort order if the query includes a $text (page 417) expression.

Examples

Reverse Order Use -1 to return documents in the reverse order as they occur on disk:

```
db.collection.find().sort( { $natural: -1 } )
```

Natural Order Comparison To demonstrate natural ordering:

- Create an { normal: 1 } index on a collection (e.g. coll).
- Insert relevant objects with _id and normal values, for example, a document with _id and normal fields that both hold the same string:

  ```
  db.coll.insert( { _id: "01", normal: "01" } )
  ```

- Use values with different types for each document, but both fields in the document should have the same value:
- Use .find().sort().explain() for all operations.

This scenario returns these results when using find() (page 36) for different _id values; sorting with the $natural (page 563) operator, _id index, and normal index; and a description of the explain() (page 85) method output:

find() (page 36)	$natural (page 563):1	sort() (page 95) *_id*:1	normal:1
_id:ObjectId()	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor
_id:Object()	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor
_id:string()	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor
_id:integer()	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor
_id:BinData()	explain() (page 85) scanned entire collection	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor
normal:(any query)	explain() (page 85) scanned entire collection	explain() (page 85) used *B-Tree* cursor	explain() (page 85) used *B-Tree* cursor

Additional Information cursor.sort() (page 95)

2.4 Aggregation Reference

Aggregation Pipeline Quick Reference **(page 564)** Quick reference card for aggregation pipeline.

Aggregation Pipeline Operators **(page 482)** Aggregation pipeline operations have a collection of operators available to define and manipulate documents in pipeline stages.

Aggregation Commands Comparison **(page 569)** A comparison of group (page 216), mapReduce (page 220) and aggregate (page 210) that explores the strengths and limitations of each aggregation modality.

SQL to Aggregation Mapping Chart **(page 580)** An overview common aggregation operations in SQL and MongoDB using the aggregation pipeline and operators in MongoDB and common SQL statements.

Aggregation Interfaces **(page 573)** The data aggregation interfaces document the invocation format and output for MongoDB's aggregation commands and methods.

Variables in Aggregation Expressions **(page 573)** Use of variables in aggregation pipeline expressions.

2.4.1 Aggregation Pipeline Quick Reference

Stages

Pipeline stages appear in an array. Documents pass through the stages in sequence. All except the $out (page 491) and $geoNear (page 484) stages can appear multiple times in a pipeline.

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

Name	Description
$geoNear (page 484)	Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of $match (page 490), $sort (page 499), and $limit (page 489) for geospatial data. The output documents include an additional distance field and can include a location identifier field.
$group (page 486)	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
$limit (page 489)	Passes the first *n* documents unmodified to the pipeline where *n* is the specified limit. For each input document, outputs either one document (for the first *n* documents) or zero documents (after the first *n* documents).
$match (page 490)	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match (page 490) uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$out (page 491)	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out (page 491) stage, it must be the last stage in the pipeline.
$project (page 492)	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
$redact (page 495)	Reshapes each document in the stream by restricting the content for each document based on information stored in the documents themselves. Incorporates the functionality of $project (page 492) and $match (page 490). Can be used to implement field level redaction. For each input document, outputs either one or zero document.
$skip (page 499)	Skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents).
$sort (page 499)	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
$unwind (page 501)	Deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array.

Expressions

Expressions can include *field paths and system variables* (page 565), *literals* (page 566), *expression objects* (page 566), and *expression operators* (page 566). Expressions can be nested.

Field Path and System Variables

Aggregation expressions use *field path* to access fields in the input documents. To specify a field path, use a string that prefixes with a dollar sign `$` the field name or the dotted field name, if the field is in embedded document. For example, `"$user"` to specify the field path for the `user` field or `"$user.name"` to specify the field path to `"user.name"` field.

`"$<field>"` is equivalent to `"$$CURRENT.<field>"` where the `CURRENT` (page 574) is a system variable that defaults to the root of the current object in the most stages, unless stated otherwise in specific stages. `CURRENT` (page 574) can be rebound.

Along with the `CURRENT` (page 574) system variable, other *system variables* (page 573) are also available for use in expressions. To use user-defined variables, use `$let` (page 531) and `$map` (page 532) expressions. To access variables in expressions, use a string that prefixes the variable name with `$$`.

Literals

Literals can be of any type. However, MongoDB parses string literals that start with a dollar sign `$` as a path to a field and numeric/boolean literals in *expression objects* (page 566) as projection flags. To avoid parsing literals, use the `$literal` (page 533) expression.

Expression Objects

Expression objects have the following form:

```
{ <field1>: <expression1>, ... }
```

If the expressions are numeric or boolean literals, MongoDB treats the literals as projection flags (e.g. `1` or `true` to include the field), valid only in the `$project` (page 492) stage. To avoid treating numeric or boolean literals as projection flags, use the `$literal` (page 533) expression to wrap the numeric or boolean literals.

Operator Expressions

Operator expressions are similar to functions that take arguments. In general, these expressions take an array of arguments and have the following form:

```
{ <operator>: [ <argument1>, <argument2> ... ] }
```

If operator accepts a single argument, you can omit the outer array designating the argument list:

```
{ <operator>: <argument> }
```

To avoid parsing ambiguity if the argument is a literal array, you must wrap the literal array in a `$literal` (page 533) expression or keep the outer array that designates the argument list.

Boolean Expressions Boolean expressions evaluates its argument expressions as booleans and return a boolean as the result.

In addition to the `false` boolean value, Boolean expression evaluates as `false` the following: `null`, `0`, and `undefined` values. The Boolean expression evaluates all other values as `true`, including non-zero numeric values and arrays.

Name	Description
`$and` (page 503)	Returns `true` only when *all* its expressions evaluate to `true`. Accepts any number of argument expressions.
`$not` (page 504)	Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression.
`$or` (page 505)	Returns `true` when *any* of its expressions evaluates to `true`. Accepts any number of argument expressions.

Set Expressions Set expressions performs set operation on arrays, treating arrays as sets. Set expressions ignores the duplicate entries in each input array and the order of the elements.

If the set operation returns a set, the operation filters out duplicates in the result to output an array that contains only unique entries. The order of the elements in the output array is unspecified.

If a set contains a nested array element, the set expression does *not* descend into the nested array but evaluates the array at top-level.

Name	Description
$allElementsTru (page 506)	Returns true if *no* element of a set evaluates to false, otherwise, returns false. Accepts a single argument expression.
$anyElementTrue (page 507)	Returns true if *any* elements of a set evaluate to true; otherwise, returns false. Accepts a single argument expression.
$setDifference (page 509)	Returns a set with elements that appear in the first set but not in the second set; i.e. performs a relative complement[24] of the second set relative to the first. Accepts exactly two argument expressions.
$setEquals (page 510)	Returns true if the input sets have the same distinct elements. Accepts two or more argument expressions.
$setIntersectio (page 511)	Returns a set with elements that appear in *all* of the input sets. Accepts any number of argument expressions.
$setIsSubset (page 512)	Returns true if all elements of the first set appear in the second set, including when the first set equals the second set; i.e. not a strict subset[25]. Accepts exactly two argument expressions.
$setUnion (page 513)	Returns a set with elements that appear in *any* of the input sets. Accepts any number of argument expressions.

Comparison Expressions Comparison expressions return a boolean except for $cmp (page 514) which returns a number.

The comparison expressions take two argument expressions and compare both value and type, using the *specified BSON comparison order* for values of different types.

Name	Description
$cmp (page 514)	Returns: 0 if the two values are equivalent, 1 if the first value is greater than the second, and −1 if the first value is less than the second.
$eq (page 515)	Returns true if the values are equivalent.
$gt (page 516)	Returns true if the first value is greater than the second.
$gte (page 517)	Returns true if the first value is greater than or equal to the second.
$lt (page 518)	Returns true if the first value is less than the second.
$lte (page 519)	Returns true if the first value is less than or equal to the second.
$ne (page 519)	Returns true if the values are *not* equivalent.

Arithmetic Expressions Arithmetic expressions perform mathematic operations on numbers. Some arithmetic expressions can also support date arithmetic.

[24] http://en.wikipedia.org/wiki/Complement_(set_theory)
[25] http://en.wikipedia.org/wiki/Subset

Name	Description
$add (page 521)	Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.
$divide (page 522)	Returns the result of dividing the first number by the second. Accepts two argument expressions.
$mod (page 522)	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
$multiply (page 523)	Multiplies numbers to return the product. Accepts any number of argument expressions.
$subtract (page 523)	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds. If the two values are a date and a number in milliseconds, return the resulting date. Accepts two argument expressions. If the two values are a date and a number, specify the date argument first as it is not meaningful to subtract a date from a number.

String Expressions String expressions, with the exception of $concat (page 525), only have a well-defined behavior for strings of ASCII characters.

$concat (page 525) behavior is well-defined regardless of the characters used.

Name	Description
$concat (page 525)	Concatenates any number of strings.
$strcasecmp (page 525)	Performs case-insensitive string comparison and returns: 0 if two strings are equivalent, 1 if the first string is greater than the second, and −1 if the first string is less than the second.
$substr (page 526)	Returns a substring of a string, starting at a specified index position up to a specified length. Accepts three expressions as arguments: the first argument must resolve to a string, and the second and third arguments must resolve to integers.
$toLower (page 527)	Converts a string to lowercase. Accepts a single argument expression.
$toUpper (page 528)	Converts a string to uppercase. Accepts a single argument expression.

Text Search Expressions

Name	Description
$meta (page 529)	Access text search metadata.

Array Expressions

Name	Description
$size (page 530)	Returns the number of elements in the array. Accepts a single expression as argument.

Variable Expressions

Name	Description
$let (page 531)	Defines variables for use within the scope of a subexpression and returns the result of the subexpression. Accepts named parameters.
$map (page 532)	Applies a subexpression to each element of an array and returns the array of resulting values order. Accepts named parameters.

Literal Expressions

Name	Description
$literal (page 533)	Return a value without parsing. Use for values that the aggregation pipeline may interpret as a expression. For example, use a $literal (page 533) expression to a string that starts with a to avoid parsing as a field path.

Date Expressions

Name	Description
$dateToString (page 535)	Returns the date as a formatted string.
$dayOfMonth (page 536)	Returns the day of the month for a date as a number between 1 and 31.
$dayOfWeek (page 537)	Returns the day of the week for a date as a number between 1 (Sunday) and 7 (Saturday).
$dayOfYear (page 538)	Returns the day of the year for a date as a number between 1 and 366 (leap year).
$hour (page 539)	Returns the hour for a date as a number between 0 and 23.
$millisecond (page 540)	Returns the milliseconds of a date as a number between 0 and 999.
$minute (page 541)	Returns the minute for a date as a number between 0 and 59.
$month (page 542)	Returns the month for a date as a number between 1 (January) and 12 (December).
$second (page 542)	Returns the seconds for a date as a number between 0 and 60 (leap seconds).
$week (page 543)	Returns the week number for a date as a number between 0 (the partial week that precedes the first Sunday of the year) and 53 (leap year).
$year (page 544)	Returns the year for a date as a number (e.g. 2014).

Conditional Expressions

Name	Description
$cond (page 545)	A ternary operator that evaluates one expression, and depending on the result, returns the one of the other two expressions. Accepts either three expressions in an ordered list or thr named parameters.
$ifNull (page 547)	Returns either the non-null result of the first expression or the result of the second express the first expression results in a null result. Null result encompasses instances of undefined or missing fields. Accepts two expressions as arguments. The result of the second express be null.

Accumulators

Accumulators, available only for the $group (page 486) stage, compute values by combining documents that share the same group key. Accumulators take as input a single expression, evaluating the expression once for each input document, and maintain their state for the group of documents.

Name	Description
$addToSet (page 548)	Returns an array of *unique* expression values for each group. Order of the array elements is undefined.
$avg (page 549)	Returns an average for each group. Ignores non-numeric values.
$first (page 549)	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.
$last (page 550)	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.
$max (page 551)	Returns the highest expression value for each group.
$min (page 552)	Returns the lowest expression value for each group.
$push (page 553)	Returns an array of expression values for each group.
$sum (page 554)	Returns a sum for each group. Ignores non-numeric values.

2.4.2 Aggregation Commands Comparison

The following table provides a brief overview of the features of the MongoDB aggregation commands.

	aggregate (page 210)	mapReduce (page 220)	group (page 216)
Description	New in version 2.2. Designed with specific goals of improving performance and usability for aggregation tasks. Uses a "pipeline" approach where objects are transformed as they pass through a series of pipeline operators such as $group (page 486), $match (page 490), and $sort (page 499). See *Aggregation Pipeline Operators* (page 482) for more information on the pipeline operators.	Implements the Map-Reduce aggregation for processing large data sets.	Provides grouping functionality. Is slower than the aggregate (page 210) command and has less functionality than the mapReduce (page 220) command.
Key Features	Pipeline operators can be repeated as needed. Pipeline operators need not produce one output document for every input document. Can also generate new documents or filter out documents.	In addition to grouping operations, can perform complex aggregation tasks as well as perform incremental aggregation on continuously growing datasets. See http://docs.mongodb.org/manual/tutorial/map-reduce-examples/ and http://docs.mongodb.org/manual/tutorial/perform-incremental-	Can either group by existing fields or with a custom keyf JavaScript function, can group by calculated fields. See group (page 216) for information and example using the keyf function.
Flexibility	Limited to the operators and expressions supported by the aggregation pipeline. However, can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results by using the $project (page 492) pipeline operator. See $project (page 492) for more information as well as *Aggregation Pipeline Operators* (page 482) for more information on all the available pipeline operators.	Custom map, reduce and finalize JavaScript functions offer flexibility to aggregation logic. See mapReduce (page 220) for details and restrictions on the functions.	Custom reduce and finalize JavaScript functions offer flexibility to grouping logic. See group (page 216) for details and restrictions on these functions.
Output Results	Returns results in various options (inline as a document that contains the result set, a cursor to the result set) or stores the results in a collection. The result is subject to the *BSON Document size* (page 692) limit if returned inline as a document that contains the result set. Changed in version 2.6: Can return results as a cursor or store the results to a collection.	Returns results in various options (inline, new collection, merge, replace, reduce). See mapReduce (page 220) for details on the output options. Changed in version 2.2: Provides much better support for sharded map-reduce output than previous versions.	Returns results inline as an array of grouped items. The result set must fit within the *maximum BSON document size limit* (page 692). Changed in version 2.2: The returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. Previous versions had a limit of 10,000 elements.
Sharding	Supports non-sharded and sharded input collections.	Supports non-sharded and sharded input collections.	Does **not** support sharded collection.
Notes		Prior to 2.4, JavaScript code executed in a single thread.	Prior to 2.4, JavaScript code executed in a single thread.
More Information	See http://docs.mongodb.org/manual/core/aggregation-introduction and aggregate (page 210).	See http://docs.mongodb.org/manual/core/map-reduce and mapReduce (page 220).	See group (page 216).

2.4.3 SQL to Aggregation Mapping Chart

The `aggregation pipeline` allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL.

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB *aggregation operators* (page 482):

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match` (page 490)
GROUP BY	`$group` (page 486)
HAVING	`$match` (page 490)
SELECT	`$project` (page 492)
ORDER BY	`$sort` (page 499)
LIMIT	`$limit` (page 489)
SUM()	`$sum` (page 554)
COUNT()	`$sum` (page 554)
join	No direct corresponding operator; *however*, the `$unwind` (page 501) operator allows for somewhat similar functionality, but with fields embedded within the document.

Examples

The following table presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume *two* tables, `orders` and `order_lineitem` that join by the `order_lineitem.order_id` and the `orders.id` columns.
- The MongoDB examples assume *one* collection `orders` that contain documents of the following prototype:

```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
           { sku: "yyy", qty: 25, price: 1 } ]
}
```

SQL Example	MongoDB Example	Description
`SELECT COUNT(*) AS count FROM orders`	`db.orders.aggregate( [ { $group: { _id: null, count: { $sum: 1 } } } ] )`	Count all records from `orders`
`SELECT SUM(price) AS total FROM orders`	`db.orders.aggregate( [ { $group: { _id: null, total: { $sum: "$price" } } } ] )`	Sum the `price` field from `orders`
`SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id`	`db.orders.aggregate( [ { $group: { _id: "$cust_id", total: { $sum: "$price" } } } ] )`	For each unique `cust_id`, sum the `price` field.
`SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id ORDER BY total`	`db.orders.aggregate( [ { $group: { _id: "$cust_id", total: { $sum: "$price" } } }, { $sort: { total: 1 } } ] )`	For each unique `cust_id`, sum the `price` field, results sorted by sum.
`SELECT cust_id, ord_date, SUM(price) AS total FROM orders GROUP BY cust_id, ord_date`	`db.orders.aggregate( [ { $group: { _id: { cust_id: "$cust_id", ord_date: { month: { $month: "$ord_date" }, day: { $dayOfMonth: "$ord_date" }, year: { $year: "$ord_date"} } }, total: { $sum: "$price" } } } ] )`	For each unique `cust_id`, `ord_date` grouping, sum the `price` field. Excludes the time portion of the date.
`SELECT cust_id, count(*) FROM orders`	`db.orders.aggregate( [ { $group: {`	For `cust_id` with multiple records, return the `cust_id` and the corresponding record count.

2.4.4 Aggregation Interfaces

Aggregation Commands

Name	Description
aggregate (page 210)	Performs aggregation tasks such as group using the aggregation framework.
count (page 213)	Counts the number of documents in a collection.
distinct (page 215)	Displays the distinct values found for a specified key in a collection.
group (page 216)	Groups documents in a collection by the specified key and performs simple aggregation.
mapReduce (page 220)	Performs map-reduce aggregation for large data sets.

Aggregation Methods

Name	Description
db.collection.aggregate() (page 22)	Provides access to the aggregation pipeline.
db.collection.group() (page 51)	Groups documents in a collection by the specified key and performs simple aggregation.
db.collection.mapReduce() (page 58)	Performs map-reduce aggregation for large data sets.

2.4.5 Variables in Aggregation Expressions

Aggregation expressions (page 565) can use both user-defined and system variables.

Variables can hold any BSON type data. To access the value of the variable, use a string with the variable name prefixed with double dollar signs ($$).

If the variable references an object, to access a specific field in the object, use the dot notation; i.e. "$$<variable>.<field>".

User Variables

User variable names can contain the ascii characters [_a-zA-Z0-9] and any non-ascii character.

User variable names must begin with a lowercase ascii letter [a-z] or a non-ascii character.

System Variables

MongoDB offers the following system variables:

Variable	Description
ROOT	References the root document, i.e. the top-level document, currently being processed in the aggregation pipeline stage.
CURRENT	References the start of the field path being processed in the aggregation pipeline stage. Unless documented otherwise, all stages start with `CURRENT` (page 574) the same as `ROOT` (page 574). `CURRENT` (page 574) is modifiable. However, since `$<field>` is equivalent to `$$CURRENT.<field>`, rebinding `CURRENT` (page 574) changes the meaning of `$` accesses.
DESCEND	One of the allowed results of a `$redact` (page 495) expression.
PRUNE	One of the allowed results of a `$redact` (page 495) expression.
KEEP	One of the allowed results of a `$redact` (page 495) expression.

See also:

`$let` (page 531), `$redact` (page 495)

CHAPTER 3

MongoDB and SQL Interface Comparisons

3.1 SQL to MongoDB Mapping Chart

In addition to the charts that follow, you might want to consider the `http://docs.mongodb.org/manual/faq` section for a selection of common questions about MongoDB.

3.1.1 Terminology and Concepts

The following table presents the various SQL terminology and concepts and the corresponding MongoDB terminology and concepts.

SQL Terms/Concepts	MongoDB Terms/Concepts
database	*database*
table	*collection*
row	*document* or *BSON* document
column	*field*
index	*index*
table joins	embedded documents and linking
primary key	*primary key*
Specify any unique column or column combination as primary key.	In MongoDB, the primary key is automatically set to the *_id* field.
aggregation (e.g. group by)	aggregation pipeline See the *SQL to Aggregation Mapping Chart* (page 580).

3.1.2 Executables

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

	MongoDB	MySQL	Oracle	Informix	DB2
Database Server	`mongod` (page 583)	`mysqld`	`oracle`	`IDS`	`DB2 Server`
Database Client	`mongo` (page 610)	`mysql`	`sqlplus`	`DB-Access`	`DB2 Client`

3.1.3 Examples

The following table presents the various SQL statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume a table named `users`.
- The MongoDB examples assume a collection named `users` that contain documents of the following prototype:

```
{
  _id: ObjectId("509a8fb2f3f4948bd2f983a0"),
  user_id: "abc123",
  age: 55,
  status: 'A'
}
```

Create and Alter

The following table presents the various SQL statements related to table-level actions and the corresponding MongoDB statements.

SQL Schema Statements	MongoDB Schema Statements

`CREATE TABLE users (`
`  id MEDIUMINT NOT NULL`
`  AUTO_INCREMENT,`
`  user_id Varchar(30),`
`  age Number,`
`  status char(1),`
`  PRIMARY KEY (id)`
`)`
	Implicitly created on first `insert()` (page 55) operation. The primary key `_id` is automatically added if `_id` field is not specified.
	`db.users.insert( {`
	`  user_id: "abc123",`
	`  age: 55,`
	`  status: "A"`
	`} )`
	However, you can also explicitly create a collection:
	`db.createCollection("users")`

`ALTER TABLE users`
`ADD join_date DATETIME`
	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.
	However, at the document level, `update()` (page 72) operations can add fields to existing documents using the `$set` (page 459) operator.
	`db.users.update(`
	`  { },`
	`  { $set: { join_date: new Date() } },`
	`  { multi: true }`
	`)`

`ALTER TABLE users`
`DROP COLUMN join_date`
	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.
	However, at the document level, `update()` (page 72) operations can remove fields from documents using the `$unset` (page 461) operator.
	`db.users.update(`
	`  { },`
	`  { $unset: { join_date: "" } },`
	`  { multi: true }`
	`)`

`CREATE INDEX idx_user_id_asc` `ON users(user_id)`	`db.users.ensureIndex( { user_id: 1 } )`

`CREATE INDEX`
`  idx_user_id_asc_age_desc`
`ON users(user_id, age DESC)`
	`db.users.ensureIndex( { user_id: 1, age: -1 } )`

`DROP TABLE users`	`db.users.drop()`

For more information, see `db.collection.insert()` (page 55), `db.createCollection()` (page 104), `db.collection.update()` (page 72), `$set` (page 459), `$unset` (page 461), `db.collection.ensureIndex()` (page 30), `indexes`, `db.collection.drop()` (page 28), and `http://docs.mongodb.org/manual/core/data-models`.

Insert

The following table presents the various SQL statements related to inserting records into tables and the corresponding MongoDB statements.

SQL INSERT Statements	MongoDB insert() Statements
`INSERT INTO users(user_id, age, status) VALUES ("bcd001", 45, "A")`	`db.users.insert( { user_id: "bcd001", age: 45, status: "A" } )`

For more information, see `db.collection.insert()` (page 55).

Select

The following table presents the various SQL statements related to reading records from tables and the corresponding MongoDB statements.

SQL SELECT Statements	MongoDB find() Statements
`SELECT * FROM users`	`db.users.find()`
`SELECT id, user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1 } )`
`SELECT user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1, _id: 0 } )`
`SELECT * FROM users WHERE status = "A"`	`db.users.find( { status: "A" } )`
`SELECT user_id, status FROM users WHERE status = "A"`	`db.users.find( { status: "A" }, { user_id: 1, status: 1, _id: 0 } )`
`SELECT * FROM users WHERE status != "A"`	`db.users.find( { status: { $ne: "A" } } )`
`SELECT * FROM users WHERE status = "A" AND age = 50`	`db.users.find( { status: "A", age: 50 } )`
`SELECT * FROM users WHERE status = "A" OR age = 50`	`db.users.find( { $or: [ { status: "A" } , { age: 50 } ] } )`
`SELECT * FROM users WHERE age > 25`	`db.users.find( { age: { $gt: 25 } } )`
`SELECT * FROM users WHERE age < 25`	`db.users.find( { age: { $lt: 25 } } )`
`SELECT * FROM users WHERE age > 25 AND age <= 50`	`db.users.find( { age: { $gt: 25, $lte: 50 } } )`
`SELECT * FROM users WHERE user_id like "%bc%"`	`db.users.find( { user_id: /bc/ } )`

For more information, see db.collection.find() (page 36), db.collection.distinct() (page 28), db.collection.findOne() (page 46), $ne (page 404) $and (page 405), $or (page 408), $gt (page 401), $lt (page 403), $exists (page 409), $lte (page 403), $regex (page 414), limit() (page 88), skip() (page 94), explain() (page 85), sort() (page 95), and count() (page 83).

Update Records

The following table presents the various SQL statements related to updating existing records in tables and the corresponding MongoDB statements.

SQL Update Statements	MongoDB update() Statements

`UPDATE users`
`SET status = "C"`
`WHERE age > 25`
	`db.users.update(`
	`  { age: { $gt: 25 } },`
	`  { $set: { status: "C" } },`
	`  { multi: true }`
	`)`

`UPDATE users`
`SET age = age + 3`
`WHERE status = "A"`
	`db.users.update(`
	`  { status: "A" } ,`
	`  { $inc: { age: 3 } },`
	`  { multi: true }`
	`)`

For more information, see db.collection.update() (page 72), $set (page 459), $inc (page 452), and $gt (page 401).

Delete Records

The following table presents the various SQL statements related to deleting records from tables and the corresponding MongoDB statements.

SQL Delete Statements	MongoDB remove() Statements
`DELETE FROM users` `WHERE status = "D"`	`db.users.remove( { status: "D" } )`
`DELETE FROM users`	`db.users.remove({})`

For more information, see db.collection.remove() (page 66).

3.2 SQL to Aggregation Mapping Chart

The aggregation pipeline allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL.

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB *aggregation operators* (page 482):

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match` (page 490)
GROUP BY	`$group` (page 486)
HAVING	`$match` (page 490)
SELECT	`$project` (page 492)
ORDER BY	`$sort` (page 499)
LIMIT	`$limit` (page 489)
SUM()	`$sum` (page 554)
COUNT()	`$sum` (page 554)
join	No direct corresponding operator; *however*, the `$unwind` (page 501) operator allows for somewhat similar functionality, but with fields embedded within the document.

3.2.1 Examples

The following table presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume *two* tables, `orders` and `order_lineitem` that join by the `order_lineitem.order_id` and the `orders.id` columns.
- The MongoDB examples assume *one* collection `orders` that contain documents of the following prototype:

```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
           { sku: "yyy", qty: 25, price: 1 } ]
}
```

SQL Example	MongoDB Example	Description
`SELECT COUNT(*) AS count FROM orders`	`db.orders.aggregate( [ { $group: { _id: null, count: { $sum: 1 } } } ] )`	Count all records from `orders`
`SELECT SUM(price) AS total FROM orders`	`db.orders.aggregate( [ { $group: { _id: null, total: { $sum: "$price" } } } ] )`	Sum the `price` field from `orders`
`SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id`	`db.orders.aggregate( [ { $group: { _id: "$cust_id", total: { $sum: "$price" } } } ] )`	For each unique `cust_id`, sum the `price` field.
`SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id ORDER BY total`	`db.orders.aggregate( [ { $group: { _id: "$cust_id", total: { $sum: "$price" } } }, { $sort: { total: 1 } } ] )`	For each unique `cust_id`, sum the `price` field, results sorted by sum.
`SELECT cust_id, ord_date, SUM(price) AS total FROM orders GROUP BY cust_id, ord_date`	`db.orders.aggregate( [ { $group: { _id: { cust_id: "$cust_id", ord_date: { month: { $month: "$ord_date" }, day: { $dayOfMonth: "$ord_date" }, year: { $year: "$ord_date"} } }, total: { $sum: "$price" } } } ] )`	For each unique `cust_id`, `ord_date` grouping, sum the `price` field. Excludes the time portion of the date.
`SELECT cust_id, count(*) FROM orders`	`db.orders.aggregate( [ { $group: {`	For `cust_id` with multiple records, return the `cust_id` and the corresponding record count.

CHAPTER 4

Program and Tool Reference Pages

4.1 MongoDB Package Components

4.1.1 Core Processes

The core components in the MongoDB package are: `mongod` (page 583), the core database process; `mongos` (page 601) the controller and query router for *sharded clusters*; and `mongo` (page 610) the interactive MongoDB Shell.

mongod

Synopsis

`mongod` (page 583) is the primary daemon process for the MongoDB system. It handles data requests, manages data access, and performs background management operations.

This document provides a complete overview of all command line options for `mongod` (page 583). These command line options are primarily useful for testing: In common operation, use the `configuration file options` to control the behavior of your database.

Options

mongod

Core Options

mongod

command line option!–help, -h

--help, -h
Returns information on the options and use of `mongod` (page 583).

command line option!–version

--version
Returns the `mongod` (page 583) release number.

command line option!–config <filename>, -f <filename>

--config <filename>, **-f** <filename>
Specifies a configuration file for runtime configuration options. The configuration file is the preferred method for runtime configuration of mongod (page 583). The options are equivalent to the command-line configuration options. See http://docs.mongodb.org/manual/reference/configuration-options for more information.

Ensure the configuration file uses ASCII encoding. The mongod (page 583) instance does not support configuration files with non-ASCII encoding, including UTF-8.

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

command line option!–quiet

--quiet
Runs the mongod (page 583) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

•output from *database commands*

•replication activity

•connection accepted events

•connection closed events

command line option!–port <port>

--port <port>
Default: 27017

Specifies the TCP port on which the MongoDB instance listens for client connections.

command line option!–bind_ip <ip address>

--bind_ip <ip address>
Default: All interfaces.

Changed in version 2.6.0: The deb and rpm packages include a default configuration file that sets *--bind_ip* (page 584) to 127.0.0.1.

Specifies the IP address that mongod (page 583) binds to in order to listen for connections from applications. You may attach mongod (page 583) to any interface. When attaching mongod (page 583) to a publicly accessible interface, ensure that you have implemented proper authentication and firewall restrictions to protect the integrity of your database.

command line option!–maxConns <number>

--maxConns <number>
The maximum number of simultaneous connections that mongod (page 583) will accept. This setting has no effect if it is higher than your operating system's configured maximum connection tracking threshold.

Changed in version 2.6: MongoDB removed the upward limit on the maxIncomingConnections setting.

command line option!–syslog

--syslog
Sends all logging output to the host's *syslog* system rather than to standard output or to a log file. , as with *--logpath* (page 585).

The *--syslog* (page 584) option is not supported on Windows.

command line option!–syslogFacility <string>

--syslogFacility <string>
Default: user

Specifies the facility level used when logging messages to syslog. The value you specify must be supported by your operating system's implementation of syslog. To use this option, you must enable the *--syslog* (page 584) option.

command line option!–logpath <path>

--logpath <path>
Sends all diagnostic logging information to a log file instead of to standard output or to the host's *syslog* system. MongoDB creates the log file at the path you specify.

By default, MongoDB overwrites the log file when the process restarts. To instead append to the log file, set the *--logappend* (page 585) option.

command line option!–logappend

--logappend
Appends new entries to the end of the log file rather than overwriting the content of the log when the mongod (page 583) instance restarts.

command line option!–timeStampFormat <string>

--timeStampFormat <string>
Default: iso8601-local

The time format for timestamps in log messages. Specify one of the following values:

Value	Description
ctime	Displays timestamps as `Wed Dec 31 18:17:54.811`.
iso8601-utc	Displays timestamps in Coordinated Universal Time (UTC) in the ISO-8601 format. For example, for New York at the start of the Epoch: `1970-01-01T00:00:00.000Z`
iso8601-local	Displays timestamps in local time in the ISO-8601 format. For example, for New York at the start of the Epoch: `1969-12-31T19:00:00.000+0500`

command line option!–diaglog <value>

--diaglog <value>
Default: 0

Deprecated since version 2.6.

--diaglog (page 585) is for internal use and not intended for most users.

Creates a very verbose *diagnostic log* for troubleshooting and recording various errors. MongoDB writes these log files in the `dbPath` directory in a series of files that begin with the string `diaglog` and end with the initiation time of the logging as a hex string.

The specified value configures the level of verbosity:

Value	Setting
0	Off. No logging.
1	Log write operations.
2	Log read operations.
3	Log both read and write operations.
7	Log write and some read operations.

You can use the mongosniff (page 670) tool to replay this output for investigation. Given a typical diaglog file located at `/data/db/diaglog.4f76a58c`, you might use a command in the following form to read these files:

```
mongosniff --source DIAGLOG /data/db/diaglog.4f76a58c
```

> **Warning:** Setting the diagnostic level to `0` will cause `mongod` (page 583) to stop writing data to the *diagnostic log* file. However, the `mongod` (page 583) instance will continue to keep the file open, even if it is no longer writing data to the file. If you want to rename, move, or delete the diagnostic log you must cleanly shut down the `mongod` (page 583) instance before doing so.

command line option!–traceExceptions

`--traceExceptions`
For internal diagnostic use only.

command line option!–pidfilepath <path>

`--pidfilepath` `<path>`
Specifies a file location to hold the process ID of the `mongod` (page 583) process where `mongod` (page 583) will write its PID. This is useful for tracking the `mongod` (page 583) process in combination with the `--fork` (page 587) option. Without a specified `--pidfilepath` (page 586) option, the process creates no PID file.

command line option!–keyFile <file>

`--keyFile` `<file>`
Specifies the path to a key file that stores the shared secret that MongoDB instances use to authenticate to each other in a *sharded cluster* or *replica set*. `--keyFile` (page 586) implies `--auth` (page 587). See *inter-process-auth* for more information.

command line option!–setParameter <options>

`--setParameter` `<options>`
Specifies one of the MongoDB parameters described in `http://docs.mongodb.org/manual/reference/parameter`
You can specify multiple `setParameter` fields.

command line option!–httpinterface

`--httpinterface`
New in version 2.6.

Enables the HTTP interface. Enabling the interface can increase network exposure.

Leave the HTTP interface *disabled* for production deployments. If you *do* enable this interface, you should only allow trusted clients to access this port. See *security-firewalls*.

Note: In MongoDB Enterprise, the HTTP Console does not support Kerberos Authentication.

---

command line option!–nohttpinterface

`--nohttpinterface`
Deprecated since version 2.6: MongoDB disables the HTTP interface by default.

Disables the HTTP interface.

Do not use in conjunction with `--rest` (page 587) or `--jsonp` (page 587).

Note: In MongoDB Enterprise, the HTTP Console does not support Kerberos Authentication.

---

command line option!–nounixsocket

`--nounixsocket`
Disables listening on the UNIX domain socket. The `mongod` (page 583) process always listens on the UNIX socket unless one of the following is true:

- *--nounixsocket* (page 586) is set
- bindIp is not set
- bindIp does not specify 127.0.0.1

New in version 2.6: mongod (page 583) installed from official .deb and .rpm packages have the bind_ip configuration set to 127.0.0.1 by default.

command line option!–unixSocketPrefix <path>

--unixSocketPrefix <path>
Default: /tmp

The path for the UNIX socket. If this option has no value, the mongod (page 583) process creates a socket with http://docs.mongodb.org/manual/tmp as a prefix. MongoDB creates and listens on a UNIX socket unless one of the following is true:

- *--nounixsocket* (page 586) is set
- bindIp is not set
- bindIp does not specify 127.0.0.1

command line option!–fork

--fork
Enables a *daemon* mode that runs the mongod (page 583) process in the background. By default mongod (page 583) does not run as a daemon: typically you will run mongod (page 583) as a daemon, either by using *--fork* (page 587) or by using a controlling process that handles the daemonization process (e.g. as with upstart and systemd).

command line option!–auth

--auth
Enables authorization to control user's access to database resources and operations. When authorization is enabled, MongoDB requires all clients to authenticate themselves first in order to determine the access for the client.

Configure users via the *mongo shell* (page 610). If no users exist, the localhost interface will continue to have access to the database until you create the first user.

See Security for more information.

command line option!–noauth

--noauth
Disables authentication. Currently the default. Exists for future compatibility and clarity.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the mongod (page 583) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–jsonp

--jsonp
Permits *JSONP* access via an HTTP interface. Enabling the interface can increase network exposure. The *--jsonp* (page 587) option enables the HTTP interface, even if the HTTP interface option is disabled.

command line option!–rest

--rest
Enables the simple *REST* API. Enabling the *REST* API enables the HTTP interface, even if the `HTTP interface` option is disabled, and as a result can increase network exposure.

command line option!–slowms <integer>

--slowms `<integer>`
Default: 100

The threshold in milliseconds at which the database profiler considers a query slow. MongoDB records all slow queries to the log, even when the database profiler is off. When the profiler is on, it writes to the `system.profile` collection. See the `profile` (page 366) command for more information on the database profiler.

command line option!–profile <level>

--profile `<level>`
Default: 0

Changes the level of database profiling, which inserts information about operation performance into standard output or a log file. Specify one of the following levels:

Level	Setting
0	Off. No profiling.
1	On. Only includes slow operations.
2	On. Includes all operations.

Database profiling can impact database performance. Enable this option only after careful consideration.

command line option!–cpu

--cpu
Forces the `mongod` (page 583) process to report the percentage of CPU time in write lock, every four seconds.

command line option!–sysinfo

--sysinfo
Returns diagnostic system information and then exits. The information provides the page size, the number of physical pages, and the number of available physical pages.

command line option!–objcheck

--objcheck
Forces the `mongod` (page 583) to validate all requests from clients upon receipt to ensure that clients never insert invalid documents into the database. For objects with a high degree of sub-document nesting, the *--objcheck* (page 588) option can have a small impact on performance. You can set *--noobjcheck* (page 588) to disable object checking at runtime.

Changed in version 2.4: MongoDB enables the *--objcheck* (page 588) option by default in order to prevent any client from inserting malformed or invalid BSON into a MongoDB database.

command line option!–noobjcheck

--noobjcheck
New in version 2.4.

Disables the default document validation that MongoDB performs on all incoming BSON documents.

command line option!–noscripting

--noscripting
Disables the scripting engine.

command line option!–notablescan

--notablescan
Forbids operations that require a table scan. See `notablescan` for additional information.

command line option!–shutdown

--shutdown
The *--shutdown* (page 589) option cleanly and safely terminates the `mongod` (page 583) process. When invoking `mongod` (page 583) with this option you must set the *--dbpath* (page 589) option either directly or by way of the `configuration file` and the *--config* (page 583) option.

The *--shutdown* (page 589) option is available only on Linux systems.

Storage Options command line option!–dbpath <path>

--dbpath `<path>`
Default: `/data/db` on Linux and OS X, `\data\db` on Windows

The directory where the `mongod` (page 583) instance stores its data.

If you installed MongoDB using a package management system, check the `/etc/mongodb.conf` file provided by your packages to see the directory is specified.

Changed in version 2.8: The files in *--dbpath* (page 589) must correspond to the storage engine specified in *--storageEngine* (page 589). If the data files do not correspond to *--storageEngine* (page 589), `mongod` (page 583) will refuse to start.

command line option!–storageEngine string

--storageEngine `string`
Default: `mmapv1`

New in version 2.8.0.

Specifies the storage engine for the `mongod` (page 583) database.

If you attempt to start a `mongod` (page 583) with a *storage.dbPath* that contains data files produced by a storage engine other than the one specified by *--storageEngine* (page 589), `mongod` (page 583) will refuse to start.

command line option!–wiredTigerDirectoryForIndexes

--wiredTigerDirectoryForIndexes
New in version 2.8.0.

When you start `mongod` (page 583) with *--wiredTigerDirectoryForIndexes* (page 589), `mongod` (page 583) stores indexes and collections in separate directories.

command line option!–wiredTigerCacheSizeGB number

--wiredTigerCacheSizeGB `number`
Default: the maximum of half of physical RAM or 1 gigabyte

New in version 2.8.0.

Defines the maximum size of the cache that WiredTiger will use for all data. Ensure that *--wiredTigerCacheSizeGB* (page 589) is sufficient to hold the entire working set for the `mongod` (page 583) instance.

command line option!–wiredTigerCheckpointDelaySecs <seconds>

--wiredTigerCheckpointDelaySecs `<seconds>`
Default: 60

New in version 2.8.0.

Defines the interval between checkpoints when WiredTiger writes all modified data to the data files in `dbPath`. If the `mongod` (page 583) exits between checkpoints and you do not have `storage.journal.enabled` set to `true`, any data modified since the last checkpoint will not persist. The data files are *always* valid even if `mongod` (page 583) exits between or during a checkpoint.

command line option!–wiredTigerStatisticsLogDelaySecs <seconds>

--wiredTigerStatisticsLogDelaySecs `<seconds>`
Default: 0

New in version 2.8.0.

When `0` WiredTiger will not log statistics. Otherwise WiredTiger will log statistics to a file in the `dbPath` on the interval defined by *`--wiredTigerStatisticsLogDelaySecs`* (page 590).

command line option!–wiredTigerJournalCompressor <compressor>

--wiredTigerJournalCompressor `<compressor>`
Default: snappy

New in version 2.8.0.

Specifies the type of compression to use to compress the journal data (i.e. `storage.journal`.)

Available compressors are:

- `none`
- `snappy`
- `zlib`

command line option!–wiredTigerCollectionBlockCompressor <compressor>

--wiredTigerCollectionBlockCompressor `<compressor>`
Default: snappy

New in version 2.8.0.

Specifies the default type of compression to use to compress collection data. You can override this on a per-collection basis when creating collections.

Available compressors are:

- `none`
- `snappy`
- `zlib`

command line option!–wiredTigerIndexBlockCompressor <compressor>

--wiredTigerIndexBlockCompressor `<compressor>`
Default: none

New in version 2.8.0.

Specifies the default type of compression to use to compress index data. You can override this on a per-index basis when creating indexes.

Available compressors are:

- `none`
- `snappy`
- `zlib`

command line option!–wiredTigerCollectionPrefixCompression <boolean>

--wiredTigerCollectionPrefixCompression <boolean>
Default: false

New in version 2.8.0.

Specify `true` for *--wiredTigerCollectionPrefixCompression* (page 591) to enable prefix compression for collection data.

command line option!–wiredTigerIndexPrefixCompression <boolean>

--wiredTigerIndexPrefixCompression <boolean>
Default: true

New in version 2.8.0.

Specify `true` for *--wiredTigerIndexPrefixCompression* (page 591) to enable prefix compression for index data.

command line option!–directoryperdb

--directoryperdb
Stores each database's files in its own folder in the *data directory*. When applied to an existing system, the *--directoryperdb* (page 591) option alters the storage pattern of the data directory.

Use this option in conjunction with your file system and device configuration so that MongoDB will store data on a number of distinct disk devices to increase write throughput or disk capacity.

Warning: To enable this option for an **existing** system, migrate the database-specific data files to the new directory structure before enabling *--directoryperdb* (page 591). Database-specific data files begin with the name of an existing database and end with either "ns" or a number. For example, the following data directory includes files for the `local` and `test` databases:

```
journal
mongod.lock
local.0
local.1
local.ns
test.0
test.1
test.ns
```

After migration, the data directory would have the following structure:

```
journal
mongod.lock
local/local.0
local/local.1
local/local.ns
test/test.0
test/test.1
test/test.ns
```

command line option!–noIndexBuildRetry

--noIndexBuildRetry
Stops the `mongod` (page 583) from rebuilding incomplete indexes on the next start up. This applies in cases where the `mongod` (page 583) restarts after it has shut down or stopped in the middle of an index build. In such cases, the `mongod` (page 583) always removes any incomplete indexes, and then also, by default, attempts

to rebuild them. To stop the mongod (page 583) from rebuilding incomplete indexes on start up, include this option on the command-line.

command line option!–noprealloc

--noprealloc
Deprecated since version 2.6.

Disables the preallocation of data files. Currently the default. Exists for future compatibility and clarity.

command line option!–nssize <value>

--nssize <value>
Default: 16

Specifies the default size for namespace files, which are files that end in `.ns`. Each collection and index counts as a namespace.

Use this setting to control size for newly created namespace files. This option has no impact on existing files. The maximum size for a namespace file is 2047 megabytes. The default value of 16 megabytes provides for approximately 24,000 namespaces.

command line option!–quota

--quota
Enables a maximum limit for the number data files each database can have. When running with the *--quota* (page 592) option, MongoDB has a maximum of 8 data files per database. Adjust the quota with *--quotaFiles* (page 592).

command line option!–quotaFiles <number>

--quotaFiles <number>
Default: 8

Modifies the limit on the number of data files per database. *--quotaFiles* (page 592) option requires that you set *--quota* (page 592).

command line option!–smallfiles

--smallfiles
Sets MongoDB to use a smaller default file size. The *--smallfiles* (page 592) option reduces the initial size for data files and limits the maximum size to 512 megabytes. *--smallfiles* (page 592) also reduces the size of each *journal* file from 1 gigabyte to 128 megabytes. Use *--smallfiles* (page 592) if you have a large number of databases that each holds a small quantity of data.

The *--smallfiles* (page 592) option can lead the mongod (page 583) instance to create a large number of files, which can affect performance for larger databases.

command line option!–syncdelay <value>

--syncdelay <value>
Default: 60

Controls how much time can pass before MongoDB flushes data to the data files via an *fsync* operation. **Do not set this value on production systems.** In almost every situation, you should use the default setting.

> **Warning:** If you set *--syncdelay* (page 592) to `0`, MongoDB will not sync the memory mapped files to disk.

The mongod (page 583) process writes data very quickly to the journal and lazily to the data files. `syncPeriodSecs` has no effect on the `journal` files or `journaling`.

The serverStatus (page 366) command reports the background flush thread's status via the backgroundFlushing (page 373) field.

command line option!–upgrade

--upgrade
Upgrades the on-disk data format of the files specified by the *--dbpath* (page 589) to the latest version, if needed.

This option only affects the operation of the mongod (page 583) if the data files are in an old format.

In most cases you should not set this value, so you can exercise the most control over your upgrade process. See the MongoDB release notes[1] (on the download page) for more information about the upgrade process.

command line option!–repair

--repair
Runs a repair routine on all databases. This is equivalent to shutting down and running the repairDatabase (page 341) database command on all databases.

Warning: During normal operations, only use the repairDatabase (page 341) command and wrappers including db.repairDatabase() (page 123) in the mongo (page 610) shell and *mongod --repair*, to compact database files and/or reclaim disk space. Be aware that these operations remove and do not save any corrupt data during the repair process.
If you are trying to repair a *replica set* member, and you have access to an intact copy of your data (e.g. a recent backup or an intact member of the *replica set*), you should restore from that intact copy, and **not** use repairDatabase (page 341).

When using *journaling*, there is almost never any need to run repairDatabase (page 341). In the event of an unclean shutdown, the server will be able to restore the data files to a pristine state automatically.

Changed in version 2.1.2.

If you run the repair option *and* have data in a journal file, the mongod (page 583) instance refuses to start. In these cases you should start the mongod (page 583) without the *--repair* (page 593) option, which allows the mongod (page 583) to recover data from the journal. This completes more quickly and is more likely to produce valid data files. To continue the repair operation despite the journal files, shut down the mongod (page 583) cleanly and restart with the *--repair* (page 593) option.

The *--repair* (page 593) option copies data from the source data files into new data files in the repairPath and then replaces the original data files with the repaired data files.

command line option!–repairpath <path>

--repairpath <path>
Default: A _tmp directory within the path specified by the dbPath option.

Specifies a working directory that MongoDB will use during the *--repair* (page 593) operation. After *--repair* (page 593) completes, the data files in dbPath and the *--repairpath* (page 593) directory is empty.

The *--repairpath* (page 593) must be within the dbPath. You can specify a symlink to *--repairpath* (page 593) to use a path on a different file system.

command line option!–journal

--journal
Enables the durability *journal* to ensure data files remain valid and recoverable. This option applies only when you specify the *--dbpath* (page 589) option. The mongod (page 583) enables journaling by default on 64-bit builds of versions after 2.0.

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

command line option!–nojournal

--nojournal
Disables the durability journaling. The mongod (page 583) instance enables journaling by default in 64-bit versions after v2.0.

command line option!–journalOptions <arguments>

--journalOptions <arguments>
Provides functionality for testing. Not for general use, and will affect data file integrity in the case of abnormal system shutdown.

command line option!–journalCommitInterval <value>

--journalCommitInterval <value>
Default: 100 or 30

The maximum amount of time the mongod (page 583) process allows between journal operations. Values can range from 2 to 300 milliseconds. Lower values increase the durability of the journal, at the expense of disk performance.

The default journal commit interval is 100 milliseconds if a single block device (e.g. physical volume, RAID device, or LVM volume) contains both the journal and the data files.

If the journal is on a different block device than the data files the default journal commit interval is 30 milliseconds.

To force mongod (page 583) to commit to the journal more frequently, you can specify `j:true`. When a write operation with `j:true` is pending, mongod (page 583) will reduce `commitIntervalMs` to a third of the set value.

Replication Options command line option!–replSet <setname>

--replSet <setname>
Configures replication. Specify a replica set name as an argument to this set. All hosts in the replica set must have the same set name.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

command line option!–oplogSize <value>

--oplogSize <value>
Specifies a maximum size in megabytes for the replication operation log (i.e., the *oplog*). The mongod (page 583) process creates an *oplog* based on the maximum amount of space available. For 64-bit systems, the oplog is typically 5% of available disk space. Once the mongod (page 583) has created the oplog for the first time, changing the *--oplogSize* (page 594) option will not affect the size of the oplog.

See *replica-set-oplog-sizing* for more information.

command line option!–replIndexPrefetch

--replIndexPrefetch
Default: all

New in version 2.2.

Storage Engine Specific Feature

--replIndexPrefetch (page 594) is only available with the `mmapv1` storage engine.

Determines which indexes *secondary* members of a *replica set* load into memory before applying operations from the oplog. By default secondaries load all indexes related to an operation into memory before applying operations from the oplog. This option can have one of the following values:

Value	Description
`none`	Secondaries do not load indexes into memory.
`all`	Secondaries load all indexes related to an operation.
`_id_only`	Secondaries load no additional indexes into memory beyond the already existing `_id` index.

Master-Slave Replication These options provide access to conventional master-slave database replication. While this functionality remains accessible in MongoDB, replica sets are the preferred configuration for database replication. command line option!–master

`--master`
Configures the `mongod` (page 583) to run as a replication *master*.

command line option!–slave

`--slave`
Configures the `mongod` (page 583) to run as a replication *slave*.

command line option!–source <host><:port>

`--source` `<host><:port>`
For use with the `--slave` (page 595) option, the `--source` option designates the server that this instance will replicate.

command line option!–only <arg>

`--only` `<arg>`
For use with the `--slave` (page 595) option, the `--only` option specifies only a single *database* to replicate.

command line option!–slavedelay <value>

`--slavedelay` `<value>`
For use with the `--slave` (page 595) option, the `--slavedelay` (page 595) option configures a "delay" in seconds, for this slave to wait to apply operations from the *master* node.

command line option!–autoresync

`--autoresync`
For use with the `--slave` (page 595) option. When set, the `--autoresync` (page 595) option allows this slave to automatically resync if it is more than 10 seconds behind the master. This setting may be problematic if the `--oplogSize` (page 594) specifies a too small oplog.

If the *oplog* is not large enough to store the difference in changes between the master's current state and the state of the slave, this instance will forcibly resync itself unnecessarily. If you don't specify `--autoresync` (page 595), the slave will not attempt an automatic resync more than once in a ten minute period.

command line option!–fastsync

`--fastsync`
In the context of *replica set* replication, set this option if you have seeded this member with an up-to-date copy of the entire `dbPath` of another member of the set. Otherwise the `mongod` (page 583) will attempt to perform an initial sync, as though the member were a new member.

> **Warning:** If the data is not perfectly synchronized *and* the `mongod` (page 583) starts with `fastsync`, then the secondary or slave will be permanently out of sync with the primary, which may cause significant consistency problems.

Sharded Cluster Options command line option!–configsvr

--configsvr

Declares that this mongod (page 583) instance serves as the *config database* of a sharded cluster. When running with this option, clients (i.e. other cluster components) will not be able to write data to any database other than config and admin. The default port for a mongod (page 583) with this option is 27019 and the default *--dbpath* (page 589) directory is /data/configdb, unless specified.

Changed in version 2.2: The *--configsvr* (page 596) option also sets *--smallfiles* (page 592).

Changed in version 2.4: The *--configsvr* (page 596) option creates a local *oplog*.

Do not use the *--configsvr* (page 596) option with *--replSet* (page 594) or *--shardsvr* (page 596). Config servers cannot be a shard server or part of a *replica set*.

command line option!–shardsvr

--shardsvr

Configures this mongod (page 583) instance as a shard in a partitioned cluster. The default port for these instances is 27018. The only effect of *--shardsvr* (page 596) is to change the port number.

SSL Options

See

http://docs.mongodb.org/manual/tutorial/configure-ssl for full documentation of MongoDB's support.

command line option!–sslOnNormalPorts

--sslOnNormalPorts

Deprecated since version 2.6.

Enables SSL for mongod (page 583).

With *--sslOnNormalPorts* (page 596), a mongod (page 583) requires SSL encryption for all connections on the default MongoDB port, or the port specified by *--port* (page 584). By default, *--sslOnNormalPorts* (page 596) is disabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslMode <mode>

--sslMode <mode>

New in version 2.6.

Enables SSL or mixed SSL used for all network connections. The argument to the *--sslMode* (page 596) option can be one of the following:

Value	Description
disabled	The server does not use SSL.
allowSSL	Connections between servers do not use SSL. For incoming connections, the server accepts both SSL and non-SSL.
preferSSL	Connections between servers use SSL. For incoming connections, the server accepts both SSL and non-SSL.
requireSSL	The server uses and accepts only SSL encrypted connections.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.2.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths.

When SSL is enabled, you must specify *--sslPEMKeyFile* (page 596).

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.2.

Specifies the password to de-crypt the certificate-key file (i.e. --sslPEMKeyFile). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongod (page 583) will redact the password from all logging and reporting output.

Changed in version 2.6: If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the mongod (page 583) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–clusterAuthMode <option>

--clusterAuthMode <option>
Default: keyFile

New in version 2.6.

The authentication mode used for cluster authentication. If you use *internal x.509 authentication*, specify so here. This option can have one of the following values:

Value	Description
keyFile	Use a keyfile for authentication. Accept only keyfiles.
sendKeyFile	For rolling upgrade purposes. Send a keyfile for authentication but can accept both keyfiles and x.509 certificates.
sendX509	For rolling upgrade purposes. Send the x.509 certificate for authentication but can accept both keyfiles and x.509 certificates.
x509	Recommended. Send the x.509 certificate for authentication and accept only x.509 certificates.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslClusterFile <filename>

--sslClusterFile <filename>
New in version 2.6.

Specifies the .pem file that contains the x.509 certificate-key file for *membership authentication* for the cluster or replica set.

If *--sslClusterFile* (page 597) does not specify the .pem file for internal cluster authentication, the cluster uses the .pem file specified in the *--sslPEMKeyFile* (page 596) option.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslClusterPassword <value>

--sslClusterPassword <value>
New in version 2.6.

Specifies the password to de-crypt the x.509 certificate-key file specified with `--sslClusterFile`. Use the *--sslClusterPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongod (page 583) will redact the password from all logging and reporting output.

If the x.509 key file is encrypted and you do not specify the *--sslClusterPassword* (page 597) option, the mongod (page 583) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.4.

Specifies the `.pem` file that contains the root certificate chain from the Certificate Authority. Specify the file name of the `.pem` file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

> **Warning:** If the *--sslCAFile* option and its target file are not specified, x.509 client and member authentication will not function. mongod (page 583), and mongos (page 601) in sharded systems, will not be able to verify the certificates of processes connecting to it against the trusted certificate authority (CA) that issued them, breaking the certificate chain.
> As of version 2.6.4, mongod (page 583) will not start with x.509 authentication enabled if the CA file is not specified.

command line option!–sslCRLFile <filename>

--sslCRLFile <filename>
New in version 2.4.

Specifies the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for SSL certificates on other servers in the cluster and allows the use of invalid certificates. When using the `allowInvalidCertificates` setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates, when connecting to other mongod (page 583) instances for inter-process authentication. This allows mongod (page 583) to connect to other mongod (page 583) instances if the hostnames in their certificates do not match their configured hostname.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowConnectionsWithoutCertificates

`--sslAllowConnectionsWithoutCertificates`

New in version 2.4.

Changed in version 2.8.0: `--sslAllowConnectionsWithoutCertificates` became *`--sslAllowConnectionsWithoutCertificates`* (page 599). For compatibility, MongoDB processes continue to accept `--sslAllowConnectionsWithoutCertificates`, but all users should update their configuration files.

Disables the requirement for SSL certificate validation that `--sslCAFile` enables. With the *`--sslAllowConnectionsWithoutCertificates`* (page 599) option, the `mongod` (page 583) will accept connections when the client does not present a certificate when establishing the connection.

If the client presents a certificate and the `mongod` (page 583) has *`--sslAllowConnectionsWithoutCertificates`* (page 599) enabled, the `mongod` (page 583) will validate the certificate using the root certificate chain specified by `--sslCAFile` and reject clients with invalid certificates.

Use the *`--sslAllowConnectionsWithoutCertificates`* (page 599) option if you have a mixed deployment that includes clients that do not or cannot present certificates to the `mongod` (page 583).

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslFIPSMode

`--sslFIPSMode`

New in version 2.4.

Directs the `mongod` (page 583) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[2]. See `http://docs.mongodb.org/manual/tutorial/configure-fips` for more information.

Audit Options command line option!–auditDestination

`--auditDestination`

New in version 2.6.

Enables `auditing`. The *`--auditDestination`* (page 599) option can have one of the following values:

Value	Description
`syslog`	Output the audit events to syslog in JSON format. Not available on Windows. Audit messages have a syslog severity level of `info` and a facility level of `user`. The syslog message limit can result in the truncation of audit messages. The auditing system will neither detect the truncation nor error upon its occurrence.
`console`	Output the audit events to `stdout` in JSON format.
`file`	Output the audit events to the file specified in *`--auditPath`* (page 600) in the format specified in *`--auditFormat`* (page 600).

Note: Available only in MongoDB Enterprise[3].

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

command line option!–auditFormat

--auditFormat
New in version 2.6.

Specifies the format of the output file for `auditing` if *--auditDestination* (page 599) is `file`. The *--auditFormat* (page 600) option can have one of the following values:

Value	Description
`JSON`	Output the audit events in JSON format to the file specified in *--auditPath* (page 600).
`BSON`	Output the audit events in BSON binary format to the file specified in *--auditPath* (page 600).

Printing audit events to a file in JSON format degrades server performance more than printing to a file in BSON format.

Note: Available only in MongoDB Enterprise[4].

command line option!–auditPath

--auditPath
New in version 2.6.

Specifies the output file for `auditing` if *--auditDestination* (page 599) has value of `file`. The *--auditPath* (page 600) option can take either a full path name or a relative path name.

Note: Available only in MongoDB Enterprise[5].

command line option!–auditFilter

--auditFilter
New in version 2.6.

Specifies the filter to limit the *types of operations* the `audit system` records. The option takes a string representation of a query document of the form:

```
{ <field1>: <expression1>, ... }
```

The `<field>` can be `any field in the audit message`, including fields returned in the *param* document. The `<expression>` is a *query condition expression* (page 400).

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

To specify the audit filter in a `configuration file`, you must use the YAML format of the configuration file.

{ atype: <expression>, “param.db”: <database> }

Note: Available only in MongoDB Enterprise[6].

SNMP Options command line option!–snmp-subagent

--snmp-subagent
Runs SNMP as a subagent. For more information, see `http://docs.mongodb.org/manual/tutorial/monitor-wit`

command line option!–snmp-master

[4] http://www.mongodb.com/products/mongodb-enterprise
[5] http://www.mongodb.com/products/mongodb-enterprise
[6] http://www.mongodb.com/products/mongodb-enterprise

--snmp-master
Runs SNMP as a master. For more information, see http://docs.mongodb.org/manual/tutorial/monitor-with

mongos

Synopsis

mongos (page 601) for "MongoDB Shard," is a routing service for MongoDB shard configurations that processes queries from the application layer, and determines the location of this data in the *sharded cluster*, in order to complete these operations. From the perspective of the application, a mongos (page 601) instance behaves identically to any other MongoDB instance.

Considerations

Never change the name of the mongos (page 601) binary.

Options

mongos

Core Options
mongos
command line option!–help, -h

--help, -h
Returns information on the options and use of mongos (page 601).

command line option!–version

--version
Returns the mongos (page 601) release number.

command line option!–config <filename>, -f <filename>

--config <filename>, **-f** <filename>
Specifies a configuration file for runtime configuration options. The configuration file is the preferred method for runtime configuration of mongos (page 601). The options are equivalent to the command-line configuration options. See http://docs.mongodb.org/manual/reference/configuration-options for more information.

Ensure the configuration file uses ASCII encoding. The mongos (page 601) instance does not support configuration files with non-ASCII encoding, including UTF-8.

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

command line option!–quiet

--quiet
Runs the mongos (page 601) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

- output from *database commands*
- replication activity
- connection accepted events
- connection closed events

command line option!–port <port>

--port <port>
Default: 27017

Specifies the TCP port on which the MongoDB instance listens for client connections.

command line option!–bind_ip <ip address>

--bind_ip <ip address>
Default: All interfaces.

Changed in version 2.6.0: The `deb` and `rpm` packages include a default configuration file that sets *--bind_ip* (page 584) to `127.0.0.1`.

Specifies the IP address that `mongos` (page 601) binds to in order to listen for connections from applications. You may attach `mongos` (page 601) to any interface. When attaching `mongos` (page 601) to a publicly accessible interface, ensure that you have implemented proper authentication and firewall restrictions to protect the integrity of your database.

command line option!–maxConns <number>

--maxConns <number>
Specifies the maximum number of simultaneous connections that `mongos` (page 601) will accept. This setting will have no effect if the value of this setting is higher than your operating system's configured maximum connection tracking threshold.

This setting is particularly useful for `mongos` (page 601) if you have a client that creates a number of connections but allows them to timeout rather than close the connections. When you set `maxIncomingConnections`, ensure the value is slightly higher than the size of the connection pool or the total number of connections to prevent erroneous connection spikes from propagating to the members of a *sharded cluster*.

Changed in version 2.6: MongoDB removed the upward limit on the `maxIncomingConnections` setting.

command line option!–syslog

--syslog
Sends all logging output to the host's *syslog* system rather than to standard output or to a log file. , as with *--logpath* (page 585).

The *--syslog* (page 584) option is not supported on Windows.

command line option!–syslogFacility <string>

--syslogFacility <string>
Default: user

Specifies the facility level used when logging messages to syslog. The value you specify must be supported by your operating system's implementation of syslog. To use this option, you must enable the *--syslog* (page 584) option.

command line option!–logpath <path>

--logpath <path>
Sends all diagnostic logging information to a log file instead of to standard output or to the host's *syslog* system. MongoDB creates the log file at the path you specify.

By default, MongoDB overwrites the log file when the process restarts. To instead append to the log file, set the *--logappend* (page 585) option.

command line option!–logappend

--logappend
Appends new entries to the end of the log file rather than overwriting the content of the log when the mongos (page 601) instance restarts.

command line option!–timeStampFormat <string>

--timeStampFormat <string>
Default: iso8601-local

The time format for timestamps in log messages. Specify one of the following values:

Value	Description
ctime	Displays timestamps as Wed Dec 31 18:17:54.811.
iso8601-utc	Displays timestamps in Coordinated Universal Time (UTC) in the ISO-8601 format. For example, for New York at the start of the Epoch: 1970-01-01T00:00:00.000Z
iso8601-local	Displays timestamps in local time in the ISO-8601 format. For example, for New York at the start of the Epoch: 1969-12-31T19:00:00.000+0500

command line option!–pidfilepath <path>

--pidfilepath <path>
Specifies a file location to hold the process ID of the mongos (page 601) process where mongos (page 601) will write its PID. This is useful for tracking the mongos (page 601) process in combination with the *--fork* (page 587) option. Without a specified *--pidfilepath* (page 586) option, the process creates no PID file.

command line option!–keyFile <file>

--keyFile <file>
Specifies the path to a key file that stores the shared secret that MongoDB instances use to authenticate to each other in a *sharded cluster* or *replica set*. *--keyFile* (page 586) implies *--auth* (page 587). See *inter-process-auth* for more information.

command line option!–setParameter <options>

--setParameter <options>
Specifies one of the MongoDB parameters described in http://docs.mongodb.org/manual/reference/parameter
You can specify multiple setParameter fields.

command line option!–httpinterface

--httpinterface
New in version 2.6.

Enables the HTTP interface. Enabling the interface can increase network exposure.

Leave the HTTP interface *disabled* for production deployments. If you *do* enable this interface, you should only allow trusted clients to access this port. See *security-firewalls*.

Note: In MongoDB Enterprise, the HTTP Console does not support Kerberos Authentication.

---

command line option!–nounixsocket

--nounixsocket
Disables listening on the UNIX domain socket. The mongos (page 601) process always listens on the UNIX socket unless one of the following is true:

- *--nounixsocket* (page 586) is set

- `bindIp` is not set
- `bindIp` does not specify `127.0.0.1`

New in version 2.6: `mongos` (page 601) installed from official `.deb` and `.rpm` packages have the `bind_ip` configuration set to `127.0.0.1` by default.

command line option!–unixSocketPrefix <path>

`--unixSocketPrefix` `<path>`
Default: /tmp

The path for the UNIX socket. If this option has no value, the `mongos` (page 601) process creates a socket with `http://docs.mongodb.org/manual/tmp` as a prefix. MongoDB creates and listens on a UNIX socket unless one of the following is true:

- *`--nounixsocket`* (page 586) is set
- `bindIp` is not set
- `bindIp` does not specify `127.0.0.1`

command line option!–fork

`--fork`
Enables a *daemon* mode that runs the `mongos` (page 601) process in the background. By default `mongos` (page 601) does not run as a daemon: typically you will run `mongos` (page 601) as a daemon, either by using *`--fork`* (page 587) or by using a controlling process that handles the daemonization process (e.g. as with `upstart` and `systemd`).

Sharded Cluster Options command line option!–configdb <config1>,<config2>,<config3>

`--configdb` `<config1>,<config2>,<config3>`
Specifies the *configuration database* for the *sharded cluster*. You must specify either 1 or 3 configuration servers, in a comma separated list. **Always** use 3 config servers in production environments.

All `mongos` (page 601) instances **must** specify the exact same value for *`--configdb`* (page 604)

If your configuration databases reside in more that one data center, order the hosts so that first config sever in the list is the closest to the majority of your `mongos` (page 601) instances.

> **Warning:** Never remove a config server from this setting, even if the config server is not available or offline.

command line option!–localThreshold

`--localThreshold`
Default: 15

Affects the logic that `mongos` (page 601) uses when selecting *replica set* members to pass read operations from clients. Specify a value in milliseconds. The default value of `15` corresponds to the default value in all of the client `drivers`.

When `mongos` (page 601) receives a request that permits reads to *secondary* members, the `mongos` (page 601) will:

- Find the member of the set with the lowest ping time.
- Construct a list of replica set members that is within a ping time of 15 milliseconds of the nearest suitable member of the set.

  If you specify a value for the *`--localThreshold`* (page 604) option, `mongos` (page 601) will construct the list of replica members that are within the latency allowed by this value.

•Select a member to read from at random from this list.

The ping time used for a member compared by the *--localThreshold* (page 604) setting is a moving average of recent ping times, calculated at most every 10 seconds. As a result, some queries may reach members above the threshold until the mongos (page 601) recalculates the average.

See the *replica-set-read-preference-behavior-member-selection* section of the read preference documentation for more information.

command line option!–upgrade

--upgrade
Updates the meta data format used by the *config database*.

command line option!–chunkSize <value>

--chunkSize <value>
Default: 64

Determines the size in megabytes of each *chunk* in the *sharded cluster*. A size of 64 megabytes is ideal in most deployments: larger chunk size can lead to uneven data distribution; smaller chunk size can lead to inefficient movement of chunks between nodes.

This option affects chunk size *only* when you initialize the cluster for the first time. If you later modify the option, the new value has no effect. See the http://docs.mongodb.org/manual/tutorial/modify-chunk-size-in-sharded-cluster procedure if you need to change the chunk size on an existing sharded cluster.

command line option!–noAutoSplit

--noAutoSplit
Prevents mongos (page 601) from automatically inserting metadata splits in a *sharded collection*. If set on all mongos (page 601) instances, this prevents MongoDB from creating new chunks as the data in a collection grows.

Because any mongos (page 601) in a cluster can create a split, to totally disable splitting in a cluster you must set *--noAutoSplit* (page 605) on all mongos (page 601).

Warning: With *--noAutoSplit* (page 605) enabled, the data in your sharded cluster may become imbalanced over time. Enable with caution.

SSL Options

See

http://docs.mongodb.org/manual/tutorial/configure-ssl for full documentation of MongoDB's support.

command line option!–sslOnNormalPorts

--sslOnNormalPorts
Deprecated since version 2.6.

Enables SSL for mongos (page 601).

With *--sslOnNormalPorts* (page 596), a mongos (page 601) requires SSL encryption for all connections on the default MongoDB port, or the port specified by *--port* (page 584). By default, *--sslOnNormalPorts* (page 596) is disabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslMode <mode>

--sslMode <mode>
New in version 2.6.

Enables SSL or mixed SSL used for all network connections. The argument to the *--sslMode* (page 596) option can be one of the following:

Value	Description
disabled	The server does not use SSL.
allowSSL	Connections between servers do not use SSL. For incoming connections, the server accepts both SSL and non-SSL.
preferSSL	Connections between servers use SSL. For incoming connections, the server accepts both SSL and non-SSL.
requireSSL	The server uses and accepts only SSL encrypted connections.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.2.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths.

When SSL is enabled, you must specify *--sslPEMKeyFile* (page 596).

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.2.

Specifies the password to de-crypt the certificate-key file (i.e. --sslPEMKeyFile). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongos (page 601) will redact the password from all logging and reporting output.

Changed in version 2.6: If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the mongos (page 601) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–clusterAuthMode <option>

--clusterAuthMode <option>
Default: keyFile

New in version 2.6.

The authentication mode used for cluster authentication. If you use *internal x.509 authentication*, specify so here. This option can have one of the following values:

Value	Description
keyFile	Use a keyfile for authentication. Accept only keyfiles.
sendKeyFile	For rolling upgrade purposes. Send a keyfile for authentication but can accept both keyfiles and x.509 certificates.
sendX509	For rolling upgrade purposes. Send the x.509 certificate for authentication but can accept both keyfiles and x.509 certificates.
x509	Recommended. Send the x.509 certificate for authentication and accept only x.509 certificates.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslClusterFile <filename>

--sslClusterFile <filename>
New in version 2.6.

Specifies the .pem file that contains the x.509 certificate-key file for *membership authentication* for the cluster or replica set.

If *--sslClusterFile* (page 597) does not specify the .pem file for internal cluster authentication, the cluster uses the .pem file specified in the *--sslPEMKeyFile* (page 596) option.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslClusterPassword <value>

--sslClusterPassword <value>
New in version 2.6.

Specifies the password to de-crypt the x.509 certificate-key file specified with --sslClusterFile. Use the *--sslClusterPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongos (page 601) will redact the password from all logging and reporting output.

If the x.509 key file is encrypted and you do not specify the *--sslClusterPassword* (page 597) option, the mongos (page 601) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.4.

Specifies the .pem file that contains the root certificate chain from the Certificate Authority. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

Warning: If the *--sslCAFile* option and its target file are not specified, x.509 client and member authentication will not function. mongod (page 583), and mongos (page 601) in sharded systems, will not be able to verify the certificates of processes connecting to it against the trusted certificate authority (CA) that issued them, breaking the certificate chain.
As of version 2.6.4, mongod (page 583) will not start with x.509 authentication enabled if the CA file is not specified.

command line option!–sslCRLFile <filename>

--sslCRLFile <filename>
New in version 2.4.

Specifies the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslWeakCertificateValidation

--sslWeakCertificateValidation
New in version 2.4.

Changed in version 2.8.0: `--sslAllowConnectionsWithoutCertificates` became *`--sslWeakCertificateValidation`* (page 608). For compatibility, MongoDB processes continue to accept `--sslAllowConnectionsWithoutCertificates`, but all users should update their configuration files.

Disables the requirement for SSL certificate validation that `--sslCAFile` enables. With the *`--sslWeakCertificateValidation`* (page 608) option, the `mongos` (page 601) will accept connections when the client does not present a certificate when establishing the connection.

If the client presents a certificate and the `mongos` (page 601) has *`--sslWeakCertificateValidation`* (page 608) enabled, the `mongos` (page 601) will validate the certificate using the root certificate chain specified by `--sslCAFile` and reject clients with invalid certificates.

Use the *`--sslWeakCertificateValidation`* (page 608) option if you have a mixed deployment that includes clients that do not or cannot present certificates to the `mongos` (page 601).

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for SSL certificates on other servers in the cluster and allows the use of invalid certificates. When using the `allowInvalidCertificates` setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates, when connecting to other `mongos` (page 601) instances for inter-process authentication. This allows `mongos` (page 601) to connect to other `mongos` (page 601) instances if the hostnames in their certificates do not match their configured hostname.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.4.

Directs the `mongos` (page 601) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[7]. See `http://docs.mongodb.org/manual/tutorial/configure-fips` for more information.

Audit Options command line option!–auditDestination

`--auditDestination`

New in version 2.6.

Enables `auditing`. The *`--auditDestination`* (page 599) option can have one of the following values:

Value	Description
`syslog`	Output the audit events to syslog in JSON format. Not available on Windows. Audit messages have a syslog severity level of `info` and a facility level of `user`. The syslog message limit can result in the truncation of audit messages. The auditing system will neither detect the truncation nor error upon its occurrence.
`console`	Output the audit events to `stdout` in JSON format.
`file`	Output the audit events to the file specified in *`--auditPath`* (page 600) in the format specified in *`--auditFormat`* (page 600).

Note: Available only in MongoDB Enterprise[8].

command line option!–auditFormat

`--auditFormat`

New in version 2.6.

Specifies the format of the output file for `auditing` if *`--auditDestination`* (page 599) is `file`. The *`--auditFormat`* (page 600) option can have one of the following values:

Value	Description
`JSON`	Output the audit events in JSON format to the file specified in *`--auditPath`* (page 600).
`BSON`	Output the audit events in BSON binary format to the file specified in *`--auditPath`* (page 600).

Printing audit events to a file in JSON format degrades server performance more than printing to a file in BSON format.

Note: Available only in MongoDB Enterprise[9].

command line option!–auditPath

`--auditPath`

New in version 2.6.

Specifies the output file for `auditing` if *`--auditDestination`* (page 599) has value of `file`. The *`--auditPath`* (page 600) option can take either a full path name or a relative path name.

Note: Available only in MongoDB Enterprise[10].

command line option!–auditFilter

`--auditFilter`

New in version 2.6.

[7] http://www.mongodb.com/products/mongodb-enterprise
[8] http://www.mongodb.com/products/mongodb-enterprise
[9] http://www.mongodb.com/products/mongodb-enterprise
[10] http://www.mongodb.com/products/mongodb-enterprise

Specifies the filter to limit the *types of operations* the `audit system` records. The option takes a string representation of a query document of the form:

```
{ <field1>: <expression1>, ... }
```

The `<field>` can be `any field in the audit message`, including fields returned in the *param* document. The `<expression>` is a *query condition expression* (page 400).

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

To specify the audit filter in a `configuration file`, you must use the YAML format of the configuration file.

{ atype: <expression>, "param.db": <database> }

Note: Available only in MongoDB Enterprise[11].

Additional Options command line option!–ipv6

--ipv6
Enables IPv6 support and allows the `mongos` (page 601) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–jsonp

--jsonp
Permits *JSONP* access via an HTTP interface. Enabling the interface can increase network exposure. The *--jsonp* (page 587) option enables the HTTP interface, even if the `HTTP interface` option is disabled.

command line option!–noscripting

--noscripting
Disables the scripting engine.

mongo

Description

mongo

`mongo` (page 610) is an interactive JavaScript shell interface to MongoDB, which provides a powerful interface for systems administrators as well as a way for developers to test queries and operations directly with the database. `mongo` (page 610) also provides a fully functional JavaScript environment for use with a MongoDB. This document addresses the basic invocation of the `mongo` (page 610) shell and an overview of its usage.

Options

Core Options
mongo
command line option!–shell

--shell
Enables the shell interface. If you invoke the `mongo` (page 610) command and specify a JavaScript file as an argument, or use *--eval* (page 611) to specify JavaScript on the command line, the *--shell* (page 610) option provides the user with a shell prompt after the file finishes executing.

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

command line option!–nodb

--nodb
Prevents the shell from connecting to any database instances. Later, to connect to a database within the shell, see *mongo-shell-new-connections*.

command line option!–norc

--norc
Prevents the shell from sourcing and evaluating `~/.mongorc.js` on start up.

command line option!–quiet

--quiet
Silences output from the shell during the connection process.

command line option!–port <port>

--port `<port>`
Specifies the port where the `mongod` (page 583) or `mongos` (page 601) instance is listening. If *--port* (page 584) is not specified, `mongo` (page 610) attempts to connect to port `27017`.

command line option!–host <hostname>

--host `<hostname>`
Specifies the name of the host machine where the `mongod` (page 583) or `mongos` (page 601) is running. If this is not specified, `mongo` (page 610) attempts to connect to a MongoDB process running on the localhost.

To connect to a replica set, specify the `replica set name` and a seed list of set members. Use the following form:

```
<replSetName>/<hostname1><:port>,<hostname2><:port>,<...>
```

command line option!–eval <javascript>

--eval `<javascript>`
Evaluates a JavaScript expression that is specified as an argument. `mongo` (page 610) does not load its own environment when evaluating code. As a result many options of the shell environment are not available.

command line option!–username <username>, -u <username>

--username `<username>`, **-u** `<username>`
Specifies a username with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the `--password` and `--authenticationDatabase` options.

command line option!–password <password>, -p <password>

--password `<password>`, **-p** `<password>`
Specifies a password with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the *--username* (page 667) and *--authenticationDatabase* (page 667) options. To force `mongo` (page 610) to prompt for a password, enter the *--password* (page 667) option as the last option and leave out the argument.

command line option!–help, -h

--help, -h
Returns information on the options and use of `mongo` (page 610).

command line option!–version

--version
Returns the `mongo` (page 610) release number.

command line option!–verbose

--verbose
Increases the verbosity of the output of the shell during the connection process.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the mongo (page 610) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

<db address>
Specifies the "database address" of the database to connect to. For example:

```
mongo admin
```

The above command will connect the mongo (page 610) shell to the *admin database* on the local machine. You may specify a remote database instance, with the resolvable hostname or IP address. Separate the database name from the hostname using a `http://docs.mongodb.org/manual/` character. See the following examples:

```
mongo mongodb1.example.net
mongo mongodb1/admin
mongo 10.8.8.10/test
```

This syntax is the *only* way to connect to a specific database.

To specify alternate hosts and a database, you must use this syntax and cannot use *--host* (page 665) or *--port* (page 584).

<file.js>
Specifies a JavaScript file to run and then exit. Generally this should be the last option specified.

Optional

To specify a JavaScript file to execute *and* allow mongo (page 610) to prompt you for a password using *--password* (page 667), pass the filename as the first parameter with *--username* (page 667) and *--password* (page 667) as the last options, as in the following:

```
mongo file.js --username username --password
```

Use the *--shell* (page 610) option to return to a shell after the file finishes running.

Authentication Options command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's credentials.

If you do not specify an authentication database, mongo (page 610) assumes that the database specified as the argument to *--authenticationDatabase* (page 667) holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
Default: MONGODB-CR

New in version 2.4.

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Specifies the authentication mechanism the mongo (page 610) instance uses to authenticate to the mongod (page 583) or mongos (page 601).

Value	Description
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB SSL certificate authentication.
PLAIN	External authentication using LDAP. You can also use `PLAIN` for authenticating in-database users. `PLAIN` transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[12].
GSSAPI	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[13].

command line option!–gssapiHostName

--gssapiHostName
New in version 2.6.

Specify the hostname of a service using `GSSAPI/Kerberos`. *Only* required if the hostname of a machine does not match the hostname resolved by DNS.

This option is available only in MongoDB Enterprise.

command line option!–gssapiServiceName

--gssapiServiceName
New in version 2.6.

Specify the name of the service using `GSSAPI/Kerberos`. Only required if the service does not use the default name of `mongodb`.

This option is available only in MongoDB Enterprise.

SSL Options command line option!–ssl

--ssl
New in version 2.2.

Enables connection to a `mongod` (page 583) or `mongos` (page 601) that has SSL support enabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile `<filename>`
New in version 2.4.

Specifies the `.pem` file that contains both the SSL certificate and key. Specify the file name of the `.pem` file using relative or absolute paths.

This option is required when using the `--ssl` option to connect to a `mongod` (page 583) or `mongos` (page 601) that has `CAFile` enabled *without* `allowConnectionsWithoutCertificates`.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword `<value>`
New in version 2.4.

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

Specifies the password to de-crypt the certificate-key file (i.e. `--sslPEMKeyFile`). Use the *`--sslPEMKeyPassword`* (page 597) option only if the certificate-key file is encrypted. In all cases, the `mongo` (page 610) will redact the password from all logging and reporting output.

Changed in version 2.6: If the private key in the PEM file is encrypted and you do not specify the *`--sslPEMKeyPassword`* (page 597) option, the `mongo` (page 610) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslCAFile <filename>

`--sslCAFile` `<filename>`
New in version 2.4.

Specifies the `.pem` file that contains the root certificate chain from the Certificate Authority. Specify the file name of the `.pem` file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

> **Warning:** If the `mongo` (page 610) shell or any other tool that connects to `mongos` (page 601) or `mongod` (page 583) is run without *`--sslCAFile`*, it will not attempt to validate server certificates. This results in vulnerability to expired `mongod` (page 583) and `mongos` (page 601) certificates as well as to foreign processes posing as valid `mongod` (page 583) or `mongos` (page 601) instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

command line option!–sslCRLFile <filename>

`--sslCRLFile` `<filename>`
New in version 2.4.

Specifies the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslFIPSMode

`--sslFIPSMode`
New in version 2.6.

Directs the `mongo` (page 610) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[14]. See `http://docs.mongodb.org/manual/tutorial/configure-fips` for more information.

---

command line option!–sslAllowInvalidCertificates

`--sslAllowInvalidCertificates`
New in version 2.6.

Bypasses the validation checks for server certificates and allows the use of invalid certificates. When using the `allowInvalidCertificates` setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

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

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates. Allows mongo (page 610) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

Files

~/.dbshell mongo (page 610) maintains a history of commands in the .dbshell file.

Note: mongo (page 610) does not recorded interaction related to authentication in the history file, including authenticate (page 263) and db.createUser() (page 152).

Warning: Versions of Windows mongo.exe earlier than 2.2.0 will save the *.dbshell* file in the mongo.exe working directory.

~/.mongorc.js mongo (page 610) will read the .mongorc.js file from the home directory of the user invoking mongo (page 610). In the file, users can define variables, customize the mongo (page 610) shell prompt, or update information that they would like updated every time they launch a shell. If you use the shell to evaluate a JavaScript file or expression either on the command line with *--eval* (page 611) or by specifying *a .js file to mongo* (page 612), mongo (page 610) will read the .mongorc.js file *after* the JavaScript has finished processing.

Specify the *--norc* (page 611) option to disable reading .mongorc.js.

/etc/mongorc.js Global mongorc.js file which the mongo (page 610) shell evaluates upon start-up. If a user also has a .mongorc.js file located in the HOME (page 615) directory, the mongo (page 610) shell evaluates the global /etc/mongorc.js file *before* evaluating the user's .mongorc.js file.

/etc/mongorc.js must have read permission for the user running the shell. The *--norc* (page 611) option for mongo (page 610) suppresses only the user's .mongorc.js file.

On Windows, the global mongorc.js </etc/mongorc.js> exists in the %ProgramData%\MongoDB directory.

http://docs.mongodb.org/manual/tmp/mongo_edit<*time_t*>.js Created by mongo (page 610) when editing a file. If the file exists, mongo (page 610) will append an integer from 1 to 10 to the time value to attempt to create a unique file.

%TEMP%mongo_edit<*time_t*>.js Created by mongo.exe on Windows when editing a file. If the file exists, mongo (page 610) will append an integer from 1 to 10 to the time value to attempt to create a unique file.

Environment

EDITOR
Specifies the path to an editor to use with the edit shell command. A JavaScript variable EDITOR will override the value of EDITOR (page 615).

HOME
Specifies the path to the home directory where mongo (page 610) will read the .mongorc.js file and write the .dbshell file.

HOMEDRIVE

On Windows systems, HOMEDRIVE (page 615) specifies the path the directory where mongo (page 610) will read the `.mongorc.js` file and write the `.dbshell` file.

HOMEPATH

Specifies the Windows path to the home directory where mongo (page 610) will read the `.mongorc.js` file and write the `.dbshell` file.

Keyboard Shortcuts

The mongo (page 610) shell supports the following keyboard shortcuts: [15]

Keybinding	Function
Up arrow	Retrieve previous command from history
Down-arrow	Retrieve next command from history
Home	Go to beginning of the line
End	Go to end of the line
Tab	Autocomplete method/command
Left-arrow	Go backward one character
Right-arrow	Go forward one character
Ctrl-left-arrow	Go backward one word
Ctrl-right-arrow	Go forward one word
Meta-left-arrow	Go backward one word
Meta-right-arrow	Go forward one word
Ctrl-A	Go to the beginning of the line
Ctrl-B	Go backward one character
Ctrl-C	Exit the mongo (page 610) shell
Ctrl-D	Delete a char (or exit the mongo (page 610) shell)
Ctrl-E	Go to the end of the line
Ctrl-F	Go forward one character
Ctrl-G	Abort
Ctrl-J	Accept/evaluate the line
Ctrl-K	Kill/erase the line
Ctrl-L or type `cls`	Clear the screen
Ctrl-M	Accept/evaluate the line
Ctrl-N	Retrieve next command from history
Ctrl-P	Retrieve previous command from history
Ctrl-R	Reverse-search command history
Ctrl-S	Forward-search command history
Ctrl-T	Transpose characters
Ctrl-U	Perform Unix line-discard
Ctrl-W	Perform Unix word-rubout
Ctrl-Y	Yank
Ctrl-Z	Suspend (job control works in linux)
Ctrl-H	Backward-delete a character
Ctrl-I	Complete, same as Tab
Meta-B	Go backward one word
Meta-C	Capitalize word
Meta-D	Kill word
Meta-F	Go forward one word
Meta-L	Change word to lowercase

Continued on next page

[15] MongoDB accommodates multiple keybinding. Since 2.0, mongo (page 610) includes support for basic emacs keybindings.

Table 4.1 – continued from previous page

Keybinding	Function
Meta-U	Change word to uppercase
Meta-Y	Yank-pop
Meta-Backspace	Backward-kill word
Meta-<	Retrieve the first command in command history
Meta->	Retrieve the last command in command history

Use

Typically users invoke the shell with the `mongo` (page 610) command at the system prompt. Consider the following examples for other scenarios.

To connect to a database on a remote host using authentication and a non-standard port, use the following form:

```
mongo --username <user> --password <pass> --host <host> --port 28015
```

Alternatively, consider the following short form:

```
mongo -u <user> -p <pass> --host <host> --port 28015
```

Replace `<user>`, `<pass>`, and `<host>` with the appropriate values for your situation and substitute or omit the *`--port`* (page 584) as needed.

To execute a JavaScript file without evaluating the `~/.mongorc.js` file before starting a shell session, use the following form:

```
mongo --shell --norc alternate-environment.js
```

To execute a JavaScript file with authentication, with password prompted rather than provided on the command-line, use the following form:

```
mongo script-file.js -u <user> -p
```

To print return a query as *JSON*, from the system prompt using the *`--eval`* option, use the following form:

```
mongo --eval 'db.collection.find().forEach(printjson)'
```

Use single quotes (e.g. `'`) to enclose the JavaScript, as well as the additional JavaScript required to generate this output.

4.1.2 Windows Services

The `mongod.exe` (page 618) and `mongos.exe` (page 619) describe the options available for configuring MongoDB when running as a Windows Service. The `mongod.exe` (page 618) and `mongos.exe` (page 619) binaries provide a superset of the `mongod` (page 583) and `mongos` (page 601) options.

mongod.exe

Synopsis

`mongod.exe` (page 618) is the build of the MongoDB daemon (i.e. `mongod` (page 583)) for the Windows platform. `mongod.exe` (page 618) has all of the features of `mongod` (page 583) on Unix-like platforms and is completely compatible with the other builds of `mongod` (page 583). In addition, `mongod.exe` (page 618) provides several options for interacting with the Windows platform itself.

This document *only* references options that are unique to mongod.exe (page 618). All mongod (page 583) options are available. See the *mongod* (page 583) and the http://docs.mongodb.org/manual/reference/configuration-options documents for more information regarding mongod.exe (page 618).

To install and use mongod.exe (page 618), read the http://docs.mongodb.org/manual/tutorial/install-mongodb document.

Options

mongod.**exe**

mongod.**exe**

command line option!–install

--install
Installs mongod.exe (page 618) as a Windows Service and exits.

If needed, you can install services for multiple instances of mongod.exe (page 618). Install each service with a unique *--serviceName* (page 620) and *--serviceDisplayName* (page 620). Use multiple instances only when sufficient system resources exist and your system design requires it.

command line option!–remove

--remove
Removes the mongod.exe (page 618) Windows Service. If mongod.exe (page 618) is running, this operation will stop and then remove the service.

--remove (page 619) requires the *--serviceName* (page 620) if you configured a non-default *--serviceName* (page 620) during the *--install* (page 619) operation.

command line option!–reinstall

--reinstall
Removes mongod.exe (page 618) and reinstalls mongod.exe (page 618) as a Windows Service.

command line option!–serviceName name

--serviceName name
Default: MongoDB

Set the service name of mongod.exe (page 618) when running as a Windows Service. Use this name with the net start <name> and net stop <name> operations.

You must use *--serviceName* (page 620) in conjunction with either the *--install* (page 619) or *--remove* (page 619) install option.

command line option!–serviceDisplayName <name>

--serviceDisplayName <name>
Default: MongoDB

Sets the name listed for MongoDB on the Services administrative application.

command line option!–serviceDescription <description>

--serviceDescription <description>
Default: MongoDB Server

Sets the mongod.exe (page 618) service description.

You must use *--serviceDescription* (page 620) in conjunction with the *--install* (page 619) option.

For descriptions that contain spaces, you must enclose the description in quotes.

command line option!–serviceUser <user>

--serviceUser <user>
Runs the mongod.exe (page 618) service in the context of a certain user. This user must have "Log on as a service" privileges.

You must use *--serviceUser* (page 620) in conjunction with the *--install* (page 619) option.

command line option!–servicePassword <password>

--servicePassword <password>
Sets the password for <user> for mongod.exe (page 618) when running with the *--serviceUser* (page 620) option.

You must use *--servicePassword* (page 620) in conjunction with the *--install* (page 619) option.

mongos.exe

Synopsis

mongos.exe (page 619) is the build of the MongoDB Shard (i.e. mongos (page 601)) for the Windows platform. mongos.exe (page 619) has all of the features of mongos (page 601) on Unix-like platforms and is completely compatible with the other builds of mongos (page 601). In addition, mongos.exe (page 619) provides several options for interacting with the Windows platform itself.

This document only references options that are unique to mongos.exe (page 619). All mongos (page 601) options are available. See the *mongos* (page 601) and the http://docs.mongodb.org/manual/reference/configuration-options documents for more information regarding mongos.exe (page 619).

To install and use mongos.exe (page 619), read the http://docs.mongodb.org/manual/tutorial/install-mongodb document.

Options

mongos.**exe**

mongos.**exe**

command line option!–install

--install
Installs mongos.exe (page 619) as a Windows Service and exits.

If needed, you can install services for multiple instances of mongos.exe (page 619). Install each service with a unique *--serviceName* (page 620) and *--serviceDisplayName* (page 620). Use multiple instances only when sufficient system resources exist and your system design requires it.

command line option!–remove

--remove
Removes the mongos.exe (page 619) Windows Service. If mongos.exe (page 619) is running, this operation will stop and then remove the service.

--remove (page 619) requires the *--serviceName* (page 620) if you configured a non-default *--serviceName* (page 620) during the *--install* (page 619) operation.

command line option!–reinstall

--reinstall
Removes mongos.exe (page 619) and reinstalls mongos.exe (page 619) as a Windows Service.

command line option!–serviceName name

--serviceName name
Default: MongoS

Set the service name of mongos.exe (page 619) when running as a Windows Service. Use this name with the net start <name> and net stop <name> operations.

You must use *--serviceName* (page 620) in conjunction with either the *--install* (page 619) or *--remove* (page 619) install option.

command line option!–serviceDisplayName <name>

--serviceDisplayName <name>
Default: Mongo DB Router

Sets the name listed for MongoDB on the Services administrative application.

command line option!–serviceDescription <description>

--serviceDescription <description>
Default: Mongo DB Sharding Router

Sets the mongos.exe (page 619) service description.

You must use *--serviceDescription* (page 620) in conjunction with the *--install* (page 619) option.

For descriptions that contain spaces, you must enclose the description in quotes.

command line option!–serviceUser <user>

--serviceUser <user>
Runs the mongos.exe (page 619) service in the context of a certain user. This user must have "Log on as a service" privileges.

You must use *--serviceUser* (page 620) in conjunction with the *--install* (page 619) option.

command line option!–servicePassword <password>

--servicePassword <password>
Sets the password for <user> for mongos.exe (page 619) when running with the *--serviceUser* (page 620) option.

You must use *--servicePassword* (page 620) in conjunction with the *--install* (page 619) option.

4.1.3 Binary Import and Export Tools

mongodump (page 622) provides a method for creating *BSON* dump files from the mongod (page 583) instances, while mongorestore (page 628) makes it possible to restore these dumps. bsondump (page 635) converts BSON dump files into *JSON*. The mongooplog (page 637) utility provides the ability to stream *oplog* entries outside of normal replication.

mongodump

Synopsis

mongodump (page 622) is a utility for creating a binary export of the contents of a database. Consider using this utility as part an effective backup strategy. Use mongodump (page 622) in conjunction with mongorestore

(page 628) to restore databases.

mongodump (page 622) can read data from either mongod (page 583) or mongos (page 601) instances, in addition to reading directly from MongoDB data files without an active mongod (page 583).

See also:

mongorestore (page 628), http://docs.mongodb.org/manual/tutorial/backup-sharded-cluster-with-da and http://docs.mongodb.org/manual/core/backups.

Behavior

mongodump (page 622) does *not* dump the content of the local database.

The data format used by mongodump (page 622) from version 2.2 or later is *incompatible* with earlier versions of mongod (page 583). Do not use recent versions of mongodump (page 622) to back up older data stores.

When running mongodump (page 622) against a mongos (page 601) instance where the *sharded cluster* consists of *replica sets*, the *read preference* of the operation will prefer reads from *secondary* members of the set.

Changed in version 2.2: When used in combination with fsync (page 336) or db.fsyncLock() (page 113), mongod (page 583) may block some reads, including those from mongodump (page 622), when queued write operation waits behind the fsync (page 336) lock.

mongodump (page 622) overwrites output files if they exist in the backup data folder. Before running the mongodump (page 622) command multiple times, either ensure that you no longer need the files in the output folder (the default is the dump/ folder) or rename the folders or files.

Required Access

Backup Collections To backup all the databases in a cluster via mongodump (page 622), you should have the backup role. The backup role provides all the needed privileges for backing up all database. The role confers no additional access, in keeping with the policy of *least privilege*.

To backup a given database, you must have read access on the database. Several roles provide this access, including the backup role.

To backup the system.profile collection in a database, you must have read access on certain system collections in the database. Several roles provide this access, including the clusterAdmin and dbAdmin roles.

Backup Users Changed in version 2.6.

To backup users and *user-defined roles* for a given database, you must have access to the admin database. MongoDB stores the user data and role definitions for all databases in the admin database.

Specifically, to backup a given database's users, you must have the find *action* on the admin database's admin.system.users (page 689) collection. The backup and userAdminAnyDatabase roles both provide this privilege.

To backup the user-defined roles on a database, you must have the find action on the admin database's admin.system.roles (page 689) collection. Both the backup and userAdminAnyDatabase roles provide this privilege.

Options

Changed in version 2.8.0: mongodump (page 622) removed the --dbpath as well as related --directoryperdb and --journal options. You must use mongodump (page 622) while connected to a mongod (page 583) instance.

mongodump

mongodump

command line option!–help

--help
Returns information on the options and use of mongodump (page 622).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

command line option!–quiet

--quiet
Runs the mongodump (page 622) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

- output from *database commands*
- replication activity
- connection accepted events
- connection closed events

command line option!–version

--version
Returns the mongodump (page 622) release number.

command line option!–host <hostname><:port>, -h <hostname><:port>

--host <hostname><:port>, **-h** <hostname><:port>
Default: localhost:27017

Specifies a resolvable hostname for the mongod (page 583) to which to connect. By default, the mongodump (page 622) attempts to connect to a MongoDB instance running on the localhost on port number 27017.

To connect to a replica set, specify the replica set name and a seed list of set members. Use the following form:

```
<replSetName>/<hostname1><:port>,<hostname2><:port>,<...>
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

Changed in version 2.8.0: If you use IPv6 and use the <address>:<port > format, you must enclose the portion of an address and port combination in brackets (e.g. [<address>]).

command line option!–port <port>

--port <port>
Default: 27017

Specifies the TCP port on which the MongoDB instance listens for client connections.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the mongodump (page 622) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.6.

Enables connection to a mongod (page 583) or mongos (page 601) that has SSL support enabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.6.

Specifies the .pem file that contains the root certificate chain from the Certificate Authority. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

> **Warning:** If the mongo (page 610) shell or any other tool that connects to mongos (page 601) or mongod (page 583) is run without *--sslCAFile*, it will not attempt to validate server certificates. This results in vulnerability to expired mongod (page 583) and mongos (page 601) certificates as well as to foreign processes posing as valid mongod (page 583) or mongos (page 601) instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.6.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths.

This option is required when using the *--ssl* (page 665) option to connect to a mongod (page 583) or mongos (page 601) that has CAFile enabled *without* allowConnectionsWithoutCertificates.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.6.

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 596)). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongodump (page 622) will redact the password from all logging and reporting output.

If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the mongodump (page 622) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCRLFile <filename>

--sslCRLFile <filename>
New in version 2.6.

Specifies the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for server certificates and allows the use of invalid certificates. When using the `allowInvalidCertificates` setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates. Allows `mongodump` (page 622) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.6.

Directs the `mongodump` (page 622) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[16]. See `http://docs.mongodb.org/manual/tutorial/configure-fips` for more information.

---

command line option!–username <username>, -u <username>

--username <username>, **-u** <username>
Specifies a username with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the `--password` and `--authenticationDatabase` options.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the `--username` and `--authenticationDatabase` options.

If you do not specify an argument for *--password* (page 667), `mongodump` (page 622) will prompt interactively for a password on the console.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's credentials.

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

If you do not specify an authentication database, mongodump (page 622) assumes that the database specified to export holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
Default: MONGODB-CR

New in version 2.4.

Changed in version 2.6: Added support for the PLAIN and MONGODB-X509 authentication mechanisms.

Specifies the authentication mechanism the mongodump (page 622) instance uses to authenticate to the mongod (page 583) or mongos (page 601).

Value	Description
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB SSL certificate authentication.
PLAIN	External authentication using LDAP. You can also use PLAIN for authenticating in-database users. PLAIN transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[17].
GSSAPI	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[18].

command line option!–gssapiServiceName

--gssapiServiceName
New in version 2.6.

Specify the name of the service using GSSAPI/Kerberos. Only required if the service does not use the default name of mongodb.

This option is available only in MongoDB Enterprise.

command line option!–gssapiHostName

--gssapiHostName
New in version 2.6.

Specify the hostname of a service using GSSAPI/Kerberos. *Only* required if the hostname of a machine does not match the hostname resolved by DNS.

This option is available only in MongoDB Enterprise.

command line option!–db <database>, -d <database>

--db <database>, **-d** <database>
Specifies a database to backup. If you do not specify a database, mongodump (page 622) copies all databases in this instance into the dump files.

command line option!–collection <collection>, -c <collection>

--collection <collection>, **-c** <collection>
Specifies a collection to backup. If you do not specify a collection, this option copies all collections in the specified database or instance to the dump files.

command line option!–query <json>, -q <json>

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

--query <json>, **-q** <json>
Provides a *JSON document* as a query that optionally limits the documents included in the output of mongodump (page 622).

command line option!–forceTableScan

--forceTableScan
Forces mongodump (page 622) to scan the data store directly: typically, mongodump (page 622) saves entries as they appear in the index of the _id field. If you specify a query *--query* (page 654), mongodump (page 622) will use the most appropriate index to support that query.

Use *--forceTableScan* (page 655) to skip the index and scan the data directly. Typically there are two cases where this behavior is preferable to the default:

1. If you have key sizes over 800 bytes that would not be present in the _id index.
2. Your database uses a custom _id field.

When you run with *--forceTableScan* (page 655), mongodump (page 622) does not use $snapshot (page 562). As a result, the dump produced by mongodump (page 622) can reflect the state of the database at many different points in time.

Important: Use *--forceTableScan* (page 655) with extreme caution and consideration.

---

command line option!–out <path>, -o <path>

--out <path>, **-o** <path>
Specifies the directory where mongodump (page 622) saves the output of the database dump. By default, mongodump (page 622) saves output files in a directory named dump in the current working directory.

To send the database dump to standard output, specify "-" instead of a path. Write to standard output if you want process the output before saving it, such as to use gzip to compress the dump. When writing standard output, mongodump (page 622) does not write the metadata that writes in a <dbname>.metadata.json file when writing to files directly.

command line option!–repair

--repair
Runs a repair option in addition to dumping the database. The repair option attempts to repair a database that may be in an invalid state as a result of an improper shutdown or mongod (page 583) crash.

The *--repair* (page 593) option uses aggressive data-recovery algorithms that may produce a large amount of duplication.

--repair (page 593) is only available for use with mongod (page 583) instances using the mmapv1 storage engine. You cannot run *--repair* (page 593) with mongos (page 601) or with mongod (page 583) instances that use the wiredTiger storage engine. To repair data in a mongod (page 583) instance using wiredTiger use *mongod --repair*.

command line option!–oplog

--oplog
Ensures that mongodump (page 622) creates a dump of the database that includes a partial *oplog* containing operations from the duration of the mongodump (page 622) operation. This oplog produces an effective point-in-time snapshot of the state of a mongod (page 583) instance. To restore to a specific point-in-time backup, use the output created with this option in conjunction with *mongorestore --oplogReplay*.

Without *--oplog* (page 626), if there are write operations during the dump operation, the dump will not reflect a single moment in time. Changes made to the database during the update process can affect the output of the backup.

--oplog (page 626) has no effect when running mongodump (page 622) against a mongos (page 601) instance to dump the entire contents of a sharded cluster. However, you can use *--oplog* (page 626) to dump individual shards.

--oplog (page 626) only works against nodes that maintain an *oplog*. This includes all members of a replica set, as well as *master* nodes in master/slave replication deployments.

--oplog (page 626) does not dump the oplog collection.

command line option!–dumpDbUsersAndRoles

--dumpDbUsersAndRoles
Includes user and role definitions when performing mongodump (page 622) on a specific database. This option applies only when you specify a database in the *--db* (page 632) option. MongoDB always includes user and role definitions when mongodump (page 622) applies to an entire instance and not just a specific database.

command line option!–excludeCollection array of strings

--excludeCollection array of strings
New in version 2.8.0.

Specifies collections to exclude from the output of mongodump (page 622) output.

command line option!–excludeCollectionsWithPrefix array of strings

--excludeCollectionsWithPrefix array of strings
New in version 2.8.0.

Excludes all collections from the output of mongodump (page 622) with a specified prefix.

Use

See the http://docs.mongodb.org/manual/tutorial/backup-with-mongodump for a larger overview of mongodump (page 622) usage. Also see the *mongorestore* (page 627) document for an overview of the mongorestore (page 628), which provides the related inverse functionality.

The following command creates a dump file that contains only the collection named collection in the database named test. In this case the database is running on the local interface on port 27017:

```
mongodump --collection collection --db test
```

In the next example, mongodump (page 622) creates a database dump located at /opt/backup/mongodump-2011-10-24, from a database running on port 37017 on the host mongodb1.example.net and authenticating using the username user and the password pass, as follows:

```
mongodump --host mongodb1.example.net --port 37017 --username user --password pass --out /opt/backup/
```

mongorestore

Synopsis

The mongorestore (page 628) program writes data from a binary database dump created by mongodump (page 622) to a MongoDB instance. mongorestore (page 628) can create a new database or add data to an existing database.

mongorestore (page 628) can write data to either *mongod* or mongos (page 601) instances, in addition to writing directly to MongoDB data files without an active mongod (page 583).

Behavior

If you restore to an existing database, mongorestore (page 628) will only insert into the existing database, and does not perform updates of any kind. If existing documents have the same value _id field in the target database and collection, mongorestore (page 628) will *not* overwrite those documents.

Remember the following properties of mongorestore (page 628) behavior:

- mongorestore (page 628) recreates indexes recorded by mongodump (page 622).
- all operations are inserts, not updates.
- mongorestore (page 628) does not wait for a response from a mongod (page 583) to ensure that the MongoDB process has received or recorded the operation.

  The mongod (page 583) will record any errors to its log that occur during a restore operation, but mongorestore (page 628) will not receive errors.

The data format used by mongodump (page 622) from version 2.2 or later is *incompatible* with earlier versions of mongod (page 583). Do not use recent versions of mongodump (page 622) to back up older data stores.

New in version 2.8.0: mongorestore (page 628) also accepts input via standard input.

Required Access to Restore User Data

Changed in version 2.6.

To restore users and *user-defined roles* on a given database, you must have access to the admin database. MongoDB stores the user data and role definitions for all databases in the admin database.

Specifically, to restore users to a given database, you must have the insert *action* on the admin database's admin.system.users (page 689) collection. The restore role provides this privilege.

To restore user-defined roles to a database, you must have the insert action on the admin database's admin.system.roles (page 689) collection. The restore role provides this privilege.

Options

mongorestore

mongorestore

command line option!–help

--help
Returns information on the options and use of mongorestore (page 628).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

command line option!–quiet

--quiet
Runs the mongorestore (page 628) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

•output from *database commands*

- replication activity
- connection accepted events
- connection closed events

command line option!–version

--version
Returns the mongorestore (page 628) release number.

command line option!–host <hostname><:port>, -h <hostname><:port>

--host <hostname><:port>, **-h** <hostname><:port>
Default: localhost:27017

Specifies a resolvable hostname for the mongod (page 583) to which to connect. By default, the mongorestore (page 628) attempts to connect to a MongoDB instance running on the localhost on port number 27017.

To connect to a replica set, specify the replica set name and a seed list of set members. Use the following form:

```
<replSetName>/<hostname1><:port>,<hostname2><:port>,<...>
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

Changed in version 2.8.0: If you use IPv6 and use the <address>:<port > format, you must enclose the portion of an address and port combination in brackets (e.g. [<address>]).

command line option!–port <port>

--port <port>
Default: 27017

Specifies the TCP port on which the MongoDB instance listens for client connections.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the mongorestore (page 628) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.6.

Enables connection to a mongod (page 583) or mongos (page 601) that has SSL support enabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.6.

Specifies the .pem file that contains the root certificate chain from the Certificate Authority. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

> **Warning:** If the mongo (page 610) shell or any other tool that connects to mongos (page 601) or mongod (page 583) is run without *--sslCAFile*, it will not attempt to validate server certificates. This results in vulnerability to expired mongod (page 583) and mongos (page 601) certificates as well as to foreign processes posing as valid mongod (page 583) or mongos (page 601) instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.6.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths.

This option is required when using the *--ssl* (page 665) option to connect to a mongod (page 583) or mongos (page 601) that has CAFile enabled *without* allowConnectionsWithoutCertificates.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.6.

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 596)). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongorestore (page 628) will redact the password from all logging and reporting output.

If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the mongorestore (page 628) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCRLFile <filename>

--sslCRLFile <filename>
New in version 2.6.

Specifies the .pem file that contains the Certificate Revocation List. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for server certificates and allows the use of invalid certificates. When using the allowInvalidCertificates setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates. Allows mongorestore (page 628) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.6.

Directs the mongorestore (page 628) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[19]. See http://docs.mongodb.org/manual/tutorial/configure-fips for more information.

---

command line option!–username <username>, -u <username>

--username <username>, **-u** <username>
Specifies a username with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --password and --authenticationDatabase options.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --username and --authenticationDatabase options.

If you do not specify an argument for *--password* (page 667), mongorestore (page 628) will prompt interactively for a password on the console.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's credentials.

If you do not specify an authentication database, mongorestore (page 628) assumes that the database specified as the argument to *--authenticationDatabase* (page 667) holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
Default: MONGODB-CR

New in version 2.4.

Changed in version 2.6: Added support for the PLAIN and MONGODB-X509 authentication mechanisms.

Specifies the authentication mechanism the mongorestore (page 628) instance uses to authenticate to the mongod (page 583) or mongos (page 601).

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

Value	Description
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB SSL certificate authentication.
PLAIN	External authentication using LDAP. You can also use `PLAIN` for authenticating in-database users. `PLAIN` transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[20].
GSSAPI	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[21].

command line option!–gssapiServiceName

--gssapiServiceName
New in version 2.6.

Specify the name of the service using `GSSAPI/Kerberos`. Only required if the service does not use the default name of `mongodb`.

This option is available only in MongoDB Enterprise.

command line option!–gssapiHostName

--gssapiHostName
New in version 2.6.

Specify the hostname of a service using `GSSAPI/Kerberos`. *Only* required if the hostname of a machine does not match the hostname resolved by DNS.

This option is available only in MongoDB Enterprise.

command line option!–db <database>, -d <database>

--db `<database>`**, -d** `<database>`
Specifies a database for `mongorestore` (page 628) to restore data *into*. If the database does not exist, `mongorestore` (page 628) creates the database. If you do not specify a `<db>`, `mongorestore` (page 628) creates new databases that correspond to the databases where data originated and data may be overwritten. Use this option to restore data into a MongoDB instance that already has data.

--db (page 632) does *not* control which *BSON* files `mongorestore` (page 628) restores. You must use the `mongorestore` (page 628) *path option* (page 634) to limit that restored data.

command line option!–collection <collection>, -c <collection>

--collection `<collection>`**, -c** `<collection>`
Specifies a single collection for `mongorestore` (page 628) to restore. If you do not specify *--collection* (page 632), `mongorestore` (page 628) takes the collection name from the input filename. If the input file has an extension, MongoDB omits the extension of the file from the collection name.

command line option!–objcheck

--objcheck
Forces `mongorestore` (page 628) to validate all requests from clients upon receipt to ensure that clients never insert invalid documents into the database. For objects with a high degree of sub-document nesting, *--objcheck* (page 588) can have a small impact on performance. You can set *--noobjcheck* (page 588) to disable object checking at run-time.

Changed in version 2.4: MongoDB enables *--objcheck* (page 588) by default, to prevent any client from inserting malformed or invalid BSON into a MongoDB database.

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

command line option!–noobjcheck

--noobjcheck
New in version 2.4.

Disables the default document validation that MongoDB performs on all incoming BSON documents.

command line option!–drop

--drop
Modifies the restoration procedure to drop every collection from the target database before restoring the collection from the dumped backup.

With *--drop* (page 633) specified, mongorestore (page 628) removes all user credentials and replaces them with users defined in the dump file. Therefore, in systems with authorization enabled, mongorestore (page 628) must be able to authenticate to an existing user *and* to a user defined in the dump file. If mongorestore (page 628) can't authenticate to a user defined in the dump file, the restoration process will fail, leaving an empty database.

command line option!–oplogReplay

--oplogReplay
Replays the *oplog* after restoring the dump to ensure that the current state of the database reflects the point-in-time backup captured with the "*mongodump --oplog*" command. For an example of *--oplogReplay* (page 633), see *backup-restore-oplogreplay*.

command line option!–oplogLimit <timestamp>

--oplogLimit <timestamp>
New in version 2.2.

Prevents mongorestore (page 628) from applying *oplog* entries with timestamp newer than or equal to <timestamp>. Specify <timestamp> values in the form of <time_t>:<ordinal>, where <time_t> is the seconds since the UNIX epoch, and <ordinal> represents a counter of operations in the oplog that occurred in the specified second.

You must use *--oplogLimit* (page 633) in conjunction with the *--oplogReplay* (page 633) option.

command line option!–keepIndexVersion

--keepIndexVersion
Prevents mongorestore (page 628) from upgrading the index to the latest version during the restoration process.

command line option!–noIndexRestore

--noIndexRestore
New in version 2.2.

Prevents mongorestore (page 628) from restoring and building indexes as specified in the corresponding mongodump (page 622) output.

command line option!–noOptionsRestore

--noOptionsRestore
New in version 2.2.

Prevents mongorestore (page 628) from setting the collection options, such as those specified by the collMod (page 321) *database command*, on restored collections.

command line option!–restoreDbUsersAndRoles

--restoreDbUsersAndRoles
Restore user and role definitions for the given database. See http://docs.mongodb.org/manual/reference/system

and http://docs.mongodb.org/manual/reference/system-users-collection for more information.

command line option!–w <number of replicas per write>

--w <number of replicas per write>
New in version 2.2.

Specifies the *write concern* for each write operation that mongorestore (page 628) writes to the target database. By default, mongorestore (page 628) does not wait for a response for *write acknowledgment*.

command line option!–writeConcern <document>

--writeConcern <document>
Default: majority

Specifies the *write concern* for each write operation that mongorestore (page 628) writes to the target database.

Specify the write concern as a document with *w options*.

command line option!–maintainInsertionOrder

--maintainInsertionOrder
Default: False

If specified, mongorestore (page 628) inserts the documents in the order of their appearance in the input source, otherwise mongorestore (page 628) may perform the insertions in an arbitrary order.

command line option!–numParallelCollections int, -j int

--numParallelCollections int, **-j** int
Default: 4

Number of collections mongorestore (page 628) should restore in parallel.

<path>
The final argument of the mongorestore (page 628) command is a directory path. This argument specifies the location of the database dump from which to restore.

command line option!–dir string

--dir string
Specifies the dump directory.

Use

See http://docs.mongodb.org/manual/tutorial/backup-with-mongodump for a larger overview of mongorestore (page 628) usage. Also see the *mongodump* (page 620) document for an overview of the mongodump (page 622), which provides the related inverse functionality.

Consider the following example:

```
mongorestore --collection people --db accounts dump/accounts/people.bson
```

Here, mongorestore (page 628) reads the database dump in the dump/ sub-directory of the current directory, and restores *only* the documents in the collection named people from the database named accounts. mongorestore (page 628) restores data to the instance running on the localhost interface on port 27017.

In the final example, mongorestore (page 628) restores a database dump located at /opt/backup/mongodump-2011-10-24, to a database running on port 37017 on the host mongodb1.example.net. The mongorestore (page 628) command authenticates to the MongoDB instance using the username user and the password pass, as follows:

```
mongorestore --host mongodb1.example.net --port 37017 --username user --password pass /opt/backup/mor
```

You can also *pipe* data directly into to mongorestore (page 628) through standard input, as in the following example:

```
zcat /opt/backup/mongodump-2014-12-03/accounts.people.bson.gz | mongorestore --collection people --db
```

bsondump

Synopsis

The bsondump (page 635) converts *BSON* files into human-readable formats, including *JSON*. For example, bsondump (page 635) is useful for reading the output files generated by mongodump (page 622).

Important: bsondump (page 635) is a diagnostic tool for inspecting BSON files, not a tool for data ingestion or other application use.

Options

Changed in version 2.8.0: bsondump (page 635) removed the `--filter` option.

bsondump

bsondump

command line option!–help

--help
Returns information on the options and use of bsondump (page 635).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

command line option!–quiet

--quiet
Runs the bsondump (page 635) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

- output from *database commands*
- replication activity
- connection accepted events
- connection closed events

command line option!–version

--version
Returns the bsondump (page 635) release number.

command line option!–objcheck

--objcheck
Validates each *BSON* object before outputting it in *JSON* format. By default, bsondump (page 635) enables --objcheck (page 588). For objects with a high degree of sub-document nesting, --objcheck (page 588) can have a small impact on performance. You can set --noobjcheck (page 588) to disable object checking.

Changed in version 2.4: MongoDB enables --objcheck (page 588) by default, to prevent any client from inserting malformed or invalid BSON into a MongoDB database.

command line option!–noobjcheck

--noobjcheck
New in version 2.4.

Disables the default document validation that MongoDB performs on all incoming BSON documents.

command line option!–type <=json|=debug>

--type <=json|=debug>
Changes the operation of bsondump (page 635) from outputting "*JSON*" (the default) to a debugging format.

command line option!–pretty

--pretty
New in version 2.8.0.

Outputs documents in a pretty-printed format JSON.

<bsonFilename>
The final argument to bsondump (page 635) is a document containing *BSON*. This data is typically generated by bsondump (page 635) or by MongoDB in a *rollback* operation.

Use

By default, bsondump (page 635) outputs data to standard output. To create corresponding *JSON* files, you will need to use the shell redirect. See the following command:

```
bsondump collection.bson > collection.json
```

Use the following command (at the system shell) to produce debugging output for a *BSON* file:

```
bsondump --type=debug collection.bson
```

mongooplog

New in version 2.2.

Synopsis

mongooplog (page 637) is a simple tool that polls operations from the *replication oplog* of a remote server, and applies them to the local server. This capability supports certain classes of real-time migrations that require that the source server remain online and in operation throughout the migration process.

Typically this command will take the following form:

```
mongooplog  --from mongodb0.example.net --host mongodb1.example.net
```

This command copies oplog entries from the mongod (page 583) instance running on the host mongodb0.example.net and duplicates operations to the host mongodb1.example.net. If you do not need to keep the *--from* host running during the migration, consider using mongodump (page 622) and mongorestore (page 628) or another backup operation, which may be better suited to your operation.

Note: If the mongod (page 583) instance specified by the *--from* argument is running with authentication, then mongooplog (page 637) will not be able to copy oplog entries.

See also:

mongodump (page 622), mongorestore (page 628), http://docs.mongodb.org/manual/core/backups, http://docs.mongodb.org/manual/core/replica-set-oplog.

Options

Changed in version 2.8.0: mongooplog (page 637) removed the --dbpath as well as related --directoryperdb and --journal options. You must use mongooplog (page 637) while connected to a mongod (page 583) instance.

mongooplog

mongooplog

command line option!–help

--help
Returns information on the options and use of mongooplog (page 637).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

command line option!–quiet

--quiet
Runs the mongooplog (page 637) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

- connection accepted events
- connection closed events

command line option!–version

--version
Returns the mongooplog (page 637) release number.

command line option!–host <hostname><:port>, -h <hostname><:port>

--host <hostname><:port>, **-h** <hostname><:port>
Specifies a resolvable hostname for the mongod (page 583) instance to which mongooplog (page 637) will apply *oplog* operations retrieved from the server specified by the *--from* option.

By default mongooplog (page 637) attempts to connect to a MongoDB instance running on the localhost on port number 27017.

To connect to a replica set, specify the replica set name and a seed list of set members. Use the following form:

```
<replSetName>/<hostname1><:port>,<hostname2><:port>,<...>
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

command line option!–port

--port
Specifies the port number of the mongod (page 583) instance where mongooplog (page 637) will apply *oplog* entries. Specify this option only if the MongoDB instance to connect to is not running on the standard port of 27017. You may also specify a port number using the *--host* command.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the mongooplog (page 637) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.6.

Enables connection to a mongod (page 583) or mongos (page 601) that has SSL support enabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.6.

Specifies the .pem file that contains the root certificate chain from the Certificate Authority. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

> **Warning:** If the mongo (page 610) shell or any other tool that connects to mongos (page 601) or mongod (page 583) is run without *--sslCAFile*, it will not attempt to validate server certificates. This results in vulnerability to expired mongod (page 583) and mongos (page 601) certificates as well as to foreign processes posing as valid mongod (page 583) or mongos (page 601) instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.6.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths.

This option is required when using the *--ssl* (page 665) option to connect to a mongod (page 583) or mongos (page 601) that has CAFile enabled *without* allowConnectionsWithoutCertificates.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.6.

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 596)). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongooplog (page 637) will redact the password from all logging and reporting output.

If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the mongooplog (page 637) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCRLFile <filename>

--sslCRLFile <filename>
New in version 2.6.

Specifies the .pem file that contains the Certificate Revocation List. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for server certificates and allows the use of invalid certificates. When using the allowInvalidCertificates setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates. Allows mongooplog (page 637) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.6.

Directs the mongooplog (page 637) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[22]. See http://docs.mongodb.org/manual/tutorial/configure-fips for more information.

---

command line option!–username <username>, -u <username>

--username <username>, **-u** <username>
Specifies a username with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --password and --authenticationDatabase options.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --username and --authenticationDatabase options.

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

If you do not specify an argument for *--password* (page 667), mongooplog (page 637) will prompt interactively for a password on the console.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's credentials.

If you do not specify an authentication database, mongooplog (page 637) assumes that the database specified as the argument to *--authenticationDatabase* (page 667) holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
Default: MONGODB-CR

New in version 2.4.

Changed in version 2.6: Added support for the PLAIN and MONGODB-X509 authentication mechanisms.

Specifies the authentication mechanism the mongooplog (page 637) instance uses to authenticate to the mongod (page 583) or mongos (page 601).

Value	Description
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB SSL certificate authentication.
PLAIN	External authentication using LDAP. You can also use PLAIN for authenticating in-database users. PLAIN transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[23].
GSSAPI	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[24].

command line option!–gssapiServiceName

--gssapiServiceName
New in version 2.6.

Specify the name of the service using GSSAPI/Kerberos. Only required if the service does not use the default name of mongodb.

This option is available only in MongoDB Enterprise.

command line option!–gssapiHostName

--gssapiHostName
New in version 2.6.

Specify the hostname of a service using GSSAPI/Kerberos. *Only* required if the hostname of a machine does not match the hostname resolved by DNS.

This option is available only in MongoDB Enterprise.

command line option!–db <database>, -d <database>

--db <database>, **-d** <database>
Specifies the name of the database on which to run the mongooplog (page 637).

command line option!–collection <collection>, -c <collection>

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

--collection <collection>, **-c** <collection>
Specifies the collection to export.

command line option!–seconds <number>, -s <number>

--seconds <number>, **-s** <number>
Specify a number of seconds of operations for mongooplog (page 637) to pull from the *remote host*. Unless specified the default value is 86400 seconds, or 24 hours.

command line option!–from <host[:port]>

--from <host[:port]>
Specify the host for mongooplog (page 637) to retrieve *oplog* operations from. mongooplog (page 637) *requires* this option.

Unless you specify the *--host* option, mongooplog (page 637) will apply the operations collected with this option to the oplog of the mongod (page 583) instance running on the localhost interface connected to port 27017.

command line option!–oplogns <namespace>

--oplogns <namespace>
Specify a namespace in the *--from* host where the oplog resides. The default value is local.oplog.rs, which is the where *replica set* members store their operation log. However, if you've copied *oplog* entries into another database or collection or are pulling oplog entries from a master-slave deployment, use *--oplogns* (page 641) to apply oplog entries stored in another location. Namespaces take the form of [database].[collection].

Use

Consider the following prototype mongooplog (page 637) command:

```
mongooplog  --from mongodb0.example.net --host mongodb1.example.net
```

Here, entries from the *oplog* of the mongod (page 583) running on port 27017. This only pull entries from the last 24 hours.

Use the *--seconds* argument to capture a greater or smaller amount of time. Consider the following example:

```
mongooplog  --from mongodb0.example.net --seconds 172800
```

In this operation, mongooplog (page 637) captures 2 full days of operations. To migrate 12 hours of *oplog* entries, use the following form:

```
mongooplog  --from mongodb0.example.net --seconds 43200
```

4.1.4 Data Import and Export Tools

mongoimport (page 642) provides a method for taking data in *JSON*, *CSV*, or *TSV* and importing it into a mongod (page 583) instance. mongoexport (page 649) provides a method to export data from a mongod (page 583) instance into JSON, CSV, or TSV.

Note: The conversion between BSON and other formats lacks full type fidelity. Therefore you cannot use mongoimport (page 642) and mongoexport (page 649) for round-trip import and export operations.

---

mongoimport

Synopsis

The mongoimport (page 642) tool provides a route to import content from a JSON, CSV, or TSV export created by mongoexport (page 649), or potentially, another third-party export tool. See the http://docs.mongodb.org/manual/core/import-export document for a more in depth usage overview, and the *mongoexport* (page 649) document for more information regarding mongoexport (page 649), which provides the inverse "exporting" capability.

Considerations

Do not use mongoimport (page 642) and mongoexport (page 649) for full instance, production backups because they will not reliably capture data type information. Use mongodump (page 622) and mongorestore (page 628) as described in http://docs.mongodb.org/manual/core/backups for this kind of functionality.

mongoimport (page 642) is single-threaded and inserts one document at a time into MongoDB. Custom import tools for data ingestion may have better performance for specific workloads.

Options

Changed in version 2.8.0: mongoimport (page 642) removed the --dbpath as well as related --directoryperdb and --journal options. You must use mongoimport (page 642) while connected to a mongod (page 583) instance.

mongoimport

mongoimport

command line option!–help

--help
Returns information on the options and use of mongoimport (page 642).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

command line option!–quiet

--quiet
Runs the mongoimport (page 642) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

- output from *database commands*
- replication activity
- connection accepted events
- connection closed events

command line option!–version

--version
Returns the mongoimport (page 642) release number.

command line option!–host <hostname><:port>, -h <hostname><:port>

--host <hostname><:port>, **-h** <hostname><:port>
Default: localhost:27017

Specifies a resolvable hostname for the mongod (page 583) to which to connect. By default, the mongoimport (page 642) attempts to connect to a MongoDB instance running on the localhost on port number 27017.

To connect to a replica set, specify the replica set name and a seed list of set members. Use the following form:

```
<replSetName>/<hostname1><:port>,<hostname2><:port>,<...>
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

Changed in version 2.8.0: If you use IPv6 and use the <address>:<port > format, you must enclose the portion of an address and port combination in brackets (e.g. [<address>]).

command line option!–port <port>

--port <port>
Default: 27017

Specifies the TCP port on which the MongoDB instance listens for client connections.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the mongoimport (page 642) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.6.

Enables connection to a mongod (page 583) or mongos (page 601) that has SSL support enabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.6.

Specifies the .pem file that contains the root certificate chain from the Certificate Authority. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

> **Warning:** If the mongo (page 610) shell or any other tool that connects to mongos (page 601) or mongod (page 583) is run without *--sslCAFile*, it will not attempt to validate server certificates. This results in vulnerability to expired mongod (page 583) and mongos (page 601) certificates as well as to foreign processes posing as valid mongod (page 583) or mongos (page 601) instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.6.

Specifies the `.pem` file that contains both the SSL certificate and key. Specify the file name of the `.pem` file using relative or absolute paths.

This option is required when using the *--ssl* (page 665) option to connect to a `mongod` (page 583) or `mongos` (page 601) that has `CAFile` enabled *without* `allowConnectionsWithoutCertificates`.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword `<value>`
New in version 2.6.

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 596)). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the `mongoimport` (page 642) will redact the password from all logging and reporting output.

If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the `mongoimport` (page 642) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslCRLFile <filename>

--sslCRLFile `<filename>`
New in version 2.6.

Specifies the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for server certificates and allows the use of invalid certificates. When using the `allowInvalidCertificates` setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates. Allows `mongoimport` (page 642) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.6.

Directs the `mongoimport` (page 642) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[25]. See

http://docs.mongodb.org/manual/tutorial/configure-fips for more information.

command line option!–username <username>, -u <username>

--username <username>, **-u** <username>
Specifies a username with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --password and --authenticationDatabase options.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --username and --authenticationDatabase options.

If you do not specify an argument for *--password* (page 667), mongoimport (page 642) will prompt interactively for a password on the console.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's credentials.

If you do not specify an authentication database, mongoimport (page 642) assumes that the database specified as the argument to *--authenticationDatabase* (page 667) holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
Default: MONGODB-CR

New in version 2.4.

Changed in version 2.6: Added support for the PLAIN and MONGODB-X509 authentication mechanisms.

Specifies the authentication mechanism the mongoimport (page 642) instance uses to authenticate to the mongod (page 583) or mongos (page 601).

Value	Description
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB SSL certificate authentication.
PLAIN	External authentication using LDAP. You can also use PLAIN for authenticating in-database users. PLAIN transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[26].
GSSAPI	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[27].

command line option!–gssapiServiceName

--gssapiServiceName
New in version 2.6.

Specify the name of the service using GSSAPI/Kerberos. Only required if the service does not use the default name of mongodb.

This option is available only in MongoDB Enterprise.

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

command line option!–gssapiHostName

--gssapiHostName
New in version 2.6.

Specify the hostname of a service using `GSSAPI/Kerberos`. *Only* required if the hostname of a machine does not match the hostname resolved by DNS.

This option is available only in MongoDB Enterprise.

command line option!–db <database>, -d <database>

--db `<database>`, **-d** `<database>`
Specifies the name of the database on which to run the `mongoimport` (page 642).

command line option!–collection <collection>, -c <collection>

--collection `<collection>`, **-c** `<collection>`
Specifies the collection to import.

New in version 2.6: If you do not specify *--collection* (page 632), `mongoimport` (page 642) takes the collection name from the input filename. MongoDB omits the extension of the file from the collection name, if the input file has an extension.

command line option!–fields <field1[,field2]>, -f <field1[,field2]>

--fields `<field1[,field2]>`, **-f** `<field1[,field2]>`
Specify a comma separated list of field names when importing *csv* or *tsv* files that do not have field names in the first (i.e. header) line of the file.

If you attempt to include *--fields* (page 646) when importing JSON data, `mongoimport` (page 642) will return an error. *--fields* (page 646) is only for *csv* or *tsv* imports.

command line option!–fieldFile <filename>

--fieldFile `<filename>`
As an alternative to *--fields* (page 646), the *--fieldFile* (page 646) option allows you to specify a file that holds a list of field names if your *csv* or *tsv* file does not include field names in the first line of the file (i.e. header). Place one field per line.

If you attempt to include *--fieldFile* (page 646) when importing JSON data, `mongoimport` (page 642) will return an error. *--fieldFile* (page 646) is only for *csv* or *tsv* imports.

command line option!–ignoreBlanks

--ignoreBlanks
Ignores empty fields in *csv* and *tsv* exports. If not specified, `mongoimport` (page 642) creates fields without values in imported documents.

If you attempt to include *--ignoreBlanks* (page 646) when importing JSON data, `mongoimport` (page 642) will return an error. *--ignoreBlanks* (page 646) is only for *csv* or *tsv* imports.

command line option!–type <json|csv|tsv>

--type `<json|csv|tsv>`
Specifies the file type to import. The default format is *JSON*, but it's possible to import *csv* and *tsv* files.

The `csv` parser accepts that data that complies with RFC **RFC 4180**[28]. As a result, backslashes are *not* a valid escape character. If you use double-quotes to enclose fields in the CSV data, you must escape internal double-quote marks by prepending another double-quote.

command line option!–file <filename>

[28] http://tools.ietf.org/html/rfc4180.html

--file <filename>
Specifies the location and name of a file containing the data to import. If you do not specify a file, mongoimport (page 642) reads data from standard input (e.g. "stdin").

command line option!–drop

--drop
Modifies the import process so that the target instance drops the collection before importing the data from the input.

command line option!–headerline

--headerline
If using *--type csv* or *--type tsv*, uses the first line as field names. Otherwise, mongoimport (page 642) will import the first line as a distinct document.

If you attempt to include *--headerline* (page 647) when importing JSON data, mongoimport (page 642) will return an error. *--headerline* (page 647) is only for *csv* or *tsv* imports.

command line option!–upsert

--upsert
Modifies the import process to update existing objects in the database if they match an imported object, while inserting all other objects.

If you do not specify a field or fields using the *--upsertFields* (page 647) mongoimport (page 642) will upsert on the basis of the _id field.

..versionchanged:: 2.8.0 *--upsert* (page 647) is no longer needed when specifying upserts. Use *--upsertFields* (page 647), which produces the same behavior.

command line option!–upsertFields <field1[,field2]>

--upsertFields <field1[,field2]>
Specifies a list of fields for the query portion of the *upsert*. Use this option if the _id fields in the existing documents don't match the field in the document, but another field or field combination can uniquely identify documents as a basis for performing upsert operations.

..versionchanged:: 2.8.0 Modifies the import process to update existing objects in the database if they match based on the specified fields, while inserting all other objects.

If you do not specify a field, *--upsertFields* (page 647) will upsert on the basis of the _id field.

To ensure adequate performance, indexes should exist for this field or fields.

command line option!–stopOnError

--stopOnError
New in version 2.2.

Forces mongoimport (page 642) to halt the import operation at the first error rather than continuing the operation despite errors.

Changed in version 2.8.0: *--stopOnError* (page 647) interrupts the import operation when mongoimport (page 642) encounters an insert or upsert error. Other error types will not stop the import.

command line option!–jsonArray

--jsonArray
Accepts the import of data expressed with multiple MongoDB documents within a single *JSON* array. Limited to imports of 16 MB or smaller.

Use *--jsonArray* (page 647) in conjunction with *mongoexport --jsonArray*.

command line option!–maintainInsertionOrder

--maintainInsertionOrder
Default: False

If specified, mongoimport (page 642) inserts the documents in the order of their appearance in the input source, otherwise mongoimport (page 642) may perform the insertions in an arbitrary order.

command line option!–writeConcern <document>

--writeConcern <document>
Default: majority

Specifies the *write concern* for each write operation that mongoimport (page 642) writes to the target database.

Specify the write concern as a document with *w options*.

Use

In this example, mongoimport (page 642) imports the *csv* formatted data in the /opt/backups/contacts.csv into the collection contacts in the users database on the MongoDB instance running on the localhost port numbered 27017. mongoimport (page 642) determines the name of files using the first line in the CSV file, because of the *--headerline*:

```
mongoimport --db users --collection contacts --type csv --headerline --file /opt/backups/contacts.csv
```

Since mongoimport (page 642) uses the input file name, without the extension, as the collection name if -c or --collection is unspecified. The following example is equivalent:

```
mongoimport --db users --type csv --headerline --file /opt/backups/contacts.csv
```

In the following example, mongoimport (page 642) imports the data in the *JSON* formatted file contacts.json into the collection contacts on the MongoDB instance running on the localhost port number 27017.

```
mongoimport --db users --collection contacts --file contacts.json
```

In the next example, mongoimport (page 642) imports data from the file /opt/backups/mdb1-examplenet.json into the collection contacts within the database marketing on a remote MongoDB database. This mongoimport (page 642) accesses the mongod (page 583) instance running on the host mongodb1.example.net over port 37017, which requires the username user and the password pass.

```
mongoimport --host mongodb1.example.net --port 37017 --username user --password pass --collection co
```

Type Fidelity

Warning: mongoimport (page 642) and mongoexport (page 649) do not reliably preserve all rich *BSON* data types because *JSON* can only represent a subset of the types supported by BSON. As a result, data exported or imported with these tools may lose some measure of fidelity. See the Extended JSON reference for more information.

JSON can only represent a subset of the types supported by BSON. To preserve type information, mongoimport (page 642) accepts strict mode representation for certain types.

For example, to preserve type information for BSON types data_date and data_numberlong during mongoimport (page 642), the data should be in strict mode representation, as in the following:

```
{ "_id" : 1, "volume" : { "$numberLong" : "2980000" }, "date" : { "$date" : "2014-03-13T13:47:42.483-
```

For the `data_numberlong` type, `mongoimport` (page 642) converts into a float during the import.

See `http://docs.mongodb.org/manual/reference/mongodb-extended-json` for a complete list of these types and the representations used.

mongoexport

Synopsis

`mongoexport` (page 649) is a utility that produces a JSON or CSV export of data stored in a MongoDB instance. See the `http://docs.mongodb.org/manual/core/import-export` document for a more in depth usage overview, and the *mongoimport* (page 642) document for more information regarding the `mongoimport` (page 642) utility, which provides the inverse "importing" capability.

Considerations

Do not use `mongoimport` (page 642) and `mongoexport` (page 649) for full-scale production backups because they may not reliably capture data type information. Use `mongodump` (page 622) and `mongorestore` (page 628) as described in `http://docs.mongodb.org/manual/core/backups` for this kind of functionality.

Options

Changed in version 2.8.0: `mongoexport` (page 649) removed the `--dbpath` as well as related `--directoryperdb` and `--journal` options. You must use `mongoexport` (page 649) while connected to a `mongod` (page 583) instance.

Changed in version 2.8.0: `mongoexport` (page 649) also removed support for writing data to `tsv` files with the `--tsv` option.

mongoexport

mongoexport

command line option!–help

--help
Returns information on the options and use of `mongoexport` (page 649).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

command line option!–quiet

--quiet
Runs the `mongoexport` (page 649) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

- output from *database commands*
- replication activity
- connection accepted events

- connection closed events

command line option!–version

--version
Returns the mongoexport (page 649) release number.

command line option!–host <hostname><:port>, -h <hostname><:port>

--host <hostname><:port>, **-h** <hostname><:port>
Default: localhost:27017

Specifies a resolvable hostname for the mongod (page 583) to which to connect. By default, the mongoexport (page 649) attempts to connect to a MongoDB instance running on the localhost on port number 27017.

To connect to a replica set, specify the replica set name and a seed list of set members. Use the following form:

```
<replSetName>/<hostname1><:port>,<hostname2><:port>,<...>
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

Changed in version 2.8.0: If you use IPv6 and use the <address>:<port > format, you must enclose the portion of an address and port combination in brackets (e.g. [<address>]).

command line option!–port <port>

--port <port>
Default: 27017

Specifies the TCP port on which the MongoDB instance listens for client connections.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the mongoexport (page 649) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.6.

Enables connection to a mongod (page 583) or mongos (page 601) that has SSL support enabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.6.

Specifies the .pem file that contains the root certificate chain from the Certificate Authority. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

> **Warning:** If the mongo (page 610) shell or any other tool that connects to mongos (page 601) or mongod (page 583) is run without *--sslCAFile*, it will not attempt to validate server certificates. This results in vulnerability to expired mongod (page 583) and mongos (page 601) certificates as well as to foreign processes posing as valid mongod (page 583) or mongos (page 601) instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.6.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths.

This option is required when using the *--ssl* (page 665) option to connect to a mongod (page 583) or mongos (page 601) that has CAFile enabled *without* allowConnectionsWithoutCertificates.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.6.

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 596)). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongoexport (page 649) will redact the password from all logging and reporting output.

If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the mongoexport (page 649) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCRLFile <filename>

--sslCRLFile <filename>
New in version 2.6.

Specifies the .pem file that contains the Certificate Revocation List. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for server certificates and allows the use of invalid certificates. When using the allowInvalidCertificates setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates. Allows mongoexport (page 649) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.6.

Directs the mongoexport (page 649) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[29]. See http://docs.mongodb.org/manual/tutorial/configure-fips for more information.

---

command line option!–username <username>, -u <username>

--username <username>, **-u** <username>
Specifies a username with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --password and --authenticationDatabase options.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --username and --authenticationDatabase options.

If you do not specify an argument for *--password* (page 667), mongoexport (page 649) will prompt interactively for a password on the console.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's credentials.

If you do not specify an authentication database, mongoexport (page 649) assumes that the database specified to export holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
Default: MONGODB-CR

New in version 2.4.

Changed in version 2.6: Added support for the PLAIN and MONGODB-X509 authentication mechanisms.

Specifies the authentication mechanism the mongoexport (page 649) instance uses to authenticate to the mongod (page 583) or mongos (page 601).

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

Value	Description
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB SSL certificate authentication.
PLAIN	External authentication using LDAP. You can also use `PLAIN` for authenticating in-database users. `PLAIN` transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[30].
GSSAPI	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[31].

command line option!–gssapiServiceName

`--gssapiServiceName`
New in version 2.6.

Specify the name of the service using `GSSAPI/Kerberos`. Only required if the service does not use the default name of `mongodb`.

This option is available only in MongoDB Enterprise.

command line option!–gssapiHostName

`--gssapiHostName`
New in version 2.6.

Specify the hostname of a service using `GSSAPI/Kerberos`. *Only* required if the hostname of a machine does not match the hostname resolved by DNS.

This option is available only in MongoDB Enterprise.

command line option!–db <database>, -d <database>

`--db` `<database>`**, `-d` `<database>`**
Specifies the name of the database on which to run the `mongoexport` (page 649).

command line option!–collection <collection>, -c <collection>

`--collection` `<collection>`**, `-c` `<collection>`**
Specifies the collection to export.

command line option!–fields <field1[,field2]>, -f <field1[,field2]>

`--fields` `<field1[,field2]>`**, `-f` `<field1[,field2]>`**
Specifies a field or fields to *include* in the export. Use a comma separated list of fields to specify multiple fields.

For *`--csv`* output formats, `mongoexport` (page 649) includes only the specified field(s), and the specified field(s) can be a field within a sub-document.

For *JSON* output formats, `mongoexport` (page 649) includes only the specified field(s) **and** the `_id` field, and if the specified field(s) is a field within a sub-document, the `mongoexport` (page 649) includes the sub-document with all its fields, not just the specified field within the document.

command line option!–fieldFile <filename>

`--fieldFile` `<filename>`
An alternative to *`--fields`*. The *`--fieldFile`* (page 646) option allows you to specify in a file the field or fields to *include* in the export and is **only valid** with the *`--csv`* option. The file must have only one field per line, and the line(s) must end with the LF character (`0x0A`).

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

mongoexport (page 649) includes only the specified field(s). The specified field(s) can be a field within a sub-document.

command line option!–query <JSON>, -q <JSON>

--query <JSON>, **-q** <JSON>
Provides a *JSON document* as a query that optionally limits the documents returned in the export. Specify JSON in strict format.

For example, given a collection named records in the database test with the following documents:

```
{ "_id" : ObjectId("51f0188846a64a1ed98fde7c"), "a" : 1 }
{ "_id" : ObjectId("520e61b0c6646578e3661b59"), "a" : 1, "b" : 2 }
{ "_id" : ObjectId("520e642bb7fa4ea22d6b1871"), "a" : 2, "b" : 3, "c" : 5 }
{ "_id" : ObjectId("520e6431b7fa4ea22d6b1872"), "a" : 3, "b" : 3, "c" : 6 }
{ "_id" : ObjectId("520e6445b7fa4ea22d6b1873"), "a" : 5, "b" : 6, "c" : 8 }
```

The following mongoexport (page 649) uses the *-q* (page **??**) option to export only the documents with the field a greater than or equal to ($gte (page 401)) to 3:

```
mongoexport -d test -c records -q "{ a: { \$gte: 3 } }" --out exportdir/myRecords.json
```

The resulting file contains the following documents:

```
{ "_id" : { "$oid" : "520e6431b7fa4ea22d6b1872" }, "a" : 3, "b" : 3, "c" : 6 }
{ "_id" : { "$oid" : "520e6445b7fa4ea22d6b1873" }, "a" : 5, "b" : 6, "c" : 8 }
```

You can sort the results with the *--sort* (page 655) option to mongoexport (page 649).

command line option!–csv

--csv
Changes the export format to a comma-separated-values (CSV) format. By default mongoexport (page 649) writes data using one *JSON* document for every MongoDB document.

If you specify *--csv* (page 654), then you must also use either the *--fields* (page 646) or the *--fieldFile* (page 646) option to declare the fields to export from the collection.

command line option!–out <file>, -o <file>

--out <file>, **-o** <file>
Specifies a file to write the export to. If you do not specify a file name, the mongoexport (page 649) writes data to standard output (e.g. stdout).

command line option!–jsonArray

--jsonArray
Modifies the output of mongoexport (page 649) to write the entire contents of the export as a single *JSON* array. By default mongoexport (page 649) writes data using one JSON document for every MongoDB document.

command line option!–pretty

--pretty
New in version 2.8.0.

Outputs documents in a pretty-printed format JSON.

command line option!–slaveOk, -k

--slaveOk, -k
Allows mongoexport (page 649) to read data from secondary or slave nodes when using mongoexport (page 649) with a replica set. This option is only available if connected to a mongod (page 583) or mongos (page 601) and is not available when used with the "*mongoexport --dbpath*" option.

This is the default behavior.

command line option!–forceTableScan

--forceTableScan
New in version 2.2.

Forces mongoexport (page 649) to scan the data store directly: typically, mongoexport (page 649) saves entries as they appear in the index of the _id field. Use --forceTableScan (page 655) to skip the index and scan the data directly. Typically there are two cases where this behavior is preferable to the default:

1. If you have key sizes over 800 bytes that would not be present in the _id index.
2. Your database uses a custom _id field.

When you run with --forceTableScan (page 655), mongoexport (page 649) does not use $snapshot (page 562). As a result, the export produced by mongoexport (page 649) can reflect the state of the database at many different points in time.

Warning: Use --forceTableScan (page 655) with extreme caution and consideration.

command line option!–skip <number>

--skip <number>
Use --skip (page 655) to control where mongoexport (page 649) begins exporting documents. See skip() (page 94) for information about the underlying operation.

command line option!–limit <number>

--limit <number>
Specifies a maximum number of documents to include in the export. See limit() (page 88) for information about the underlying operation.

command line option!–sort <JSON>

--sort <JSON>
Specifies an ordering for exported results. If an index does **not** exist that can support the sort operation, the results must be *less than* 32 megabytes.

Use --sort (page 655) conjunction with --skip (page 655) and --limit (page 655) to limit number of exported documents.

```
mongoexport -d test -c records --sort '{a: 1}' --limit 100 --out export.0.json
mongoexport -d test -c records --sort '{a: 1}' --limit 100 --skip 100 --out export.1.json
mongoexport -d test -c records --sort '{a: 1}' --limit 100 --skip 200 --out export.2.json
```

See sort() (page 95) for information about the underlying operation.

Use

Export in CSV Format In the following example, mongoexport (page 649) exports the collection contacts from the users database from the mongod (page 583) instance running on the localhost port number 27017. This command writes the export data in *CSV* format into a file located at /opt/backups/contacts.csv. The fields.txt file contains a line-separated list of fields to export.

```
mongoexport --db users --collection contacts --csv --fieldFile fields.txt --out /opt/backups/contacts
```

Export in JSON Format The next example creates an export of the collection `contacts` from the MongoDB instance running on the localhost port number `27017`, with journaling explicitly enabled. This writes the export to the `contacts.json` file in *JSON* format.

```
mongoexport --db sales --collection contacts --out contacts.json --journal
```

Export from Remote Host Running with Authentication The following example exports the collection `contacts` from the database `marketing` . This data resides on the MongoDB instance located on the host `mongodb1.example.net` running on port `37017`, which requires the username `user` and the password `pass`.

```
mongoexport --host mongodb1.example.net --port 37017 --username user --password pass --collection co
```

Type Fidelity

Warning: `mongoimport` (page 642) and `mongoexport` (page 649) do not reliably preserve all rich *BSON* data types because *JSON* can only represent a subset of the types supported by BSON. As a result, data exported or imported with these tools may lose some measure of fidelity. See the `Extended JSON` reference for more information.

JSON can only represent a subset of the types supported by BSON. To preserve type information, `mongoexport` (page 649) uses the `strict mode representation` for certain types.

For example, the following insert operation in the `mongo` (page 610) shell uses the `mongoShell mode representation` for the BSON types `data_date` and `data_numberlong`:

```
use test
db.traffic.insert( { _id: 1, volume: NumberLong(2980000), date: new Date() } )
```

Use `mongoexport` (page 649) to export the data:

```
mongoexport --db test --collection traffic --out traffic.json
```

The exported data is in `strict mode representation` to preserve type information:

```
{ "_id" : 1, "volume" : { "$numberLong" : "2980000" }, "date" : { "$date" : "2014-03-13T13:47:42.483-
```

See `http://docs.mongodb.org/manual/reference/mongodb-extended-json` for a complete list of these types and the representations used.

4.1.5 Diagnostic Tools

`mongostat` (page 657), `mongotop` (page 664), and `mongosniff` (page 670) provide diagnostic information related to the current operation of a `mongod` (page 583) instance.

Note: Because `mongosniff` (page 670) depends on *libpcap*, most distributions of MongoDB do *not* include `mongosniff` (page 670).

mongostat

Synopsis

The mongostat (page 657) utility provides a quick overview of the status of a currently running mongod (page 583) or mongos (page 601) instance. mongostat (page 657) is functionally similar to the UNIX/Linux file system utility vmstat, but provides data regarding mongod (page 583) and mongos (page 601) instances.

See also:

For more information about monitoring MongoDB, see http://docs.mongodb.org/manual/administration/monitori

For more background on various other MongoDB status outputs see:

- *serverStatus* (page 366)
- *replSetGetStatus* (page 296)
- *dbStats* (page 352)
- *collStats* (page 348)

For an additional utility that provides MongoDB metrics see *mongotop* (page 664).

Access Control Requirements

In order to connect to a mongod (page 583) that enforces authorization with the *--auth* option, specify the *--username* and *--password* options, and the user specified must have the serverStatus privilege action on the cluster resources.

The built-in role clusterMonitor provides this privilege as well as other privileges. To create a role with just the privilege to run mongostat (page 657), see *create-role-for-mongostat*.

Options

mongostat

mongostat

command line option!–help

--help
Returns information on the options and use of mongostat (page 657).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

command line option!–version

--version
Returns the mongostat (page 657) release number.

command line option!–host <hostname><:port>, -h <hostname><:port>

--host <hostname><:port>, **-h** <hostname><:port>
Default: localhost:27017

Specifies a resolvable hostname for the mongod (page 583) to which to connect. By default, the mongostat (page 657) attempts to connect to a MongoDB instance running on the localhost on port number 27017.

To connect to a replica set, specify the replica set name and a seed list of set members. Use the following form:

```
<replSetName>/<hostname1><:port>,<hostname2><:port>,<...>
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

Changed in version 2.8.0: If you use IPv6 and use the <address>:<port > format, you must enclose the portion of an address and port combination in brackets (e.g. [<address>]).

command line option!–port <port>

--port <port>
Default: 27017

Specifies the TCP port on which the MongoDB instance listens for client connections.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the mongostat (page 657) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.6.

Enables connection to a mongod (page 583) or mongos (page 601) that has SSL support enabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.6.

Specifies the .pem file that contains the root certificate chain from the Certificate Authority. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

> **Warning:** If the mongo (page 610) shell or any other tool that connects to mongos (page 601) or mongod (page 583) is run without *--sslCAFile*, it will not attempt to validate server certificates. This results in vulnerability to expired mongod (page 583) and mongos (page 601) certificates as well as to foreign processes posing as valid mongod (page 583) or mongos (page 601) instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.6.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths.

This option is required when using the *--ssl* (page 665) option to connect to a mongod (page 583) or mongos (page 601) that has CAFile enabled *without* allowConnectionsWithoutCertificates.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.6.

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 596)). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongostat (page 657) will redact the password from all logging and reporting output.

If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the mongostat (page 657) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCRLFile <filename>

--sslCRLFile <filename>
New in version 2.6.

Specifies the .pem file that contains the Certificate Revocation List. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for server certificates and allows the use of invalid certificates. When using the allowInvalidCertificates setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates. Allows mongostat (page 657) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.6.

Directs the mongostat (page 657) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[32]. See http://docs.mongodb.org/manual/tutorial/configure-fips for more information.

---

command line option!–username <username>, -u <username>

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

--username <username>, **-u** <username>
Specifies a username with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --password and --authenticationDatabase options.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the --username and --authenticationDatabase options.

If you do not specify an argument for *--password* (page 667), mongostat (page 657) will prompt interactively for a password on the console.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's credentials.

If you do not specify an authentication database, mongostat (page 657) assumes that the database specified as the argument to *--authenticationDatabase* (page 667) holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
Default: MONGODB-CR

New in version 2.4.

Changed in version 2.6: Added support for the PLAIN and MONGODB-X509 authentication mechanisms.

Specifies the authentication mechanism the mongostat (page 657) instance uses to authenticate to the mongod (page 583) or mongos (page 601).

Value	Description
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB SSL certificate authentication.
PLAIN	External authentication using LDAP. You can also use PLAIN for authenticating in-database users. PLAIN transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[33].
GSSAPI	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[34].

command line option!–gssapiServiceName

--gssapiServiceName
New in version 2.6.

Specify the name of the service using GSSAPI/Kerberos. Only required if the service does not use the default name of mongodb.

This option is available only in MongoDB Enterprise.

command line option!–gssapiHostName

--gssapiHostName
New in version 2.6.

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

Specify the hostname of a service using `GSSAPI/Kerberos`. *Only* required if the hostname of a machine does not match the hostname resolved by DNS.

This option is available only in MongoDB Enterprise.

command line option!–noheaders

`--noheaders`
Disables the output of column or field names.

command line option!–rowcount <number>, -n <number>

`--rowcount` `<number>`, **`-n`** `<number>`
Controls the number of rows to output. Use in conjunction with the `sleeptime` argument to control the duration of a `mongostat` (page 657) operation.

Unless *`--rowcount`* (page 668) is specified, `mongostat` (page 657) will return an infinite number of rows (e.g. value of `0`.)

command line option!–http

`--http`
Configures `mongostat` (page 657) to collect data using the HTTP interface rather than a raw database connection.

command line option!–discover

`--discover`
Discovers and reports on statistics from all members of a *replica set* or *sharded cluster*. When connected to any member of a replica set, *`--discover`* (page 661) all non-*hidden members* of the replica set. When connected to a `mongos` (page 601), `mongostat` (page 657) will return data from all *shards* in the cluster. If a replica set provides a shard in the sharded cluster, `mongostat` (page 657) will report on non-hidden members of that replica set.

The *`mongostat --host`* option is not required but potentially useful in this case.

Changed in version 2.6: When running with *`--discover`* (page 661), `mongostat` (page 657) now respects :option:–rowcount‘.

command line option!–all

`--all`
Configures `mongostat` (page 657) to return all optional *fields* (page 661).

command line option!–json

`--json`
New in version 2.8.0.

Returns output for `mongostat` (page 657) in *JSON* format.

`<sleeptime>`
The final argument is the length of time, in seconds, that `mongostat` (page 657) waits in between calls. By default `mongostat` (page 657) returns one call every second.

`mongostat` (page 657) returns values that reflect the operations over a 1 second period. For values of `<sleeptime>` greater than 1, `mongostat` (page 657) averages data to reflect average operations per second.

Fields

`mongostat` (page 657) returns values that reflect the operations over a 1 second period. When **mongostat <sleeptime>** has a value greater than 1, `mongostat` (page 657) averages the statistics to reflect average operations per

second.

mongostat (page 657) outputs the following fields:

inserts
The number of objects inserted into the database per second. If followed by an asterisk (e.g. *), the datum refers to a replicated operation.

query
The number of query operations per second.

update
The number of update operations per second.

delete
The number of delete operations per second.

getmore
The number of get more (i.e. cursor batch) operations per second.

command
The number of commands per second. On *slave* and *secondary* systems, mongostat (page 657) presents two values separated by a pipe character (e.g. |), in the form of local|replicated commands.

flushes
The number of *fsync* operations per second.

mapped
The total amount of data mapped in megabytes. This is the total data size at the time of the last mongostat (page 657) call.

size
The amount of virtual memory in megabytes used by the process at the time of the last mongostat (page 657) call.

non-mapped
The total amount of virtual memory excluding all mapped memory at the time of the last mongostat (page 657) call.

res
The amount of resident memory in megabytes used by the process at the time of the last mongostat (page 657) call.

faults
Changed in version 2.1.

The number of page faults per second.

Before version 2.1 this value was only provided for MongoDB instances running on Linux hosts.

locked
The percent of time in a global write lock.

Changed in version 2.2: The locked db field replaces the locked % field to more appropriate data regarding the database specific locks in version 2.2.

locked db
New in version 2.2.

The percent of time in the per-database context-specific lock. mongostat (page 657) will report the database that has spent the most time since the last mongostat (page 657) call with a write lock.

This value represents the amount of time that the listed database spent in a locked state *combined* with the time that the mongod (page 583) spent in the global lock. Because of this, and the sampling method, you may see some values greater than 100%.

idx miss
The percent of index access attempts that required a page fault to load a btree node. This is a sampled value.

qr
The length of the queue of clients waiting to read data from the MongoDB instance.

qw
The length of the queue of clients waiting to write data from the MongoDB instance.

ar
The number of active clients performing read operations.

aw
The number of active clients performing write operations.

netIn
The amount of network traffic, in *bytes*, received by the MongoDB instance.

This includes traffic from mongostat (page 657) itself.

netOut
The amount of network traffic, in *bytes*, sent by the MongoDB instance.

This includes traffic from mongostat (page 657) itself.

conn
The total number of open connections.

set
The name, if applicable, of the replica set.

repl
The replication status of the member.

Value	Replication Type
M	*master*
SEC	*secondary*
REC	recovering
UNK	unknown
SLV	*slave*
RTR	mongos process ("router")

Usage

In the first example, mongostat (page 657) will return data every second for 20 seconds. mongostat (page 657) collects data from the mongod (page 583) instance running on the localhost interface on port 27017. All of the following invocations produce identical behavior:

```
mongostat --rowcount 20 1
mongostat --rowcount 20
mongostat -n 20 1
mongostat -n 20
```

In the next example, mongostat (page 657) returns data every 5 minutes (or 300 seconds) for as long as the program runs. mongostat (page 657) collects data from the mongod (page 583) instance running on the localhost interface on port 27017. Both of the following invocations produce identical behavior.

```
mongostat --rowcount 0 300
mongostat -n 0 300
mongostat 300
```

In the following example, mongostat (page 657) returns data every 5 minutes for an hour (12 times.) mongostat (page 657) collects data from the mongod (page 583) instance running on the localhost interface on port 27017. Both of the following invocations produce identical behavior.

```
mongostat --rowcount 12 300
mongostat -n 12 300
```

In many cases, using the *--discover* will help provide a more complete snapshot of the state of an entire group of machines. If a mongos (page 601) process connected to a *sharded cluster* is running on port 27017 of the local machine, you can use the following form to return statistics from all members of the cluster:

```
mongostat --discover
```

mongotop

Synopsis

mongotop (page 664) provides a method to track the amount of time a MongoDB instance spends reading and writing data. mongotop (page 664) provides statistics on a per-collection level. By default, mongotop (page 664) returns values every second.

Important: In order to connect to a mongod (page 583) that enforces authorization with the *--auth* option, the *--username* and *--password* options must be used, and the user specified must have the serverStatus and top privileges.

The most appropriate built-in role that has these privileges is clusterMonitor.

See also:

For more information about monitoring MongoDB, see http://docs.mongodb.org/manual/administration/monitori

For additional background on various other MongoDB status outputs see:

- *serverStatus* (page 366)
- *replSetGetStatus* (page 296)
- *dbStats* (page 352)
- *collStats* (page 348)

For an additional utility that provides MongoDB metrics see *mongostat* (page 657).

Options

mongotop

mongotop

command line option!–help

--help
Returns information on the options and use of mongotop (page 664).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

command line option!–quiet

--quiet
Runs the `mongotop` (page 664) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

- output from *database commands*
- replication activity
- connection accepted events
- connection closed events

command line option!–version

--version
Returns the `mongotop` (page 664) release number.

command line option!–host <hostname><:port>, -h <hostname><:port>

--host `<hostname><:port>`, **-h** `<hostname><:port>`
Default: localhost:27017

Specifies a resolvable hostname for the `mongod` (page 583) to which to connect. By default, the `mongotop` (page 664) attempts to connect to a MongoDB instance running on the localhost on port number `27017`.

To connect to a replica set, specify the `replica set name` and a seed list of set members. Use the following form:

```
<replSetName>/<hostname1><:port>,<hostname2><:port>,<...>
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

Changed in version 2.8.0: If you use IPv6 and use the `<address>:<port >` format, you must enclose the portion of an address and port combination in brackets (e.g. `[<address>]`).

command line option!–port <port>

--port `<port>`
Default: 27017

Specifies the TCP port on which the MongoDB instance listens for client connections.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the `mongotop` (page 664) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.6.

Enables connection to a `mongod` (page 583) or `mongos` (page 601) that has SSL support enabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.6.

Specifies the .pem file that contains the root certificate chain from the Certificate Authority. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

Warning: If the mongo (page 610) shell or any other tool that connects to mongos (page 601) or mongod (page 583) is run without *--sslCAFile*, it will not attempt to validate server certificates. This results in vulnerability to expired mongod (page 583) and mongos (page 601) certificates as well as to foreign processes posing as valid mongod (page 583) or mongos (page 601) instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.6.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths.

This option is required when using the *--ssl* (page 665) option to connect to a mongod (page 583) or mongos (page 601) that has CAFile enabled *without* allowConnectionsWithoutCertificates.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.6.

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 596)). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongotop (page 664) will redact the password from all logging and reporting output.

If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the mongotop (page 664) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCRLFile <filename>

--sslCRLFile <filename>
New in version 2.6.

Specifies the .pem file that contains the Certificate Revocation List. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for server certificates and allows the use of invalid certificates. When using the allowInvalidCertificates setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates. Allows `mongotop` (page 664) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.6.

Directs the `mongotop` (page 664) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[35]. See `http://docs.mongodb.org/manual/tutorial/configure-fips` for more information.

---

command line option!–username <username>, -u <username>

--username `<username>`**, -u** `<username>`
Specifies a username with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the `--password` and `--authenticationDatabase` options.

command line option!–password <password>, -p <password>

--password `<password>`**, -p** `<password>`
Specifies a password with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the `--username` and `--authenticationDatabase` options.

If you do not specify an argument for *--password* (page 667), `mongotop` (page 664) will prompt interactively for a password on the console.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase `<dbname>`
New in version 2.4.

Specifies the database that holds the user's credentials.

If you do not specify an authentication database, `mongotop` (page 664) assumes that the database specified as the argument to *--authenticationDatabase* (page 667) holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism `<name>`
Default: MONGODB-CR

New in version 2.4.

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Specifies the authentication mechanism the `mongotop` (page 664) instance uses to authenticate to the `mongod` (page 583) or `mongos` (page 601).

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

Value	Description
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB SSL certificate authentication.
PLAIN	External authentication using LDAP. You can also use `PLAIN` for authenticating in-database users. `PLAIN` transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[36].
GSSAPI	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[37].

command line option!–gssapiServiceName

--gssapiServiceName
New in version 2.6.

Specify the name of the service using `GSSAPI/Kerberos`. Only required if the service does not use the default name of `mongodb`.

This option is available only in MongoDB Enterprise.

command line option!–gssapiHostName

--gssapiHostName
New in version 2.6.

Specify the hostname of a service using `GSSAPI/Kerberos`. *Only* required if the hostname of a machine does not match the hostname resolved by DNS.

This option is available only in MongoDB Enterprise.

command line option!–locks

--locks
Toggles the mode of `mongotop` (page 664) to report on use of per-database *locks* (page 368). These data are useful for measuring concurrent operations and lock percentage.

command line option!–rowcount int, -n int

--rowcount `int`, **-n** `int`
Number of lines of data that `mongotop` (page 664) should print. "0 for indefinite"

command line option!–json

--json
New in version 2.8.0.

Returns output for `mongotop` (page 664) in *JSON* format.

<sleeptime>
The final argument is the length of time, in seconds, that `mongotop` (page 664) waits in between calls. By default `mongotop` (page 664) returns data every second.

Fields

`mongotop` (page 664) returns time values specified in milliseconds (ms.)

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

mongotop (page 664) only reports active namespaces or databases, depending on the --locks (page 668) option. If you don't see a database or collection, it has received no recent activity. You can issue a simple operation in the mongo (page 610) shell to generate activity to affect the output of mongotop (page 664).

mongotop.**ns**
Contains the database namespace, which combines the database name and collection.

Changed in version 2.2: If you use the --locks (page 668), the ns (page 669) field does not appear in the mongotop (page 664) output.

mongotop.**db**
New in version 2.2.

Contains the name of the database. The database named . refers to the global lock, rather than a specific database.

This field does not appear unless you have invoked mongotop (page 664) with the --locks (page 668) option.

mongotop.**total**
Provides the total amount of time that this mongod (page 583) spent operating on this namespace.

mongotop.**read**
Provides the amount of time that this mongod (page 583) spent performing read operations on this namespace.

mongotop.**write**
Provides the amount of time that this mongod (page 583) spent performing write operations on this namespace.

mongotop.**<timestamp>**
Provides a time stamp for the returned data.

Use

By default mongotop (page 664) connects to the MongoDB instance running on the localhost port 27017. However, mongotop (page 664) can optionally connect to remote mongod (page 583) instances. See the *mongotop options* (page 664) for more information.

To force mongotop (page 664) to return less frequently specify a number, in seconds at the end of the command. In this example, mongotop (page 664) will return every 15 seconds.

```
mongotop 15
```

This command produces the following output:

```
connected to: 127.0.0.1

                    ns       total        read       write           2012-08-13T15:45:40
test.system.namespaces         0ms         0ms         0ms
  local.system.replset         0ms         0ms         0ms
  local.system.indexes         0ms         0ms         0ms
  admin.system.indexes         0ms         0ms         0ms
                admin.         0ms         0ms         0ms

                    ns       total        read       write           2012-08-13T15:45:55
test.system.namespaces         0ms         0ms         0ms
  local.system.replset         0ms         0ms         0ms
  local.system.indexes         0ms         0ms         0ms
  admin.system.indexes         0ms         0ms         0ms
                admin.         0ms         0ms         0ms
```

To return a mongotop (page 664) report every 5 minutes, use the following command:

```
mongotop 300
```

To report the use of per-database locks, use *mongotop --locks*, which produces the following output:

```
$ mongotop --locks
connected to: 127.0.0.1

                   db       total        read       write          2012-08-13T16:33:34
                local          0ms         0ms         0ms
                admin          0ms         0ms         0ms
                    .          0ms         0ms         0ms
```

mongosniff

Synopsis

mongosniff (page 670) provides a low-level operation tracing/sniffing view into database activity in real time. Think of mongosniff (page 670) as a MongoDB-specific analogue of tcpdump for TCP/IP network traffic. Typically, mongosniff (page 670) is most frequently used in driver development.

Note: mongosniff (page 670) requires libpcap and is only available for Unix-like systems.

As an alternative to mongosniff (page 670), Wireshark, a popular network sniffing tool is capable of inspecting and parsing the MongoDB wire protocol.

Options

mongosniff

mongosniff

command line option!–help

--help
Returns information on the options and use of mongosniff (page 670).

command line option!–forward <host><:port>

--forward <host><:port>
Declares a host to forward all parsed requests that the mongosniff (page 670) intercepts to another mongod (page 583) instance and issue those operations on that database instance.

Specify the target host name and port in the <host><:port> format.

To connect to a replica set, specify the replica set name and a seed list of set members. Use the following form:

```
<replSetName>/<hostname1><:port>,<hostname2><:port>,<...>
```

command line option!–source <NET [interface]>, <FILE [filename]>, <DIAGLOG [filename]>

--source <NET [interface]>
Specifies source material to inspect. Use --source NET [interface] to inspect traffic from a network interface (e.g. eth0 or lo.) Use --source FILE [filename] to read captured packets in *pcap* format.

You may use the `--source DIAGLOG [filename]` option to read the output files produced by the *`--diaglog`* option.

command line option!–objcheck

`--objcheck`
Displays invalid BSON objects only and nothing else. Use this option for troubleshooting driver development. This option has some performance impact on the performance of `mongosniff` (page 670).

`<port>`
Specifies alternate ports to sniff for traffic. By default, `mongosniff` (page 670) watches for MongoDB traffic on port `27017`. Append multiple port numbers to the end of `mongosniff` (page 670) to monitor traffic on multiple ports.

Use

Use the following command to connect to a `mongod` (page 583) or `mongos` (page 601) running on port 27017 *and* 27018 on the localhost interface:

```
mongosniff --source NET lo 27017 27018
```

Use the following command to only log invalid *BSON* objects for the `mongod` (page 583) or `mongos` (page 601) running on the localhost interface and port 27018, for driver development and troubleshooting:

```
mongosniff --objcheck --source NET lo 27018
```

Build `mongosniff`

To build `mongosniff` yourself, Linux users can use the following procedure:

1. Obtain prerequisites using your operating systems package management software. Dependencies include:
   - `libpcap` - to capture network packets.
   - `git` - to download the MongoDB source code.
   - `scons` and a C++ compiler - to build `mongosniff` (page 670).
2. Download a copy of the MongoDB source code using `git`:

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

3. Issue the following sequence of commands to change to the `mongo/` directory and build `mongosniff` (page 670):

   ```
   cd mongo
   scons mongosniff
   ```

Note: If you run `scons mongosniff` before installing `libpcap` you must run `scons clean` before you can build `mongosniff` (page 670).

`mongoperf`

Synopsis

`mongoperf` (page 672) is a utility to check disk I/O performance independently of MongoDB.

It times tests of random disk I/O and presents the results. You can use `mongoperf` (page 672) for any case apart from MongoDB. The `mmf` (page 673) `true` mode is completely generic. In that mode it is somewhat analogous to tools such as bonnie++[38] (albeit mongoperf is simpler).

Specify options to `mongoperf` (page 672) using a JavaScript document.

See also:

- bonnie[39]
- bonnie++[40]
- Output from an example run[41]
- Checking Disk Performance with the mongoperf Utility[42]

Options

mongoperf

mongoperf

command line option!–help, -h

--help, -h
Returns information on the options and use of `mongoperf` (page 672).

<jsonconfig>
`mongoperf` (page 672) accepts configuration options in the form of a file that holds a *JSON* document. You must stream the content of this file into `mongoperf` (page 672), as in the following operation:

```
mongoperf < config
```

In this example `config` is the name of a file that holds a JSON document that resembles the following example:

```
{
  nThreads:<n>,
  fileSizeMB:<n>,
  sleepMicros:<n>,
  mmf:<bool>,
  r:<bool>,
  w:<bool>,
  recSizeKB:<n>,
  syncDelay:<n>
}
```

See the *Configuration Fields* (page 672) section for documentation of each of these fields.

Configuration Fields

`mongoperf.nThreads`
Type: Integer.

Default: 1

[38] http://sourceforge.net/projects/bonnie/
[39] http://www.textuality.com/bonnie/
[40] http://sourceforge.net/projects/bonnie/
[41] https://gist.github.com/1694664
[42] http://blog.mongodb.org/post/40769806981/checking-disk-performance-with-the-mongoperf-utility

Defines the number of threads mongoperf (page 672) will use in the test. To saturate your system's storage system you will need multiple threads. Consider setting nThreads (page 672) to 16.

mongoperf.**fileSizeMB**
Type: Integer.

Default: 1 megabyte (i.e. 1024^2 bytes)

Test file size.

mongoperf.**sleepMicros**
Type: Integer.

Default: 0

mongoperf (page 672) will pause for the number of specified sleepMicros (page 673) divided by the nThreads (page 672) between each operation.

mongoperf.**mmf**
Type: Boolean.

Default: false

Set mmf (page 673) to true to use memory mapped files for the tests.

Generally:

- when mmf (page 673) is false, mongoperf (page 672) tests direct, physical, I/O, without caching. Use a large file size to test heavy random I/O load and to avoid I/O coalescing.
- when mmf (page 673) is true, mongoperf (page 672) runs tests of the caching system, and can use normal file system cache. Use mmf (page 673) in this mode to test file system cache behavior with memory mapped files.

mongoperf.**r**
Type: Boolean.

Default: false

Set r (page 673) to true to perform reads as part of the tests.

Either r (page 673) or w (page 673) must be true.

mongoperf.**w**
Type: Boolean.

Default: false

Set w (page 673) to true to perform writes as part of the tests.

Either r (page 673) or w (page 673) must be true.

mongoperf.**recSizeKB**
New in version 2.4.

Type: Integer.

Default: 4 kb

The size of each write operation.

mongoperf.**syncDelay**
Type: Integer.

Default: 0

Seconds between disk flushes. `mongoperf.syncDelay` (page 673) is similar to *`--syncdelay`* for `mongod` (page 583).

The `syncDelay` (page 673) controls how frequently `mongoperf` (page 672) performs an asynchronous disk flush of the memory mapped file used for testing. By default, `mongod` (page 583) performs this operation every 60 seconds. Use `syncDelay` (page 673) to test basic system performance of this type of operation.

Only use `syncDelay` (page 673) in conjunction with `mmf` (page 673) set to `true`.

The default value of `0` disables this.

Use

```
mongoperf < jsonconfigfile
```

Replace `jsonconfigfile` with the path to the `mongoperf` (page 672) configuration. You may also invoke `mongoperf` (page 672) in the following form:

```
echo "{nThreads:16,fileSizeMB:10000,r:true,w:true}" | mongoperf
```

In this operation:

- `mongoperf` (page 672) tests direct physical random read and write io's, using 16 concurrent reader threads.
- `mongoperf` (page 672) uses a 10 gigabyte test file.

Consider using `iostat`, as invoked in the following example to monitor I/O performance during the test.

```
iostat -xtm 1
```

4.1.6 GridFS

`mongofiles` (page 674) provides a command-line interact to a MongoDB *GridFS* storage system.

mongofiles

mongofiles

Synopsis

The `mongofiles` (page 674) utility makes it possible to manipulate files stored in your MongoDB instance in *GridFS* objects from the command line. It is particularly useful as it provides an interface between objects stored in your file system and GridFS.

All `mongofiles` (page 674) commands have the following form:

```
mongofiles <options> <commands> <filename>
```

The components of the `mongofiles` (page 674) command are:

1. *Options* (page 675). You may use one or more of these options to control the behavior of `mongofiles` (page 674).
2. *Commands* (page 679). Use one of these commands to determine the action of `mongofiles` (page 674).
3. A filename which is either: the name of a file on your local's file system, or a GridFS object.

mongofiles (page 674), like mongodump (page 622), mongoexport (page 649), mongoimport (page 642), and mongorestore (page 628), can access data stored in a MongoDB data directory without requiring a running mongod (page 583) instance, if no other mongod (page 583) is running.

Important: For *replica sets*, mongofiles (page 674) can only read from the set's '*primary*.

Options

Changed in version 2.8.0: mongofiles (page 674) removed the --dbpath as well as related --directoryperdb and --journal options. You must use mongofiles (page 674) while connected to a mongod (page 583) instance.

mongofiles

command line option!–help

--help
Returns information on the options and use of mongofiles (page 674).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in log files. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

command line option!–quiet

--quiet
Runs the mongofiles (page 674) in a quiet mode that attempts to limit the amount of output.

This option suppresses:

- output from *database commands*
- replication activity
- connection accepted events
- connection closed events

command line option!–version

--version
Returns the mongofiles (page 674) release number.

command line option!–host <hostname><:port>

--host <hostname><:port>
Specifies a resolvable hostname for the mongod (page 583) that holds your GridFS system. By default mongofiles (page 674) attempts to connect to a MongoDB process running on the localhost port number 27017.

Optionally, specify a port number to connect a MongoDB instance running on a port other than 27017.

command line option!–port <port>

--port <port>
Default: 27017

Specifies the TCP port on which the MongoDB instance listens for client connections.

command line option!–ipv6

--ipv6
Enables IPv6 support and allows the mongofiles (page 674) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.6.

Enables connection to a mongod (page 583) or mongos (page 601) that has SSL support enabled.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.6.

Specifies the .pem file that contains the root certificate chain from the Certificate Authority. Specify the file name of the .pem file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

> **Warning:** If the mongo (page 610) shell or any other tool that connects to mongos (page 601) or mongod (page 583) is run without *--sslCAFile*, it will not attempt to validate server certificates. This results in vulnerability to expired mongod (page 583) and mongos (page 601) certificates as well as to foreign processes posing as valid mongod (page 583) or mongos (page 601) instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.6.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths.

This option is required when using the *--ssl* (page 665) option to connect to a mongod (page 583) or mongos (page 601) that has CAFile enabled *without* allowConnectionsWithoutCertificates.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.6.

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 596)). Use the *--sslPEMKeyPassword* (page 597) option only if the certificate-key file is encrypted. In all cases, the mongofiles (page 674) will redact the password from all logging and reporting output.

If the private key in the PEM file is encrypted and you do not specify the *--sslPEMKeyPassword* (page 597) option, the mongofiles (page 674) will prompt for a passphrase. See *ssl-certificate-password*.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see http://docs.mongodb.org/manual/tutorial/configure-ssl.

command line option!–sslCRLFile <filename>

--sslCRLFile <filename>
New in version 2.6.

Specifies the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates
New in version 2.6.

Bypasses the validation checks for server certificates and allows the use of invalid certificates. When using the `allowInvalidCertificates` setting, MongoDB logs as a warning the use of the invalid certificate.

The default distribution of MongoDB does not contain support for SSL. For more information on MongoDB and SSL, see `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames
New in version 2.8.

Disables the validation of the hostnames in SSL certificates. Allows `mongofiles` (page 674) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.6.

Directs the `mongofiles` (page 674) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 599) option.

Note: FIPS Compatible SSL is available only in MongoDB Enterprise[43]. See `http://docs.mongodb.org/manual/tutorial/configure-fips` for more information.

---

command line option!–username <username>, -u <username>

--username <username>, **-u** <username>
Specifies a username with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the `--password` and `--authenticationDatabase` options.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password with which to authenticate to a MongoDB database that uses authentication. Use in conjunction with the `--username` and `--authenticationDatabase` options.

If you do not specify an argument for *--password* (page 667), `mongofiles` (page 674) will prompt interactively for a password on the console.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's credentials.

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

If you do not specify an authentication database, mongofiles (page 674) assumes that the database specified as the argument to --authenticationDatabase (page 667) holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
Default: MONGODB-CR

New in version 2.4.

Changed in version 2.6: Added support for the PLAIN and MONGODB-X509 authentication mechanisms.

Specifies the authentication mechanism the mongofiles (page 674) instance uses to authenticate to the mongod (page 583) or mongos (page 601).

Value	Description
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB SSL certificate authentication.
PLAIN	External authentication using LDAP. You can also use PLAIN for authenticating in-database users. PLAIN transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[44].
GSSAPI	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[45].

command line option!–gssapiServiceName

--gssapiServiceName
New in version 2.6.

Specify the name of the service using GSSAPI/Kerberos. Only required if the service does not use the default name of mongodb.

This option is available only in MongoDB Enterprise.

command line option!–gssapiHostName

--gssapiHostName
New in version 2.6.

Specify the hostname of a service using GSSAPI/Kerberos. *Only* required if the hostname of a machine does not match the hostname resolved by DNS.

This option is available only in MongoDB Enterprise.

command line option!–db <database>, -d <database>

--db <database>, **-d** <database>
Specifies the name of the database on which to run the mongofiles (page 674).

command line option!–collection <collection>, -c <collection>

--collection <collection>, **-c** <collection>
This option has no use in this context and a future release may remove it. See SERVER-4931[46] for more information.

command line option!–local <filename>, -l <filename>

[44] http://www.mongodb.com/products/mongodb-enterprise
[45] http://www.mongodb.com/products/mongodb-enterprise
[46] https://jira.mongodb.org/browse/SERVER-4931

--local <filename>, **-l** <filename>
Specifies the local filesystem name of a file for get and put operations.

In the **mongofiles put** and **mongofiles get** commands, the required <filename> modifier refers to the name the object will have in GridFS. mongofiles (page 674) assumes that this reflects the file's name on the local file system. This setting overrides this default.

command line option!–type <MIME>

--type <MIME>
Provides the ability to specify a *MIME* type to describe the file inserted into GridFS storage. mongofiles (page 674) omits this option in the default operation.

Use only with **mongofiles put** operations.

command line option!–replace, -r

--replace, -r
Alters the behavior of **mongofiles put** to replace existing GridFS objects with the specified local file, rather than adding an additional object with the same name.

In the default operation, files will not be overwritten by a **mongofiles put** option.

command line option!–prefix string

--prefix string
Default: fs

GridFS prefix to use.

command line option!–writeConcern <document>

--writeConcern <document>
Default: majority

Specifies the *write concern* for each write operation that mongofiles (page 674) writes to the target database.

Specify the write concern as a document with *w options*.

Commands

list <prefix>
Lists the files in the GridFS store. The characters specified after list (e.g. <prefix>) optionally limit the list of returned items to files that begin with that string of characters.

search <string>
Lists the files in the GridFS store with names that match any portion of <string>.

put <filename>
Copy the specified file from the local file system into GridFS storage.

Here, <filename> refers to the name the object will have in GridFS, and mongofiles (page 674) assumes that this reflects the name the file has on the local file system. If the local filename is different use the *mongofiles --local* option.

get <filename>
Copy the specified file from GridFS storage to the local file system.

Here, <filename> refers to the name the object will have in GridFS, and mongofiles (page 674) assumes that this reflects the name the file has on the local file system. If the local filename is different use the *mongofiles --local* option.

delete <filename>
Delete the specified file from GridFS storage.

Examples

To return a list of all files in a *GridFS* collection in the `records` database, use the following invocation at the system shell:

```
mongofiles -d records list
```

This `mongofiles` (page 674) instance will connect to the `mongod` (page 583) instance running on the `27017` localhost interface to specify the same operation on a different port or hostname, and issue a command that resembles one of the following:

```
mongofiles --port 37017 -d records list
mongofiles --hostname db1.example.net -d records list
mongofiles --hostname db1.example.net --port 37017 -d records list
```

Modify any of the following commands as needed if you're connecting the `mongod` (page 583) instances on different ports or hosts.

To upload a file named `32-corinth.lp` to the GridFS collection in the `records` database, you can use the following command:

```
mongofiles -d records put 32-corinth.lp
```

To delete the `32-corinth.lp` file from this GridFS collection in the `records` database, you can use the following command:

```
mongofiles -d records delete 32-corinth.lp
```

To search for files in the GridFS collection in the `records` database that have the string `corinth` in their names, you can use following command:

```
mongofiles -d records search corinth
```

To list all files in the GridFS collection in the `records` database that begin with the string `32`, you can use the following command:

```
mongofiles -d records list 32
```

To fetch the file from the GridFS collection in the `records` database named `32-corinth.lp`, you can use the following command:

```
mongofiles -d records get 32-corinth.lp
```

CHAPTER 5

Internal Metadata

5.1 Config Database

The `config` database supports *sharded cluster* operation. See the `http://docs.mongodb.org/manual/sharding` section of this manual for full documentation of sharded clusters.

Important: Consider the schema of the `config` database *internal* and may change between releases of MongoDB. The `config` database is not a dependable API, and users should not write data to the `config` database in the course of normal operation or maintenance.

Warning: Modification of the `config` database on a functioning system may lead to instability or inconsistent data sets. If you must modify the `config` database, use `mongodump` (page 622) to create a full backup of the `config` database.

To access the `config` database, connect to a `mongos` (page 601) instance in a sharded cluster, and use the following helper:

```
use config
```

You can return a list of the collections, with the following helper:

```
show collections
```

5.1.1 Collections

config

config.**changelog**

Internal MongoDB Metadata

The `config` (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `changelog` (page 681) collection stores a document for each change to the metadata of a sharded collection.

Example

The following example displays a single record of a chunk split from a changelog (page 681) collection:

```
{
 "_id" : "<hostname>-<timestamp>-<increment>",
 "server" : "<hostname><:port>",
 "clientAddr" : "127.0.0.1:63381",
 "time" : ISODate("2012-12-11T14:09:21.039Z"),
 "what" : "split",
 "ns" : "<database>.<collection>",
 "details" : {
    "before" : {
       "min" : {
          "<database>" : { $minKey : 1 }
       },
       "max" : {
          "<database>" : { $maxKey : 1 }
       },
       "lastmod" : Timestamp(1000, 0),
       "lastmodEpoch" : ObjectId("000000000000000000000000")
    },
    "left" : {
       "min" : {
          "<database>" : { $minKey : 1 }
       },
       "max" : {
          "<database>" : "<value>"
       },
       "lastmod" : Timestamp(1000, 1),
       "lastmodEpoch" : ObjectId(<...>)
    },
    "right" : {
       "min" : {
          "<database>" : "<value>"
       },
       "max" : {
          "<database>" : { $maxKey : 1 }
       },
       "lastmod" : Timestamp(1000, 2),
       "lastmodEpoch" : ObjectId("<...>")
    }
 }
}
```

Each document in the changelog (page 681) collection contains the following fields:

`config.changelog._id`
The value of `changelog._id` is: `<hostname>-<timestamp>-<increment>`.

`config.changelog.server`
The hostname of the server that holds this data.

`config.changelog.clientAddr`
A string that holds the address of the client, a mongos (page 601) instance that initiates this change.

`config.changelog.time`
A *ISODate* timestamp that reflects when the change occurred.

`config.changelog.what`
Reflects the type of change recorded. Possible values are:

- dropCollection
- dropCollection.start
- dropDatabase
- dropDatabase.start
- moveChunk.start
- moveChunk.commit
- split
- multi-split

`config.changelog.ns`
: Namespace where the change occurred.

`config.changelog.details`
: A *document* that contains additional details regarding the change. The structure of the `details` (page 683) document depends on the type of change.

config.chunks

Internal MongoDB Metadata

The `config` (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `chunks` (page 683) collection stores a document for each chunk in the cluster. Consider the following example of a document for a chunk named `records.pets-animal_\"cat\"`:

```
{
   "_id" : "mydb.foo-a_\"cat\"",
   "lastmod" : Timestamp(1000, 3),
   "lastmodEpoch" : ObjectId("5078407bd58b175c5c225fdc"),
   "ns" : "mydb.foo",
   "min" : {
         "animal" : "cat"
   },
   "max" : {
         "animal" : "dog"
   },
   "shard" : "shard0004"
}
```

These documents store the range of values for the shard key that describe the chunk in the `min` and `max` fields. Additionally the `shard` field identifies the shard in the cluster that "owns" the chunk.

config.collections

Internal MongoDB Metadata

The `config` (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `collections` (page 683) collection stores a document for each sharded collection in the cluster. Given a collection named `pets` in the `records` database, a document in the `collections` (page 683) collection would resemble the following:

```
{
   "_id" : "records.pets",
   "lastmod" : ISODate("1970-01-16T15:00:58.107Z"),
   "dropped" : false,
   "key" : {
         "a" : 1
   },
   "unique" : false,
   "lastmodEpoch" : ObjectId("5078407bd58b175c5c225fdc")
}
```

config.databases

Internal MongoDB Metadata

The config (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The databases (page 684) collection stores a document for each database in the cluster, and tracks if the database has sharding enabled. databases (page 684) represents each database in a distinct document. When a databases have sharding enabled, the primary field holds the name of the *primary shard*.

```
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "mydb", "partitioned" : true, "primary" : "shard0000" }
```

config.lockpings

Internal MongoDB Metadata

The config (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The lockpings (page 684) collection keeps track of the active components in the sharded cluster. Given a cluster with a mongos (page 601) running on example.com:30000, the document in the lockpings (page 684) collection would resemble:

```
{ "_id" : "example.com:30000:1350047994:16807", "ping" : ISODate("2012-10-12T18:32:54.892Z") }
```

config.locks

Internal MongoDB Metadata

The config (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The locks (page 684) collection stores a distributed lock. This ensures that only one mongos (page 601) instance can perform administrative tasks on the cluster at once. The mongos (page 601) acting as *balancer* takes a lock by inserting a document resembling the following into the locks collection.

```
{
    "_id" : "balancer",
    "process" : "example.net:40000:1350402818:16807",
    "state" : 2,
    "ts" : ObjectId("507daeedf40e1879df62e5f3"),
    "when" : ISODate("2012-10-16T19:01:01.593Z"),
```

```
    "who" : "example.net:40000:1350402818:16807:Balancer:282475249",
    "why" : "doing balance round"
}
```

If a mongos (page 601) holds the balancer lock, the state field has a value of 2, which means that balancer is active. The when field indicates when the balancer began the current operation.

Changed in version 2.0: The value of the state field was 1 before MongoDB 2.0.

config.mongos

Internal MongoDB Metadata

The config (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The mongos (page 685) collection stores a document for each mongos (page 601) instance affiliated with the cluster. mongos (page 601) instances send pings to all members of the cluster every 30 seconds so the cluster can verify that the mongos (page 601) is active. The ping field shows the time of the last ping, while the up field reports the uptime of the mongos (page 601) as of the last ping. The cluster maintains this collection for reporting purposes.

The following document shows the status of the mongos (page 601) running on example.com:30000.

```
{ "_id" : "example.com:30000", "ping" : ISODate("2012-10-12T17:08:13.538Z"), "up" : 13699, "wait
```

config.settings

Internal MongoDB Metadata

The config (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The settings (page 685) collection holds the following sharding configuration settings:

- Chunk size. To change chunk size, see http://docs.mongodb.org/manual/tutorial/modify-chunk-size
- Balancer status. To change status, see *sharding-balancing-disable-temporarily*.

The following is an example settings collection:

```
{ "_id" : "chunksize", "value" : 64 }
{ "_id" : "balancer", "stopped" : false }
```

config.shards

Internal MongoDB Metadata

The config (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The shards (page 685) collection represents each shard in the cluster in a separate document. If the shard is a replica set, the host field displays the name of the replica set, then a slash, then the hostname, as in the following example:

```
{ "_id" : "shard0000", "host" : "shard1/localhost:30000" }
```

If the shard has *tags* assigned, this document has a `tags` field, that holds an array of the tags, as in the following example:

```
{ "_id" : "shard0001", "host" : "localhost:30001", "tags": [ "NYC" ] }
```

config.tags

Internal MongoDB Metadata

The `config` (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `tags` (page 686) collection holds documents for each tagged shard key range in the cluster. The documents in the `tags` (page 686) collection resemble the following:

```
{
    "_id" : { "ns" : "records.users", "min" : { "zipcode" : "10001" } },
    "ns" : "records.users",
    "min" : { "zipcode" : "10001" },
    "max" : { "zipcode" : "10281" },
    "tag" : "NYC"
}
```

config.version

Internal MongoDB Metadata

The `config` (page 681) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `version` (page 686) collection holds the current metadata version number. This collection contains only one document:

To access the `version` (page 686) collection you must use the `db.getCollection()` (page 114) method. For example, to display the collection's document:

```
mongos> db.getCollection("version").find()
{ "_id" : 1, "version" : 3 }
```

Note: Like all databases in MongoDB, the `config` database contains a `system.indexes` (page 689) collection contains metadata for all indexes in the database for information on indexes, see `http://docs.mongodb.org/manual/indexes`.

5.2 The `local` Database

5.2.1 Overview

Every `mongod` (page 583) instance has its own `local` database, which stores data used in the replication process, and other instance-specific data. The `local` database is invisible to replication: collections in the `local` database are not replicated.

In replication, the `local` database store stores internal replication data for each member of a *replica set*. The `local` stores the following collections:

Changed in version 2.4: When running with authentication (i.e. `authorization`), authenticating to the `local` database is **not** equivalent to authenticating to the `admin` database. In previous versions, authenticating to the `local` database provided access to all databases.

5.2.2 Collection on all `mongod` Instances

`local.startup_log`

On startup, each `mongod` (page 583) instance inserts a document into `startup_log` (page 687) with diagnostic information about the `mongod` (page 583) instance itself and host information. `startup_log` (page 687) is a capped collection. This information is primarily useful for diagnostic purposes.

Example

Consider the following prototype of a document from the `startup_log` (page 687) collection:

```
{
  "_id" : "<string>",
  "hostname" : "<string>",
  "startTime" : ISODate("<date>"),
  "startTimeLocal" : "<string>",
  "cmdLine" : {
        "dbpath" : "<path>",
        "<option>" : <value>
  },
  "pid" : <number>,
  "buildinfo" : {
        "version" : "<string>",
        "gitVersion" : "<string>",
        "sysInfo" : "<string>",
        "loaderFlags" : "<string>",
        "compilerFlags" : "<string>",
        "allocator" : "<string>",
        "versionArray" : [ <num>, <num>, <...> ],
        "javascriptEngine" : "<string>",
        "bits" : <number>,
        "debug" : <boolean>,
        "maxBsonObjectSize" : <number>
  }
}
```

Documents in the `startup_log` (page 687) collection contain the following fields:

`local.startup_log._id`
Includes the system hostname and a millisecond epoch value.

`local.startup_log.hostname`
The system's hostname.

`local.startup_log.startTime`
A UTC *ISODate* value that reflects when the server started.

`local.startup_log.startTimeLocal`
A string that reports the `startTime` (page 687) in the system's local time zone.

`local.startup_log.cmdLine`
A sub-document that reports the `mongod` (page 583) runtime options and their values.

`local.startup_log.pid`
The process identifier for this process.

`local.startup_log.`**`buildinfo`**
A sub-document that reports information about the build environment and settings used to compile this `mongod` (page 583). This is the same output as `buildInfo` (page 346). See `buildInfo` (page 347).

5.2.3 Collections on Replica Set Members

`local.system.`**`replset`**
`local.system.replset` (page 688) holds the replica set's configuration object as its single document. To view the object's configuration information, issue `rs.conf()` (page 174) from the `mongo` (page 610) shell. You can also query this collection directly.

`local.oplog.`**`rs`**
`local.oplog.rs` (page 688) is the capped collection that holds the *oplog*. You set its size at creation using the `oplogSizeMB` setting. To resize the oplog after replica set initiation, use the `http://docs.mongodb.org/manual/tutorial/change-oplog-size` procedure. For additional information, see the *replica-set-oplog-sizing* section.

`local.replset.`**`minvalid`**
This contains an object used internally by replica sets to track replication status.

`local.`**`slaves`**
Deprecated since version 2.8.0.

This contains information about each member of the set and the latest point in time that this member has synced to. If this collection becomes out of date, you can refresh it by dropping the collection and allowing MongoDB to automatically refresh it during normal replication:

```
db.getSiblingDB("local").slaves.drop()
```

5.2.4 Collections used in Master/Slave Replication

In *master*/*slave* replication, the `local` database contains the following collections:

- On the master:

  `local.oplog.`**`$main`**
  This is the oplog for the master-slave configuration.

  `local.`**`slaves`**
  This contains information about each slave.

- On each slave:

  `local.`**`sources`**
  This contains information about the slave's master server.

5.3 System Collections

5.3.1 Synopsis

MongoDB stores system information in collections that use the `<database>.system.*` *namespace*, which MongoDB reserves for internal use. Do not create collections that begin with `system`.

MongoDB also stores some additional instance-local metadata in the *local database* (page 686), specifically for replication purposes.

5.3.2 Collections

System collections include these collections stored in the `admin` database:

`admin.system.`**`roles`**
New in version 2.6.

The `admin.system.roles` (page 689) collection stores custom roles that administrators create and assign to users to provide access to specific resources.

`admin.system.`**`users`**
Changed in version 2.6.

The `admin.system.users` (page 689) collection stores the user's authentication credentials as well as any roles assigned to the user. Users may define authorization roles in the `admin.system.roles` (page 689) collection.

`admin.system.`**`version`**
New in version 2.6.

Stores the schema version of the user credential documents.

System collections also include these collections stored directly in each database:

`<database>.system.`**`namespaces`**
The `<database>.system.namespaces` (page 689) collection contains information about all of the database's collections. Additional namespace metadata exists in the `database.ns` files and is opaque to database users.

`<database>.system.`**`indexes`**
The `<database>.system.indexes` (page 689) collection lists all the indexes in the database. Add and remove data from this collection via the `ensureIndex()` (page 30) and `dropIndex()` (page 29)

`<database>.system.`**`profile`**
The `<database>.system.profile` (page 689) collection stores database profiling information. For information on profiling, see *database-profiling*.

`<database>.system.`**`js`**
The `<database>.system.js` (page 689) collection holds special JavaScript code for use in `server side JavaScript`. See `http://docs.mongodb.org/manual/tutorial/store-javascript-function-or` for more information.

CHAPTER 6

General System Reference

6.1 Exit Codes and Statuses

MongoDB will return one of the following codes and statuses when exiting. Use this guide to interpret logs and when troubleshooting issues with `mongod` (page 583) and `mongos` (page 601) instances.

0

Returned by MongoDB applications upon successful exit.

2

The specified options are in error or are incompatible with other options.

3

Returned by `mongod` (page 583) if there is a mismatch between hostnames specified on the command line and in the `local.sources` (page 688) collection. `mongod` (page 583) may also return this status if *oplog* collection in the `local` database is not readable.

4

The version of the database is different from the version supported by the `mongod` (page 583) (or `mongod.exe` (page 618)) instance. The instance exits cleanly. Restart `mongod` (page 583) with the *--upgrade* option to upgrade the database to the version supported by this `mongod` (page 583) instance.

5

Returned by `mongod` (page 583) if a `moveChunk` (page 310) operation fails to confirm a commit.

12

Returned by the `mongod.exe` (page 618) process on Windows when it receives a Control-C, Close, Break or Shutdown event.

14

Returned by MongoDB applications which encounter an unrecoverable error, an uncaught exception or uncaught signal. The system exits without performing a clean shut down.

20

Message: `ERROR: wsastartup failed <reason>`

Returned by MongoDB applications on Windows following an error in the WSAStartup function.

Message: `NT Service Error`

Returned by MongoDB applications for Windows due to failures installing, starting or removing the NT Service for the application.

45

Returned when a MongoDB application cannot open a file or cannot obtain a lock on a file.

47
MongoDB applications exit cleanly following a large clock skew (32768 milliseconds) event.

48
`mongod` (page 583) exits cleanly if the server socket closes. The server socket is on port `27017` by default, or as specified to the `--port` run-time option.

49
Returned by `mongod.exe` (page 618) or `mongos.exe` (page 619) on Windows when either receives a shutdown message from the *Windows Service Control Manager*.

100
Returned by `mongod` (page 583) when the process throws an uncaught exception.

6.2 MongoDB Limits and Thresholds

This document provides a collection of hard and soft limitations of the MongoDB system.

6.2.1 BSON Documents

BSON Document Size
The maximum BSON document size is 16 megabytes.

The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth. To store documents larger than the maximum size, MongoDB provides the GridFS API. See `mongofiles` (page 674) and the documentation for your `driver` for more information about GridFS.

Nested Depth for BSON Documents
Changed in version 2.2.

MongoDB supports no more than 100 levels of nesting for *BSON documents*.

6.2.2 Namespaces

Namespace Length
Each namespace, including database and collection name, must be shorter than 123 bytes.

Number of Namespaces
The limitation on the number of namespaces is the size of the namespace file divided by 628.

A 16 megabyte namespace file can support approximately 24,000 namespaces. Each collection and index is a namespace.

Size of Namespace File
Namespace files can be no larger than 2047 megabytes.

By default namespace files are 16 megabytes. You can configure the size using the `nsSize` option.

6.2.3 Indexes

Index Key Limit
The *total size* of an index entry, which can include structural overhead depending on the BSON type, must be *less than* 1024 bytes.

Changed in version 2.6: MongoDB 2.6 implements a stronger enforcement of the limit on index key (page 692):

- MongoDB will **not** create an index (page 30) on a collection if the index entry for an existing document exceeds the index key limit (page 692). Previous versions of MongoDB would create the index but not index such documents.
- Reindexing operations will error if the index entry for an indexed field exceeds the index key limit (page 692). Reindexing operations occur as part of compact (page 322) and repairDatabase (page 341) commands as well as the db.collection.reIndex() (page 65) method.

  Because these operations drop *all* the indexes from a collection and then recreate them sequentially, the error from the index key limit (page 692) prevents these operations from rebuilding any remaining indexes for the collection and, in the case of the repairDatabase (page 341) command, from continuing with the remainder of the process.
- MongoDB will not insert into an indexed collection any document with an indexed field whose corresponding index entry would exceed the index key limit (page 692), and instead, will return an error. Previous versions of MongoDB would insert but not index such documents.
- Updates to the indexed field will error if the updated value causes the index entry to exceed the index key limit (page 692).

  If an existing document contains an indexed field whose index entry exceeds the limit, *any* update that results in the relocation of that document on disk will error.
- mongorestore (page 628) and mongoimport (page 642) will not insert documents that contain an indexed field whose corresponding index entry would exceed the index key limit (page 692).
- In MongoDB 2.6, secondary members of replica sets will continue to replicate documents with an indexed field whose corresponding index entry exceeds the index key limit (page 692) on initial sync but will print warnings in the logs.

  Secondary members also allow index build and rebuild operations on a collection that contains an indexed field whose corresponding index entry exceeds the index key limit (page 692) but with warnings in the logs.

  With *mixed version* replica sets where the secondaries are version 2.6 and the primary is version 2.4, secondaries will replicate documents inserted or updated on the 2.4 primary, but will print error messages in the log if the documents contain an indexed field whose corresponding index entry exceeds the index key limit (page 692).
- For existing sharded collections, chunk migration will fail if the chunk has a document that contains an indexed field whose index entry exceeds the index key limit (page 692).

Number of Indexes per Collection

A single collection can have *no more* than 64 indexes.

Index Name Length

Fully qualified index names, which includes the namespace and the dot separators (i.e. <database name>.<collection name>.$<index name>), cannot be longer than 128 characters.

By default, <index name> is the concatenation of the field names and index type. You can explicitly specify the <index name> to the ensureIndex() (page 30) method to ensure that the fully qualified index name does not exceed the limit.

Number of Indexed Fields in a Compound Index

There can be no more than 31 fields in a compound index.

Queries cannot use both text and Geospatial Indexes

You cannot combine the text (page 251) command, which requires a special *text index*, with a query operator

that requires a different type of special index. For example you cannot combine `text` (page 251) command with the `$near` (page 429) operator.

Fields with 2dsphere Indexes can only hold Geometries

Fields with `2dsphere` indexes must hold geometry data in the form of *coordinate pairs* or *GeoJSON* data. If you attempt to insert a document with non-geometry data in a `2dsphere` indexed field, or build a `2dsphere` index on a collection where the indexed field has non-geometry data, the operation will fail.

See also:

The unique indexes limit in *Sharding Operational Restrictions* (page 695).

6.2.4 Data

Maximum Number of Documents in a Capped Collection

Changed in version 2.4.

If you specify a maximum number of documents for a capped collection using the `max` parameter to `create` (page 333), the limit must be less than 2^{32} documents. If you do not specify a maximum number of documents when creating a capped collection, there is no limit on the number of documents.

Data Size

A single `mongod` (page 583) instance cannot manage a data set that exceeds maximum virtual memory address space provided by the underlying operating system.

Table 6.1: Virtual Memory Limitations

Operating System	Journaled	Not Journaled
Linux	64 terabytes	128 terabytes
Windows Server 2012 R2 and Windows 8.1	64 terabytes	128 terabytes
Windows (otherwise)	4 terabytes	8 terabytes

Number of Collections in a Database

The maximum number of collections in a database is a function of the size of the namespace file and the number of indexes of collections in the database.

See `Number of Namespaces` (page 692) for more information.

6.2.5 Replica Sets

Number of Members of a Replica Set

Replica sets can have no more than 12 members.

Number of Voting Members of a Replica Set

Only 7 members of a replica set can have votes at any given time. See can vote *replica-set-non-voting-members* for more information

Maximum Size of Auto-Created Oplog

Changed in version 2.6.

If you do not explicitly specify an oplog size (i.e. with `oplogSizeMB` or `--oplogSize`) MongoDB will create an oplog that is no larger than 50 gigabytes.

6.2.6 Sharded Clusters

Sharded clusters have the restrictions and thresholds described here.

Sharding Operational Restrictions

Operations Unavailable in Sharded Environments

The `group` (page 216) does not work with sharding. Use `mapReduce` (page 220) or `aggregate` (page 210) instead.

`db.eval()` (page 112) is incompatible with sharded collections. You may use `db.eval()` (page 112) with un-sharded collections in a shard cluster.

`$where` (page 421) does not permit references to the `db` object from the `$where` (page 421) function. This is uncommon in un-sharded collections.

The `$isolated` (page 482) update modifier does not work in sharded environments.

`$snapshot` (page 562) queries do not work in sharded environments.

The `geoSearch` (page 233) command is not supported in sharded environments.

Covered Queries in Sharded Clusters

MongoDB does not support *covered queries* for sharded collections.

Sharding Existing Collection Data Size

For existing collections that hold documents, MongoDB supports enabling sharding on *any* collections that contains less than 256 gigabytes of data. MongoDB *may* be able to shard collections with as many as 400 gigabytes depending on the distribution of document sizes. The precise size of the limitation is a function of the chunk size and the data size. Consider the following table:

Shard Key Size	512 bytes	200 bytes
Number of Splits	31,558	85,196
Max Collection Size (1 MB Chunk Size)	31 GB	83 GB
Max Collection Size (64 MB Chunk Size)	1.9 TB	5.3 TB

The data in this chart reflects documents with no data other than the shard key values and therefore represents the smallest possible data size that could reach this limit.

Important: Sharded collections may have *any* size, after successfully enabling sharding.

Single Document Modification Operations in Sharded Collections

All `update()` (page 72) and `remove()` (page 66) operations for a sharded collection that specify the `justOne` or `multi: false` option must include the *shard key or* the `_id` field in the query specification. `update()` (page 72) and `remove()` (page 66) operations specifying `justOne` or `multi: false` in a sharded collection without the *shard key or* the `_id` field return an error.

Unique Indexes in Sharded Collections

MongoDB does not support unique indexes across shards, except when the unique index contains the full shard key as a prefix of the index. In these situations MongoDB will enforce uniqueness across the full key, not a single field.

See

`http://docs.mongodb.org/manual/tutorial/enforce-unique-keys-for-sharded-collections` for an alternate approach.

Maximum Number of Documents Per Chunk to Migrate

MongoDB cannot move a chunk if the number of documents in the chunk exceeds either 250000 documents or 1.3 times the number of average sized documents that the *maximum chunk size* can hold.

Shard Key Limitations

Shard Key Size
A shard key cannot exceed 512 bytes.

Shard Key Index Type
A *shard key* index can be an ascending index on the shard key, a compound index that start with the shard key and specify ascending order for the shard key, or a `hashed index`.

A *shard key* index cannot be an index that specifies a `multikey index`, a `text index` or a *geospatial index* on the *shard key* fields.

Shard Key is Immutable
You cannot change a shard key after sharding the collection. If you must change a shard key:

- Dump all data from MongoDB into an external format.
- Drop the original sharded collection.
- Configure sharding using the new shard key.
- `Pre-split` the shard key range to ensure initial even distribution.
- Restore the dumped data into MongoDB.

Shard Key Value in a Document is Immutable
After you insert a document into a sharded collection, you cannot change the document's value for the field or fields that comprise the shard key. The `update()` (page 72) operation will not modify the value of a shard key in an existing document.

Monotonically Increasing Shard Keys Can Limit Insert Throughput
For clusters with high insert volumes, a shard keys with monotonically increasing and decreasing keys can affect insert throughput. If your shard key is the `_id` field, be aware that the default values of the `_id` fields are *ObjectIds* which have generally increasing values.

When inserting documents with monotonically increasing shard keys, all inserts belong to the same *chunk* on a single *shard*. The system will eventually divide the chunk range that receives all write operations and migrate its contents to distribute data more evenly. However, at any moment the cluster can direct insert operations only to a single shard, which creates an insert throughput bottleneck.

If the operations on the cluster are predominately read operations and updates, this limitation may not affect the cluster.

To avoid this constraint, use a *hashed shard key* or select a field that does not increase or decrease monotonically.

Changed in version 2.4: *Hashed shard keys* and *hashed indexes* store hashes of keys with ascending values.

6.2.7 Operations

Sorted Documents
MongoDB will only return sorted results on fields without an index *if* the combined size of all documents in the sort operation, plus a small overhead, is less than 32 megabytes.

Aggregation Pipeline Operation
Changed in version 2.6.

Pipeline stages have a limit of 100 megabytes of RAM. If a stage exceeds this limit, MongoDB will produce an error. To allow for the handling of large datasets, use the `allowDiskUse` option to enable aggregation pipeline stages to write data to temporary files.

See also:

$sort and Memory Restrictions (page 501) and *$group Operator and Memory* (page 487).

2d Geospatial queries cannot use the $or operator

See

`$or` (page 408) and `http://docs.mongodb.org/manual/core/geospatial-indexes`.

Area of GeoJSON Polygons

For `$geoIntersects` (page 423) or `$geoWithin` (page 425), if you specify a single-ringed polygon that has an area greater than a single hemisphere, include `the custom MongoDB coordinate reference system in the $geometry` (page 434) expression; otherwise, `$geoIntersects` (page 423) or `$geoWithin` (page 425) queries for the complementary geometry. For all other GeoJSON polygons with areas greater than a hemisphere, `$geoIntersects` (page 423) or `$geoWithin` (page 425) queries for the complementary geometry.

Write Command Operation Limit Size

Write commands (page 234) can accept no more than 1000 operations. The `Bulk()` (page 135) operations in the `mongo` (page 610) shell and comparable methods in the drivers do not have this limit.

6.2.8 Naming Restrictions

Database Name Case Sensitivity

MongoDB does not permit database names that differ only by the case of the characters.

Restrictions on Database Names for Windows

Changed in version 2.2: *Restrictions on Database Names for Windows* (page 798).

For MongoDB deployments running on Windows, MongoDB will not permit database names that include any of the following characters:

```
/\. "*<>:|?
```

Also, database names cannot contain the null character.

Restrictions on Database Names for Unix and Linux Systems

For MongoDB deployments running on Unix and Linux systems, MongoDB will not permit database names that include any of the following characters:

```
/\. "
```

Also, database names cannot contain the null character.

Length of Database Names

Database names cannot be empty and must have fewer than 64 characters.

Restriction on Collection Names

New in version 2.2.

Collection names should begin with an underscore or a letter character, and *cannot*:

- contain the `$`.
- be an empty string (e.g. `""`).
- contain the null character.
- begin with the `system.` prefix. (Reserved for internal use.)

In the mongo (page 610) shell, use db.getCollection() (page 114) to specify collection names that might interact with the shell or are not valid JavaScript.

Restrictions on Field Names
Field names cannot contain dots (i.e. .) or null characters, and they must not start with a dollar sign (i.e. $). See *faq-dollar-sign-escaping* for an alternate approach.

6.3 Glossary

$cmd A special virtual *collection* that exposes MongoDB's *database commands*. To use database commands, see *issue-commands*.

_id A field required in every MongoDB *document*. The *_id* field must have a unique value. You can think of the _id field as the document's *primary key*. If you create a new document without an _id field, MongoDB automatically creates the field and assigns a unique BSON *ObjectId*.

accumulator An *expression* in the *aggregation framework* that maintains state between documents in the aggregation *pipeline*. For a list of accumulator operations, see $group (page 486).

action An operation the user can perform on a resource. Actions and *resources* combine to create *privileges*. See action.

admin database A privileged database. Users must have access to the admin database to run certain administrative commands. For a list of administrative commands, see *Instance Administration Commands* (page 319).

aggregation Any of a variety of operations that reduces and summarizes large sets of data. MongoDB's aggregate() (page 22) and mapReduce() (page 58) methods are two examples of aggregation operations. For more information, see http://docs.mongodb.org/manual/core/aggregation.

aggregation framework The set of MongoDB operators that let you calculate aggregate values without having to use *map-reduce*. For a list of operators, see *Aggregation Reference* (page 564).

arbiter A member of a *replica set* that exists solely to vote in *elections*. Arbiters do not replicate data. See *replica-set-arbiter-configuration*.

authentication Verification of the user identity. See http://docs.mongodb.org/manual/core/authentication.

authorization Provisioning of access to databases and operations. See http://docs.mongodb.org/manual/core/authorization.

B-tree A data structure commonly used by database management systems to store indexes. MongoDB uses B-trees for its indexes.

balancer An internal MongoDB process that runs in the context of a *sharded cluster* and manages the migration of *chunks*. Administrators must disable the balancer for all maintenance operations on a sharded cluster. See *sharding-balancing*.

BSON A serialization format used to store *documents* and make remote procedure calls in MongoDB. "BSON" is a portmanteau of the words "binary" and "JSON". Think of BSON as a binary representation of JSON (JavaScript Object Notation) documents. See http://docs.mongodb.org/manual/reference/bson-types and http://docs.mongodb.org/manual/reference/mongodb-extended-json.

BSON types The set of types supported by the *BSON* serialization format. For a list of BSON types, see http://docs.mongodb.org/manual/reference/bson-types.

CAP Theorem Given three properties of computing systems, consistency, availability, and partition tolerance, a distributed computing system can provide any two of these features, but never all three.

capped collection A fixed-sized *collection* that automatically overwrites its oldest entries when it reaches its maximum size. The MongoDB *oplog* that is used in *replication* is a capped collection. See `http://docs.mongodb.org/manual/core/capped-collections`.

checksum A calculated value used to ensure data integrity. The *md5* algorithm is sometimes used as a checksum.

chunk A contiguous range of *shard key* values within a particular *shard*. Chunk ranges are inclusive of the lower boundary and exclusive of the upper boundary. MongoDB splits chunks when they grow beyond the configured chunk size, which by default is 64 megabytes. MongoDB migrates chunks when a shard contains too many chunks of a collection relative to other shards. See *sharding-data-partitioning* and `http://docs.mongodb.org/manual/core/sharded-cluster-mechanics`.

client The application layer that uses a database for data persistence and storage. *Drivers* provide the interface level between the application layer and the database server.

cluster See *sharded cluster*.

collection A grouping of MongoDB *documents*. A collection is the equivalent of an *RDBMS* table. A collection exists within a single *database*. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection have a similar or related purpose. See *faq-dev-namespace*.

collection scan Collection scans are a query execution strategy where MongoDB must inspect every document in a collection to see if it matches the query criteria. These queries are very inefficient and do not use indexes. See `http://docs.mongodb.org/manual/core/query-optimization` for details about query execution strategies.

compound index An *index* consisting of two or more keys. See *index-type-compound*.

config database An internal database that holds the metadata associated with a *sharded cluster*. Applications and administrators should not modify the `config` database in the course of normal operation. See *Config Database* (page 681).

config server A `mongod` (page 583) instance that stores all the metadata associated with a *sharded cluster*. A production sharded cluster requires three config servers, each on a separate machine. See *sharding-config-server*.

control script A simple shell script, typically located in the `/etc/rc.d` or `/etc/init.d` directory, and used by the system's initialization process to start, restart or stop a *daemon* process.

CRUD An acronym for the fundamental operations of a database: Create, Read, Update, and Delete. See `http://docs.mongodb.org/manual/crud`.

CSV A text-based data format consisting of comma-separated values. This format is commonly used to exchange data between relational databases since the format is well-suited to tabular data. You can import CSV files using `mongoimport` (page 642).

cursor A pointer to the result set of a *query*. Clients can iterate through a cursor to retrieve results. By default, cursors timeout after 10 minutes of inactivity. See *read-operations-cursors*.

daemon The conventional name for a background, non-interactive process.

data directory The file-system location where the `mongod` (page 583) stores data files. The `dbPath` option specifies the data directory.

data-center awareness A property that allows clients to address members in a system based on their locations. *Replica sets* implement data-center awareness using *tagging*. See `/data-center-awareness`.

database A physical container for *collections*. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

database command A MongoDB operation, other than an insert, update, remove, or query. For a list of database commands, see *Database Commands* (page 210). To use database commands, see *issue-commands*.

database profiler A tool that, when enabled, keeps a record on all long-running operations in a database's `system.profile` collection. The profiler is most often used to diagnose slow queries. See *database-profiling*.

datum A set of values used to define measurements on the earth. MongoDB uses the *WGS84* datum in certain *geospatial* calculations. See `http://docs.mongodb.org/manual/applications/geospatial-indexes`.

dbpath The location of MongoDB's data file storage. See `dbPath`.

delayed member A *replica set* member that cannot become primary and applies operations at a specified delay. The delay is useful for protecting data from human error (i.e. unintentionally deleted databases) or updates that have unforeseen effects on the production database. See *replica-set-delayed-members*.

diagnostic log A verbose log of operations stored in the *dbpath*. See the *--diaglog* option.

document A record in a MongoDB *collection* and the basic unit of data in MongoDB. Documents are analogous to *JSON* objects but exist in the database in a more type-rich format known as *BSON*. See `http://docs.mongodb.org/manual/core/document`.

dot notation MongoDB uses the dot notation to access the elements of an array and to access the fields of an embedded document. See *document-dot-notation*.

draining The process of removing or "shedding" *chunks* from one *shard* to another. Administrators must drain shards before removing them from the cluster. See `http://docs.mongodb.org/manual/tutorial/remove-shards-from-cluster`.

driver A client library for interacting with MongoDB in a particular language. See `http://docs.mongodb.org/manual/applications/drivers`.

election The process by which members of a *replica set* select a *primary* on startup and in the event of a failure. See *replica-set-elections*.

eventual consistency A property of a distributed system that allows changes to the system to propagate gradually. In a database system, this means that readable members are not required to reflect the latest writes at all times. In MongoDB, reads to a primary have *strict consistency*; reads to secondaries have *eventual consistency*.

expression In the context of *aggregation framework*, expressions are the stateless transformations that operate on the data that passes through a *pipeline*. See `http://docs.mongodb.org/manual/core/aggregation`.

failover The process that allows a *secondary* member of a *replica set* to become *primary* in the event of a failure. See *replica-set-failover*.

field A name-value pair in a *document*. A document has zero or more fields. Fields are analogous to columns in relational databases. See *document-structure*.

field path Path to a field in the document. To specify a field path, use a string that prefixes the field name with a dollar sign (`$`).

firewall A system level networking filter that restricts access based on, among other things, IP address. Firewalls form a part of an effective network security strategy. See *security-firewalls*.

fsync A system call that flushes all dirty, in-memory pages to disk. MongoDB calls `fsync()` on its database files at least every 60 seconds. See `fsync` (page 336).

geohash A geohash value is a binary representation of the location on a coordinate grid. See *geospatial-indexes-geohash*.

GeoJSON A *geospatial* data interchange format based on JavaScript Object Notation (*JSON*). GeoJSON is used in `geospatial queries`. For supported GeoJSON objects, see *geo-overview-location-data*. For the GeoJSON format specification, see http://geojson.org/geojson-spec.html.

geospatial Data that relates to geographical location. In MongoDB, you may store, index, and query data according to geographical parameters. See `http://docs.mongodb.org/manual/applications/geospatial-indexes`.

GridFS A convention for storing large files in a MongoDB database. All of the official MongoDB drivers support this convention, as does the `mongofiles` (page 674) program. See `http://docs.mongodb.org/manual/core/gridfs` and `http://docs.mongodb.org/manual/reference/gridfs`.

hashed shard key A special type of *shard key* that uses a hash of the value in the shard key field to distribute documents among members of the *sharded cluster*. See *index-type-hashed*.

haystack index A *geospatial* index that enhances searches by creating "buckets" of objects grouped by a second criterion. See `http://docs.mongodb.org/manual/core/geohaystack`.

hidden member A *replica set* member that cannot become *primary* and are invisible to client applications. See *replica-set-hidden-members*.

idempotent The quality of an operation to produce the same result given the same input, whether run once or run multiple times.

index A data structure that optimizes queries. See `http://docs.mongodb.org/manual/core/indexes`.

initial sync The *replica set* operation that replicates data from an existing replica set member to a new or restored replica set member. See *replica-set-initial-sync*.

interrupt point A point in an operation's lifecycle when it can safely abort. MongoDB only terminates an operation at designated interrupt points. See `http://docs.mongodb.org/manual/tutorial/terminate-running-operations`.

IPv6 A revision to the IP (Internet Protocol) standard that provides a significantly larger address space to more effectively support the number of hosts on the contemporary Internet.

ISODate The international date format used by `mongo` (page 610) to display dates. The format is: `YYYY-MM-DD HH:MM.SS.millis`.

JavaScript A popular scripting language originally designed for web browsers. The MongoDB shell and certain server-side functions use a JavaScript interpreter. See `http://docs.mongodb.org/manual/core/server-side-javascript` for more information.

journal A sequential, binary transaction log used to bring the database into a valid state in the event of a hard shutdown. Journaling writes data first to the journal and then to the core data files. MongoDB enables journaling by default for 64-bit builds of MongoDB version 2.0 and newer. Journal files are pre-allocated and exist as files in the *data directory*. See `http://docs.mongodb.org/manual/core/journaling/`.

JSON JavaScript Object Notation. A human-readable, plain text format for expressing structured data with support in many programming languages. For more information, see http://www.json.org. Certain MongoDB tools render an approximation of MongoDB *BSON* documents in JSON format. See `http://docs.mongodb.org/manual/reference/mongodb-extended-json`.

JSON document A *JSON* document is a collection of fields and values in a structured format. For sample JSON documents, see http://json.org/example.html.

JSONP *JSON* with Padding. Refers to a method of injecting JSON into applications. **Presents potential security concerns**.

least privilege An authorization policy that gives a user only the amount of access that is essential to that user's work and no more.

legacy coordinate pairs The format used for *geospatial* data prior to MongoDB version 2.4. This format stores geospatial data as points on a planar coordinate system (e.g. `[ x, y ]`). See `http://docs.mongodb.org/manual/applications/geospatial-indexes`.

LineString A LineString is defined by an array of two or more positions. A closed LineString with four or more positions is called a LinearRing, as described in the GeoJSON LineString specification: http://geojson.org/geojson-spec.html#linestring. To use a LineString in MongoDB, see *geospatial-indexes-store-geojson*.

lock MongoDB uses locks to ensure `concurrency`. MongoDB uses both *read locks* and *write locks*. For more information, see *faq-concurrency-locking*.

LVM Logical volume manager. LVM is a program that abstracts disk images from physical devices and provides a number of raw disk manipulation and snapshot capabilities useful for system management. For information on LVM and MongoDB, see *lvm-backup-and-restore*.

map-reduce A data processing and aggregation paradigm consisting of a "map" phase that selects data and a "reduce" phase that transforms the data. In MongoDB, you can run arbitrary aggregations over data using map-reduce. For map-reduce implementation, see `http://docs.mongodb.org/manual/core/map-reduce`. For all approaches to aggregation, see `http://docs.mongodb.org/manual/core/aggregation`.

mapping type A Structure in programming languages that associate keys with values, where keys may nest other pairs of keys and values (e.g. dictionaries, hashes, maps, and associative arrays). The properties of these structures depend on the language specification and implementation. Generally the order of keys in mapping types is arbitrary and not guaranteed.

master The database that receives all writes in a conventional master-*slave* replication. In MongoDB, *replica sets* replace master-slave replication for most use cases. For more information on master-slave replication, see `http://docs.mongodb.org/manual/core/master-slave`.

md5 A hashing algorithm used to efficiently provide reproducible unique strings to identify and *checksum* data. MongoDB uses md5 to identify chunks of data for *GridFS*. See *filemd5* (page 335).

MIB Management Information Base. MongoDB uses MIB files to define the type of data tracked by SNMP in the MongoDB Enterprise edition.

MIME Multipurpose Internet Mail Extensions. A standard set of type and encoding definitions used to declare the encoding and type of data in multiple data storage, transmission, and email contexts. The `mongofiles` (page 674) tool provides an option to specify a MIME type to describe a file inserted into *GridFS* storage.

mongo The MongoDB shell. The `mongo` (page 610) process starts the MongoDB shell as a daemon connected to either a `mongod` (page 583) or `mongos` (page 601) instance. The shell has a JavaScript interface. See *mongo* (page 610) and *mongo Shell Methods* (page 21).

mongod The MongoDB database server. The `mongod` (page 583) process starts the MongoDB server as a daemon. The MongoDB server manages data requests and formats and manages background operations. See *mongod* (page 583).

MongoDB An open-source document-based database system. "MongoDB" derives from the word "humongous" because of the database's ability to scale up with ease and hold very large amounts of data. MongoDB stores *documents* in *collections* within databases.

MongoDB Enterprise A commercial edition of MongoDB that includes additional features. For more information, see MongoDB Subscriptions[1].

mongos The routing and load balancing process that acts an interface between an application and a MongoDB *sharded cluster*. See *mongos* (page 601).

namespace The canonical name for a collection or index in MongoDB. The namespace is a combination of the database name and the name of the collection or index, like so:

[1] https://www.mongodb.com/products/mongodb-subscriptions

`[database-name].[collection-or-index-name]`. All documents belong to a namespace. See *faq-dev-namespace*.

natural order The order in which the database refers to documents on disk. This is the default sort order. See `$natural` (page 563) and *Return in Natural Order* (page 98).

ObjectId A special 12-byte *BSON* type that guarantees uniqueness within the *collection*. The ObjectId is generated based on timestamp, machine ID, process ID, and a process-local incremental counter. MongoDB uses ObjectId values as the default values for *_id* fields.

operator A keyword beginning with a `$` used to express an update, complex query, or data transformation. For example, `$gt` is the query language's "greater than" operator. For available operators, see *Operators* (page 400).

oplog A *capped collection* that stores an ordered history of logical writes to a MongoDB database. The oplog is the basic mechanism enabling *replication* in MongoDB. See `http://docs.mongodb.org/manual/core/replica-set-oplog`.

ordered query plan A query plan that returns results in the order consistent with the `sort()` (page 95) order. See *read-operations-query-optimization*.

orphaned document In a sharded cluster, orphaned documents are those documents on a shard that also exist in chunks on other shards as a result of failed migrations or incomplete migration cleanup due to abnormal shutdown. Delete orphaned documents using `cleanupOrphaned` (page 305) to reclaim disk space and reduce confusion.

padding The extra space allocated to document on the disk to prevent moving a document when it grows as the result of `update()` (page 72) operations. See *record-allocation-strategies*.

padding factor An automatically-calibrated constant used to determine how much extra space MongoDB should allocate per document container on disk. A padding factor of 1 means that MongoDB will allocate only the amount of space needed for the document. A padding factor of 2 means that MongoDB will allocate twice the amount of space required by the document. See *record-allocation-strategies*.

page fault Page faults can occur as MongoDB reads from or writes data to parts of its data files that are not currently located in physical memory. In contrast, operating system page faults happen when physical memory is exhausted and pages of physical memory are swapped to disk.

See *administration-monitoring-page-faults* and *faq-storage-page-faults*.

partition A distributed system architecture that splits data into ranges. *Sharding* uses partitioning. See *sharding-data-partitioning*.

passive member A member of a *replica set* that cannot become primary because its `priority` is `0`. See `http://docs.mongodb.org/manual/core/replica-set-priority-0-member`.

pcap A packet-capture format used by `mongosniff` (page 670) to record packets captured from network interfaces and display them as human-readable MongoDB operations. See *Options* (page 670).

PID A process identifier. UNIX-like systems assign a unique-integer PID to each running process. You can use a PID to inspect a running process and send signals to it. See *proc-file-system*.

pipe A communication channel in UNIX-like systems allowing independent processes to send and receive data. In the UNIX shell, piped operations allow users to direct the output of one command into the input of another.

pipeline A series of operations in an *aggregation* process. See `http://docs.mongodb.org/manual/core/aggregation`.

Point A single coordinate pair as described in the GeoJSON Point specification: http://geojson.org/geojson-spec.html#point. To use a Point in MongoDB, see *geospatial-indexes-store-geojson*.

Polygon An array of *LinearRing* coordinate arrays, as described in the GeoJSON Polygon specification: http://geojson.org/geojson-spec.html#polygon. For Polygons with multiple rings, the first must be the exterior ring and any others must be interior rings or holes.

MongoDB does not permit the exterior ring to self-intersect. Interior rings must be fully contained within the outer loop and cannot intersect or overlap with each other. See *geospatial-indexes-store-geojson*.

powerOf2Sizes A per-collection setting that changes and normalizes the way MongoDB allocates space for each *document*, in an effort to maximize storage reuse and to reduce fragmentation. This is the default for `TTL Collections`. See *collMod* (page 321) and `usePowerOf2Sizes` (page 322).

pre-splitting An operation performed before inserting data that divides the range of possible shard key values into chunks to facilitate easy insertion and high write throughput. In some cases pre-splitting expedites the initial distribution of documents in *sharded cluster* by manually dividing the collection rather than waiting for the MongoDB *balancer* to do so. See `http://docs.mongodb.org/manual/tutorial/create-chunks-in-sharded-cluster`.

primary In a *replica set*, the primary member is the current *master* instance, which receives all write operations. See *replica-set-primary-member*.

primary key A record's unique immutable identifier. In an *RDBMS*, the primary key is typically an integer stored in each row's `id` field. In MongoDB, the *_id* field holds a document's primary key which is usually a BSON *ObjectId*.

primary shard The *shard* that holds all the un-sharded collections. See *primary-shard*.

priority A configurable value that helps determine which members in a *replica set* are most likely to become *primary*. See `priority`.

privilege A combination of specified *resource* and *actions* permitted on the resource. See *privilege*.

projection A document given to a *query* that specifies which fields MongoDB returns in the result set. See *projection*. For a list of projection operators, see *Projection Operators* (page 444).

query A read request. MongoDB uses a *JSON*-like query language that includes a variety of *query operators* with names that begin with a `$` character. In the `mongo` (page 610) shell, you can issue queries using the `find()` (page 36) and `findOne()` (page 46) methods. See *read-operations-queries*.

query optimizer A process that generates query plans. For each query, the optimizer generates a plan that matches the query to the index that will return results as efficiently as possible. The optimizer reuses the query plan each time the query runs. If a collection changes significantly, the optimizer creates a new query plan. See *read-operations-query-optimization*.

query shape A combination of query predicate, sort, and projection specifications.

For the query predicate, only the structure of the predicate, including the field names, are significant; the values in the query predicate are insignificant. As such, a query predicate `{ type: 'food' }` is equivalent to the query predicate `{ type: 'utensil' }` for a query shape.

RDBMS Relational Database Management System. A database management system based on the relational model, typically using *SQL* as the query language.

read lock In the context of a reader-writer lock, a lock that while held allows concurrent readers but no writers. See *faq-concurrency-locking*.

read preference A setting that determines how clients direct read operations. Read preference affects all replica sets, including shards. By default, MongoDB directs reads to *primaries* for *strict consistency*. However, you may also direct reads to secondaries for *eventually consistent* reads. See `Read Preference`.

record size The space allocated for a document including the padding. For more information on padding, see *record-allocation-strategies* and *compact* (page 322).

recovering A *replica set* member status indicating that a member is not ready to begin normal activities of a secondary or primary. Recovering members are unavailable for reads.

replica pairs The precursor to the MongoDB *replica sets*.

Deprecated since version 1.6.

replica set A cluster of MongoDB servers that implements master-slave replication and automated failover. MongoDB's recommended replication strategy. See http://docs.mongodb.org/manual/replication.

replication A feature allowing multiple database servers to share the same data, thereby ensuring redundancy and facilitating load balancing. See http://docs.mongodb.org/manual/replication.

replication lag The length of time between the last operation in the *primary's oplog* and the last operation applied to a particular *secondary*. In general, you want to keep replication lag as small as possible. See *Replication Lag*.

resident memory The subset of an application's memory currently stored in physical RAM. Resident memory is a subset of *virtual memory*, which includes memory mapped to physical RAM and to disk.

resource A database, collection, set of collections, or cluster. A *privilege* permits *actions* on a specified resource. See *resource*.

REST An API design pattern centered around the idea of resources and the *CRUD* operations that apply to them. Typically REST is implemented over HTTP. MongoDB provides a simple HTTP REST interface that allows HTTP clients to run commands against the server. See *rest-interface* and *rest-api*.

role A set of privileges that permit *actions* on specified *resources*. Roles assigned to a user determine the user's access to resources and operations. See http://docs.mongodb.org/manual/core/security-introduction.

rollback A process that reverts writes operations to ensure the consistency of all replica set members. See *replica-set-rollback*.

secondary A *replica set* member that replicates the contents of the master database. Secondary members may handle read requests, but only the *primary* members can handle write operations. See *replica-set-secondary-members*.

secondary index A database *index* that improves query performance by minimizing the amount of work that the query engine must perform to fulfill a query. See http://docs.mongodb.org/manual/indexes.

set name The arbitrary name given to a replica set. All members of a replica set must have the same name specified with the `replSetName` setting or the `--replSet` option.

shard A single `mongod` (page 583) instance or *replica set* that stores some portion of a *sharded cluster's* total data set. In production, all shards should be replica sets. See http://docs.mongodb.org/manual/core/sharded-cluster-shards.

shard key The field MongoDB uses to distribute documents among members of a *sharded cluster*. See *shard-key*.

sharded cluster The set of nodes comprising a *sharded* MongoDB deployment. A sharded cluster consists of three config processes, one or more replica sets, and one or more `mongos` (page 601) routing processes. See http://docs.mongodb.org/manual/core/sharded-cluster-components.

sharding A database architecture that partitions data by key ranges and distributes the data among two or more database instances. Sharding enables horizontal scaling. See http://docs.mongodb.org/manual/sharding.

shell helper A method in the `mongo` shell that provides a more concise syntax for a *database command* (page 210). Shell helpers improve the general interactive experience. See *mongo Shell Methods* (page 21).

single-master replication A *replication* topology where only a single database instance accepts writes. Single-master replication ensures consistency and is the replication topology employed by MongoDB. See http://docs.mongodb.org/manual/core/replica-set-primary.

slave A read-only database that replicates operations from a *master* database in conventional master/slave replication. In MongoDB, *replica sets* replace master/slave replication for most use cases. However, for information on master/slave replication, see http://docs.mongodb.org/manual/core/master-slave.

split The division between *chunks* in a *sharded cluster*. See http://docs.mongodb.org/manual/core/sharding-chunk

SQL Structured Query Language (SQL) is a common special-purpose programming language used for interaction with a relational database, including access control, insertions, updates, queries, and deletions. There are some similar elements in the basic SQL syntax supported by different database vendors, but most implementations have their own dialects, data types, and interpretations of proposed SQL standards. Complex SQL is generally not directly portable between major *RDBMS* products. SQL is often used as metonym for relational databases.

SSD Solid State Disk. A high-performance disk drive that uses solid state electronics for persistence, as opposed to the rotating platters and movable read/write heads used by traditional mechanical hard drives.

stale Refers to the amount of time a *secondary* member of a *replica set* trails behind the current state of the *primary'soplog*. If a secondary becomes too stale, it can no longer use replication to catch up to the current state of the primary. See http://docs.mongodb.org/manual/core/replica-set-oplog and http://docs.mongodb.org/manual/core/replica-set-sync for more information.

standalone An instance of mongod (page 583) that is running as a single server and not as part of a *replica set*. To convert a standalone into a replica set, see http://docs.mongodb.org/manual/tutorial/convert-standalone-to-replica-set.

storage order See *natural order*.

strict consistency A property of a distributed system requiring that all members always reflect the latest changes to the system. In a database system, this means that any system that can provide data must reflect the latest writes at all times. In MongoDB, reads from a primary have *strict consistency*; reads from secondary members have *eventual consistency*.

sync The *replica set* operation where members replicate data from the *primary*. Sync first occurs when MongoDB creates or restores a member, which is called *initial sync*. Sync then occurs continually to keep the member updated with changes to the replica set's data. See http://docs.mongodb.org/manual/core/replica-set-sync.

syslog On UNIX-like systems, a logging process that provides a uniform standard for servers and processes to submit logging information. MongoDB provides an option to send output to the host's syslog system. See syslogFacility.

tag A label applied to a replica set member or shard and used by clients to issue data-center-aware operations. For more information on using tags with replica sets and with shards, see the following sections of this manual: *replica-set-read-preference-tag-sets* and *shards-tag-sets*.

tailable cursor For a *capped collection*, a tailable cursor is a cursor that remains open after the client exhausts the results in the initial cursor. As clients insert new documents into the capped collection, the tailable cursor continues to retrieve documents. See http://docs.mongodb.org/manual/tutorial/create-tailable-cursor.

topology The state of a deployment of MongoDB instances, including the type of deployment (i.e. standalone, replica set, or sharded cluster) as well as the availability of servers, and the role of each server (i.e. *primary*, *secondary*, *config server*, or mongos (page 601).)

TSV A text-based data format consisting of tab-separated values. This format is commonly used to exchange data between relational databases, since the format is well-suited to tabular data. You can import TSV files using mongoimport (page 642).

TTL Stands for "time to live" and represents an expiration time or period for a given piece of information to remain in a cache or other temporary storage before the system deletes it or ages it out. MongoDB has a TTL collection feature. See http://docs.mongodb.org/manual/tutorial/expire-data.

unique index An index that enforces uniqueness for a particular field across a single collection. See *index-type-unique*.

unix epoch January 1st, 1970 at 00:00:00 UTC. Commonly used in expressing time, where the number of seconds or milliseconds since this point is counted.

unordered query plan A query plan that returns results in an order inconsistent with the `sort()` (page 95) order. See *read-operations-query-optimization*.

upsert An option for update operations. If set to true, the update operation will either update the first document matched by a query or insert a new document if none matches. The new document will have the fields implied by the operation. The `update()` (page 72) and `findAndModify()` (page 42) have the option. See *Upsert Option* (page 74).

virtual memory An application's working memory, typically residing on both disk an in physical RAM.

WGS84 The default *datum* MongoDB uses to calculate geometry over an Earth-like sphere. MongoDB uses the WGS84 datum for *geospatial* queries on *GeoJSON* objects. See the "EPSG:4326: WGS 84" specification: http://spatialreference.org/ref/epsg/4326/.

working set The data that MongoDB uses most often. This data is preferably held in RAM, solid-state drive (SSD), or other fast media. See *faq-working-set*.

write concern Specifies whether a write operation has succeeded. Write concern allows your application to detect insertion errors or unavailable `mongod` (page 583) instances. For *replica sets*, you can configure write concern to confirm replication to a specified number of members. See `http://docs.mongodb.org/manual/core/write-concern`.

write lock A lock on the database for a given writer. When a process writes to the database, it takes an exclusive write lock to prevent other processes from writing or reading. For more information on locks, see `http://docs.mongodb.org/manual/faq/concurrency`.

writeBacks The process within the sharding system that ensures that writes issued to a *shard* that *is not* responsible for the relevant chunk get applied to the proper shard. For related information, see *faq-writebacklisten* and *writeBacksQueued* (page 379).

CHAPTER 7

Release Notes

Always install the latest, stable version of MongoDB. See *release-version-numbers* for more information.

See the following release notes for an account of the changes in major versions. Release notes also include instructions for upgrade.

7.1 Current Development Release

(*2.8-series*)

7.1.1 Development Release Notes for MongoDB 2.8.0 Release Candidates

MongoDB 2.8 is currently in development. While 2.8 release candidates are currently available, these versions of MongoDB are for **testing only and not for production use**.

Major Changes

Storage Engines: Pluggable API, Improved Concurrency, Document-Level Locking with Compression

MongoDB 2.8 includes a pluggable storage engine API that allows third parties to develop storage engines with MongoDB.

MongoDB 2.8 includes support for two storage engines: MMAPv1, the storage engine available in previous versions of MongoDB, and WiredTiger[1]. By default, in 2.8 MongoDB uses the MMAPv1 engine.

The 2.8 MMAPv1 storage engine adds support for collection-level locking by default.

The 2.8 WiredTiger storage engine provides document-level locking and compression. WiredTiger supports all MongoDB features and can interoperate with MMAPv1 in the same replica set or sharded cluster, but requires a change to the on-disk storage format. To convert an existing data set to WiredTiger, you can either:

- upgrade replica sets in a rolling manner. See *Upgrade a Replica Set to 2.8* (page 719) or *Upgrade a Sharded Cluster to 2.8* (page 720), or
- use `mongodump` (page 622) and `mongorestore` (page 628) to import data.

To enable WiredTiger, start MongoDB with the following option:

[1] http://wiredtiger.com

```
mongod --storageEngine wiredTiger
```

If you attempt to start a mongod (page 583) with a storage.dbPath that contains data files produced by a storage engine other than the one specified by --storageEngine, mongod (page 583) will refuse to start.

The behavior and properties of the wiredTiger storage engine are configurable using the following storage.wiredTiger configuration options. You can set *WiredTiger options on the command line*.

All existing server, database, and collection reporting exposes statistics from WiredTiger, including: db.serverStatus() (page 124), db.stats() (page 126), db.collection.stats() (page 71), and db.collection.validate() (page 79).

WiredTiger compresses collection data by default using snappy.

WiredTiger uses prefix compression on all indexes by default.

Increased Number of Replica Set Members

MongoDB replica sets now support larger maximum replica set sizes up to 50 members. As in earlier releases, replica sets may only have a maximum of 7 voting members.

Use the following drivers with larger replica sets:

- C# (.NET) Driver 1.10
- Java Driver 2.13
- Python Driver (PyMongo) 3.0+
- Ruby Driver 2.0+
- Node.JS Driver 2.0+

Because the C, C++, Perl, PHP, and earlier versions of the Ruby, Python, and Node.JS drivers discover and monitor replica set members serially, these drivers are not suitable for use with large replica sets.

Pymongo 3.0, the Ruby Driver 2.0, and the Node.JS Driver 2.0 are currently in development.

New Query Introspection System

MongoDB 2.8 includes a new query introspection system that provides an improved output format and a finer-grained introspection into both query plan and query execution.

For details, see the new db.collection.explain() (page 33) method and the new explain (page 354) command as well as the updated cursor.explain() (page 85) method.

For information on the format of the new output, see http://docs.mongodb.org/manual/reference/explain-results

Security Improvements

New SCRAM-SHA-1 Password Authentication Mechanism

MongoDB 2.8 adds a new SCRAM-SHA-1 password authentication mechanism.

Changes to the Localhost Exception

MongoDB 2.8 increases restrictions when using the *localhost-exception* to access MongoDB. For details, see *Localhost Exception Changed* (page 715).

Additional Changes

General Improvements

General Improvements in MongoDB 2.8 MongoDB 2.8 is currently in development. While 2.8 release candidates are currently available, these versions of MongoDB are for **testing only and not for production use**.

Replica Sets

Replica Set Step Down Behavior Changes The process that a *primary* member of a *replica set* uses to step down has the following changes:

- Before stepping down, `replSetStepDown` (page 301) will attempt to terminate long running user operations that would block the primary from stepping down, such as an index build, a write operation or a map-reduce job.
- The `replSetStepDown` (page 301) will wait for an electable secondary to catch up to the state of the primary before stepping down. Previously a primary would wait 10 seconds for a secondary to catch up before stepping down. This change will help prevent rollbacks.
- `replSetStepDown` (page 301) now allows users to specify a `secondaryCatchUpPeriodSecs` parameter to specify how long the primary should wait for a secondary to catch up before stepping down.

Other Replica Set Operational Changes

- Initial sync builds indexes more efficiently for each collection and applies oplog entries in batches using threads.
- Definition of *w: "majority"* write concern changed to mean majority of *voting* nodes.
- Stronger restrictions on `http://docs.mongodb.org/manual/reference/replica-configuration`. For details, see *Replica Set Configuration Validation* (page 714).
- For pre-existing collections on secondary members, no longer build missing `_id` indexes automatically.

Sharded Clusters

Enhancement to Tag Management MongoDB 2.8 adds a new `sh.removeTagRange()` (page 188) helper to improve management of sharded collections with tags. The new `sh.removeTagRange()` (page 188) method acts as a complement to `sh.addTagRange()` (page 184).

More Predictable Read Preference Behavior `mongos` (page 601) instances no longer pin connections to members of replica sets when performing read operations. Instead, `mongos` (page 601) reevaluates `read preferences` for every operation to provide a more predictable read preference behavior when read preferences change.

Configurable Write Concern for Chunk Migration MongoDB 2.8 adds a new `writeConcern` setting to configure the `write concern` of chunk migration operations. You can configure the `writeConcern` setting for the *balancer* as well as for `moveChunk` (page 310) and `cleanupOrphaned` (page 305) commands.

Improvements

Enhanced Logging To improve usability of the log messages for diagnosis, MongoDB categorizes some log messages under specific components, or operations, and provides the ability to set the verbosity level for these components. For information, see `http://docs.mongodb.org/manual/reference/log-messages`.

MongoDB Tools Enhancements All MongoDB tools are now written in go and maintained as a separate project.

- Smaller binaries.
- New options for parallelized `mongodump` (page 622) and `mongorestore` (page 628). You can control the number of collections that `mongorestore` (page 628) will restore at a time with the `--numParallelCollections` option.
- New options `-excludeCollection` and `--excludeCollectionsWithPrefix` for `mongodump` (page 622) to exclude collections
- `mongorestore` (page 628) can now accept BSON data input from standard input in addition to reading BSON data from file.
- `mongostat` (page 657) and `mongotop` (page 664) can now return output in JSON format with the `--json` option.
- Added configurable *write concern* to `mongoimport` (page 642), `mongorestore` (page 628), and `mongofiles` (page 674). Use the `--writeConcern` option.
- `mongofiles` (page 674) now allows you to configure the GridFS prefix with the `--prefix` option so that you can use custom namespaces and store multiple GridFS namespaces in a single database.

Covered Queries For sharded collections, indexes can now cover queries that execute against the primary.

Geospatial Enhancement Add support for "big" polygons for `$geoIntersects` (page 423) and `$geoWithin` (page 425) queries. "Big" polygons are single-ringed GeoJSON polygons with areas greater than that of a single hemisphere. See `$geometry` (page 434), `$geoIntersects` (page 423), and `$geoWithin` (page 425) for details.

See also:

2d Indexes and Geospatial Near Queries (page 716)

Aggregation Enhancement MongoDB 2.8 adds a new `$dateToString` (page 535) operator to facilitate converting a date to a formatted string.

`$eq` Query Operator MongoDB 2.8 adds a `$eq` operator to query for equality conditions.

Operational Changes

Background Indexes Background index builds will no longer automatically interrupt if `dropDatabase` (page 334), `dropIndexes` (page 335), `drop` (page 335) occur. The `dropDatabase` (page 334), `dropIndexes` (page 335), `drop` (page 335) commands will still fail with `background job in progress`, as in 2.6

Parallel index builds with createIndexes cmd

`mmapv1` Record Allocation Behavior Changed The default allocation strategy for collections in instances that use `mmapv1` is `powerOf2Sizes`. To remove all record padding set `noPadding` (page 322) or specify the `noPadding` option to `db.createCollection()` (page 104).

Windows Performance Improvements

Security Improvements

SSL Certificate Validation By default, MongoDB instances will *only* start if its certificate (i.e. `net.ssl.PemKeyFile`) is valid. For details, see *SSL Certificates Validation* (page 716).

SSL Certificate Hostname Validation MongoDB 2.8 adds a new `--sslAllowInvalidHostnames` option to disable the validation of the hostname specified in the SSL certificate. For details, see *SSL Certificate Hostname Validation* (page 716).

MongoDB Enterprise Features

Audit Enhancements `http://docs.mongodb.org/manual/core/auditing` in MongoDB Enterprise can filter on CRUD operations. For information, see *audit-filter*.

MongoDB 2.8 introduces numerous enhancements, such as:

- Changes to the step down behavior for replica sets;
- More efficient index builds during initial sync;
- Configurable write concern for chunk migration;
- Enhanced logging; and
- Changes to MongoDB tools.

For details on these and additional improvements, see *General Improvements in MongoDB 2.8* (page 711).

Changes Affecting Compatibility

Compatibility Changes in MongoDB 2.8 MongoDB 2.8 is currently in development. While 2.8 release candidates are currently available, these versions of MongoDB are for **testing only and not for production use**.

Storage Engine

Configuration File Options With the introduction of additional storage engines in 2.8, some `configuration file options` have changed:

Previous Setting	New Setting
storage.journal.commitIntervalMs	storage.mmapv1.journal.commitIntervalMs
storage.journal.debugFlags	storage.mmapv1.journal.debugFlags
storage.nsSize	storage.mmapv1.nsSize
storage.preallocDataFiles	storage.mmapv1.preallocDataFiles
storage.quota.enforced	storage.mmapv1.quota.enforced
storage.quota.maxFilesPerDB	storage.mmapv1.quota.maxFilesPerDB
storage.smallFiles	storage.mmapv1.smallFiles
storage.syncPeriodSecs	storage.mmapv1.syncPeriodSecs

2.8 mongod (page 583) instances are backwards compatible with existing configuration files, but will issue warnings when if you attempt to use the old settings.

Data Files Must Correspond to Configured Storage Engine The files in the dbPath directory must correspond to the configured storage engine (i.e. --storageEngine). mongod (page 583) will not start if dbPath contains data files created by a storage engine other than the one specified by --storageEngine.

Support for `touch` Command If a storage engine does not support the touch (page 344), then the touch (page 344) command will return an error:

- The WiredTiger storage engine *does not* support the touch (page 344).
- The MMAPv1 storage engine supports touch (page 344).

Dynamic Record Allocation MongoDB 2.8 no longer supports dynamic record allocation and deprecates paddingFactor. For mroe information, see *mmapv1 Record Allocation Behavior Changed* (page 713).

Replication Changes

Replica Set State Change The FATAL replica set state does not exist as of 2.8.0.

Replica Set Oplog Format Change MongoDB 2.8 is not compatible with oplog entries generated by versions of MongoDB before 2.2.1. If you upgrade from one of these versions, you must wait for new oplog entries to overwrite *all* old oplog entries generated by one of these versions before upgrading to 2.8.0 or earlier.

Secondaries may abort if they replay an oplog from 2.4 or prior with an index build operation that would normally fail on a 2.6 or later primary.

Replica Set Configuration Validation MongoDB now provides more strict validation of replica set configuration objects. Of particular note are the following alterations:

- Arbiters may only have 1 vote. Previously arbiters could have 0 votes, which is no longer supported. You must fix on the primary and restart node.
- Nodes can **only** have votes value of 0 or 1. Will fail to load. Must fix on primary and restart node
- http://docs.mongodb.org/manual/reference/replica-configuration must specify the same _id name as that specified by --replSet or replication.replSetName;
- Unrecognized configuration fields produce an invalid config and an error. Previously, ignored these fields. For example, tokutek adds a protocolVersion field, which will now result in error.
- Disallows getLastErrorDefaults: 0 in a config.

Remove `local.slaves` Collection MongoDB 2.8 removes the `local.slaves` collection that tracked the slaves' replication progress. To track slave's replication progress, refer to the `serverStatus.repl` (page 374) section of `serverStatus` (page 366).

MongoDB Tools Changes

Require a Running MongoDB Instance The 2.8 versions of MongoDB tools, `mongodump` (page 622), `mongorestore` (page 628), `mongoexport` (page 649), `mongoimport` (page 642), `mongofiles` (page 674), and `mongooplog` (page 637), must connect to running MongoDB instances and *cannot* modify MongoDB data files as in previous versions.

Removed Options

- Removed `--dbpath` and `--filter` options for `mongorestore` (page 628), `mongoimport` (page 642), `mongoexport` (page 649), and `bsondump` (page 635).
- Removed `--locks` option for `mongotop` (page 664).

See also:

MongoDB Tools Enhancements (page 712)

Sharded Cluster Setting

Remove `releaseConnectionsAfterResponse` Parameter MongoDB now always releases connections after response. `releaseConnectionsAfterResponse` parameter is no longer available.

Security Changes

MongoDB 2.4 User Model Removed After deprecating the 2.4 user model in 2.6, MongoDB 2.8 completely removes support for the 2.4 user model. MongoDB will exit with an error message there is user data with the 2.4 schema. If your deployment still uses the 2.4 user model, see *Upgrade User Authorization Data to 2.6 Format* (page 764) to upgrade to the 2.6 user model before upgrading to 2.8.

Localhost Exception Changed In 2.8, the localhost exception changed so that these connections *only* have access to create the first user on the `admin` database. In previous versions, connections that gained access using the localhost exception had unrestricted access to the MongoDB instance.

See *localhost-exception* for more information.

`db.addUser()` Removed 2.8 removes the legacy `db.addUser()` method. Use `db.createUser()` (page 152) and `db.updateUser()` (page 159) instead.

SSL Configuration Option Changes MongoDB 2.8 introduced new `net.ssl.allowConnectionsWithoutCertificates` configuration file setting and `--sslAllowConnectionsWithoutCertificates` command line option for `mongod` (page 583) and `mongos` (page 601). These options replace previous `net.ssl.weakCertificateValidation` and `--sslWeakCertificateValidation` options, which became aliases. Update your configuration to ensure future compatibility.

SSL Certificates Validation By default, MongoDB instances will *only* start if its certificate (i.e. `net.ssl.PemKeyFile`) is valid. You can disable this behavior with the `net.ssl.allowInvalidCertificates` setting or the `--sslAllowInvalidCertificates` command line option.

SSL Certificate Hostname Validation By default, MongoDB validates the hostnames of hosts attempting to connect using certificates against the hostnames listed in those certificates. In certain deployment situations this behavior may be undesirable. It is now possible to disable such hostname validation without disabling validation of the rest of the certificate information with the `net.ssl.allowInvalidHostnames` setting or the `--sslAllowInvalidHostnames` command line option.

SSLv3 Ciphers Disabled In light of vulnerabilities in legacy SSL ciphers[2], these ciphers have been explicitly disabled in MongoDB. No configuration changes are necessary.

`mongo` Shell Version Compatibility Versions of the `mongo` (page 610) shell before 2.8 are *not* compatible with 2.8 deployments of MongoDB that enforce access control. If you have a 2.8 MongoDB deployment that requires access control, you must use 2.8 versions of the `mongo` (page 610) shell.

Indexes

Remove `dropDups` Option `dropDups` option is no longer available for `ensureIndex()` (page 30) and `createIndexes` (page 330).

Changes to Restart Behavior during Background Indexing For 2.8 `mongod` (page 583) instances, if a background index build is in progress when the `mongod` (page 583) process terminates, when the instance restarts the index build will restart as foreground index build. If the index build encounters any errors, such as a duplicate key error, the `mongod` (page 583) will exit with an error.

To start the `mongod` (page 583) after a failed index build, use the `storage.indexBuildRetry` or *`--noIndexBuildRetry`* to skip the index build on start up.

`2d` Indexes and Geospatial Near Queries For `$near` (page 429) queries that use a `2d` index:

- MongoDB no longer uses a default limit of 100 documents.
- Specifying a `batchSize()` (page 82) is no longer analogous to specifying a `limit()` (page 88).

For `$nearSphere` (page 428) queries that use a `2d` index, MongoDB no longer uses a default limit of 100 documents.

`mongo` Shell Version Compatibility

WiredTiger and Previous Versions For 2.8 MongoDB deployments using the WiredTiger storage engine, the following operations return no output when issued in previous versions of the `mongo` (page 610) shell:

- `db.getCollectionNames()` (page 114)
- `db.collection.getIndexes()` (page 48)
- `show collections`

[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566

- `show tables`

Use the 2.8 `mongo` (page 610) shell when connecting to 2.8 `mongod` (page 583) instances that use WiredTiger.

General Compatibility Changes

Deprecate Access to `system.indexes` and `system.namespaces` MongoDB 2.8 deprecates *direct* access to `system.indexes` and `system.namespaces` collections. Use the `createIndexes` (page 330) and `listIndexes` (page 338) commands instead.

Collection Name Validation MongoDB 2.8 more consistently enforces the `collection naming restrictions` (page 697). Ensure your application does not create or depend on invalid collection names.

Platform Support No longer provides commercial support for MongoDB on Linux32 and Win32 platforms; however, will continue to build the MongoDB distributions for the platforms.

Removed/Deprecated Commands The following commands are no longer available in MongoDB 2.8:

- `closeAllDatabases`
- `getoptime`

The following command is deprecated in MongoDB 2.8:

- `diagLogging` (page 354)

Some changes in 2.8 can affect *compatibility* (page 713) and may require user actions. For a detailed list of compatibility changes, see *Compatibility Changes in MongoDB 2.8* (page 713).

Upgrade Process

Upgrade MongoDB to 2.8

MongoDB 2.8 is currently in development. While 2.8-release-candidates are currently available, these versions of MongoDB are for **testing only and not for production use**.

In the general case, the upgrade from MongoDB 2.6 to 2.8 is a binary-compatible "drop-in" upgrade: shut down the `mongod` (page 583) instances and replace them with `mongod` (page 583) instances running 2.8. **However**, before you attempt any upgrade please familiarize yourself with the content of this document, particularly the procedure for *upgrading sharded clusters* (page 720).

Upgrade Recommendations and Checklists When upgrading, consider the following:

Upgrade Requirements To upgrade an existing MongoDB deployment to 2.8, you must be running 2.6. If you're running a version of MongoDB before 2.6, you *must* upgrade to 2.6 before upgrading to 2.8. See *Upgrade MongoDB to 2.6* (page 760) for the procedure to upgrade from 2.4 to 2.6. Once upgraded to MongoDB 2.6, you **cannot** downgrade to any version earlier than MongoDB 2.4.

Preparedness Before upgrading MongoDB, always test your application in a staging environment before deploying the upgrade to your production environment.

Downgrade Limitations Once upgraded to MongoDB 2.8, you **cannot** downgrade to a version lower than **2.6.5**.

If you upgrade to 2.8 and have run `authSchemaUpgrade` (page 263), you **cannot** downgrade to 2.6 without disabling `--auth` (page 587).

Package Upgrades If you installed MongoDB from the MongoDB `apt` or `yum` repositories, upgrade to 2.8 using the package manager.

For Debian, Ubuntu, and related operating systems, type these commands:

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

For Red Hat Enterprise, CentOS, Fedora, or Amazon Linux:

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

If you did not install the `mongodb-org` package, and installed a subset of MongoDB components replace `mongodb-org` in the commands above with the appropriate package names.

See installation instructions for `Ubuntu`, `RHEL`, `Debian`, or `other Linux Systems` for a list of the available packages and complete MongoDB installation instructions.

Upgrade MongoDB Processes

Upgrade Standalone `mongod` Instance to MongoDB 2.8

Upgrade Binaries The following steps outline the procedure to upgrade a standalone `mongod` (page 583) from version 2.6 to 2.8. To upgrade from version 2.4 to 2.8, *upgrade to version 2.6* (page 760) *first*, and then follow the procedure to upgrade from 2.6 to 2.8.

Step 1: Download 2.8 binaries. Download binaries of the latest release in the 2.8 series from the MongoDB Download Page[3]. See `http://docs.mongodb.org/manual/installation` for more information.

Step 2: Replace 2.6 binaries. Shut down your `mongod` (page 583) instance. Replace the existing binary with the 2.8 `mongod` (page 583) binary and restart `mongod` (page 583).

Change Storage Engine to WiredTiger To change storage engine to WiredTiger, you will need to manually export and upload the data using `mongodump` (page 622) and `mongorestore` (page 628).

Step 1: Start 2.8 `mongod`. Ensure that the 2.8 `mongod` (page 583) is running with the default MMAPv1 engine.

Step 2: Export the data using `mongodump`.

```
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See `mongodump` (page 622) for available options.

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

Step 3: Create data directory for WiredTiger. Create a new data directory for WiredTiger. Ensure that the user account running `mongod` (page 583) has read and write permissions for the new directory.

`mongod` (page 583) with WiredTiger will not start with data files created with a different storage engine.

Step 4: Restart the `mongod` with WiredTiger. Restart the 2.8 `mongod` (page 583), specifying WiredTiger as the *`--storageEngine`* (page 589) and the newly created data directory for WiredTiger as the *`--dbpath`* (page 589).

```
mongod --storageEngine wiredTiger --dbpath <newWiredTigerDBPath>
```

Specify additional options as appropriate.

Step 5: Upload the exported data using `mongorestore`.

```
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` (page 628) for available options.

Upgrade a Replica Set to 2.8

Note: If the oplog contains entries generated by versions of MongoDB that precede version 2.2.1, you must wait for the entries to be overwritten by later versions *before* you can upgrade to MongoDB 2.8. See also *Replica Set Oplog Format Change* (page 714).

To upgrade a replica set from MongoDB 2.6 to 2.8, *upgrade all members of the replica set to version 2.6* (page 760) *first*, and then follow the procedure to upgrade from MongoDB 2.6 to 2.8.

Upgrade Binaries You can upgrade from MongoDB 2.6 to 2.8 using a "rolling" upgrade to minimize downtime by upgrading the members individually while the other members are available:

Step 1: Upgrade secondary members of the replica set. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` (page 583) and replacing the 2.6 binary with the 2.8 binary. After upgrading a `mongod` (page 583) 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 179) in the `mongo` (page 610) shell.

Step 2: Step down the replica set primary. Use `rs.stepDown()` (page 179) in the `mongo` (page 610) shell to step down the *primary* and force the set to *failover*. `rs.stepDown()` (page 179) expedites the failover procedure and is preferable to shutting down the primary directly.

Step 3: Upgrade the primary. When `rs.status()` (page 179) shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` (page 583) binary with the 2.8 binary and start the new instance.

Replica set failover is not instant and will render the set unavailable to accept writes until the failover process completes. This may take 30 seconds or more: schedule the upgrade procedure during a scheduled maintenance window.

Change Storage Engine to WiredTiger In MongoDB 2.8, replica sets can have members with different storage engines. As such, you can update members to use the WiredTiger storage engine in a rolling manner.

Note: Before changing all the members to use WiredTiger, you may prefer to run with mixed storage engines for some period. However, performance can vary according to workload.

To change the storage engine to WiredTiger for an existing secondary replica set member, remove the member's data and perform an `initial sync`:

Step 1: Shutdown the secondary member. Stop the `mongod` (page 583) instance for the secondary member.

Step 2: Prepare data directory for WiredTiger. `mongod` (page 583) with WiredTiger will not start if the *`--dbpath`* (page 589) directory contains data files created with a different storage engine.

For the stopped secondary member, either delete the content of the data directory or create a new data directory. If creating a new directory, ensure that the user account running `mongod` (page 583) has read and write permissions for the new directory.

Step 3: Restart the secondary member with WiredTiger. Restart the 2.8 `mongod` (page 583), specifying WiredTiger as the *`--storageEngine`* (page 589) and the data directory for WiredTiger as the *`--dbpath`* (page 589). Specify additional options as appropriate for the member.

```
mongod --storageEngine wiredTiger --dbpath <newWiredTigerDBPath>
```

Since no data exists in the `--dbpath`, the `mongod` (page 583) will perform an initial sync. The length of the initial sync process depends on the size of the database and network connection between members of the replica set.

To update all members of the replica set to use WiredTiger, update the the secondary members first. Then step down the primary, and update the stepped-down member.

Upgrade a Sharded Cluster to 2.8 Only upgrade sharded clusters to 2.8 if **all** members of the cluster are currently running instances of 2.6. The only supported upgrade path for sharded clusters running 2.4 is via 2.6. The upgrade process checks all components of the cluster and will produce warnings if any component is running version 2.4.

Considerations The upgrade process does not require any downtime. However, while you upgrade the sharded cluster, ensure that clients do not make changes to the collection meta-data. For example, during the upgrade, do **not** do any of the following:

- `sh.enableSharding()` (page 186)
- `sh.shardCollection()` (page 189)
- `sh.addShard()` (page 183)
- `db.createCollection()` (page 104)
- `db.collection.drop()` (page 28)
- `db.dropDatabase()` (page 111)
- any operation that creates a database
- any other operation that modifies the cluster metadata in any way. See `http://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `http://docs.mongodb.org/manual/reference/sharding` page modifies the cluster meta-data.

Upgrade Sharded Clusters *Optional but Recommended.* As a precaution, take a backup of the `config` database *before* upgrading the sharded cluster.

Step 1: Disable the Balancer. Turn off the *balancer* in the sharded cluster, as described in *sharding-balancing-disable-temporarily*.

Step 2: Upgrade the cluster's meta data. Start a single 2.8 mongos (page 601) instance with the configDB pointing to the cluster's config servers and with the *--upgrade* option.

To run a mongos (page 601) with the *--upgrade* option, you can upgrade an existing mongos (page 601) instance to 2.8, or if you need to avoid reconfiguring a production mongos (page 601) instance, you can use a new 2.8 mongos (page 601) that can reach all the config servers.

To upgrade the meta data, run:

```
mongos --configdb <configDB string> --upgrade
```

You can include the *--logpath* option to output the log messages to a file instead of the standard output. Also include any other options required to start mongos (page 601) instances in your cluster, such as *--sslOnNormalPorts* or *--sslPEMKeyFile*.

The mongos (page 601) will exit upon completion of the *--upgrade* process.

The upgrade will prevent any chunk moves or splits from occurring during the upgrade process. If the data files have many sharded collections or if failed processes hold stale locks, acquiring the locks for all collections can take seconds or minutes. Watch the log for progress updates.

Step 3: Ensure `mongos --upgrade` process completes successfully. The mongos (page 601) will exit upon completion of the meta data upgrade process. If successful, the process will log the following messages:

```
upgrade of config server to v6 successful
Config database is at version v6
```

After a successful upgrade, restart the mongos (page 601) instance. If mongos (page 601) fails to start, check the log for more information.

If the mongos (page 601) instance loses its connection to the config servers during the upgrade or if the upgrade is otherwise unsuccessful, you may always safely retry the upgrade.

Step 4: Upgrade the remaining `mongos` instances to v2.8. Upgrade and restart **without** the *--upgrade* (page 593) option the other mongos (page 601) instances in the sharded cluster.

After you have successfully upgraded *all* mongos (page 601) instances, you can proceed to upgrade the other components in your shareded cluster.

> **Warning:** Do not upgrade the mongod (page 583) instances until after you have upgraded *all* the mongos (page 601) instances.

Step 5: Upgrade the config servers. After you have successfully upgraded *all* mongos (page 601) instances, upgrade all 3 mongod (page 583) config server instances, leaving the *first* config server listed in the *mongos --configdb* (page 604) argument to upgrade *last*.

Step 6: Upgrade the shards. Upgrade each shard, one at a time, upgrading the mongod (page 583) secondaries before running replSetStepDown (page 301) and upgrading the primary of each shard.

Step 7: Re-enable the balancer. Once the upgrade of sharded cluster components is complete, *Re-enable the balancer*.

To change the storage engine to WiredTiger, refer to the procedure in *Change Storage Engine to WiredTiger for replica sets* (page 719) and *Change Storage Engine to WiredTiger for standalone mongod* (page 718) .

General Upgrade Procedure **Except** as described on this page, moving between 2.6 and 2.8 is a drop-in replacement:

Step 1: Stop the existing `mongod` instance. For example, on Linux, run 2.6 `mongod` (page 583) with the `--shutdown` (page 589) option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath`. See also the *terminate-mongod-processes* for alternate methods of stopping a `mongod` (page 583) instance.

Step 2: Start the new `mongod` instance. Ensure you start the 2.8 `mongod` (page 583) with the same `dbPath`:

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath`.

Downgrade MongoDB from 2.8

Before you attempt any downgrade, familiarize yourself with the content of this document, particularly the *Downgrade Recommendations and Checklist* (page 722) and the procedure for *downgrading sharded clusters* (page 725).

Downgrade Recommendations and Checklist When downgrading, consider the following:

Downgrade Path Once upgraded to MongoDB 2.8, you **cannot** downgrade to a version lower than **2.6.5**.

Important: If you upgrade to 2.8 and have run `authSchemaUpgrade` (page 263), you **cannot** downgrade to 2.6 without disabling `--auth` (page 587).

Procedures Follow the downgrade procedures:

- To downgrade sharded clusters, see *Downgrade a 2.8 Sharded Cluster* (page 725).
- To downgrade replica sets, see *Downgrade a 2.8 Replica Set* (page 723).
- To downgrade a standalone MongoDB instance, see *Downgrade a Standalone mongod Instance* (page 722).

Downgrade MongoDB Processes

Downgrade a Standalone `mongod` Instance If you have changed the storage engine to `WiredTiger`, change the storage engine to MMAPv1 before downgrading to 2.6.

Change Storage Engine to MMAPv1 To change storage engine to MMAPv1 for a standalone `mongod` (page 583) instance, you will need to manually export and upload the data using `mongodump` (page 622) and `mongorestore` (page 628).

Step 1: Ensure 2.8 `mongod` is running with WiredTiger.

Step 2: Export the data using `mongodump`.

```
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See `mongodump` (page 622) for available options.

Step 3: Create data directory for MMAPv1. Create a new data directory for MMAPv1. Ensure that the user account running `mongod` (page 583) has read and write permissions for the new directory.

Step 4: Restart the `mongod` with MMAPv1. Restart the 2.8 `mongod` (page 583), specifying the newly created data directory for MMAPv1 as the *`--dbpath`* (page 589). You do not have to specify *`--storageEngine`* (page 589) as MMAPv1 is the default.

```
mongod --dbpath <newMMAPv1DBPath>
```

Specify additional options as appropriate.

Step 5: Upload the exported data using `mongorestore`.

```
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` (page 628) for available options.

Downgrade Binaries The following steps outline the procedure to downgrade a standalone `mongod` (page 583) from version 2.8 to 2.6.

Once upgraded to MongoDB 2.8, you **cannot** downgrade to a version lower than **2.6.5**.

Step 1: Download 2.6 binaries. Download binaries of the latest release in the 2.6 series from the MongoDB Download Page[4]. See `http://docs.mongodb.org/manual/installation` for more information.

Step 2: Replace with 2.6 binaries. Shut down your `mongod` (page 583) instance. Replace the existing binary with the 2.6 `mongod` (page 583) binary and restart `mongod` (page 583).

Downgrade a 2.8 Replica Set If you have changed the storage engine to `WiredTiger`, change the storage engine to MMAPv1 before downgrading to 2.6.

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

Change Storage Engine to MMAPv1 You can update members to use the MMAPv1 storage engine in a rolling manner.

Note: When running a replica set with mixed storage engines, performance can vary according to workload.

To change the storage engine to MMAPv1 for an existing secondary replica set member, remove the member's data and perform an `initial sync`:

Step 1: Shutdown the secondary member. Stop the `mongod` (page 583) instance for the secondary member.

Step 2: Prepare data directory for MMAPv1. Prepare *`--dbpath`* (page 589) directory for initial sync.

For the stopped secondary member, either delete the content of the data directory or create a new data directory. If creating a new directory, ensure that the user account running `mongod` (page 583) has read and write permissions for the new directory.

Step 3: Restart the secondary member with MMAPv1. Restart the 2.8 `mongod` (page 583), specifying the MMAPv1 data directory as the *`--dbpath`* (page 589). Specify additional options as appropriate for the member. You do not have to specify *`--storageEngine`* (page 589) since MMAPv1 is the default.

```
mongod --dbpath <preparedMMAPv1DBPath>
```

Since no data exists in the `--dbpath`, the `mongod` (page 583) will perform an initial sync. The length of the initial sync process depends on the size of the database and network connection between members of the replica set.

Repeat for the remaining the secondary members. Once all the secondary members have switched to MMAPv1, step down the primary, and update the stepped-down member.

Downgrade Binaries Once upgraded to MongoDB 2.8, you **cannot** downgrade to a version lower than **2.6.5**.

The following steps outline a "rolling" downgrade process for the replica set. The "rolling" downgrade process minimizes downtime by downgrading the members individually while the other members are available:

Step 1: Downgrade secondary members of the replica set. Downgrade each *secondary* member of the replica set, one at a time:

1. Shut down the `mongod` (page 583). See *terminate-mongod-processes* for instructions on safely terminating `mongod` (page 583) processes.
2. Replace the 2.8 binary with the 2.6 binary and restart.
3. Wait for the member to recover to `SECONDARY` state before downgrading the next secondary. To check the member's state, use the `rs.status()` (page 179) method in the `mongo` (page 610) shell.

Step 2: Step down the primary. Use `rs.stepDown()` (page 179) in the `mongo` (page 610) shell to step down the *primary* and force the normal *failover* procedure.

```
rs.stepDown()
```

`rs.stepDown()` (page 179) expedites the failover procedure and is preferable to shutting down the primary directly.

Step 3: Replace and restart former primary `mongod`. When `rs.status()` (page 179) shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` (page 583) binary with the 2.6 binary and start the new instance.

Replica set failover is not instant but will render the set unavailable to writes and interrupt reads until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the downgrade during a predetermined maintenance window.

Downgrade a 2.8 Sharded Cluster

Requirements While the downgrade is in progress, you cannot make changes to the collection meta-data. For example, during the downgrade, do **not** do any of the following:

- `sh.enableSharding()` (page 186)
- `sh.shardCollection()` (page 189)
- `sh.addShard()` (page 183)
- `db.createCollection()` (page 104)
- `db.collection.drop()` (page 28)
- `db.dropDatabase()` (page 111)
- any operation that creates a database
- any other operation that modifies the cluster meta-data in any way. See `http://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `http://docs.mongodb.org/manual/reference/sharding` page modifies the cluster meta-data.

Change Storage Engine to MMAPv1 If you have changed the storage engine to `WiredTiger`, change the storage engine to MMAPv1 before downgrading to 2.6.

To change the storage engine to MMAPv1, refer to the procedure in *Change Storage Engine to MMAPv1 for replica set members* (page 724) and *Change Storage Engine to MMAPv1 for standalone mongod* (page 723).

Downgrade Binaries Once upgraded to MongoDB 2.8, you **cannot** downgrade to a version lower than **2.6.5**.

The downgrade procedure for a sharded cluster reverses the order of the upgrade procedure.

Step 1: Disable the Balancer. Turn off the *balancer* in the sharded cluster, as described in *sharding-balancing-disable-temporarily*.

Step 2: Downgrade each shard, one at a time. For each shard:

1. Downgrade the `mongod` (page 583) secondaries *before* downgrading the primary.
2. To downgrade the primary, run `replSetStepDown` (page 301) and downgrade.

Step 3: Downgrade the config servers. Downgrade all 3 `mongod` (page 583) config server instances, leaving the *first* system in the `mongos --configdb` argument to downgrade *last*.

Step 4: Downgrade the `mongos` instances. Downgrade and restart each `mongos` (page 601), one at a time. The downgrade process is a binary drop-in replacement.

Step 5: Re-enable the balancer. Once the upgrade of sharded cluster components is complete, *re-enable the balancer*.

General Downgrade Procedure **Except** as described on this page, moving between 2.6 and 2.8 is a drop-in replacement:

Step 1: Stop the existing `mongod` instance. For example, on Linux, run 2.8 `mongod` (page 583) with the `--shutdown` (page 589) option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath`. See also the *terminate-mongod-processes* for alternate methods of stopping a `mongod` (page 583) instance.

Step 2: Start the new `mongod` instance. Ensure you start the 2.6 `mongod` (page 583) with the same `dbPath`:

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath`.

See *Upgrade MongoDB to 2.8* (page 717) for full upgrade instructions.

Download

To download MongoDB 2.8 release candidates, go to the downloads page[5].

Additional Resources

- All Third Party License Notices[6].

7.2 Current Stable Release

(*2.6-series*)

7.2.1 Release Notes for MongoDB 2.6

April 8, 2014

MongoDB 2.6 is now available. Key features include aggregation enhancements, text-search integration, query-engine improvements, a new write-operation protocol, and security enhancements.

MMS 1.4, which includes On-Prem Backup in addition to Monitoring, is now also available. See MMS 1.4 documentation[7] and the MMS 1.4 release notes[8] for more information.

[5] http://www.mongodb.org/downloads
[6] https://github.com/mongodb/mongo/blob/v2.8/distsrc/THIRD-PARTY-NOTICES
[7] https://mms.mongodb.com/help-hosted/v1.4/
[8] https://mms.mongodb.com/help-hosted/v1.4/management/changelog/

Minor Releases

2.6 Changelog

2.6.6 – Changes

Security

- SERVER-15673[9] Disable SSLv3 ciphers
- SERVER-15515[10] New test for mixed version replSet, 2.4 primary, user updates
- SERVER-15500[11] New test for system.user operations

Stability

- SERVER-12061[12] Do not silently ignore read errors when syncing a replica set node
- SERVER-12058[13] Primary should abort if encountered problems writing to the oplog

Querying

- SERVER-16291[14] Cannot set/list/clear index filters on the secondary
- SERVER-15958[15] The "isMultiKey" value is not correct in the output of aggregation explain plan
- SERVER-15899[16] Querying against path in document containing long array of subdocuments with nested arrays causes stack overflow
- SERVER-15696[17] `$regex` (page 414), `$in` (page 402) and `$sort` with index returns too many results
- SERVER-15639[18] Text queries can return incorrect results and leak memory when multiple predicates given on same text index prefix field
- SERVER-15580[19] Evaluating candidate query plans with concurrent writes on same collection may crash :program:"mongod
- SERVER-15528[20] Distinct queries can scan many index keys without yielding read lock
- SERVER-15485[21] CanonicalQuery::canonicalize can leak a LiteParsedQuery
- SERVER-15403[22] `$min` (page 559) and `$max` (page 558) equal errors in 2.6 but not in 2.4
- SERVER-15233[23] Cannot run `planCacheListQueryShapes` on a Secondary
- SERVER-14799[24] `count` (page 213) with hint doesn't work when hint is a document

[9] https://jira.mongodb.org/browse/SERVER-15673
[10] https://jira.mongodb.org/browse/SERVER-15515
[11] https://jira.mongodb.org/browse/SERVER-15500
[12] https://jira.mongodb.org/browse/SERVER-12061
[13] https://jira.mongodb.org/browse/SERVER-12058
[14] https://jira.mongodb.org/browse/SERVER-16291
[15] https://jira.mongodb.org/browse/SERVER-15958
[16] https://jira.mongodb.org/browse/SERVER-15899
[17] https://jira.mongodb.org/browse/SERVER-15696
[18] https://jira.mongodb.org/browse/SERVER-15639
[19] https://jira.mongodb.org/browse/SERVER-15580
[20] https://jira.mongodb.org/browse/SERVER-15528
[21] https://jira.mongodb.org/browse/SERVER-15485
[22] https://jira.mongodb.org/browse/SERVER-15403
[23] https://jira.mongodb.org/browse/SERVER-15233
[24] https://jira.mongodb.org/browse/SERVER-14799

Replication

- SERVER-16107[25] 2.6 `mongod` crashes with segfault when added to a 2.8 replica set with >= 12 nodes.
- SERVER-15994[26] `listIndexes` and `listCollections` can be run on secondaries without slaveOk bit
- SERVER-15849[27] do not forward replication progress for nodes that are no longer part of a replica set
- SERVER-15491[28] `SyncSourceFeedback` can crash due to a `SocketException` in `authenticateInternalUser`

Sharding

- SERVER-15318[29] `copydb` (page 326) should not use exhaust flag when used against `mongos` (page 601)
- SERVER-14728[30] Shard depends on string comparison of replica set connection string
- SERVER-14506[31] special top chunk logic can move max chunk to a shard with incompatible tag
- SERVER-14299[32] For sharded limit=N queries with sort, mongos can request >N results from shard
- SERVER-14080[33] Have migration result reported in the changelog correctly
- SERVER-12472[34] Fail MoveChunk if an index is needed on TO shard and data exists

Storage

- SERVER-16283[35] Can't start new wiredtiger node with log file or config file in data directory - false detection of old `mmapv1` files
- SERVER-15986[36] Starting with different storage engines in the same dbpath should error/warn
- SERVER-14057[37] Changing TTL expiration time with collMod does not correctly update index definition

Indexing and write Operations

- SERVER-14287[38] ensureIndex can abort reIndex and lose indexes
- SERVER-14886[39] Updates against paths composed with array index notation and positional operator fail with error

Data Aggregation SERVER-15552[40] Errors writing to temporary collections during `mapReduce` (page 220) command execution should be operation-fatal

[25] https://jira.mongodb.org/browse/SERVER-16107
[26] https://jira.mongodb.org/browse/SERVER-15994
[27] https://jira.mongodb.org/browse/SERVER-15849
[28] https://jira.mongodb.org/browse/SERVER-15491
[29] https://jira.mongodb.org/browse/SERVER-15318
[30] https://jira.mongodb.org/browse/SERVER-14728
[31] https://jira.mongodb.org/browse/SERVER-14506
[32] https://jira.mongodb.org/browse/SERVER-14299
[33] https://jira.mongodb.org/browse/SERVER-14080
[34] https://jira.mongodb.org/browse/SERVER-12472
[35] https://jira.mongodb.org/browse/SERVER-16283
[36] https://jira.mongodb.org/browse/SERVER-15986
[37] https://jira.mongodb.org/browse/SERVER-14057
[38] https://jira.mongodb.org/browse/SERVER-14287
[39] https://jira.mongodb.org/browse/SERVER-14886
[40] https://jira.mongodb.org/browse/SERVER-15552

Build and Packaging

- SERVER-14184[41] Unused preprocessor macros from s2 conflict on OS X Yosemite
- SERVER-14015[42] S2 Compilation on GCC 4.9/Solaris fails
- SERVER-16017[43] Suse11 enterprise packages fail due to unmet dependencies
- SERVER-15598[44] Ubuntu 14.04 Enterprise packages depend on unavailable libsnmp15 package
- SERVER-13595[45] Red Hat init.d script error: YAML config file parsing

Logging and Diagnostics

- SERVER-13471[46] Increase log level of "did reduceInMemory" message in map/reduce
- SERVER-16324[47] Command execution log line displays "`query not recording (too large)`" instead of abbreviated command object
- SERVER-10069[48] Improve errorcodes.py so it captures multiline messages

Testing and Internals

- SERVER-15632[49] `MultiHostQueryOp::PendingQueryContext::doBlockingQuery` can leak a cursor object
- SERVER-15629[50] `GeoParser::parseMulti{Line|Polygon}` does not clear objects owned by out parameter
- SERVER-16316[51] Remove unsupported behavior in shard3.js
- SERVER-14763[52] Update jstests/sharding/split_large_key.js
- SERVER-14249[53] Add tests for querying oplog via mongodump using –dbpath
- SERVER-13726[54] indexbg_drop.js

2.6.5 – Changes

Security

- SERVER-15465[55] OpenSSL crashes on stepdown
- SERVER-15360[56] User document changes made on a 2.4 primary and replicated to a 2.6 secondary don't make the 2.6 secondary invalidate its user cache

[41] https://jira.mongodb.org/browse/SERVER-14184
[42] https://jira.mongodb.org/browse/SERVER-14015
[43] https://jira.mongodb.org/browse/SERVER-16017
[44] https://jira.mongodb.org/browse/SERVER-15598
[45] https://jira.mongodb.org/browse/SERVER-13595
[46] https://jira.mongodb.org/browse/SERVER-13471
[47] https://jira.mongodb.org/browse/SERVER-16324
[48] https://jira.mongodb.org/browse/SERVER-10069
[49] https://jira.mongodb.org/browse/SERVER-15632
[50] https://jira.mongodb.org/browse/SERVER-15629
[51] https://jira.mongodb.org/browse/SERVER-16316
[52] https://jira.mongodb.org/browse/SERVER-14763
[53] https://jira.mongodb.org/browse/SERVER-14249
[54] https://jira.mongodb.org/browse/SERVER-13726
[55] https://jira.mongodb.org/browse/SERVER-15465
[56] https://jira.mongodb.org/browse/SERVER-15360

- SERVER-14887[57] Allow user document changes made on a 2.4 primary to replicate to a 2.6 secondary
- SERVER-14727[58] Details of SASL failures aren't logged
- SERVER-12551[59] Audit DML/CRUD operations

Stability SERVER-9032[60] mongod fails when launched with misconfigured locale

Querying

- SERVER-15287[61] Query planner sort analysis incorrectly allows index key pattern plugin fields to provide sort
- SERVER-15286[62] Assertion in date indexes when opposite-direction-sorted and double "or" filtered
- SERVER-15279[63] Disable hash-based index intersection (AND_HASH) by default
- SERVER-15152[64] When evaluating plans, some index candidates cause complete index scan
- SERVER-15015[65] Assertion failure when combining `$max` (page **??**) and `$min` (page **??**) and reverse index scan
- SERVER-15012[66] Server crashes on indexed rooted $or queries using a 2d index
- SERVER-14969[67] Dropping index during active aggregation operation can crash server
- SERVER-14961[68] Plan ranker favors intersection plans if predicate generates empty range index scan
- SERVER-14892[69] Invalid `{$elemMatch: {$where}}` query causes memory leak
- SERVER-14706[70] Queries that use negated $type predicate over a field may return incomplete results when an index is present on that field
- SERVER-13104[71] Plan enumerator doesn't enumerate all possibilities for a nested `$or` (page 408)
- SERVER-14984[72] Server aborts when running `$centerSphere` (page 432) query with `NaN` radius
- SERVER-14981[73] Server aborts when querying against `2dsphere` index with `coarsestIndexedLevel:0`
- SERVER-14831[74] Text search trips assertion when default language only supported in `textIndexVersion=1` used

Replication

- SERVER-15038[75] Multiple background index builds may not interrupt cleanly for commands, on secondaries

[57] https://jira.mongodb.org/browse/SERVER-14887
[58] https://jira.mongodb.org/browse/SERVER-14727
[59] https://jira.mongodb.org/browse/SERVER-12551
[60] https://jira.mongodb.org/browse/SERVER-9032
[61] https://jira.mongodb.org/browse/SERVER-15287
[62] https://jira.mongodb.org/browse/SERVER-15286
[63] https://jira.mongodb.org/browse/SERVER-15279
[64] https://jira.mongodb.org/browse/SERVER-15152
[65] https://jira.mongodb.org/browse/SERVER-15015
[66] https://jira.mongodb.org/browse/SERVER-15012
[67] https://jira.mongodb.org/browse/SERVER-14969
[68] https://jira.mongodb.org/browse/SERVER-14961
[69] https://jira.mongodb.org/browse/SERVER-14892
[70] https://jira.mongodb.org/browse/SERVER-14706
[71] https://jira.mongodb.org/browse/SERVER-13104
[72] https://jira.mongodb.org/browse/SERVER-14984
[73] https://jira.mongodb.org/browse/SERVER-14981
[74] https://jira.mongodb.org/browse/SERVER-14831
[75] https://jira.mongodb.org/browse/SERVER-15038

- SERVER-14887[76] Allow user document changes made on a 2.4 primary to replicate to a 2.6 secondary
- SERVER-14805[77] Use multithreaded oplog replay during initial sync

Sharding

- SERVER-15056[78] Sharded connection cleanup on setup error can crash mongos
- SERVER-13702[79] Commands without optional query may target to wrong shards on mongos
- SERVER-15156[80] MongoDB upgrade 2.4 to 2.6 check returns error in `config.changelog collection`

Storage

- SERVER-15369[81] explicitly zero .ns files on creation
- SERVER-15319[82] Verify 2.8 freelist is upgrade-downgrade safe with 2.6
- SERVER-15111[83] partially written journal last section causes recovery to fail

Indexing

- SERVER-14848[84] Port `index_id_desc.js` to v2.6 and master branches
- SERVER-14205[85] ensureIndex failure reports `ok: 1` on some failures

Write Operations

- SERVER-15106[86] Incorrect nscanned and nscannedObjects for idhack updates in 2.6.4 profiler or slow query log
- SERVER-15029[87] The `$rename` (page 457) modifier uses incorrect dotted source path
- SERVER-14829[88] `UpdateIndexData::clear()` should reset all member variables

Data Aggregation

- SERVER-15087[89] Server crashes when running concurrent mapReduce and dropDatabase commands
- SERVER-14969[90] Dropping index during active aggregation operation can crash server
- SERVER-14168[91] Warning logged when incremental MR collections are unsuccessfully dropped on secondaries

[76] https://jira.mongodb.org/browse/SERVER-14887
[77] https://jira.mongodb.org/browse/SERVER-14805
[78] https://jira.mongodb.org/browse/SERVER-15056
[79] https://jira.mongodb.org/browse/SERVER-13702
[80] https://jira.mongodb.org/browse/SERVER-15156
[81] https://jira.mongodb.org/browse/SERVER-15369
[82] https://jira.mongodb.org/browse/SERVER-15319
[83] https://jira.mongodb.org/browse/SERVER-15111
[84] https://jira.mongodb.org/browse/SERVER-14848
[85] https://jira.mongodb.org/browse/SERVER-14205
[86] https://jira.mongodb.org/browse/SERVER-15106
[87] https://jira.mongodb.org/browse/SERVER-15029
[88] https://jira.mongodb.org/browse/SERVER-14829
[89] https://jira.mongodb.org/browse/SERVER-15087
[90] https://jira.mongodb.org/browse/SERVER-14969
[91] https://jira.mongodb.org/browse/SERVER-14168

Packaging

- SERVER-14679[92] (CentOS 7/RHEL 7) `init.d` script should create directory for `pid` file if it is missing
- SERVER-14023[93] Support for RHEL 7 Enterprise `.rpm` packages
- SERVER-13243[94] Support for Ubuntu 14 "Trusty" Enterprise `.deb` packages
- SERVER-11077[95] Support for Debian 7 Enterprise `.deb` packages
- SERVER-10642[96] Generate Community and Enterprise packages for SUSE 11

Logging and Diagnostics

- SERVER-14964[97] nscanned not written to the logs at `logLevel` 1 unless `slowms` exceeded or profiling enabled
- SERVER-12551[98] Audit DML/CRUD operations
- SERVER-14904[99] Adjust dates in `tool/exportimport_date.js` to account for different timezones

Internal Code and Testing

- SERVER-13770[100] `Helpers::removeRange` should check all runner states
- SERVER-14284[101] jstests should not leave profiler enabled at test run end
- SERVER-14076[102] remove test `replset_remove_node.js`
- SERVER-14778[103] Hide function and data pointers for natively-injected v8 functions

2.6.4 – Changes

Security

- SERVER-14701[104] The "backup" auth role should allow running the "collstats" command for all resources
- SERVER-14518[105] Allow disabling hostname validation for SSL
- SERVER-14268[106] Potential information leak
- SERVER-14170[107] Cannot read from secondary if both audit and auth are enabled in a sharded cluster
- SERVER-13833[108] userAdminAnyDatabase role should be able to create indexes on admin.system.users and admin.system.roles

[92] https://jira.mongodb.org/browse/SERVER-14679
[93] https://jira.mongodb.org/browse/SERVER-14023
[94] https://jira.mongodb.org/browse/SERVER-13243
[95] https://jira.mongodb.org/browse/SERVER-11077
[96] https://jira.mongodb.org/browse/SERVER-10642
[97] https://jira.mongodb.org/browse/SERVER-14964
[98] https://jira.mongodb.org/browse/SERVER-12551
[99] https://jira.mongodb.org/browse/SERVER-14904
[100] https://jira.mongodb.org/browse/SERVER-13770
[101] https://jira.mongodb.org/browse/SERVER-14284
[102] https://jira.mongodb.org/browse/SERVER-14076
[103] https://jira.mongodb.org/browse/SERVER-14778
[104] https://jira.mongodb.org/browse/SERVER-14701
[105] https://jira.mongodb.org/browse/SERVER-14518
[106] https://jira.mongodb.org/browse/SERVER-14268
[107] https://jira.mongodb.org/browse/SERVER-14170
[108] https://jira.mongodb.org/browse/SERVER-13833

- SERVER-12512[109] Add role-based, selective audit logging.
- SERVER-9482[110] Add build flag for sslFIPSMode

Querying

- SERVER-14625[111] Query planner can construct incorrect bounds for negations inside $elemMatch
- SERVER-14607[112] hash intersection of fetched and non-fetched data can discard data from a result
- SERVER-14532[113] Improve logging in the case of plan ranker ties
- SERVER-14350[114] Server crash when $centerSphere has non-positive radius
- SERVER-14317[115] Dead code in IDHackRunner::applyProjection
- SERVER-14311[116] skipping of index keys is not accounted for in plan ranking by the index scan stage
- SERVER-14123[117] some operations can create BSON object larger than the 16MB limit
- SERVER-14034[118] Sorted $in query with large number of elements can't use merge sort
- SERVER-13994[119] do not aggressively pre-fetch data for parallelCollectionScan

Replication

- SERVER-14665[120] Build failure for v2.6 in closeall.js caused by access violation reading _me
- SERVER-14505[121] cannot dropAllIndexes when index builds in progress assertion failure
- SERVER-14494[122] Dropping collection during active background index build on secondary triggers segfault
- SERVER-13822[123] Running resync before replset config is loaded can crash `mongod` (page 583)
- SERVER-11776[124] Replication 'isself' check should allow mapped ports

Sharding

- SERVER-14551[125] Runner yield during migration cleanup (removeRange) results in fassert
- SERVER-14431[126] Invalid chunk data after splitting on a key that's too large
- SERVER-14261[127] stepdown during migration range delete can abort `mongod` (page 583)
- SERVER-14032[128] v2.6 `mongos` (page 601) doesn't verify _id is present for config server upserts

[109] https://jira.mongodb.org/browse/SERVER-12512
[110] https://jira.mongodb.org/browse/SERVER-9482
[111] https://jira.mongodb.org/browse/SERVER-14625
[112] https://jira.mongodb.org/browse/SERVER-14607
[113] https://jira.mongodb.org/browse/SERVER-14532
[114] https://jira.mongodb.org/browse/SERVER-14350
[115] https://jira.mongodb.org/browse/SERVER-14317
[116] https://jira.mongodb.org/browse/SERVER-14311
[117] https://jira.mongodb.org/browse/SERVER-14123
[118] https://jira.mongodb.org/browse/SERVER-14034
[119] https://jira.mongodb.org/browse/SERVER-13994
[120] https://jira.mongodb.org/browse/SERVER-14665
[121] https://jira.mongodb.org/browse/SERVER-14505
[122] https://jira.mongodb.org/browse/SERVER-14494
[123] https://jira.mongodb.org/browse/SERVER-13822
[124] https://jira.mongodb.org/browse/SERVER-11776
[125] https://jira.mongodb.org/browse/SERVER-14551
[126] https://jira.mongodb.org/browse/SERVER-14431
[127] https://jira.mongodb.org/browse/SERVER-14261
[128] https://jira.mongodb.org/browse/SERVER-14032

- SERVER-13648[129] better stats from migration cleanup
- SERVER-12750[130] `mongos` (page 601) shouldn't accept initial query with "exhaust" flag set
- SERVER-9788[131] `mongos` (page 601) does not re-evaluate read preference once a valid replica set member is chosen
- SERVER-9526[132] Log messages regarding chunks not very informative when the shard key is of type BinData

Storage

- SERVER-14198[133] Std::set<pointer> and Windows Heap Allocation Reuse produces non-deterministic results
- SERVER-13975[134] Creating index on collection named "system" can cause server to abort
- SERVER-13729[135] Reads & Writes are blocked during data file allocation on Windows
- SERVER-13681[136] `mongod` (page 583) B stalls during background flush on Windows

Indexing SERVER-14494[137] Dropping collection during active background index build on secondary triggers segfault

Write Ops

- SERVER-14257[138] "remove" command can cause process termination by throwing unhandled exception if profiling is enabled
- SERVER-14024[139] Update fails when query contains part of a DBRef and results in an insert (upsert:true)
- SERVER-13764[140] debug mechanisms report incorrect nscanned / nscannedObjects for updates

Networking SERVER-13734[141] Remove catch (...) from handleIncomingMsg

Geo

- SERVER-14039[142] $nearSphere query with 2d index, skip, and limit returns incomplete results
- SERVER-13701[143] Query using 2d index throws exception when using explain()

[129] https://jira.mongodb.org/browse/SERVER-13648
[130] https://jira.mongodb.org/browse/SERVER-12750
[131] https://jira.mongodb.org/browse/SERVER-9788
[132] https://jira.mongodb.org/browse/SERVER-9526
[133] https://jira.mongodb.org/browse/SERVER-14198
[134] https://jira.mongodb.org/browse/SERVER-13975
[135] https://jira.mongodb.org/browse/SERVER-13729
[136] https://jira.mongodb.org/browse/SERVER-13681
[137] https://jira.mongodb.org/browse/SERVER-14494
[138] https://jira.mongodb.org/browse/SERVER-14257
[139] https://jira.mongodb.org/browse/SERVER-14024
[140] https://jira.mongodb.org/browse/SERVER-13764
[141] https://jira.mongodb.org/browse/SERVER-13734
[142] https://jira.mongodb.org/browse/SERVER-14039
[143] https://jira.mongodb.org/browse/SERVER-13701

Text Search

- SERVER-14738[144] Updates to documents with text-indexed fields may lead to incorrect entries
- SERVER-14027[145] Renaming collection within same database fails if wildcard text index present

Tools

- SERVER-14212[146] `mongorestore` (page 628) may drop system users and roles
- SERVER-14048[147] `mongodump` (page 622) against `mongos` (page 601) can't send dump to standard output

Admin

- SERVER-14556[148] Default dbpath for `mongod` (page 583) `--configsvr` (page 596) changes in 2.6
- SERVER-14355[149] Allow dbAdmin role to manually create system.profile collections

Packaging SERVER-14283[150] Parameters in installed config file are out of date

JavaScript

- SERVER-14254[151] Do not store native function pointer as a property in function prototype
- SERVER-13798[152] v8 garbage collection can cause crash due to independent lifetime of DBClient and Cursor objects
- SERVER-13707[153] mongo shell may crash when converting invalid regular expression

Shell

- SERVER-14341[154] negative opcounter values in serverStatus
- SERVER-14107[155] Querying for a document containing a value of either type Javascript or JavascriptWithScope crashes the shell

Usability SERVER-13833[156] userAdminAnyDatabase role should be able to create indexes on admin.system.users and admin.system.roles

[144] https://jira.mongodb.org/browse/SERVER-14738
[145] https://jira.mongodb.org/browse/SERVER-14027
[146] https://jira.mongodb.org/browse/SERVER-14212
[147] https://jira.mongodb.org/browse/SERVER-14048
[148] https://jira.mongodb.org/browse/SERVER-14556
[149] https://jira.mongodb.org/browse/SERVER-14355
[150] https://jira.mongodb.org/browse/SERVER-14283
[151] https://jira.mongodb.org/browse/SERVER-14254
[152] https://jira.mongodb.org/browse/SERVER-13798
[153] https://jira.mongodb.org/browse/SERVER-13707
[154] https://jira.mongodb.org/browse/SERVER-14341
[155] https://jira.mongodb.org/browse/SERVER-14107
[156] https://jira.mongodb.org/browse/SERVER-13833

Logging and Diagnostics

- SERVER-12512[157] Add role-based, selective audit logging.
- SERVER-14341[158] negative opcounter values in serverStatus

Testing

- SERVER-14731[159] plan_cache_ties.js sometimes fails
- SERVER-14147[160] make index_multi.js retry on connection failure
- SERVER-13615[161] sharding_rs2.js intermittent failure due to reliance on opcounters

2.6.3 – Changes

- SERVER-14302[162] Fixed: "Equality queries on `_id` with projection may return no results on sharded collections"
- SERVER-14304[163] Fixed: "Equality queries on `_id` with projection on `_id` may return orphan documents on sharded collections"

2.6.2 – Changes

Security

- SERVER-13727[164] The `backup` authorization role now includes privileges to run the `collStats` (page 348) command.
- SERVER-13804[165] The built-in role `restore` now has privileges on `system.roles` collection.
- SERVER-13612[166] Fixed: "SSL-enabled server appears not to be sending the list of supported certificate issuers to the client"
- SERVER-13753[167] Fixed: "`mongod` (page 583) may terminate if x.509 authentication certificate is invalid"
- SERVER-13945[168] For *replica set/sharded cluster member authentication*, now matches x.509 cluster certificates by attributes instead of by substring comparison.
- SERVER-13868[169] Now marks V1 users as probed on databases that do not have surrogate user documents.
- SERVER-13850[170] Now ensures that the user cache entry is up to date before using it to determine a user's roles in user management commands on `mongos` (page 601).
- SERVER-13588[171] Fixed: "Shell prints startup warning when auth enabled"

[157] https://jira.mongodb.org/browse/SERVER-12512
[158] https://jira.mongodb.org/browse/SERVER-14341
[159] https://jira.mongodb.org/browse/SERVER-14731
[160] https://jira.mongodb.org/browse/SERVER-14147
[161] https://jira.mongodb.org/browse/SERVER-13615
[162] https://jira.mongodb.org/browse/SERVER-14302
[163] https://jira.mongodb.org/browse/SERVER-14304
[164] https://jira.mongodb.org/browse/SERVER-13727
[165] https://jira.mongodb.org/browse/SERVER-13804
[166] https://jira.mongodb.org/browse/SERVER-13612
[167] https://jira.mongodb.org/browse/SERVER-13753
[168] https://jira.mongodb.org/browse/SERVER-13945
[169] https://jira.mongodb.org/browse/SERVER-13868
[170] https://jira.mongodb.org/browse/SERVER-13850
[171] https://jira.mongodb.org/browse/SERVER-13588

Querying

- SERVER-13731[172] Fixed: "Stack overflow when parsing deeply nested `$not` (page 407) query"
- SERVER-13890[173] Fixed: "Index bounds builder constructs invalid bounds for multiple negations joined by an `$or` (page 408)"
- SERVER-13752[174] Verified assertion on empty `$in` (page 402) clause and sort on second field in a compound index.
- SERVER-13337[175] Re-enabled `idhack` for queries with projection.
- SERVER-13715[176] Fixed: "Aggregation pipeline execution can fail with $or and blocking sorts"
- SERVER-13714[177] Fixed: "non-top-level indexable `$not` (page 407) triggers query planning bug"
- SERVER-13769[178] Fixed: "`distinct` (page 215) command on indexed field with geo predicate fails to execute"
- SERVER-13675[179] Fixed "Plans with differing performance can tie during plan ranking"
- SERVER-13899[180] Fixed: "'Whole index scan' query solutions can use incompatible indexes, return incorrect results"
- SERVER-13852[181] Fixed "IndexBounds::endKeyInclusive not initialized by constructor"
- SERVER-14073[182] planSummary no longer truncated at 255 characters
- SERVER-14174[183] Fixed: "If ntoreturn is a limit (rather than batch size) extra data gets buffered during plan ranking"
- SERVER-13789[184] Some nested queries no longer trigger an assertion error
- SERVER-14064[185] Added planSummary information for `count` (page 213) command log message.
- SERVER-13960[186] Queries containing `$or` (page 408) no longer miss results if multiple clauses use the same index.
- SERVER-14180[187] Fixed: "Crash with 'and' clause, `$elemMatch` (page 442), and nested `$mod` (page 412) or regex"
- SERVER-14176[188] Natural order sort specification no longer ignored if query is specified.
- SERVER-13754[189] Bounds no longer combined for `$or` (page 408) queries that can use merge sort.

[172] https://jira.mongodb.org/browse/SERVER-13731
[173] https://jira.mongodb.org/browse/SERVER-13890
[174] https://jira.mongodb.org/browse/SERVER-13752
[175] https://jira.mongodb.org/browse/SERVER-13337
[176] https://jira.mongodb.org/browse/SERVER-13715
[177] https://jira.mongodb.org/browse/SERVER-13714
[178] https://jira.mongodb.org/browse/SERVER-13769
[179] https://jira.mongodb.org/browse/SERVER-13675
[180] https://jira.mongodb.org/browse/SERVER-13899
[181] https://jira.mongodb.org/browse/SERVER-13852
[182] https://jira.mongodb.org/browse/SERVER-14073
[183] https://jira.mongodb.org/browse/SERVER-14174
[184] https://jira.mongodb.org/browse/SERVER-13789
[185] https://jira.mongodb.org/browse/SERVER-14064
[186] https://jira.mongodb.org/browse/SERVER-13960
[187] https://jira.mongodb.org/browse/SERVER-14180
[188] https://jira.mongodb.org/browse/SERVER-14176
[189] https://jira.mongodb.org/browse/SERVER-13754

Geospatial SERVER-13687[190] Results of `$near` (page 429) query on compound multi-key 2dsphere index are now sorted by distance.

Write Operations SERVER-13802[191] Insert field validation no longer stops at first `Timestamp()` field.

Replication

- SERVER-13993[192] Fixed: "log a message when `shouldChangeSyncTarget()` believes a node should change sync targets"
- SERVER-13976[193] Fixed: "Cloner needs to detect failure to create collection"

Sharding

- SERVER-13616[194] Resolved: "'type 7' (OID) error when acquiring distributed lock for first time"
- SERVER-13812[195] Now catches exception thrown by `getShardsForQuery` for geo query.
- SERVER-14138[196] `mongos` (page 601) will now correctly target multiple shards for nested field shard key predicates.
- SERVER-11332[197] Fixed: "Authentication requests delayed if first config server is unresponsive"

Map/Reduce

- SERVER-14186[198] Resolved: "`rs.stepDown` (page 179) during mapReduce causes fassert in logOp"
- SERVER-13981[199] Temporary map/reduce collections are now correctly replicated to secondaries.

Storage

- SERVER-13750[200] `convertToCapped` (page 326) on empty collection no longer aborts after `invariant()` failure.
- SERVER-14056[201] Moving large collection across databases with renameCollection no longer triggers fatal assertion.
- SERVER-14082[202] Fixed: "Excessive freelist scanning for MaxBucket"
- SERVER-13737[203] CollectionOptions parser now skips non-numeric for "size"/"max" elements if values non-numeric.

[190] https://jira.mongodb.org/browse/SERVER-13687
[191] https://jira.mongodb.org/browse/SERVER-13802
[192] https://jira.mongodb.org/browse/SERVER-13993
[193] https://jira.mongodb.org/browse/SERVER-13976
[194] https://jira.mongodb.org/browse/SERVER-13616
[195] https://jira.mongodb.org/browse/SERVER-13812
[196] https://jira.mongodb.org/browse/SERVER-14138
[197] https://jira.mongodb.org/browse/SERVER-11332
[198] https://jira.mongodb.org/browse/SERVER-14186
[199] https://jira.mongodb.org/browse/SERVER-13981
[200] https://jira.mongodb.org/browse/SERVER-13750
[201] https://jira.mongodb.org/browse/SERVER-14056
[202] https://jira.mongodb.org/browse/SERVER-14082
[203] https://jira.mongodb.org/browse/SERVER-13737

Build and Packaging

- SERVER-13950[204] MongoDB Enterprise now includes required dependency list.
- SERVER-13862[205] Support for mongodb-org-server installation 2.6.1-1 on RHEL5 via RPM.
- SERVER-13724[206] Added SCons flag to override treating all warnings as errors.

Diagnostics

- SERVER-13587[207] Resolved: "`ndeleted` in `system.profile` documents reports 1 too few documents removed"
- SERVER-13368[208] Improved exposure of timing information in `currentOp`.

Administration SERVER-13954[209] `security.javascriptEnabled` option is now available in the YAML configuration file.

Tools

- SERVER-10464[210] `mongodump` (page 622) can now query `oplog.$main` and `oplog.rs` when using *`--dbpath`*.
- SERVER-13760[211] `mongoexport` (page 649) can now handle large timestamps on Windows.

Shell

- SERVER-13865[212] Shell now returns correct `WriteResult` for compatibility-mode upsert with non-OID equality predicate on `_id` field.
- SERVER-13037[213] Fixed typo in error message for "compatibility mode".

Internal Code

- SERVER-13794[214] Fixed: "Unused snapshot history consuming significant heap space"
- SERVER-13446[215] Removed Solaris builds dependency on ILLUMOS libc.
- SERVER-14092[216] MongoDB upgrade 2.4 to 2.6 check no longer returns an error in internal collections.
- SERVER-14000[217] Added new lsb file location for Debian 7.1

Testing

- SERVER-13723[218] Stabilized `tags.js` after a change in its timeout when it was ported to use write commands.

[204] https://jira.mongodb.org/browse/SERVER-13950
[205] https://jira.mongodb.org/browse/SERVER-13862
[206] https://jira.mongodb.org/browse/SERVER-13724
[207] https://jira.mongodb.org/browse/SERVER-13587
[208] https://jira.mongodb.org/browse/SERVER-13368
[209] https://jira.mongodb.org/browse/SERVER-13954
[210] https://jira.mongodb.org/browse/SERVER-10464
[211] https://jira.mongodb.org/browse/SERVER-13760
[212] https://jira.mongodb.org/browse/SERVER-13865
[213] https://jira.mongodb.org/browse/SERVER-13037
[214] https://jira.mongodb.org/browse/SERVER-13794
[215] https://jira.mongodb.org/browse/SERVER-13446
[216] https://jira.mongodb.org/browse/SERVER-14092
[217] https://jira.mongodb.org/browse/SERVER-14000
[218] https://jira.mongodb.org/browse/SERVER-13723

- SERVER-13494[219] Fixed: "`setup_multiversion_mongodb.py` doesn't download 2.4.10 because of non-numeric version sorting"
- SERVER-13603[220] Fixed: "Test suites with options tests fail when run with *`--noprealloc`*"
- SERVER-13948[221] Fixed: "`awaitReplication()` failures related to getting a config version from master causing test failures"
- SERVER-13839[222] Fixed `sync2.js` failure.
- SERVER-13972[223] Fixed `connections_opened.js` failure.
- SERVER-13712[224] Reduced peak disk usage of test suites.
- SERVER-14249[225] Added tests for querying oplog via `mongodump` (page 622) using *`--dbpath`*
- SERVER-10462[226] Fixed: "Windows file locking related buildbot failures"

2.6.1 – Changes

Stability SERVER-13739[227] Repair database failure can delete database files

Build and Packaging

- SERVER-13287[228] Addition of debug symbols has doubled compile time
- SERVER-13563[229] Upgrading from 2.4.x to 2.6.0 via `yum` clobbers configuration file
- SERVER-13691[230] yum and apt "stable" repositories contain release candidate 2.6.1-rc0 packages
- SERVER-13515[231] Cannot install MongoDB as a service on Windows

Querying

- SERVER-13066[232] Negations over multikey fields do not use index
- SERVER-13495[233] Concurrent `GETMORE` and `KILLCURSORS` operations can cause race condition and server crash
- SERVER-13503[234] The `$where` (page 421) operator should not be allowed under `$elemMatch` (page 442)
- SERVER-13537[235] Large skip and and limit values can cause crash in blocking sort stage
- SERVER-13557[236] Incorrect negation of $elemMatch value in 2.6

[219] https://jira.mongodb.org/browse/SERVER-13494
[220] https://jira.mongodb.org/browse/SERVER-13603
[221] https://jira.mongodb.org/browse/SERVER-13948
[222] https://jira.mongodb.org/browse/SERVER-13839
[223] https://jira.mongodb.org/browse/SERVER-13972
[224] https://jira.mongodb.org/browse/SERVER-13712
[225] https://jira.mongodb.org/browse/SERVER-14249
[226] https://jira.mongodb.org/browse/SERVER-10462
[227] https://jira.mongodb.org/browse/SERVER-13739
[228] https://jira.mongodb.org/browse/SERVER-13287
[229] https://jira.mongodb.org/browse/SERVER-13563
[230] https://jira.mongodb.org/browse/SERVER-13691
[231] https://jira.mongodb.org/browse/SERVER-13515
[232] https://jira.mongodb.org/browse/SERVER-13066
[233] https://jira.mongodb.org/browse/SERVER-13495
[234] https://jira.mongodb.org/browse/SERVER-13503
[235] https://jira.mongodb.org/browse/SERVER-13537
[236] https://jira.mongodb.org/browse/SERVER-13557

- SERVER-13562[237] Queries that use tailable cursors do not stream results if skip() is applied
- SERVER-13566[238] Using the OplogReplay flag with extra predicates can yield incorrect results
- SERVER-13611[239] Missing sort order for compound index leads to unnecessary in-memory sort
- SERVER-13618[240] Optimization for sorted $in queries not applied to reverse sort order
- SERVER-13661[241] Increase the maximum allowed depth of query objects
- SERVER-13664[242] Query with `$elemMatch` (page 442) using a compound multikey index can generate incorrect results
- SERVER-13677[243] Query planner should traverse through $all while handling $elemMatch object predicates
- SERVER-13766[244] Dropping index or collection while $or query is yielding triggers fatal assertion

Geospatial

- SERVER-13666[245] `$near` (page 429) queries with out-of-bounds points in legacy format can lead to crashes
- SERVER-13540[246] The `geoNear` (page 229) command no longer returns distance in radians for legacy point
- SERVER-13486[247]: The `geoNear` (page 229) command can create too large BSON objects for aggregation.

Replication

- SERVER-13500[248] Changing replica set configuration can crash running members
- SERVER-13589[249] Background index builds from a 2.6.0 primary fail to complete on 2.4.x secondaries
- SERVER-13620[250] Replicated data definition commands will fail on secondaries during background index build
- SERVER-13496[251] Creating index with same name but different spec in mixed version replicaset can abort replication

Sharding

- SERVER-12638[252] Initial sharding with hashed shard key can result in duplicate split points
- SERVER-13518[253] The `_id` field is no longer automatically generated by `mongos` (page 601) when missing
- SERVER-13777[254] Migrated ranges waiting for deletion do not report cursors still open

[237] https://jira.mongodb.org/browse/SERVER-13562
[238] https://jira.mongodb.org/browse/SERVER-13566
[239] https://jira.mongodb.org/browse/SERVER-13611
[240] https://jira.mongodb.org/browse/SERVER-13618
[241] https://jira.mongodb.org/browse/SERVER-13661
[242] https://jira.mongodb.org/browse/SERVER-13664
[243] https://jira.mongodb.org/browse/SERVER-13677
[244] https://jira.mongodb.org/browse/SERVER-13766
[245] https://jira.mongodb.org/browse/SERVER-13666
[246] https://jira.mongodb.org/browse/SERVER-13540
[247] https://jira.mongodb.org/browse/SERVER-13486
[248] https://jira.mongodb.org/browse/SERVER-13500
[249] https://jira.mongodb.org/browse/SERVER-13589
[250] https://jira.mongodb.org/browse/SERVER-13620
[251] https://jira.mongodb.org/browse/SERVER-13496
[252] https://jira.mongodb.org/browse/SERVER-12638
[253] https://jira.mongodb.org/browse/SERVER-13518
[254] https://jira.mongodb.org/browse/SERVER-13777

Security

- SERVER-9358[255] Log rotation can overwrite previous log files
- SERVER-13644[256] Sensitive credentials in startup options are not redacted and may be exposed
- SERVER-13441[257] Inconsistent error handling in user management shell helpers

Write Operations

- SERVER-13466[258] Error message in collection creation failure contains incorrect namespace
- SERVER-13499[259] Yield policy for batch-inserts should be the same as for batch-updates/deletes
- SERVER-13516[260] Array updates on documents with more than 128 BSON elements may crash `mongod` (page 583)

2.6.6 – December 09, 2014

- Fixed: Evaluating candidate query plans with concurrent writes on same collection may crash `mongod` (page 583) SERVER-15580[261]
- Fixed: 2.6 `mongod` (page 583) crashes with segfault when added to a 2.8 replica set with 12 or more members SERVER-16107[262]
- Fixed: `$regex` (page 414), `$in` (page 402) and `$sort` with index returns too many results SERVER-15696[263]
- Change: `moveChunk` (page 310) will fail if there is data on the target shard and a required index does not exist. SERVER-12472[264]
- Primary should abort if encountered problems writing to the oplog SERVER-12058[265]
- All issues closed in 2.6.6[266]

2.6.5 – October 07, 2014

- `$rename` now uses correct dotted source paths SERVER-15029[267]
- Partially written journal last section does not affect recovery SERVER-15111[268]
- Explicitly zero `.ns` files on creation SERVER-15369[269]
- Plan ranker will no longer favor intersection plans if predicate generates empty range index scan SERVER-14961[270]

[255] https://jira.mongodb.org/browse/SERVER-9358
[256] https://jira.mongodb.org/browse/SERVER-13644
[257] https://jira.mongodb.org/browse/SERVER-13441
[258] https://jira.mongodb.org/browse/SERVER-13466
[259] https://jira.mongodb.org/browse/SERVER-13499
[260] https://jira.mongodb.org/browse/SERVER-13516
[261] https://jira.mongodb.org/browse/SERVER-15580
[262] https://jira.mongodb.org/browse/SERVER-16107
[263] https://jira.mongodb.org/browse/SERVER-15696
[264] https://jira.mongodb.org/browse/SERVER-12472
[265] https://jira.mongodb.org/browse/SERVER-12058
[266] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.6%22%20AND%20project%20%3D%20SERVER
[267] https://jira.mongodb.org/browse/SERVER-15029
[268] https://jira.mongodb.org/browse/SERVER-15111
[269] https://jira.mongodb.org/browse/SERVER-15369
[270] https://jira.mongodb.org/browse/SERVER-14961

- Generate Community and Enterprise packages for SUSE 11 SERVER-10642[271]
- All issues closed in 2.6.5[272]

2.6.4 – August 11, 2014

- Fix for `text` index where under specific circumstances, in-place updates to a `text`-indexed field may result in incorrect/incomplete results SERVER-14738[273]
- Check the size of the split point before performing a manual split chunk operation SERVER-14431[274]
- Ensure read preferences are re-evaluated by drawing secondary connections from a global pool and releasing back to the pool at the end of a query/command SERVER-9788[275]
- Allow read from secondaries when both audit and authorization are enabled in a sharded cluster SERVER-14170[276]
- All issues closed in 2.6.4[277]

2.6.3 – June 19, 2014

- Equality queries on `_id` with projection may return no results on sharded collections SERVER-14302[278].
- Equality queries on `_id` with projection on `_id` may return orphan documents on sharded collections SERVER-14304[279].
- All issues closed in 2.6.3[280].

2.6.2 – June 16, 2014

- Query plans with differing performance can tie during plan ranking SERVER-13675[281].
- `mongod` (page 583) may terminate if x.509 authentication certificate is invalid SERVER-13753[282].
- Temporary map/reduce collections are incorrectly replicated to secondaries SERVER-13981[283].
- `mongos` (page 601) incorrectly targets multiple shards for nested field shard key predicates SERVER-14138[284].
- `rs.stepDown()` (page 179) during mapReduce causes `fassert` when writing to op log SERVER-14186[285].
- All issues closed in 2.6.2[286].

[271] https://jira.mongodb.org/browse/SERVER-10642
[272] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.5%22%20AND%20project%20%3D%20SERVER
[273] https://jira.mongodb.org/browse/SERVER-14738
[274] https://jira.mongodb.org/browse/SERVER-14431
[275] https://jira.mongodb.org/browse/SERVER-9788
[276] https://jira.mongodb.org/browse/SERVER-14170
[277] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.4%22%20AND%20project%20%3D%20SERVER
[278] https://jira.mongodb.org/browse/SERVER-14302
[279] https://jira.mongodb.org/browse/SERVER-14304
[280] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.3%22%20AND%20project%20%3D%20SERVER
[281] https://jira.mongodb.org/browse/SERVER-13675
[282] https://jira.mongodb.org/browse/SERVER-13753
[283] https://jira.mongodb.org/browse/SERVER-13981
[284] https://jira.mongodb.org/browse/SERVER-14138
[285] https://jira.mongodb.org/browse/SERVER-14186
[286] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.2%22%20AND%20project%20%3D%20SERVER

2.6.1 – May 5, 2014

- Fix to install MongoDB service on Windows with the `--install` option SERVER-13515[287].
- Allow direct upgrade from 2.4.x to 2.6.0 via `yum` SERVER-13563[288].
- Fix issues with background index builds on secondaries: SERVER-13589[289] and SERVER-13620[290].
- Redact credential information passed as startup options SERVER-13644[291].
- *2.6.1 Changelog* (page 740).
- All issues closed in 2.6.1[292].

Major Changes

The following changes in MongoDB affect both the standard and Enterprise editions:

Aggregation Enhancements

The aggregation pipeline adds the ability to return result sets of any size, either by returning a cursor or writing the output to a collection. Additionally, the aggregation pipeline supports variables and adds new operations to handle sets and redact data.

- The `db.collection.aggregate()` (page 22) now returns a cursor, which enables the aggregation pipeline to return result sets of any size.
- Aggregation pipelines now support an `explain` operation to aid analysis of aggregation operations.
- Aggregation can now use a more efficient external-disk-based sorting process.
- New pipeline stages:
  - `$out` (page 491) stage to output to a collection.
  - `$redact` (page 495) stage to allow additional control to accessing the data.
- New or modified operators:
  - *set expression operators* (page 506).
  - `$let` (page 531) and `$map` (page 532) operators to allow for the use of variables.
  - `$literal` (page 533) operator and `$size` (page 530) operator.
  - `$cond` (page 545) expression now accepts either an object or an array.

Text Search Integration

Text search is now enabled by default, and the query system, including the aggregation pipeline `$match` (page 490) stage, includes the `$text` (page 417) operator, which resolves text-search queries.

MongoDB 2.6 includes an updated `text index` format and deprecates the `text` (page 251) command.

[287] https://jira.mongodb.org/browse/SERVER-13515
[288] https://jira.mongodb.org/browse/SERVER-13563
[289] https://jira.mongodb.org/browse/SERVER-13589
[290] https://jira.mongodb.org/browse/SERVER-13620
[291] https://jira.mongodb.org/browse/SERVER-13644
[292] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.1%22%20AND%20project%20%3D%20SERVER

Insert and Update Improvements

Improvements to the update and insert systems include additional operations and improvements that increase consistency of modified data.

- MongoDB preserves the order of the document fields following write operations *except* for the following cases:
  - The `_id` field is always the first field in the document.
  - Updates that include `renaming` (page 457) of field names may result in the reordering of fields in the document.
- New or enhanced update operators:
  - `$bit` (page 480) operator supports bitwise `xor` operation.
  - `$min` (page 454) and `$max` (page 453) operators that perform conditional update depending on the relative size of the specified value and the current value of a field.
  - `$push` (page 470) operator has enhanced support for the `$sort` (page 477), `$slice` (page 474), and `$each` (page 472) modifiers and supports a new `$position` (page 473) modifier.
  - `$currentDate` (page 451) operator to set the value of a field to the current date.
- The `$mul` (page 456) operator for multiplicative increments for insert and update operations.

See also:

Update Operator Syntax Validation (page 755)

New Write Operation Protocol

A new write protocol integrates write operations with write concerns. The protocol also provides improved support for bulk operations.

MongoDB 2.6 adds the write commands `insert` (page 247), `update` (page 251), and `delete` (page 234), which provide the basis for the improved bulk insert. All officially supported MongoDB drivers support the new write commands.

The `mongo` (page 610) shell now includes methods to perform bulk-write operations. See `Bulk()` (page 135) for more information.

See also:

Write Method Acknowledgements (page 751)

MSI Package for MongoDB Available for Windows

MongoDB now distributes MSI packages for Microsoft Windows. This is the recommended method for MongoDB installation under Windows.

Security Improvements

MongoDB 2.6 enhances support for secure deployments through improved SSL support, x.509-based authentication, an improved authorization system with more granular controls, as well as centralized credential storage, and improved user management tools.

Specifically these changes include:

- A new *authorization model* that provides the ability to create custom *user-defined-roles* and the ability to specify user privileges at a collection-level granularity.
- Global user management, which stores all user and user-defined role data in the `admin` database and provides a new set of commands for managing users and roles.
- x.509 certificate authentication for `client authentication` as well as for `internal authentication` of sharded and/or replica set cluster members. x.509 authentication is only available for deployments using SSL.
- Enhanced SSL Support:
  - `Rolling upgrades of clusters` to use SSL.
  - *mongodb-tools-support-ssl* support connections to `mongod` (page 583) and `mongos` (page 601) instances using SSL connections.
  - *Prompt for passphrase* by `mongod` (page 583) or `mongos` (page 601) at startup.
  - Require the use of strong SSL ciphers, with a minimum 128-bit key length for all connections. The strong-cipher requirement prevents an old or malicious client from forcing use of a weak cipher.
- MongoDB disables the http interface by default, limiting `network exposure`. To enable the interface, see `enabled`.

See also:

New Authorization Model (page 753), *SSL Certificate Hostname Validation* (page 753), and `http://docs.mongodb.org/manual/administration/security-checklist`.

Query Engine Improvements

- MongoDB can now use `index intersection` to fulfill queries supported by more than one index.
- *index-filters* to limit which indexes can become the winning plan for a query.
- *Query Plan Cache Methods* (page 130) methods to view and clear the `query plans` cached by the query optimizer.
- MongoDB can now use `count()` (page 83) with `hint()` (page 87). See `count()` (page 83) for details.

Improvements

Geospatial Enhancements

- *2dsphere indexes version 2*.
- Support for *geojson-multipoint*, *geojson-multilinestring*, *geojson-multipolygon*, and *geojson-geometrycollection*.
- Support for geospatial query clauses in `$or` (page 408) expressions.

See also:

2dsphere Index Version 2 (page 754), *$maxDistance Changes* (page 756), *Deprecated $uniqueDocs* (page 757), *Stronger Validation of Geospatial Queries* (page 757)

Index Build Enhancements

- *Background index build* allowed on secondaries. If you initiate a background index build on a *primary*, the secondaries will replicate the index build in the background.
- Automatic rebuild of interrupted index builds after a restart.
  - If a standalone or a primary instance terminates during an index build *without a clean shutdown*, `mongod` (page 583) now restarts the index build when the instance restarts. If the instance shuts down cleanly or if a user kills the index build, the interrupted index builds do not automatically restart upon the restart of the server.
  - If a secondary instance terminates during an index build, the `mongod` (page 583) instance will now restart the interrupted index build when the instance restarts.

  To disable this behavior, use the `--noIndexBuildRetry` (page 591) command-line option.
- `ensureIndex()` (page 30) now wraps a new `createIndex` command.
- The `dropDups` option to `ensureIndex()` (page 30) and `createIndex` is deprecated.

See also:

Enforce Index Key Length Limit (page 749)

Enhanced Sharding and Replication Administration

- New `cleanupOrphaned` (page 305) command to remove *orphaned documents* from a shard.
- New `mergeChunks` (page 309) command to combine contiguous chunks located on a single shard. See `mergeChunks` (page 309) and `http://docs.mongodb.org/manual/tutorial/merge-chunks-in-sharded-cluster`.
- New `rs.printReplicationInfo()` (page 175) and `rs.printSlaveReplicationInfo()` (page 176) methods to provide a formatted report of the status of a replica set from the perspective of the primary and the secondary, respectively.

Configuration Options YAML File Format

MongoDB 2.6 supports a YAML-based configuration file format in addition to the previous configuration file format. See the documentation of the `Configuration File` for more information.

Operational Changes

Storage

`usePowerOf2Sizes` (page 322) is now the default allocation strategy for all new collections. The new allocation strategy uses more storage relative to total document size but results in lower levels of storage fragmentation and more predictable storage capacity planning over time.

To use the previous *exact-fit allocation strategy*:

- For a specific collection, use `collMod` (page 321) with `usePowerOf2Sizes` (page 322) set to `false`.
- For all new collections on an entire `mongod` (page 583) instance, set `newCollectionsUsePowerOf2Sizes` to `false`.

New collections include those: created during *initial sync*, as well as those created by the mongorestore (page 628) and mongoimport (page 642) tools, by running mongod (page 583) with the *--repair* option, as well as the restoreDatabase command.

See http://docs.mongodb.org/manual/core/storage for more information about MongoDB's storage system.

Networking

- Removed upward limit for the maxIncomingConnections for mongod (page 583) and mongos (page 601). Previous versions capped the maximum possible maxIncomingConnections setting at 20,000 connections.
- Connection pools for a mongos (page 601) instance may be used by multiple MongoDB servers. This can reduce the number of connections needed for high-volume workloads and reduce resource consumption in sharded clusters.
- The C++ driver now monitors *replica set* health with the isMaster (page 289) command instead of replSetGetStatus (page 296). This allows the C++ driver to support systems that require authentication.
- New cursor.maxTimeMS() (page 90) and corresponding maxTimeMS option for commands to specify a time limit.

Tool Improvements

- mongo (page 610) shell supports a global */etc/mongorc.js* (page 615).
- All MongoDB *executable files* (page 583) now support the --quiet option to suppress all logging output except for error messages.
- mongoimport (page 642) uses the input filename, without the file extension if any, as the collection name if run without the -c or --collection specification.
- mongoexport (page 649) can now constrain export data using *--skip* (page 655) and *--limit* (page 655), as well as order the documents in an export using the *--sort* (page 655) option.
- mongostat (page 657) can support the use of *--rowcount* (page 668) (*-n* (page **??**)) with the *--discover* (page 661) option to produce the specified number of output lines.
- Add strict mode representation for data_numberlong for use by mongoexport (page 649) and mongoimport (page 642).

MongoDB Enterprise Features

The following changes are specific to MongoDB Enterprise Editions:

MongoDB Enterprise for Windows

MongoDB Enterprise for Windows is now available. It includes support for Kerberos, SSL, and SNMP.

MongoDB Enterprise for Windows does **not** include LDAP support for authentication. However, MongoDB Enterprise for Linux supports using LDAP authentication with an ActiveDirectory server.

MongoDB Enterprise for Windows includes OpenSSL version 1.0.1g.

Auditing

MongoDB Enterprise adds `auditing` capability for `mongod` (page 583) and `mongos` (page 601) instances. See *auditing* for details.

LDAP Support for Authentication

MongoDB Enterprise provides support for proxy authentication of users. This allows administrators to configure a MongoDB cluster to authenticate users by proxying authentication requests to a specified Lightweight Directory Access Protocol (LDAP) service. See `http://docs.mongodb.org/manual/tutorial/configure-ldap-sasl-openldap` and `http://docs.mongodb.org/manual/tutorial/configure-ldap-sasl-activedirectory` for details.

MongoDB Enterprise for Windows does **not** include LDAP support for authentication. However, MongoDB Enterprise for Linux supports using LDAP authentication with an ActiveDirectory server.

MongoDB does **not** support LDAP authentication in mixed sharded cluster deployments that contain both version 2.4 and version 2.6 shards. See *Upgrade MongoDB to 2.6* (page 760) for upgrade instructions.

Expanded SNMP Support

MongoDB Enterprise has greatly expanded its SNMP support to provide SNMP access to nearly the full range of metrics provided by `db.serverStatus()` (page 124).

See also:

SNMP Changes (page 754)

Additional Information

Changes Affecting Compatibility

Compatibility Changes in MongoDB 2.6 The following 2.6 changes can affect the compatibility with older versions of MongoDB. See *Release Notes for MongoDB 2.6* (page 726) for the full list of the 2.6 changes.

Index Changes

Enforce Index Key Length Limit

Description MongoDB 2.6 implements a stronger enforcement of the limit on `index key`.

Creating indexes will error if an index key in an existing document exceeds the limit:

- `db.collection.ensureIndex()` (page 30), `db.collection.reIndex()` (page 65), `compact` (page 322), and `repairDatabase` (page 341) will error and not create the index. Previous versions of MongoDB would create the index but not index such documents.
- Because `db.collection.reIndex()` (page 65), `compact` (page 322), and `repairDatabase` (page 341) drop *all* the indexes from a collection and then recreate them sequentially, the error from the index key limit prevents these operations from rebuilding any remaining indexes for the collection and, in the case of the `repairDatabase` (page 341) command, from continuing with the remainder of the process.

Inserts will error:

- db.collection.insert() (page 55) and other operations that perform inserts (e.g. db.collection.save() (page 70) and db.collection.update() (page 72) with upsert that result in inserts) will fail to insert if the new document has an indexed field whose corresponding index entry exceeds the limit. Previous versions of MongoDB would insert but not index such documents.
- mongorestore (page 628) and mongoimport (page 642) will fail to insert if the new document has an indexed field whose corresponding index entry exceeds the limit.

Updates will error:

- db.collection.update() (page 72) and db.collection.save() (page 70) operations on an indexed field will error if the updated value causes the index entry to exceed the limit.
- If an existing document contains an indexed field whose index entry exceeds the limit, updates on other fields that result in the relocation of a document on disk will error.

Chunk Migration will fail:

- Migrations will fail for a chunk that has a document with an indexed field whose index entry exceeds the limit.
- If left unfixed, the chunk will repeatedly fail migration, effectively ceasing chunk balancing for that collection. Or, if chunk splits occur in response to the migration failures, this response would lead to unnecessarily large number of chunks and an overly large config databases.

Secondary members of replica sets will warn:

- Secondaries will continue to replicate documents with an indexed field whose corresponding index entry exceeds the limit on initial sync but will print warnings in the logs.
- Secondaries allow index build and rebuild operations on a collection that contains an indexed field whose corresponding index entry exceeds the limit but with warnings in the logs.
- With *mixed version* replica sets where the secondaries are version 2.6 and the primary is version 2.4, secondaries will replicate documents inserted or updated on the 2.4 primary, but will print error messages in the log if the documents contain an indexed field whose corresponding index entry exceeds the limit.

Solution Run db.upgradeCheckAllDBs() (page 128) to find current keys that violate this limit and correct as appropriate. Preferably, run the test before upgrading; i.e. connect the 2.6 mongo (page 610) shell to your MongoDB 2.4 database and run the method.

If you have an existing data set and want to disable the default index key length validation so that you can upgrade before resolving these indexing issues, use the failIndexKeyTooLong parameter.

Index Specifications Validate Field Names

Description In MongoDB 2.6, create and re-index operations fail when the index key refers to an empty field, e.g. "a..b" : 1 or the field name starts with a dollar sign ($).

- db.collection.ensureIndex() (page 30) will not create a new index with an invalid or empty key name.
- db.collection.reIndex() (page 65), compact (page 322), and repairDatabase (page 341) will error if an index exists with an invalid or empty key name.
- Chunk migration will fail if an index exists with an invalid or empty key name.

Previous versions of MongoDB allow the index.

Solution Run `db.upgradeCheckAllDBs()` (page 128) to find current keys that violate this limit and correct as appropriate. Preferably, run the test before upgrading; i.e. connect the 2.6 `mongo` (page 610) shell to your MongoDB 2.4 database and run the method.

`ensureIndex` and Existing Indexes

Description `db.collection.ensureIndex()` (page 30) now errors:

- if you try to create an existing index but with different options; e.g. in the following example, the second `db.collection.ensureIndex()` (page 30) will error.

```
db.mycollection.ensureIndex( { x: 1 } )
db.mycollection.ensureIndex( { x: 1 }, { unique: 1 } )
```

- if you specify an index name that already exists but the key specifications differ; e.g. in the following example, the second `db.collection.ensureIndex()` (page 30) will error.

```
db.mycollection.ensureIndex( { a: 1 }, { name: "myIdx" } )
db.mycollection.ensureIndex( { z: 1 }, { name: "myIdx" } )
```

Previous versions did not create the index but did not error.

Write Method Acknowledgements

Description The `mongo` (page 610) shell write methods `db.collection.insert()` (page 55), `db.collection.update()` (page 72), `db.collection.save()` (page 70) and `db.collection.remove()` (page 66) now integrate the `write concern` directly into the method rather than with a separate `getLastError` (page 245) command to provide *safe writes* whether run interactively in the `mongo` (page 610) shell or non-interactively in a script. In previous versions, these methods exhibited a "fire-and-forget" behavior. [293]

- Existing scripts for the `mongo` (page 610) shell that used these methods will now observe safe writes which take **longer** than the previous "fire-and-forget" behavior.
- The write methods now return a `WriteResult` (page 201) object that contains the results of the operation, including any write errors and write concern errors, and obviates the need to call `getLastError` (page 245) command to get the status of the results. See `db.collection.insert()` (page 55), `db.collection.update()` (page 72), `db.collection.save()` (page 70) and `db.collection.remove()` (page 66) for details.
- In sharded environments, `mongos` (page 601) no longer supports "fire-and-forget" behavior. This limits throughput when writing data to sharded clusters.

Solution Scripts that used these `mongo` (page 610) shell methods for bulk write operations with "fire-and-forget" behavior should use the `Bulk()` (page 135) methods.

In sharded environments, applications using any driver or `mongo` (page 610) shell should use `Bulk()` (page 135) methods for optimal performance when inserting or modifying groups of documents.

For example, instead of:

```
for (var i = 1; i <= 1000000; i++) {
    db.test.insert( { x : i } );
}
```

In MongoDB 2.6, replace with `Bulk()` (page 135) operation:

[293] In previous versions, when using the `mongo` (page 610) shell interactively, the `mongo` (page 610) shell automatically called the `getLastError` (page 245) command after a write method to provide "safe writes". Scripts, however, would observe "fire-and-forget" behavior in previous versions unless the scripts included an **explicit** call to the `getLastError` (page 245) command after a write method.

```
var bulk = db.test.initializeUnorderedBulkOp();

for (var i = 1; i <= 1000000; i++) {
    bulk.insert( { x : i} );
}

bulk.execute( { w: 1 } );
```

Bulk method returns a `BulkWriteResult` (page 198) object that contains the result of the operation.

See also:

New Write Operation Protocol (page 745), `Bulk()` (page 135), `Bulk.execute()` (page 137), `db.collection.initializeUnorderedBulkOp()` (page 150), `db.collection.initializeOrderedBulkOp()` (page 149)

`db.collection.aggregate()` Change

Description The `db.collection.aggregate()` (page 22) method in the `mongo` (page 610) shell defaults to returning a cursor to the results set. This change enables the aggregation pipeline to return result sets of any size and requires cursor iteration to access the result set. For example:

```
var myCursor = db.orders.aggregate( [
    {
      $group: {
         _id: "$cust_id",
         total: { $sum: "$price" }
      }
    }
] );

myCursor.forEach( function(x) { printjson (x); } );
```

Previous versions returned a single document with a field `results` that contained an array of the result set, subject to the *BSON Document size* (page 692) limit. Accessing the result set in the previous versions of MongoDB required accessing the `results` field and iterating the array. For example:

```
var returnedDoc = db.orders.aggregate( [
    {
      $group: {
         _id: "$cust_id",
         total: { $sum: "$price" }
      }
    }
] );

var myArray = returnedDoc.result; // access the result field

myArray.forEach( function(x) { printjson (x); } );
```

Solution Update scripts that currently expect `db.collection.aggregate()` (page 22) to return a document with a `results` array to handle cursors instead.

See also:

Aggregation Enhancements (page 744), `db.collection.aggregate()` (page 22),

Write Concern Validation

Description Specifying a write concern that includes `j: true` to a `mongod` (page 583) or `mongos` (page 601) instance running with `--nojournal` (page 594) option now errors. Previous versions would ignore the `j: true`.

Solution Either remove the `j: true` specification from the write concern when issued against a `mongod` (page 583) or `mongos` (page 601) instance with `--nojournal` (page 594) or run `mongod` (page 583) or `mongos` (page 601) with journaling.

Security Changes

New Authorization Model

Description MongoDB 2.6 *authorization model* changes how MongoDB stores and manages user privilege information:

- Before the upgrade, MongoDB 2.6 requires at least one user in the admin database.
- MongoDB versions using older models cannot create/modify users or create user-defined roles.

Solution Ensure that at least one user exists in the admin database. If no user exists in the admin database, add a user. Then upgrade to MongoDB 2.6. Finally, upgrade the user privilege model. See *Upgrade MongoDB to 2.6* (page 760).

Important: Before upgrading the authorization model, you should first upgrade MongoDB binaries to 2.6. For sharded clusters, ensure that **all** cluster components are 2.6. If there are users in any database, be sure you have at least one user in the `admin` database **before** upgrading the MongoDB binaries.

See also:

Security Improvements (page 745)

SSL Certificate Hostname Validation

Description The SSL certificate validation now checks the Common Name (`CN`) and the Subject Alternative Name (`SAN`) fields to ensure that either the `CN` or one of the `SAN` entries matches the hostname of the server. As a result, if you currently use SSL and *neither* the `CN` nor any of the `SAN` entries of your current SSL certificates match the hostnames, upgrading to version 2.6 will cause the SSL connections to fail.

Solution To allow for the continued use of these certificates, MongoDB provides the `allowInvalidCertificates` setting. The setting is available for:

- `mongod` (page 583) and `mongos` (page 601) to bypass the validation of SSL certificates on other servers in the cluster.
- `mongo` (page 610) shell, *MongoDB tools that support SSL*, and the C++ driver to bypass the validation of server certificates.

When using the `allowInvalidCertificates` setting, MongoDB logs as a warning the use of the invalid certificates.

> **Warning:** The `allowInvalidCertificates` setting bypasses the other certificate validation, such as checks for expiration and valid signatures.

`2dsphere` Index Version 2

Description MongoDB 2.6 introduces a version 2 of the `2dsphere index`. If a document lacks a `2dsphere` index field (or the field is `null` or an empty array), MongoDB does not add an entry for the document to the `2dsphere` index. For inserts, MongoDB inserts the document but does not add to the `2dsphere` index.

Previous version would not insert documents where the `2dsphere` index field is a `null` or an empty array. For documents that lack the `2dsphere` index field, previous versions would insert and index the document.

Solution To revert to old behavior, create the `2dsphere` index with `{ "2dsphereIndexVersion" : 1 }` to create a version 1 index. However, version 1 index cannot use the new GeoJSON geometries.

See also:

2dsphere-v2

Log Messages

Timestamp Format Change

Description Each message now starts with the timestamp format given in *Time Format Changes* (page 759). Previous versions used the `ctime` format.

Solution MongoDB adds a new option *`--timeStampFormat`* (page 585) which supports timestamp format in *`ctime`* (page 585), *`iso8601-utc`* (page 585), and *`iso8601-local`* (page 585) (new default).

Package Configuration Changes

Default `bindIp` for RPM/DEB Packages

Description In the official MongoDB packages in RPM (Red Hat, CentOS, Fedora Linux, and derivatives) and DEB (Debian, Ubuntu, and derivatives), the default `bindIp` value attaches MongoDB components to the localhost interface *only*. These packages set this default in the default configuration file (i.e. `/etc/mongodb.conf`.)

Solution If you use one of these packages and have *not* modified the default `/etc/mongodb.conf` file, you will need to set `bindIp` before or during the upgrade.

There is no default `bindIp` setting in any other official MongoDB packages.

SNMP Changes

Description

- The IANA enterprise identifier for MongoDB changed from 37601 to 34601.
- MongoDB changed the MIB field name `globalopcounts` to `globalOpcounts`.

Solution

- Users of SNMP monitoring must modify their SNMP configuration (i.e. MIB) from 37601 to 34601.
- Update references to `globalopcounts` to `globalOpcounts`.

Remove Method Signature Change

Description `db.collection.remove()` (page 66) requires a query document as a parameter. In previous versions, the method invocation without a query document deleted all documents in a collection.

Solution For existing `db.collection.remove()` (page 66) invocations without a query document, modify the invocations to include an empty document `db.collection.remove({})`.

Update Operator Syntax Validation

Description

- *Update operators (e.g $set)* (page 451) must specify a non-empty operand expression. For example, the following expression is now invalid:

  ```
  { $set: { } }
  ```

- *Update operators (e.g $set)* (page 451) cannot repeat in the update statement. For example, the following expression is invalid:

  ```
  { $set: { a: 5 }, $set: { b: 5 } }
  ```

Updates Enforce Field Name Restrictions

Description

- Updates cannot use *update operators (e.g $set)* (page 451) to target fields with empty field names (i.e. `""`).
- Updates no longer support saving field names that contain a dot (`.`) or a field name that starts with a dollar sign (`$`).

Solution

- For existing documents that have fields with empty names `""`, replace the whole document. See `db.collection.update()` (page 72) and `db.collection.save()` (page 70) for details on replacing an existing document.
- For existing documents that have fields with names that contain a dot (`.`), either replace the whole document or `unset` (page 461) the field. To find fields whose names contain a dot, run `db.upgradeCheckAllDBs()` (page 128).
- For existing documents that have fields with names that start with a dollar sign (`$`), `unset` (page 461) or `rename` (page 457) those fields. To find fields whose names start with a dollar sign, run `db.upgradeCheckAllDBs()` (page 128).

See *New Write Operation Protocol* (page 745) for the changes to the write operation protocol, and *Insert and Update Improvements* (page 745) for the changes to the insert and update operations. Also consider the documentation of the `Restrictions on Field Names` (page 698).

Query and Sort Changes

Enforce Field Name Restrictions

Description Queries cannot specify conditions on fields with names that start with a dollar sign (`$`).

Solution `Unset` (page 461) or `rename` (page 457) existing fields whose names start with a dollar sign (`$`). Run `db.upgradeCheckAllDBs()` (page 128) to find fields whose names start with a dollar sign.

Sparse Index and Incomplete Results

Description If a `sparse index` results in an incomplete result set for queries and sort operations, MongoDB will not use that index unless a `hint()` (page 87) explicitly specifies the index.

For example, the query `{ x: { $exists: false } }` will no longer use a sparse index on the `x` field, unless explicitly hinted.

Solution To override the behavior to use the sparse index and return incomplete results, explicitly specify the index with a `hint()` (page 87).

See *sparse-index-incomplete-results* for an example that details the new behavior.

`sort()` Specification Values

Description The `sort()` (page 95) method **only** accepts the following values for the sort keys:

- `1` to specify ascending order for a field,
- `-1` to specify descending order for a field, or
- `$meta` (page 449) expression to specify sort by the text search score.

Any other value will result in an error.

Previous versions also accepted either `true` or `false` for ascending.

Solution Update sort key values that use `true` or `false` to `1`.

`skip()` and `_id` Queries

Description Equality match on the `_id` field obeys `skip()` (page 94).

Previous versions ignored `skip()` (page 94) when performing an equality match on the `_id` field.

`explain()` Retains Query Plan Cache

Description `explain()` (page 85) no longer clears the `query plans` cached for that *query shape*.

In previous versions, `explain()` (page 85) would have the side effect of clearing the query plan cache for that query shape.

See also:

The `PlanCache()` (page 134) reference.

Geospatial Changes

`$maxDistance` Changes

Description

- For `$near` (page 429) queries on GeoJSON data, if the queries specify a `$maxDistance` (page 434), `$maxDistance` (page 434) must be inside of the `$near` (page 429) document.

  In previous version, `$maxDistance` (page 434) could be either inside or outside the `$near` (page 429) document.

- `$maxDistance` (page 434) must be a positive value.

Solution

- Update any existing $near (page 429) queries on GeoJSON data that currently have the $maxDistance (page 434) outside the $near (page 429) document
- Update any existing queries where $maxDistance (page 434) is a negative value.

Deprecated $uniqueDocs

Description MongoDB 2.6 deprecates $uniqueDocs (page 437), and geospatial queries no longer return duplicated results when a document matches the query multiple times.

Stronger Validation of Geospatial Queries

Description MongoDB 2.6 enforces a stronger validation of geospatial queries, such as validating the options or GeoJSON specifications, and errors if the geospatial query is invalid. Previous versions allowed/ignored invalid options.

Query Operator Changes

$not Query Behavior Changes

Description

- Queries with $not (page 407) expressions on an indexed field now match:
  - Documents that are missing the indexed field. Previous versions would not return these documents using the index.
  - Documents whose indexed field value is a different type than that of the specified value. Previous versions would not return these documents using the index.

  For example, if a collection orders contains the following documents:

  ```
  { _id: 1, status: "A", cust_id: "123", price: 40 }
  { _id: 2, status: "A", cust_id: "xyz", price: "N/A" }
  { _id: 3, status: "D", cust_id: "xyz" }
  ```

  If the collection has an index on the price field:

  ```
  db.orders.ensureIndex( { price: 1 } )
  ```

  The following query uses the index to search for documents where price is not greater than or equal to 50:

  ```
  db.orders.find( { price: { $not: { $gte: 50 } } } )
  ```

  In 2.6, the query returns the following documents:

  ```
  { "_id" : 3, "status" : "D", "cust_id" : "xyz" }
  { "_id" : 1, "status" : "A", "cust_id" : "123", "price" : 40 }
  { "_id" : 2, "status" : "A", "cust_id" : "xyz", "price" : "N/A" }
  ```

  In previous versions, indexed plans would only return matching documents where the type of the field matches the type of the query predicate:

  ```
  { "_id" : 1, "status" : "A", "cust_id" : "123", "price" : 40 }
  ```

  If using a collection scan, previous versions would return the same results as those in 2.6.

- MongoDB 2.6 allows chaining of $not (page 407) expressions.

`null` Comparison Queries

Description

- `$lt` (page 403) and `$gt` (page 401) comparisons to `null` no longer match documents that are missing the field.
- `null` equality conditions on array elements (e.g. `"a.b": null`) no longer match document missing the nested field `a.b` (e.g. `a: [ 2, 3 ]`).
- `null` equality queries (i.e. `field: null`) now match fields with values `undefined`.

`$all` Operator Behavior Change

Description

- The `$all` (page 439) operator is now equivalent to an `$and` (page 405) operation of the specified values. This change in behavior can allow for more matches than previous versions when passed an array of a single nested array (e.g. `[ [ "A" ] ]`). When passed an array of a nested array, `$all` (page 439) can now match documents where the field contains the nested array as an element (e.g. `field: [ [ "A" ], ... ]`), *or* the field equals the nested array (e.g. `field: [ "A", "B" ]`). Earlier version could only match documents where the field contains the nested array.
- The `$all` (page 439) operator returns no match if the array field contains nested arrays (e.g. `field: [ "a", ["b"] ]`) and `$all` (page 439) on the nested field is the element of the nested array (e.g. `"field.1": { $all: [ "b" ] }`). Previous versions would return a match.

`$mod` Operator Enforces Strict Syntax

Description The `$mod` (page 412) operator now only accepts an array with exactly two elements, and errors when passed an array with fewer or more elements. See *Not Enough Elements Error* (page 413) and *Too Many Elements Error* (page 414) for details.

In previous versions, if passed an array with one element, the `$mod` (page 412) operator uses `0` as the second element, and if passed an array with more than two elements, the `$mod` (page 412) ignores all but the first two elements. Previous versions do return an error when passed an empty array.

Solution Ensure that the array passed to `$mod` (page 412) contains exactly two elements:

- If the array contains the a single element, add `0` as the second element.
- If the array contains more than two elements, remove the extra elements.

`$where` Must Be Top-Level

Description `$where` (page 421) expressions can now only be at top level and cannot be nested within another expression, such as `$elemMatch` (page 442).

Solution Update existing queries that nest `$where` (page 421).

`$exists` and `notablescan` If the MongoDB server has disabled collection scans, i.e. `notablescan`, then `$exists` (page 409) queries that have no *indexed solution* will error.

`MinKey` and `MaxKey` Queries

Description Equality match for either `MinKey` or `MaxKey` no longer match documents missing the field.

Nested Array Queries with $elemMatch

Description The `$elemMatch` (page 442) query operator no longer traverses recursively into nested arrays.

For example, if a collection `test` contains the following document:

```
{ "_id": 1, "a" : [ [ 1, 2, 5 ] ] }
```

In 2.6, the following `$elemMatch` (page 442) query does *not* match the document:

```
db.test.find( { a: { $elemMatch: { $gt: 1, $lt: 5 } } } )
```

Solution Update existing queries that rely upon the old behavior.

Text Search Compatibility MongoDB does not support the use of the `$text` (page 417) query operator in mixed sharded cluster deployments that contain both version 2.4 and version 2.6 shards. See *Upgrade MongoDB to 2.6* (page 760) for upgrade instructions.

Replica Set/Sharded Cluster Validation

Shard Name Checks on Metadata Refresh

Description For sharded clusters, MongoDB 2.6 disallows a shard from refreshing the metadata if the shard name has not been explicitly set.

For mixed sharded cluster deployments that contain both version 2.4 and version 2.6 shards, this change can cause errors when migrating chunks **from** version 2.4 shards **to** version 2.6 shards if the shard name is unknown to the version 2.6 shards. MongoDB does not support migrations in mixed sharded cluster deployments.

Solution Upgrade all components of the cluster to 2.6. See *Upgrade MongoDB to 2.6* (page 760).

Replica Set Vote Configuration Validation

Description MongoDB now deprecates giving any *replica set* member more than a single vote. During configuration, `local.system.replset.members[n].votes` should only have a value of 1 for voting members and 0 for non-voting members. MongoDB treats values other than 1 or 0 as a value of 1 and produces a warning message.

Solution Update `local.system.replset.members[n].votes` with values other than 1 or 0 to 1 or 0 as appropriate.

Time Format Changes MongoDB now uses `iso8601-local` when formatting time data in many outputs. This format follows the template `YYYY-MM-DDTHH:mm:ss.mmm<+/-Offset>`. For example, `2014-03-04T20:13:38.944-0500`.

This change impacts all clients using `Extended JSON` in *Strict mode*, such as `mongoexport` (page 649) and the REST and HTTP Interfaces[294].

Other Resources

- All backwards incompatible changes (JIRA)[295].
- *Release Notes for MongoDB 2.6* (page 726).

[294] http://docs.mongodb.org/ecosystem/tools/http-interfaces

[295] https://jira.mongodb.org/issues/?jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(%222.5.0%22%2C%20%222.5.1%22%2C%20%222.5.2%22
rc1%22%2C%20%222.6.0-rc2%22)%20AND%20%22Backwards%20Compatibility%22%20in%20%20(%22Major%20Change%22%2C%20%22Minor%20Change%2

- *Upgrade MongoDB to 2.6* (page 760) for the upgrade process.

Some changes in 2.6 can affect *compatibility* (page 749) and may require user actions. The 2.6 `mongo` (page 610) shell provides a `db.upgradeCheckAllDBs()` (page 128) method to perform a check for upgrade preparedness for some of these changes.

See *Compatibility Changes in MongoDB 2.6* (page 749) for a detailed list of compatibility changes.

See also:

All Backwards incompatible changes (JIRA)[296].

Upgrade Process

Upgrade MongoDB to 2.6 In the general case, the upgrade from MongoDB 2.4 to 2.6 is a binary-compatible "drop-in" upgrade: shut down the `mongod` (page 583) instances and replace them with `mongod` (page 583) instances running 2.6. **However**, before you attempt any upgrade, familiarize yourself with the content of this document, particularly the *Upgrade Recommendations and Checklists* (page 760), the procedure for *upgrading sharded clusters* (page 762), and the considerations for *reverting to 2.4 after running 2.6* (page 765).

Upgrade Recommendations and Checklists When upgrading, consider the following:

Upgrade Requirements To upgrade an existing MongoDB deployment to 2.6, you must be running 2.4. If you're running a version of MongoDB before 2.4, you *must* upgrade to 2.4 before upgrading to 2.6. See *Upgrade MongoDB to 2.4* (page 785) for the procedure to upgrade from 2.2 to 2.4.

If you use MMS Backup, ensure that you're running *at least* version `v20131216.1` of the Backup agent before upgrading. Version `1.4.0` of the backup agent followed `v20131216.1`

Preparedness Before upgrading MongoDB always test your application in a staging environment before deploying the upgrade to your production environment.

To begin the upgrade procedure, connect a 2.6 `mongo` (page 610) shell to your MongoDB 2.4 `mongos` (page 601) or `mongod` (page 583) and run the `db.upgradeCheckAllDBs()` (page 128) to check your data set for compatibility. This is a preliminary automated check. Assess and resolve all issues identified by `db.upgradeCheckAllDBs()` (page 128).

Some changes in MongoDB 2.6 require manual checks and intervention. See *Compatibility Changes in MongoDB 2.6* (page 749) for an explanation of these changes. Resolve all incompatibilities in your deployment before continuing.

Authentication MongoDB 2.6 includes significant changes to the authorization model, which requires changes to the way that MongoDB stores users' credentials. As a result, in addition to upgrading MongoDB processes, if your deployment uses authentication and authorization, after upgrading all MongoDB process to 2.6 you **must** also upgrade the authorization model.

After you begin to upgrade a MongoDB deployment that uses authentication to 2.6, you *cannot* modify existing user data until you complete the *authorization user schema upgrade* (page 764).

Before beginning the upgrade process for a deployment that uses authentication and authorization:

- Ensure that at least one user exists in the `admin` database.

[296] https://jira.mongodb.org/issues/?jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(%222.5.0%22%2C%20%222.5.1%22%2C%20%222.5.2%22
rc1%22%2C%20%222.6.0-rc2%22%2C%20%222.6.0-rc3%22)%20AND%20%22Backwards%20Compatibility%22%20in%20(%20%22Minor%20Change%22%2C%2

- If your application performs CRUD operations on the <database>.system.users collection or uses a db.addUser()-like method, then you **must** upgrade those drivers (i.e. client libraries) **before** mongod (page 583) or mongos (page 601) instances.
- You must fully complete the upgrade procedure for *all* MongoDB processes before upgrading the authorization model.

See *Upgrade User Authorization Data to 2.6 Format* (page 764) for a complete discussion of the upgrade procedure for the authorization model including additional requirements and procedures.

Downgrade Limitations Once upgraded to MongoDB 2.6, you **cannot** downgrade to **any** version earlier than MongoDB 2.4. If you created text or 2dsphere indexes while running 2.6, you can only downgrade to MongoDB 2.4.10 or later.

Package Upgrades If you installed MongoDB from the MongoDB apt or yum repositories, upgrade to 2.6 using the package manager.

For Debian, Ubuntu, and related operating systems, type these commands:

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

For Red Hat Enterprise, CentOS, Fedora, or Amazon Linux:

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

If you did not install the mongodb-org package, and installed a subset of MongoDB components replace mongodb-org in the commands above with the appropriate package names.

See installation instructions for Ubuntu, RHEL, Debian, or other Linux Systems for a list of the available packages and complete MongoDB installation instructions.

Upgrade MongoDB Processes

Upgrade Standalone mongod Instance to MongoDB 2.6 The following steps outline the procedure to upgrade a standalone mongod (page 583) from version 2.4 to 2.6. To upgrade from version 2.2 to 2.6, *upgrade to version 2.4* (page 785) *first*, and then follow the procedure to upgrade from 2.4 to 2.6.

1. Download binaries of the latest release in the 2.6 series from the MongoDB Download Page[297]. See http://docs.mongodb.org/manual/installation for more information.
2. Shut down your mongod (page 583) instance. Replace the existing binary with the 2.6 mongod (page 583) binary and restart mongod (page 583).

Upgrade a Replica Set to 2.6 The following steps outline the procedure to upgrade a replica set from MongoDB 2.4 to MongoDB 2.6. To upgrade from MongoDB 2.2 to 2.6, *upgrade all members of the replica set to version 2.4* (page 785) *first*, and then follow the procedure to upgrade from MongoDB 2.4 to 2.6.

You can upgrade from MongoDB 2.4 to 2.6 using a "rolling" upgrade to minimize downtime by upgrading the members individually while the other members are available:

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

Step 1: Upgrade secondary members of the replica set. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` (page 583) and replacing the 2.4 binary with the 2.6 binary. After upgrading a `mongod` (page 583) 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 179) in the `mongo` (page 610) shell.

Step 2: Step down the replica set primary. Use `rs.stepDown()` (page 179) in the `mongo` (page 610) shell to step down the *primary* and force the set to *failover*. `rs.stepDown()` (page 179) expedites the failover procedure and is preferable to shutting down the primary directly.

Step 3: Upgrade the primary. When `rs.status()` (page 179) shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` (page 583) binary with the 2.6 binary and start the new instance.

Replica set failover is not instant but will render the set unavailable accept writes until the failover process completes. Typically this takes 30 seconds or more: schedule the upgrade procedure during a scheduled maintenance window.

Upgrade a Sharded Cluster to 2.6 Only upgrade sharded clusters to 2.6 if **all** members of the cluster are currently running instances of 2.4. The only supported upgrade path for sharded clusters running 2.2 is via 2.4. The upgrade process checks all components of the cluster and will produce warnings if any component is running version 2.2.

Considerations The upgrade process does not require any downtime. However, while you upgrade the sharded cluster, ensure that clients do not make changes to the collection meta-data. For example, during the upgrade, do **not** do any of the following:

- `sh.enableSharding()` (page 186)
- `sh.shardCollection()` (page 189)
- `sh.addShard()` (page 183)
- `db.createCollection()` (page 104)
- `db.collection.drop()` (page 28)
- `db.dropDatabase()` (page 111)
- any operation that creates a database
- any other operation that modifies the cluster metadata in any way. See `http://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `http://docs.mongodb.org/manual/reference/sharding` page modifies the cluster meta-data.

Upgrade Sharded Clusters *Optional but Recommended.* As a precaution, take a backup of the `config` database *before* upgrading the sharded cluster.

Step 1: Disable the Balancer. Turn off the *balancer* in the sharded cluster, as described in *sharding-balancing-disable-temporarily*.

Step 2: Upgrade the cluster's meta data. Start a single 2.6 mongos (page 601) instance with the configDB pointing to the cluster's config servers and with the *--upgrade* option.

To run a mongos (page 601) with the *--upgrade* option, you can upgrade an existing mongos (page 601) instance to 2.6, or if you need to avoid reconfiguring a production mongos (page 601) instance, you can use a new 2.6 mongos (page 601) that can reach all the config servers.

To upgrade the meta data, run:

```
mongos --configdb <configDB string> --upgrade
```

You can include the *--logpath* option to output the log messages to a file instead of the standard output. Also include any other options required to start mongos (page 601) instances in your cluster, such as *--sslOnNormalPorts* or *--sslPEMKeyFile*.

The mongos (page 601) will exit upon completion of the *--upgrade* process.

The upgrade will prevent any chunk moves or splits from occurring during the upgrade process. If the data files have many sharded collections or if failed processes hold stale locks, acquiring the locks for all collections can take seconds or minutes. Watch the log for progress updates.

Step 3: Ensure `mongos --upgrade` process completes successfully. The mongos (page 601) will exit upon completion of the meta data upgrade process. If successful, the process will log the following messages:

```
upgrade of config server to v5 successful
Config database is at version v5
```

After a successful upgrade, restart the mongos (page 601) instance. If mongos (page 601) fails to start, check the log for more information.

If the mongos (page 601) instance loses its connection to the config servers during the upgrade or if the upgrade is otherwise unsuccessful, you may always safely retry the upgrade.

Step 4: Upgrade the remaining `mongos` instances to v2.6. Upgrade and restart **without** the *--upgrade* (page 593) option the other mongos (page 601) instances in the sharded cluster. After upgrading all the mongos (page 601), see *Complete Sharded Cluster Upgrade* (page 763) for information on upgrading the other cluster components.

Complete Sharded Cluster Upgrade After you have successfully upgraded *all* mongos (page 601) instances, you can upgrade the other instances in your MongoDB deployment.

> **Warning:** Do not upgrade mongod (page 583) instances until after you have upgraded *all* mongos (page 601) instances.

While the balancer is still disabled, upgrade the components of your sharded cluster in the following order:

- Upgrade all 3 mongod (page 583) config server instances, leaving the *first* system in the *mongos --configdb* argument to upgrade *last*.
- Upgrade each shard, one at a time, upgrading the mongod (page 583) secondaries before running replSetStepDown (page 301) and upgrading the primary of each shard.

When this process is complete, *re-enable the balancer*.

Upgrade Procedure Once upgraded to MongoDB 2.6, you **cannot** downgrade to **any** version earlier than MongoDB 2.4. If you have `text` or `2dsphere` indexes, you can only downgrade to MongoDB 2.4.10 or later.

Except as described on this page, moving between 2.4 and 2.6 is a drop-in replacement:

Step 1: Stop the existing `mongod` instance. For example, on Linux, run 2.4 `mongod` (page 583) with the *`--shutdown`* (page 589) option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath`. See also the *terminate-mongod-processes* for alternate methods of stopping a `mongod` (page 583) instance.

Step 2: Start the new `mongod` instance. Ensure you start the 2.6 `mongod` (page 583) with the same `dbPath`:

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath`.

Upgrade User Authorization Data to 2.6 Format MongoDB 2.6 includes significant changes to the authorization model, which requires changes to the way that MongoDB stores users' credentials. As a result, in addition to upgrading MongoDB processes, if your deployment uses authentication and authorization, after upgrading all MongoDB process to 2.6 you **must** also upgrade the authorization model.

Considerations

Complete all other Upgrade Requirements Before upgrading the authorization model, you should first upgrade MongoDB binaries to 2.6. For sharded clusters, ensure that **all** cluster components are 2.6. If there are users in any database, be sure you have at least one user in the `admin` database **before** upgrading the MongoDB binaries.

Timing Because downgrades are more difficult after you upgrade the user authorization model, once you upgrade the MongoDB binaries to version 2.6, allow your MongoDB deployment to run a day or two **without** upgrading the user authorization model.

This allows 2.6 some time to "burn in" and decreases the likelihood of downgrades occurring after the user privilege model upgrade. The user authentication and access control will continue to work as it did in 2.4, **but** it will be impossible to create or modify users or to use user-defined roles until you run the authorization upgrade.

If you decide to upgrade the user authorization model immediately instead of waiting the recommended "burn in" period, then for sharded clusters, you must wait at least 10 seconds after upgrading the sharded clusters to run the authorization upgrade script.

Replica Sets For a replica set, it is only necessary to run the upgrade process on the *primary* as the changes will automatically replicate to the secondaries.

Sharded Clusters For a sharded cluster, connect to a `mongos` (page 601) and run the upgrade procedure to upgrade the cluster's authorization data. By default, the procedure will upgrade the authorization data of the shards as well.

To override this behavior, run the upgrade command with the additional parameter `upgradeShards: false`. If you choose to override, you must run the upgrade procedure on the `mongos` (page 601) first, and then run the procedure on the *primary* members of each shard.

For a sharded cluster, do **not** run the upgrade process directly against the `config servers`. Instead, perform the upgrade process using one `mongos` (page 601) instance to interact with the config database.

Requirements To upgrade the authorization model, you must have a user in the `admin` database with the role `userAdminAnyDatabase`.

Procedure

Step 1: Connect to MongoDB instance. Connect and authenticate to the `mongod` (page 583) instance for a single deployment or a `mongos` (page 601) for a sharded cluster as an `admin` database user with the role `userAdminAnyDatabase`.

Step 2: Upgrade authorization schema. Use the `authSchemaUpgrade` (page 263) command in the `admin` database to update the user data using the `mongo` (page 610) shell.

Run `authSchemaUpgrade` command.

```
db.getSiblingDB("admin").runCommand({authSchemaUpgrade: 1 });
```

In case of error, you may safely rerun the `authSchemaUpgrade` (page 263) command.

Sharded cluster `authSchemaUpgrade` consideration. For a sharded cluster, `authSchemaUpgrade` (page 263) will upgrade the authorization data of the shards as well and the upgrade is complete. You can, however, override this behavior by including `upgradeShards: false` in the command, as in the following example:

```
db.getSiblingDB("admin").runCommand({authSchemaUpgrade: 1,
upgradeShards: false });
```

If you override the behavior, after running `authSchemaUpgrade` (page 263) on a `mongos` (page 601) instance, you will need to connect to the primary for each shard and repeat the upgrade process after upgrading on the `mongos` (page 601).

Result All users in a 2.6 system are stored in the `admin.system.users` (page 689) collection. To manipulate these users, use the *user management methods* (page 151).

The upgrade procedure copies the version 2.4 `admin.system.users` collection to `admin.system.backup_users`.

The upgrade procedure leaves the version 2.4 `<database>.system.users` collection(s) intact.

Downgrade MongoDB from 2.6 Before you attempt any downgrade, familiarize yourself with the content of this document, particularly the *Downgrade Recommendations and Checklist* (page 765) and the procedure for *downgrading sharded clusters* (page 770).

Downgrade Recommendations and Checklist When downgrading, consider the following:

Downgrade Path Once upgraded to MongoDB 2.6, you **cannot** downgrade to **any** version earlier than MongoDB 2.4. If you created `text` or `2dsphere` indexes while running 2.6, you can only downgrade to MongoDB 2.4.10 or later.

Preparedness

- *Remove or downgrade version 2 text indexes* (page 768) before downgrading MongoDB 2.6 to 2.4.
- *Remove or downgrade version 2 2dsphere indexes* (page 769) before downgrading MongoDB 2.6 to 2.4.
- *Downgrade 2.6 User Authorization Model* (page 766). If you have upgraded to the 2.6 user authorization model, you must downgrade the user model to 2.4 before downgrading MongoDB 2.6 to 2.4.

Procedures Follow the downgrade procedures:

- To downgrade sharded clusters, see *Downgrade a 2.6 Sharded Cluster* (page 770).
- To downgrade replica sets, see *Downgrade a 2.6 Replica Set* (page 769).
- To downgrade a standalone MongoDB instance, see *Downgrade 2.6 Standalone mongod Instance* (page 769).

Downgrade 2.6 User Authorization Model If you have upgraded to the 2.6 user authorization model, you **must first** downgrade the user authorization model to 2.4 **before** before downgrading MongoDB 2.6 to 2.4.

Considerations

- For a replica set, it is only necessary to run the downgrade process on the *primary* as the changes will automatically replicate to the secondaries.
- For sharded clusters, although the procedure lists the downgrade of the cluster's authorization data first, you may downgrade the authorization data of the cluster or shards first.
- You *must* have the `admin.system.backup_users` and `admin.system.new_users` collections created during the upgrade process.
- **Important**. The downgrade process returns the user data to its state prior to upgrading to 2.6 authorization model. Any changes made to the user/role data using the 2.6 users model will be lost.

Access Control Prerequisites To downgrade the authorization model, you must connect as a user with the following *privileges*:

```
{ resource: { db: "admin", collection: "system.new_users" }, actions: [ "find", "insert", "update" ]
{ resource: { db: "admin", collection: "system.backup_users" }, actions: [  "find" ] }
{ resource: { db: "admin", collection: "system.users" }, actions: [ "find", "remove", "insert"] }
{ resource: { db: "admin", collection: "system.version" }, actions: [ "find", "update" ] }
```

If no user exists with the appropriate *privileges*, create an authorization model downgrade user:

Step 1: Connect as user with privileges to manage users and roles. Connect and authenticate as a user with `userAdminAnyDatabase`.

Step 2: Create a role with required privileges. Using the `db.createRole` (page 161) method, create a *role* with the required privileges.

```
use admin
db.createRole(
  {
    role: "downgradeAuthRole",
    privileges: [
      { resource: { db: "admin", collection: "system.new_users" }, actions: [ "find", "insert", "upda
```

```
      { resource: { db: "admin", collection: "system.backup_users" }, actions: [  "find" ] },
      { resource: { db: "admin", collection: "system.users" }, actions: [ "find", "remove", "insert"
      { resource: { db: "admin", collection: "system.version" }, actions: [ "find", "update" ] }
    ],
    roles: [ ]
  }
)
```

Step 3: Create a user with the new role. Create a user and assign the user the `downgradeRole`.

```
use admin
db.createUser(
   {
     user: "downgradeAuthUser",
     pwd: "somePass123",
     roles: [ { role: "downgradeAuthRole", db: "admin" } ]
   }
)
```

Note: Instead of creating a new user, you can also grant the role to an existing user. See `db.grantRolesToUser()` (page 156) method.

Step 4: Authenticate as the new user. Authenticate as the newly created user.

```
use admin
db.auth( "downgradeAuthUser", "somePass123" )
```

The method returns `1` upon successful authentication.

Procedure The following downgrade procedure requires `<database>.system.users` collections used in version 2.4. to be intact for non-`admin` databases.

Step 1: Connect and authenticate to MongoDB instance. Connect and authenticate to the `mongod` (page 583) instance for a single deployment or a `mongos` (page 601) for a sharded cluster with the appropriate privileges. See *Access Control Prerequisites* (page 766) for details.

Step 2: Create backup of 2.6 `admin.system.users` collection. Copy all documents in the `admin.system.users` (page 689) collection to the `admin.system.new_users` collection:

```
db.getSiblingDB("admin").system.users.find().forEach( function(userDoc) {
    status = db.getSiblingDB("admin").system.new_users.save( userDoc );
    if (status.hasWriteError()) {
        print(status.writeError);
    }
  }
);
```

Step 3: Update the version document for the `authSchema`.

```
db.getSiblingDB("admin").system.version.update(
   { _id: "authSchema" },
   { $set: { currentVersion: 2 } }
);
```

The method returns a WriteResult (page 201) object with the status of the operation. Upon successful update, the WriteResult (page 201) object should have "nModified" equal to 1.

Step 4: Remove existing documents from the admin.system.users collection.

```
db.getSiblingDB("admin").system.users.remove( {} )
```

The method returns a WriteResult (page 201) object with the number of documents removed in the "nRemoved" field.

Step 5: Copy documents from the admin.system.backup_users collection. Copy all documents from the admin.system.backup_users, created during the 2.6 upgrade, to admin.system.users.

```
db.getSiblingDB("admin").system.backup_users.find().forEach(
   function (userDoc) {
      status = db.getSiblingDB("admin").system.users.insert( userDoc );
      if (status.hasWriteError()) {
          print(status.writeError);
      }
   }
);
```

Step 6: Update the version document for the authSchema.

```
db.getSiblingDB("admin").system.version.update(
   { _id: "authSchema" },
   { $set: { currentVersion: 1 } }
)
```

For a sharded cluster, repeat the downgrade process by connecting to the *primary* replica set member for each shard.

Note: The cluster's mongos (page 601) instances will fail to detect the authorization model downgrade until the user cache is refreshed. You can run invalidateUserCache (page 279) on each mongos (page 601) instance to refresh immediately, or you can wait until the cache is refreshed automatically at the end of the user cache invalidation interval. To run invalidateUserCache (page 279), you must have privilege with invalidateUserCache action, which is granted by userAdminAnyDatabase and hostManager roles.

Result The downgrade process returns the user data to its state prior to upgrading to 2.6 authorization model. Any changes made to the user/role data using the 2.6 users model will be lost.

Downgrade Updated Indexes

Text Index Version Check If you have *version 2* text indexes (i.e. the default version for text indexes in MongoDB 2.6), drop the *version 2* text indexes before downgrading MongoDB. After the downgrade, enable text search and recreate the dropped text indexes.

To determine the version of your text indexes, run db.collection.getIndexes() (page 48) to view index specifications. For text indexes, the method returns the version information in the field textIndexVersion. For example, the following shows that the text index on the quotes collection is version 2.

```
{
   "v" : 1,
   "key" : {
      "_fts" : "text",
      "_ftsx" : 1
   },
   "name" : "quote_text_translation.quote_text",
   "ns" : "test.quotes",
   "weights" : {
      "quote" : 1,
      "translation.quote" : 1
   },
   "default_language" : "english",
   "language_override" : "language",
   "textIndexVersion" : 2
}
```

2dsphere Index Version Check If you have *version 2* 2dsphere indexes (i.e. the default version for 2dsphere indexes in MongoDB 2.6), drop the *version 2* 2dsphere indexes before downgrading MongoDB. After the downgrade, recreate the 2dsphere indexes.

To determine the version of your 2dsphere indexes, run db.collection.getIndexes() (page 48) to view index specifications. For 2dsphere indexes, the method returns the version information in the field 2dsphereIndexVersion. For example, the following shows that the 2dsphere index on the locations collection is version 2.

```
{
   "v" : 1,
   "key" : {
      "geo" : "2dsphere"
   },
   "name" : "geo_2dsphere",
   "ns" : "test.locations",
   "sparse" : true,
   "2dsphereIndexVersion" : 2
}
```

Downgrade MongoDB Processes

Downgrade 2.6 Standalone mongod Instance The following steps outline the procedure to downgrade a standalone mongod (page 583) from version 2.6 to 2.4.

1. Download binaries of the latest release in the 2.4 series from the MongoDB Download Page[298]. See http://docs.mongodb.org/manual/installation for more information.
2. Shut down your mongod (page 583) instance. Replace the existing binary with the 2.4 mongod (page 583) binary and restart mongod (page 583).

Downgrade a 2.6 Replica Set The following steps outline a "rolling" downgrade process for the replica set. The "rolling" downgrade process minimizes downtime by downgrading the members individually while the other members are available:

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

Step 1: Downgrade each secondary member, one at a time. For each *secondary* in a replica set:

Replace and restart secondary `mongod` instances. First, shut down the `mongod` (page 583), then replace these binaries with the 2.4 binary and restart `mongod` (page 583). See *terminate-mongod-processes* for instructions on safely terminating `mongod` (page 583) processes.

Allow secondary to recover. Wait for the member to recover to `SECONDARY` state before upgrading the next secondary.

To check the member's state, use the `rs.status()` (page 179) method in the `mongo` (page 610) shell.

Step 2: Step down the primary. Use `rs.stepDown()` (page 179) in the `mongo` (page 610) shell to step down the *primary* and force the normal *failover* procedure.

```
rs.stepDown()
```

`rs.stepDown()` (page 179) expedites the failover procedure and is preferable to shutting down the primary directly.

Step 3: Replace and restart former primary `mongod`. When `rs.status()` (page 179) shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` (page 583) binary with the 2.4 binary and start the new instance.

Replica set failover is not instant but will render the set unavailable to writes and interrupt reads until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the downgrade during a predetermined maintenance window.

Downgrade a 2.6 Sharded Cluster

Requirements While the downgrade is in progress, you cannot make changes to the collection meta-data. For example, during the downgrade, do **not** do any of the following:

- `sh.enableSharding()` (page 186)
- `sh.shardCollection()` (page 189)
- `sh.addShard()` (page 183)
- `db.createCollection()` (page 104)
- `db.collection.drop()` (page 28)
- `db.dropDatabase()` (page 111)
- any operation that creates a database
- any other operation that modifies the cluster meta-data in any way. See `http://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `http://docs.mongodb.org/manual/reference/sharding` page modifies the cluster meta-data.

Procedure The downgrade procedure for a sharded cluster reverses the order of the upgrade procedure.

1. Turn off the *balancer* in the sharded cluster, as described in *sharding-balancing-disable-temporarily*.
2. Downgrade each shard, one at a time. For each shard,
   (a) Downgrade the `mongod` (page 583) secondaries *before* downgrading the primary.
   (b) To downgrade the primary, run `replSetStepDown` (page 301) and downgrade.
3. Downgrade all 3 `mongod` (page 583) config server instances, leaving the *first* system in the *mongos* `--configdb` argument to downgrade *last*.
4. Downgrade and restart each `mongos` (page 601), one at a time. The downgrade process is a binary drop-in replacement.
5. Turn on the balancer, as described in *sharding-balancing-enable*.

Downgrade Procedure Once upgraded to MongoDB 2.6, you **cannot** downgrade to **any** version earlier than MongoDB 2.4. If you have `text` or `2dsphere` indexes, you can only downgrade to MongoDB 2.4.10 or later.

Except as described on this page, moving between 2.4 and 2.6 is a drop-in replacement:

Step 1: Stop the existing `mongod` instance. For example, on Linux, run 2.6 `mongod` (page 583) with the *`--shutdown`* (page 589) option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath`. See also the *terminate-mongod-processes* for alternate methods of stopping a `mongod` (page 583) instance.

Step 2: Start the new `mongod` instance. Ensure you start the 2.4 `mongod` (page 583) with the same `dbPath`:

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath`.

See *Upgrade MongoDB to 2.6* (page 760) for full upgrade instructions.

Download

To download MongoDB 2.6, go to the downloads page[299].

Other Resources

- All JIRA issues resolved in 2.6[300].
- All Third Party License Notices[301].

[299] http://www.mongodb.org/downloads
[300] https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.5.0%22%2C+%222.5.1%22%2rc1%22%2C+%222.6.0-rc2%22%2C+%222.6.0-rc3%22%29
[301] https://github.com/mongodb/mongo/blob/v2.6/distsrc/THIRD-PARTY-NOTICES

7.3 Previous Stable Releases

7.3.1 Release Notes for MongoDB 2.4

March 19, 2013

MongoDB 2.4 includes enhanced geospatial support, switch to V8 JavaScript engine, security enhancements, and text search (beta) and hashed index.

Minor Releases

2.4 Changelog

2.4.12 - Changes

- Sharding: Sharded connection cleanup on setup error can crash mongos (SERVER-15056[302])
- Sharding: "type 7" (OID) error when acquiring distributed lock for first time (SERVER-13616[303])
- Storage: explicitly zero .ns files on creation (SERVER-15369[304])
- Storage: partially written journal last section causes recovery to fail (SERVER-15111[305])

2.4.11 - Changes

- Security: Potential information leak (SERVER-14268[306])
- Replication: `_id` with `$prefix` field causes replication failure due to unvalidated insert (SERVER-12209[307])
- Sharding: Invalid access: seg fault in `SplitChunkCommand::run` (SERVER-14342[308])
- Indexing: Creating descending index on `_id` can corrupt namespace (SERVER-14833[309])
- Text Search: Updates to documents with text-indexed fields may lead to incorrect entries (SERVER-14738[310])
- Build: Add SCons flag to override treating all warnings as errors (SERVER-13724[311])
- Packaging: Fix mongodb enterprise 2.4 init script to allow multiple processes per host (SERVER-14336[312])
- JavaScript: Do not store native function pointer as a property in function prototype (SERVER-14254[313])

2.4.10 - Changes

- Indexes: Fixed issue that can cause index corruption when building indexes concurrently (SERVER-12990[314])

[302] https://jira.mongodb.org/browse/SERVER-15056
[303] https://jira.mongodb.org/browse/SERVER-13616
[304] https://jira.mongodb.org/browse/SERVER-15369
[305] https://jira.mongodb.org/browse/SERVER-15111
[306] https://jira.mongodb.org/browse/SERVER-14268
[307] https://jira.mongodb.org/browse/SERVER-12209
[308] https://jira.mongodb.org/browse/SERVER-14342
[309] https://jira.mongodb.org/browse/SERVER-14833
[310] https://jira.mongodb.org/browse/SERVER-14738
[311] https://jira.mongodb.org/browse/SERVER-13724
[312] https://jira.mongodb.org/browse/SERVER-14336
[313] https://jira.mongodb.org/browse/SERVER-14254
[314] https://jira.mongodb.org/browse/SERVER-12990

- Indexes: Fixed issue that can cause index corruption when shutting down secondary node during index build (SERVER-12956[315])
- Indexes: Mongod now recognizes incompatible "future" text and geo index versions and exits gracefully (SERVER-12914[316])
- Indexes: Fixed issue that can cause secondaries to fail replication when building the same index multiple times concurrently (SERVER-12662[317])
- Indexes: Fixed issue that can cause index corruption on the tenth index in a collection if the index build fails (SERVER-12481[318])
- Indexes: Introduced versioning for text and geo indexes to ensure backwards compatibility (SERVER-12175[319])
- Indexes: Disallowed building indexes on the system.indexes collection, which can lead to initial sync failure on secondaries (SERVER-10231[320])
- Sharding: Avoid frequent immediate balancer retries when config servers are out of sync (SERVER-12908[321])
- Sharding: Add indexes to locks collection on config servers to avoid long queries in case of large numbers of collections (SERVER-12548[322])
- Sharding: Fixed issue that can corrupt the config metadata cache when sharding collections concurrently (SERVER-12515[323])
- Sharding: Don't move chunks created on collections with a hashed shard key if the collection already contains data (SERVER-9259[324])
- Replication: Fixed issue where node appears to be down in a replica set during a compact operation (SERVER-12264[325])
- Replication: Fixed issue that could cause delays in elections when a node is not vetoing an election (SERVER-12170[326])
- Replication: Step down all primaries if multiple primaries are detected in replica set to ensure correct election result (SERVER-10793[327])
- Replication: Upon clock skew detection, secondaries will switch to sync directly from the primary to avoid sync cycles (SERVER-8375[328])
- Runtime: The SIGXCPU signal is now caught and mongod writes a log message and exits gracefully (SERVER-12034[329])
- Runtime: Fixed issue where mongod fails to start on Linux when /sys/dev/block directory is not readable (SERVER-9248[330])

[315] https://jira.mongodb.org/browse/SERVER-12956
[316] https://jira.mongodb.org/browse/SERVER-12914
[317] https://jira.mongodb.org/browse/SERVER-12662
[318] https://jira.mongodb.org/browse/SERVER-12481
[319] https://jira.mongodb.org/browse/SERVER-12175
[320] https://jira.mongodb.org/browse/SERVER-10231
[321] https://jira.mongodb.org/browse/SERVER-12908
[322] https://jira.mongodb.org/browse/SERVER-12548
[323] https://jira.mongodb.org/browse/SERVER-12515
[324] https://jira.mongodb.org/browse/SERVER-9259
[325] https://jira.mongodb.org/browse/SERVER-12264
[326] https://jira.mongodb.org/browse/SERVER-12170
[327] https://jira.mongodb.org/browse/SERVER-10793
[328] https://jira.mongodb.org/browse/SERVER-8375
[329] https://jira.mongodb.org/browse/SERVER-12034
[330] https://jira.mongodb.org/browse/SERVER-9248

- Windows: No longer zero-fill newly allocated files on systems other than Windows 7 or Windows Server 2008 R2 (SERVER-8480[331])
- GridFS: Chunk size is decreased to 255 KB (from 256 KB) to avoid overhead with usePowerOf2Sizes option (SERVER-13331[332])
- SNMP: Fixed MIB file validation under smilint (SERVER-12487[333])
- Shell: Fixed issue in V8 memory allocation that could cause long-running shell commands to crash (SERVER-11871[334])
- Shell: Fixed memory leak in the md5sumFile shell utility method (SERVER-11560[335])

Previous Releases

- All 2.4.9 improvements[336].
- All 2.4.8 improvements[337].
- All 2.4.7 improvements[338].
- All 2.4.6 improvements[339].
- All 2.4.5 improvements[340].
- All 2.4.4 improvements[341].
- All 2.4.3 improvements[342].
- All 2.4.2 improvements[343]
- All 2.4.1 improvements[344].

2.4.12 – October 16, 2014

- Partially written journal last section causes recovery to fail SERVER-15111[345].
- Explicitly zero `.ns` files on creation SERVER-15369[346].
- *2.4.12 Changelog* (page 772).
- All 2.4.12 improvements[347].

[331] https://jira.mongodb.org/browse/SERVER-8480
[332] https://jira.mongodb.org/browse/SERVER-13331
[333] https://jira.mongodb.org/browse/SERVER-12487
[334] https://jira.mongodb.org/browse/SERVER-11871
[335] https://jira.mongodb.org/browse/SERVER-11560
[336] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.9%22%20AND%20project%20%3D%20SERVER
[337] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.8%22%20AND%20project%20%3D%20SERVER
[338] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.7%22%20AND%20project%20%3D%20SERVER
[339] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.6%22%20AND%20project%20%3D%20SERVER
[340] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.5%22%20AND%20project%20%3D%20SERVER
[341] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.4%22%20AND%20project%20%3D%20SERVER
[342] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.3%22%20AND%20project%20%3D%20SERVER
[343] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.2%22%20AND%20project%20%3D%20SERVER
[344] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.1%22%20AND%20project%20%3D%20SERVER
[345] https://jira.mongodb.org/browse/SERVER-15111
[346] https://jira.mongodb.org/browse/SERVER-15369
[347] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.12%22%20AND%20project%20%3D%20SERVER

2.4.11 – August 18, 2014

- Fixed potential information leak: SERVER-14268[348].
- Resolved issue were an `_id` with a `$prefix` field caused replication failure due to unvalidated insert SERVER-12209[349].
- Addressed issue where updates to documents with text-indexed fields could lead to incorrect entries SERVER-14738[350].
- Resolved issue where creating descending index on `_id` could corrupt namespace SERVER-14833[351].
- *2.4.11 Changelog* (page 772).
- All 2.4.11 improvements[352].

2.4.10 – April 4, 2014

- Performs fast file allocation on Windows when available SERVER-8480[353].
- Start elections if more than one primary is detected SERVER-10793[354].
- Changes to allow safe downgrading from v2.6 to v2.4 SERVER-12914[355], SERVER-12175[356].
- Fixes for edge cases in index creation SERVER-12481[357], SERVER-12956[358].
- *2.4.10 Changelog* (page 772).
- All 2.4.10 improvements[359].

2.4.9 – January 10, 2014

- Fix for instances where `mongos` (page 601) incorrectly reports a successful write SERVER-12146[360].
- Make non-primary read preferences consistent with `slaveOK` versioning logic SERVER-11971[361].
- Allow new sharded cluster connections to read from secondaries when primary is down SERVER-7246[362].
- All 2.4.9 improvements[363].

2.4.8 – November 1, 2013

- Increase future compatibility for 2.6 authorization features SERVER-11478[364].

[348] https://jira.mongodb.org/browse/SERVER-14268
[349] https://jira.mongodb.org/browse/SERVER-12209
[350] https://jira.mongodb.org/browse/SERVER-14738
[351] https://jira.mongodb.org/browse/SERVER-14833
[352] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.11%22%20AND%20project%20%3D%20SERVER
[353] https://jira.mongodb.org/browse/SERVER-8480
[354] https://jira.mongodb.org/browse/SERVER-10793
[355] https://jira.mongodb.org/browse/SERVER-12914
[356] https://jira.mongodb.org/browse/SERVER-12175
[357] https://jira.mongodb.org/browse/SERVER-12481
[358] https://jira.mongodb.org/browse/SERVER-12956
[359] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.10%22%20AND%20project%20%3D%20SERVER
[360] https://jira.mongodb.org/browse/SERVER-12146
[361] https://jira.mongodb.org/browse/SERVER-11971
[362] https://jira.mongodb.org/browse/SERVER-7246
[363] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.9%22%20AND%20project%20%3D%20SERVER
[364] https://jira.mongodb.org/browse/SERVER-11478

- Fix `dbhash` cache issue for config servers SERVER-11421[365].
- All 2.4.8 improvements[366].

2.4.7 – October 21, 2013

- Fixed over-aggressive caching of V8 Isolates SERVER-10596[367].
- Removed extraneous initial count during mapReduce SERVER-9907[368].
- Cache results of dbhash command SERVER-11021[369].
- Fixed memory leak in aggregation SERVER-10554[370].
- All 2.4.7 improvements[371].

2.4.6 – August 20, 2013

- Fix for possible loss of documents during the chunk migration process if a document in the chunk is very large SERVER-10478[372].
- Fix for C++ client shutdown issues SERVER-8891[373].
- Improved replication robustness in presence of high network latency SERVER-10085[374].
- Improved Solaris support SERVER-9832[375], SERVER-9786[376], and SERVER-7080[377].
- All 2.4.6 improvements[378].

2.4.5 – July 3, 2013

- Fix for CVE-2013-4650 Improperly grant user system privileges on databases other than local SERVER-9983[379].
- Fix for CVE-2013-3969 Remotely triggered segmentation fault in Javascript engine SERVER-9878[380].
- Fix to prevent identical background indexes from being built SERVER-9856[381].
- Config server performance improvements SERVER-9864[382] and SERVER-5442[383].
- Improved initial sync resilience to network failure SERVER-9853[384].

[365] https://jira.mongodb.org/browse/SERVER-11421
[366] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.8%22%20AND%20project%20%3D%20SERVER
[367] https://jira.mongodb.org/browse/SERVER-10596
[368] https://jira.mongodb.org/browse/SERVER-9907
[369] https://jira.mongodb.org/browse/SERVER-11021
[370] https://jira.mongodb.org/browse/SERVER-10554
[371] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.7%22%20AND%20project%20%3D%20SERVER
[372] https://jira.mongodb.org/browse/SERVER-10478
[373] https://jira.mongodb.org/browse/SERVER-8891
[374] https://jira.mongodb.org/browse/SERVER-10085
[375] https://jira.mongodb.org/browse/SERVER-9832
[376] https://jira.mongodb.org/browse/SERVER-9786
[377] https://jira.mongodb.org/browse/SERVER-7080
[378] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.6%22%20AND%20project%20%3D%20SERVER
[379] https://jira.mongodb.org/browse/SERVER-9983
[380] https://jira.mongodb.org/browse/SERVER-9878
[381] https://jira.mongodb.org/browse/SERVER-9856
[382] https://jira.mongodb.org/browse/SERVER-9864
[383] https://jira.mongodb.org/browse/SERVER-5442
[384] https://jira.mongodb.org/browse/SERVER-9853

- All 2.4.5 improvements[385].

2.4.4 – June 4, 2013

- Performance fix for Windows version SERVER-9721[386]
- Fix for config upgrade failure SERVER-9661[387].
- Migration to Cyrus SASL library for MongoDB Enterprise SERVER-8813[388].
- All 2.4.4 improvements[389].

2.4.3 – April 23, 2013

- Fix for `mongo` shell ignoring modified object's `_id` field SERVER-9385[390].
- Fix for race condition in log rotation SERVER-4739[391].
- Fix for `copydb` command with authorization in a sharded cluster SERVER-9093[392].
- All 2.4.3 improvements[393].

2.4.2 – April 17, 2013

- Several V8 memory leak and performance fixes SERVER-9267[394] and SERVER-9230[395].
- Fix for upgrading sharded clusters SERVER-9125[396].
- Fix for high volume connection crash SERVER-9014[397].
- All 2.4.2 improvements[398]

2.4.1 – April 17, 2013

- Fix for losing index changes during initial sync SERVER-9087[399]
- All 2.4.1 improvements[400].

Major New Features

The following changes in MongoDB affect both standard and Enterprise editions:

[385] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.5%22%20AND%20project%20%3D%20SERVER
[386] https://jira.mongodb.org/browse/SERVER-9721
[387] https://jira.mongodb.org/browse/SERVER-9661
[388] https://jira.mongodb.org/browse/SERVER-8813
[389] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.4%22%20AND%20project%20%3D%20SERVER
[390] https://jira.mongodb.org/browse/SERVER-9385
[391] https://jira.mongodb.org/browse/SERVER-4739
[392] https://jira.mongodb.org/browse/SERVER-9093
[393] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.3%22%20AND%20project%20%3D%20SERVER
[394] https://jira.mongodb.org/browse/SERVER-9267
[395] https://jira.mongodb.org/browse/SERVER-9230
[396] https://jira.mongodb.org/browse/SERVER-9125
[397] https://jira.mongodb.org/browse/SERVER-9014
[398] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.2%22%20AND%20project%20%3D%20SERVER
[399] https://jira.mongodb.org/browse/SERVER-9087
[400] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.1%22%20AND%20project%20%3D%20SERVER

Text Search

Add support for text search of content in MongoDB databases as a *beta* feature. See `http://docs.mongodb.org/manual/core/index-text` for more information.

Geospatial Support Enhancements

- Add new `2dsphere index`. The new index supports GeoJSON[401] objects `Point`, `LineString`, and `Polygon`. See `http://docs.mongodb.org/manual/core/2dsphere` and `http://docs.mongodb.org/manual/applications/geospatial-indexes`.
- Introduce operators `$geometry` (page 434), `$geoWithin` (page 425) and `$geoIntersects` (page 423) to work with the GeoJSON data.

Hashed Index

Add new *hashed index* to index documents using hashes of field values. When used to index a shard key, the hashed index ensures an evenly distributed shard key. See also *sharding-hashed-sharding*.

Improvements to the Aggregation Framework

- Improve support for geospatial queries. See the `$geoWithin` (page 425) operator and the `$geoNear` (page 484) pipeline stage.
- Improve sort efficiency when the `$sort` (page 499) stage immediately precedes a `$limit` (page 489) in the pipeline.
- Add new operators `$millisecond` (page 540) and `$concat` (page 525) and modify how `$min` (page 552) operator processes `null` values.

Changes to Update Operators

- Add new `$setOnInsert` (page 458) operator for use with `upsert` (page 72) .
- Enhance functionality of the `$push` (page 470) operator, supporting its use with the `$each` (page 472), the `$sort` (page 477), and the `$slice` (page 474) modifiers.

Additional Limitations for Map-Reduce and `$where` Operations

The `mapReduce` (page 220) command, `group` (page 216) command, and the `$where` (page 421) operator expressions cannot access certain global functions or properties, such as `db`, that are available in the `mongo` (page 610) shell. See the individual command or operator for details.

Improvements to `serverStatus` Command

Provide additional metrics and customization for the `serverStatus` (page 366) command. See `db.serverStatus()` (page 124) and `serverStatus` (page 366) for more information.

[401] http://geojson.org/geojson-spec.html

Security Enhancements

- Introduce a role-based access control system User Privileges[402] now use a new format for `Privilege Documents`.
- Enforce uniqueness of the user in user privilege documents per database. Previous versions of MongoDB did not enforce this requirement, and existing databases may have duplicates.
- Support encrypted connections using SSL certificates signed by a Certificate Authority. See `http://docs.mongodb.org/manual/tutorial/configure-ssl`.

For more information on security and risk management strategies, see `MongoDB Security Practices and Procedures`.

Performance Improvements

V8 JavaScript Engine

JavaScript Changes in MongoDB 2.4 Consider the following impacts of *V8 JavaScript Engine* (page 779) in MongoDB 2.4:

Tip

Use the new `interpreterVersion()` method in the `mongo` (page 610) shell and the `javascriptEngine` (page 347) field in the output of `db.serverBuildInfo()` (page 124) to determine which JavaScript engine a MongoDB binary uses.

Improved Concurrency Previously, MongoDB operations that required the JavaScript interpreter had to acquire a lock, and a single `mongod` (page 583) could only run a single JavaScript operation at a time. The switch to V8 improves concurrency by permitting multiple JavaScript operations to run at the same time.

Modernized JavaScript Implementation (ES5) The 5th edition of ECMAscript[403], abbreviated as ES5, adds many new language features, including:

- standardized JSON[404],
- strict mode[405],
- function.bind()[406],
- array extensions[407], and
- getters and setters.

With V8, MongoDB supports the ES5 implementation of Javascript with the following exceptions.

Note: The following features do not work as expected on documents **returned from MongoDB queries**:

- `Object.seal()` throws an exception on documents returned from MongoDB queries.
- `Object.freeze()` throws an exception on documents returned from MongoDB queries.

[402] http://docs.mongodb.org/v2.4/reference/user-privileges
[403] http://www.ecma-international.org/publications/standards/Ecma-262.htm
[404] http://www.ecma-international.org/ecma-262/5.1/#sec-15.12.1
[405] http://www.ecma-international.org/ecma-262/5.1/#sec-4.2.2
[406] http://www.ecma-international.org/ecma-262/5.1/#sec-15.3.4.5
[407] http://www.ecma-international.org/ecma-262/5.1/#sec-15.4.4.16

- `Object.preventExtensions()` incorrectly allows the addition of new properties on documents returned from MongoDB queries.
- `enumerable` properties, when added to documents returned from MongoDB queries, are not saved during write operations.

See SERVER-8216[408], SERVER-8223[409], SERVER-8215[410], and SERVER-8214[411] for more information.

For objects that have not been returned from MongoDB queries, the features work as expected.

Removed Non-Standard SpiderMonkey Features V8 does **not** support the following *non-standard* SpiderMonkey[412] JavaScript extensions, previously supported by MongoDB's use of SpiderMonkey as its JavaScript engine.

E4X Extensions V8 does not support the *non-standard* E4X[413] extensions. E4X provides a native XML[414] object to the JavaScript language and adds the syntax for embedding literal XML documents in JavaScript code.

You need to use alternative XML processing if you used any of the following constructors/methods:

- `XML()`
- `Namespace()`
- `QName()`
- `XMLList()`
- `isXMLName()`

Destructuring Assignment V8 does not support the non-standard destructuring assignments. Destructuring assignment "extract[s] data from arrays or objects using a syntax that mirrors the construction of array and object literals." - Mozilla docs[415]

Example

The following destructuring assignment is **invalid** with V8 and throws a `SyntaxError`:

```
original = [4, 8, 15];
var [b, ,c] = a;  // <== destructuring assignment
print(b) // 4
print(c) // 15
```

`Iterator()`, `StopIteration()`, and Generators V8 does not support Iterator(), StopIteration(), and generators[416].

`InternalError()` V8 does not support `InternalError()`. Use `Error()` instead.

[408] https://jira.mongodb.org/browse/SERVER-8216
[409] https://jira.mongodb.org/browse/SERVER-8223
[410] https://jira.mongodb.org/browse/SERVER-8215
[411] https://jira.mongodb.org/browse/SERVER-8214
[412] https://developer.mozilla.org/en-US/docs/SpiderMonkey
[413] https://developer.mozilla.org/en-US/docs/E4X
[414] https://developer.mozilla.org/en-US/docs/E4X/Processing_XML_with_E4X
[415] https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7#Destructuring_assignment_(Merge_into_own_page.2Fsection)
[416] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Iterators_and_Generators

`for each...in` Construct V8 does not support the use of for each...in[417] construct. Use `for (var x in y)` construct instead.

Example

The following `for each (var x in y)` construct is **invalid** with V8:

```
var o = { name: 'MongoDB', version: 2.4 };

for each (var value in o) {
  print(value);
}
```

Instead, in version 2.4, you can use the `for (var x in y)` construct:

```
var o = { name: 'MongoDB', version: 2.4 };

for (var prop in o) {
  var value = o[prop];
  print(value);
}
```

You can also use the array *instance* method `forEach()` with the ES5 method `Object.keys()`:

```
Object.keys(o).forEach(function (key) {
  var value = o[key];
  print(value);
});
```

Array Comprehension V8 does not support Array comprehensions[418].

Use other methods such as the `Array` instance methods `map()`, `filter()`, or `forEach()`.

Example

With V8, the following array comprehension is **invalid**:

```
var a = { w: 1, x: 2, y: 3, z: 4 }

var arr = [i * i for each (i in a) if (i > 2)]
printjson(arr)
```

Instead, you can implement using the `Array` *instance* method `forEach()` and the ES5 method `Object.keys()`:

```
var a = { w: 1, x: 2, y: 3, z: 4 }

var arr = [];
Object.keys(a).forEach(function (key) {
  var val = a[key];
  if (val > 2) arr.push(val * val);
})
printjson(arr)
```

Note: The new logic uses the `Array` *instance* method `forEach()` and not the *generic* method

[417] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/for_each...in
[418] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Predefined_Core_Objects#Array_comprehensions

`Array.forEach()`; V8 does **not** support `Array` *generic* methods. See *Array Generic Methods* (page 783) for more information.

Multiple Catch Blocks V8 does not support multiple `catch` blocks and will throw a `SyntaxError`.

Example

The following multiple catch blocks is **invalid** with V8 and will throw `"SyntaxError: Unexpected token if"`:

```
try {
  something()
} catch (err if err instanceof SomeError) {
  print('some error')
} catch (err) {
  print('standard error')
}
```

Conditional Function Definition V8 will produce different outcomes than SpiderMonkey with conditional function definitions[419].

Example

The following conditional function definition produces different outcomes in SpiderMonkey versus V8:

```
function test () {
   if (false) {
      function go () {};
   }
   print(typeof go)
}
```

With SpiderMonkey, the conditional function outputs `undefined`, whereas with V8, the conditional function outputs `function`.

If your code defines functions this way, it is highly recommended that you refactor the code. The following example refactors the conditional function definition to work in both SpiderMonkey and V8.

```
function test () {
  var go;
  if (false) {
    go = function () {}
  }
  print(typeof go)
}
```

The refactored code outputs `undefined` in both SpiderMonkey and V8.

Note: ECMAscript prohibits conditional function definitions. To force V8 to throw an `Error`, enable strict mode[420].

```
function test () {
  'use strict';
```

[419] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Functions
[420] http://www.nczonline.net/blog/2012/03/13/its-time-to-start-using-javascript-strict-mode/

```
  if (false) {
    function go () {}
  }
}
```

The JavaScript code throws the following syntax error:

```
SyntaxError: In strict mode code, functions can only be declared at top level or immediately within a
```

String Generic Methods V8 does not support String generics[421]. String generics are a set of methods on the `String` class that mirror instance methods.

Example

The following use of the generic method `String.toLowerCase()` is **invalid** with V8:

```
var name = 'MongoDB';

var lower = String.toLowerCase(name);
```

With V8, use the `String` instance method `toLowerCase()` available through an *instance* of the `String` class instead:

```
var name = 'MongoDB';

var lower = name.toLowerCase();
print(name + ' becomes ' + lower);
```

With V8, use the `String` *instance* methods instead of following *generic* methods:

`String.charAt()`	`String.quote()`	`String.toLocaleLowerCase()`
`String.charCodeAt()`	`String.replace()`	`String.toLocaleUpperCase()`
`String.concat()`	`String.search()`	`String.toLowerCase()`
`String.endsWith()`	`String.slice()`	`String.toUpperCase()`
`String.indexOf()`	`String.split()`	`String.trim()`
`String.lastIndexOf()`	`String.startsWith()`	`String.trimLeft()`
`String.localeCompare()`	`String.substr()`	`String.trimRight()`
`String.match()`	`String.substring()`	

Array Generic Methods V8 does not support Array generic methods[422]. Array generics are a set of methods on the `Array` class that mirror instance methods.

Example

The following use of the generic method `Array.every()` is **invalid** with V8:

```
var arr = [4, 8, 15, 16, 23, 42];

function isEven (val) {
   return 0 === val % 2;
}
```

[421] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String#String_generic_methods
[422] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array#Array_generic_methods

```
var allEven = Array.every(arr, isEven);
print(allEven);
```

With V8, use the `Array` instance method `every()` available through an *instance* of the `Array` class instead:

```
var allEven = arr.every(isEven);
print(allEven);
```

With V8, use the `Array` *instance* methods instead of the following *generic* methods:

`Array.concat()`	`Array.lastIndexOf()`	`Array.slice()`
`Array.every()`	`Array.map()`	`Array.some()`
`Array.filter()`	`Array.pop()`	`Array.sort()`
`Array.forEach()`	`Array.push()`	`Array.splice()`
`Array.indexOf()`	`Array.reverse()`	`Array.unshift()`
`Array.join()`	`Array.shift()`	

Array Instance Method `toSource()` V8 does not support the `Array` instance method toSource()[423]. Use the `Array` instance method `toString()` instead.

`uneval()` V8 does not support the non-standard method `uneval()`. Use the standardized JSON.stringify()[424] method instead.

Change default JavaScript engine from SpiderMonkey to V8. The change provides improved concurrency for JavaScript operations, modernized JavaScript implementation, and the removal of non-standard SpiderMonkey features, and affects all JavaScript behavior including the commands `mapReduce` (page 220), `group` (page 216), and `eval` (page 237) and the query operator `$where` (page 421).

See *JavaScript Changes in MongoDB 2.4* (page 779) for more information about all changes .

BSON Document Validation Enabled by Default for `mongod` and `mongorestore`

Enable basic *BSON* object validation for `mongod` (page 583) and `mongorestore` (page 628) when writing to MongoDB data files. See `wireObjectCheck` for details.

Index Build Enhancements

- Add support for multiple concurrent index builds in the background by a single `mongod` (page 583) instance. See *building indexes in the background* for more information on background index builds.
- Allow the `db.killOp()` (page 119) method to terminate a foreground index build.
- Improve index validation during index creation. See *Compatibility and Index Type Changes in MongoDB 2.4* (page 792) for more information.

Set Parameters as Command Line Options

Provide `--setParameter` as a command line option for `mongos` (page 601) and `mongod` (page 583). See `mongod` (page 583) and `mongos` (page 601) for list of available options for `setParameter`.

[423] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/toSource
[424] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/JSON/stringify

Changed Replication Behavior for Chunk Migration

By default, each document move during *chunk migration* in a *sharded cluster* propagates to at least one secondary before the balancer proceeds with its next operation. See *chunk-migration-replication*.

Improved Chunk Migration Queue Behavior

Increase performance for moving multiple chunks off an overloaded shard. The balancer no longer waits for the current migration's delete phase to complete before starting the next chunk migration. See *chunk-migration-queuing* for details.

Enterprise

The following changes are specific to MongoDB Enterprise Editions:

SASL Library Change

In 2.4.4, MongoDB Enterprise uses Cyrus SASL. Earlier 2.4 Enterprise versions use GNU SASL (`libgsasl`). To upgrade to 2.4.4 MongoDB Enterprise or greater, you **must** install all package dependencies related to this change, including the appropriate Cyrus SASL `GSSAPI` library. See `http://docs.mongodb.org/manual/administration/install-enterprise` for details of the dependencies.

New Modular Authentication System with Support for Kerberos

In 2.4, the MongoDB Enterprise now supports authentication via a Kerberos mechanism. See `http://docs.mongodb.org/manual/tutorial/control-access-to-mongodb-with-kerberos-authenticat` for more information. For drivers that provide support for Kerberos authentication to MongoDB, refer to *kerberos-and-drivers*.

For more information on security and risk management strategies, see `MongoDB Security Practices and Procedures`.

Additional Information

Platform Notes

For OS X, MongoDB 2.4 only supports OS X versions 10.6 (Snow Leopard) and later. There are no other platform support changes in MongoDB 2.4. See the downloads page[425] for more information on platform support.

Upgrade Process

Upgrade MongoDB to 2.4 In the general case, the upgrade from MongoDB 2.2 to 2.4 is a binary-compatible "drop-in" upgrade: shut down the `mongod` (page 583) instances and replace them with `mongod` (page 583) instances running 2.4. **However**, before you attempt any upgrade please familiarize yourself with the content of this document, particularly the procedure for *upgrading sharded clusters* (page 786) and the considerations for *reverting to 2.2 after running 2.4* (page 791).

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

Upgrade Recommendations and Checklist When upgrading, consider the following:

- For all deployments using authentication, upgrade the drivers (i.e. client libraries), before upgrading the `mongod` (page 583) instance or instances.
- To upgrade to 2.4 sharded clusters *must* upgrade following the *meta-data upgrade procedure* (page 786).
- If you're using 2.2.0 and running with `authorization` enabled, you will need to upgrade first to 2.2.1 and then upgrade to 2.4. See *Rolling Upgrade Limitation for 2.2.0 Deployments Running with auth Enabled* (page 790).
- If you have `system.users` documents (i.e. for `authorization`) that you created before 2.4 you *must* ensure that there are no duplicate values for the `user` field in the `system.users` collection in *any* database. If you *do* have documents with duplicate user fields, you must remove them before upgrading.

  See *Security Enhancements* (page 779) for more information.

Upgrade Standalone `mongod` Instance to MongoDB 2.4

1. Download binaries of the latest release in the 2.4 series from the MongoDB Download Page[426]. See `http://docs.mongodb.org/manual/installation` for more information.
2. Shutdown your `mongod` (page 583) instance. Replace the existing binary with the 2.4 `mongod` (page 583) binary and restart `mongod` (page 583).

Upgrade a Replica Set from MongoDB 2.2 to MongoDB 2.4 You can upgrade to 2.4 by performing a "rolling" upgrade of the set by upgrading the members individually while the other members are available to minimize downtime. Use the following procedure:

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` (page 583) and replacing the 2.2 binary with the 2.4 binary. After upgrading a `mongod` (page 583) 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 179) in the `mongo` (page 610) shell.
2. Use the `mongo` (page 610) shell method `rs.stepDown()` (page 179) to step down the *primary* to allow the normal *failover* procedure. `rs.stepDown()` (page 179) 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 179), shut down the previous primary and replace `mongod` (page 583) binary with the 2.4 binary and start the new process.

   Note: Replica set failover is not instant but will render the set unavailable to read or accept writes until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the upgrade during a predefined maintenance window.

Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4

Important: Only upgrade sharded clusters to 2.4 if **all** members of the cluster are currently running instances of 2.2. The only supported upgrade path for sharded clusters running 2.0 is via 2.2.

Overview Upgrading a *sharded cluster* from MongoDB version 2.2 to 2.4 (or 2.3) requires that you run a 2.4 `mongos` (page 601) with the `--upgrade` option, described in this procedure. The upgrade process does not require downtime.

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

The upgrade to MongoDB 2.4 adds epochs to the meta-data for all collections and chunks in the existing cluster. MongoDB 2.2 processes are capable of handling epochs, even though 2.2 did not require them. This procedure applies only to upgrades from version 2.2. Earlier versions of MongoDB do not correctly handle epochs. See *Cluster Meta-data Upgrade* (page 787) for more information.

After completing the meta-data upgrade you can fully upgrade the components of the cluster. With the balancer disabled:

- Upgrade all `mongos` (page 601) instances in the cluster.
- Upgrade all 3 `mongod` (page 583) config server instances.
- Upgrade the `mongod` (page 583) instances for each shard, one at a time.

See *Upgrade Sharded Cluster Components* (page 790) for more information.

Cluster Meta-data Upgrade

Considerations Beware of the following properties of the cluster upgrade process:

- Before you start the upgrade, ensure that the amount of free space on the filesystem for the *config database* (page 681) is at least 4 to 5 times the amount of space currently used by the *config database* (page 681) data files.

  Additionally, ensure that all indexes in the *config database* (page 681) are `{v:1}` indexes. If a critical index is a `{v:0}` index, chunk splits can fail due to known issues with the `{v:0}` format. `{v:0}` indexes are present on clusters created with MongoDB 2.0 or earlier.

  The duration of the metadata upgrade depends on the network latency between the node performing the upgrade and the three config servers. Ensure low latency between the upgrade process and the config servers.

- While the upgrade is in progress, you cannot make changes to the collection meta-data. For example, during the upgrade, do **not** perform:
  - `sh.enableSharding()` (page 186),
  - `sh.shardCollection()` (page 189),
  - `sh.addShard()` (page 183),
  - `db.createCollection()` (page 104),
  - `db.collection.drop()` (page 28),
  - `db.dropDatabase()` (page 111),
  - any operation that creates a database, or
  - any other operation that modifies the cluster meta-data in any way. See `http://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `http://docs.mongodb.org/manual/reference/sharding` page modifies the cluster meta-data.
- Once you upgrade to 2.4 and complete the upgrade procedure **do not** use 2.0 `mongod` (page 583) and `mongos` (page 601) processes in your cluster. 2.0 process may re-introduce old meta-data formats into cluster meta-data.

The upgraded config database will require more storage space than before, to make backup and working copies of the `config.chunks` (page 683) and `config.collections` (page 683) collections. As always, if storage requirements increase, the `mongod` (page 583) might need to pre-allocate additional data files. See *faq-tools-for-measuring-storage-use* for more information.

Meta-data Upgrade Procedure Changes to the meta-data format for sharded clusters, stored in the *config database* (page 681), require a special meta-data upgrade procedure when moving to 2.4.

Do not perform operations that modify meta-data while performing this procedure. See *Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4* (page 786) for examples of prohibited operations.

1. Before you start the upgrade, ensure that the amount of free space on the filesystem for the *config database* (page 681) is at least 4 to 5 times the amount of space currently used by the *config database* (page 681) data files.

   Additionally, ensure that all indexes in the *config database* (page 681) are `{v:1}` indexes. If a critical index is a `{v:0}` index, chunk splits can fail due to known issues with the `{v:0}` format. `{v:0}` indexes are present on clusters created with MongoDB 2.0 or earlier.

   The duration of the metadata upgrade depends on the network latency between the node performing the upgrade and the three config servers. Ensure low latency between the upgrade process and the config servers.

   To check the version of your indexes, use `db.collection.getIndexes()` (page 48).

   If any index **on the config database** is `{v:0}`, you should rebuild those indexes by connecting to the `mongos` (page 601) and either: rebuild all indexes using the `db.collection.reIndex()` (page 65) method, or drop and rebuild specific indexes using `db.collection.dropIndex()` (page 29) and then `db.collection.ensureIndex()` (page 30). If you need to upgrade the `_id` index to `{v:1}` use `db.collection.reIndex()` (page 65).

   You may have `{v:0}` indexes on other databases in the cluster.

2. Turn off the *balancer* in the *sharded cluster*, as described in *sharding-balancing-disable-temporarily*.

   Optional

   For additional security during the upgrade, you can make a backup of the config database using `mongodump` (page 622) or other backup tools.

   ---

3. Ensure there are no version 2.0 `mongod` (page 583) or `mongos` (page 601) processes still active in the sharded cluster. The automated upgrade process checks for 2.0 processes, but network availability can prevent a definitive check. Wait 5 minutes after stopping or upgrading version 2.0 `mongos` (page 601) processes to confirm that none are still active.

4. Start a single 2.4 `mongos` (page 601) process with `configDB` pointing to the sharded cluster's *config servers* and with the `--upgrade` option. The upgrade process happens before the process becomes a daemon (i.e. before `--fork`.)

   You can upgrade an existing `mongos` (page 601) instance to 2.4 or you can start a new *mongos* instance that can reach all config servers if you need to avoid reconfiguring a production `mongos` (page 601).

   Start the `mongos` (page 601) with a command that resembles the following:

   ```
   mongos --configdb <config servers> --upgrade
   ```

   Without the `--upgrade` option 2.4 `mongos` (page 601) processes will fail to start until the upgrade process is complete.

   The upgrade will prevent any chunk moves or splits from occurring during the upgrade process. If there are very many sharded collections or there are stale locks held by other failed processes, acquiring the locks for all collections can take seconds or minutes. See the log for progress updates.

5. When the `mongos` (page 601) process starts successfully, the upgrade is complete. If the `mongos` (page 601) process fails to start, check the log for more information.

   If the `mongos` (page 601) terminates or loses its connection to the config servers during the upgrade, you may always safely retry the upgrade.

However, if the upgrade failed during the short critical section, the mongos (page 601) will exit and report that the upgrade will require manual intervention. To continue the upgrade process, you must follow the *Resync after an Interruption of the Critical Section* (page 789) procedure.

Optional

If the mongos (page 601) logs show the upgrade waiting for the upgrade lock, a previous upgrade process may still be active or may have ended abnormally. After 15 minutes of no remote activity mongos (page 601) will force the upgrade lock. If you can verify that there are no running upgrade processes, you may connect to a 2.2 mongos (page 601) process and force the lock manually:

```
mongo <mongos.example.net>

db.getMongo().getCollection("config.locks").findOne({ _id : "configUpgrade" })
```

If the process specified in the process field of this document is *verifiably* offline, run the following operation to force the lock.

```
db.getMongo().getCollection("config.locks").update({ _id : "configUpgrade" }, { $set : { state :
```

It is always more safe to wait for the mongos (page 601) to verify that the lock is inactive, if you have any doubts about the activity of another upgrade operation. In addition to the configUpgrade, the mongos (page 601) may need to wait for specific collection locks. Do not force the specific collection locks.

---

6. Upgrade and restart other mongos (page 601) processes in the sharded cluster, *without* the *--upgrade* option.

   See *Upgrade Sharded Cluster Components* (page 790) for more information.

7. *Re-enable the balancer*. You can now perform operations that modify cluster meta-data.

Once you have upgraded, *do not* introduce version 2.0 MongoDB processes into the sharded cluster. This can reintroduce old meta-data formats into the config servers. The meta-data change made by this upgrade process will help prevent errors caused by cross-version incompatibilities in future versions of MongoDB.

Resync after an Interruption of the Critical Section During the short critical section of the upgrade that applies changes to the meta-data, it is unlikely but possible that a network interruption can prevent all three config servers from verifying or modifying data. If this occurs, the *config servers* must be re-synced, and there may be problems starting new mongos (page 601) processes. The *sharded cluster* will remain accessible, but avoid all cluster meta-data changes until you resync the config servers. Operations that change meta-data include: adding shards, dropping databases, and dropping collections.

Note: Only perform the following procedure *if* something (e.g. network, power, etc.) interrupts the upgrade process during the short critical section of the upgrade. Remember, you may always safely attempt the *meta data upgrade procedure* (page 788).

To resync the config servers:

1. Turn off the *balancer* in the sharded cluster and stop all meta-data operations. If you are in the middle of an upgrade process (*Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4* (page 786)), you have already disabled the balancer.

2. Shut down two of the three config servers, preferably the last two listed in the configDB string. For example, if your configDB string is configA:27019,configB:27019,configC:27019, shut down configB and configC. Shutting down the last two config servers ensures that most mongos (page 601) instances will have uninterrupted access to cluster meta-data.

3. mongodump (page 622) the data files of the active config server (configA).

4. Move the data files of the deactivated config servers (`configB` and `configC`) to a backup location.
5. Create new, empty *data directories*.
6. Restart the disabled config servers with *--dbpath* pointing to the now-empty data directory and *--port* pointing to an alternate port (e.g. `27020`).
7. Use `mongorestore` (page 628) to repopulate the data files on the disabled documents from the active config server (`configA`) to the restarted config servers on the new port (`configB:27020,configC:27020`). These config servers are now re-synced.
8. Restart the restored config servers on the old port, resetting the port back to the old settings (`configB:27019` and `configC:27019`).
9. In some cases connection pooling may cause spurious failures, as the `mongos` (page 601) disables old connections only after attempted use. 2.4 fixes this problem, but to avoid this issue in version 2.2, you can restart all `mongos` (page 601) instances (one-by-one, to avoid downtime) and use the `rs.stepDown()` (page 179) method before restarting each of the shard *replica set primaries*.
10. The sharded cluster is now fully resynced; however before you attempt the upgrade process again, you must manually reset the upgrade state using a version 2.2 `mongos` (page 601). Begin by connecting to the 2.2 `mongos` (page 601) with the `mongo` (page 610) shell:

    ```
    mongo <mongos.example.net>
    ```

    Then, use the following operation to reset the upgrade process:

    ```
    db.getMongo().getCollection("config.version").update({ _id : 1 }, { $unset : { upgradeState : 1
    ```

11. Finally retry the upgrade process, as in *Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4* (page 786).

Upgrade Sharded Cluster Components After you have successfully completed the meta-data upgrade process described in *Meta-data Upgrade Procedure* (page 788), and the 2.4 `mongos` (page 601) instance starts, you can upgrade the other processes in your MongoDB deployment.

While the balancer is still disabled, upgrade the components of your sharded cluster in the following order:

- Upgrade all `mongos` (page 601) instances in the cluster, in any order.
- Upgrade all 3 `mongod` (page 583) config server instances, upgrading the *first* system in the *mongos --configdb* argument *last*.
- Upgrade each shard, one at a time, upgrading the `mongod` (page 583) secondaries before running `replSetStepDown` (page 301) and upgrading the primary of each shard.

When this process is complete, you can now *re-enable the balancer*.

Rolling Upgrade Limitation for 2.2.0 Deployments Running with `auth` Enabled MongoDB *cannot* support deployments that mix 2.2.0 and 2.4.0, or greater, components. MongoDB version 2.2.1 and later processes *can* exist in mixed deployments with 2.4-series processes. Therefore you cannot perform a rolling upgrade from MongoDB 2.2.0 to MongoDB 2.4.0. To upgrade a cluster with 2.2.0 components, use one of the following procedures.

1. Perform a rolling upgrade of all 2.2.0 processes to the latest 2.2-series release (e.g. 2.2.3) so that there are no processes in the deployment that predate 2.2.1. When there are no 2.2.0 processes in the deployment, perform a rolling upgrade to 2.4.0.
2. Stop all processes in the cluster. Upgrade all processes to a 2.4-series release of MongoDB, and start all processes at the same time.

Upgrade from 2.3 to 2.4 If you used a mongod (page 583) from the 2.3 or 2.4-rc (release candidate) series, you can safely transition these databases to 2.4.0 or later; *however*, if you created 2dsphere or text indexes using a mongod (page 583) before v2.4-rc2, you will need to rebuild these indexes. For example:

```
db.records.dropIndex( { loc: "2dsphere" } )
db.records.dropIndex( "records_text" )

db.records.ensureIndex( { loc: "2dsphere" } )
db.records.ensureIndex( { records: "text" } )
```

Downgrade MongoDB from 2.4 to Previous Versions For some cases the on-disk format of data files used by 2.4 and 2.2 mongod (page 583) is compatible, and you can upgrade and downgrade if needed. However, several new features in 2.4 are incompatible with previous versions:

- 2dsphere indexes are incompatible with 2.2 and earlier mongod (page 583) instances.
- text indexes are incompatible with 2.2 and earlier mongod (page 583) instances.
- using a hashed index as a shard key are incompatible with 2.2 and earlier mongos (page 601) instances.
- hashed indexes are incompatible with 2.0 and earlier mongod (page 583) instances.

Important: Collections sharded using hashed shard keys, should **not** use 2.2 mongod (page 583) instances, which cannot correctly support cluster operations for these collections.

If you completed the *meta-data upgrade for a sharded cluster* (page 786), you can safely downgrade to 2.2 MongoDB processes. **Do not** use 2.0 processes after completing the upgrade procedure.

Note: In sharded clusters, once you have completed the *meta-data upgrade procedure* (page 786), you cannot use 2.0 mongod (page 583) or mongos (page 601) instances in the same cluster.

If you complete the meta-data upgrade, you can safely downgrade components in any order. When upgrade again, always upgrade mongos (page 601) instances before mongod (page 583) instances.

Do not create 2dsphere or text indexes in a cluster that has 2.2 components.

Considerations and Compatibility If you upgrade to MongoDB 2.4, and then need to run MongoDB 2.2 with the same data files, consider the following limitations.

- If you use a hashed index as the shard key index, which is only possible under 2.4 you will not be able to query data in this sharded collection. Furthermore, a 2.2 mongos (page 601) cannot properly route an insert operation for a collections sharded using a hashed index for the shard key index: any data that you insert using a 2.2 mongos (page 601), will not arrive on the correct shard and will not be reachable by future queries.
- If you *never* create an 2dsphere or text index, you can move between a 2.4 and 2.2 mongod (page 583) for a given data set; however, after you create the first 2dsphere or text index with a 2.4 mongod (page 583) you will need to run a 2.2 mongod (page 583) with the *--upgrade* option and drop any 2dsphere or text index.

Upgrade and Downgrade Procedures

Basic Downgrade and Upgrade **Except** as described below, moving between 2.2 and 2.4 is a drop-in replacement:

- stop the existing mongod (page 583), using the *--shutdown* option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath`.

- start the new `mongod` (page 583) processes with the same `dbPath` setting, for example:

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath`.

Downgrade to 2.2 After Creating a `2dsphere` or `text` Index If you have created `2dsphere` or `text` indexes while running a 2.4 `mongod` (page 583) instance, you can downgrade at any time, by starting the `2.2` `mongod` (page 583) with the *`--upgrade`* option as follows:

```
mongod --dbpath /var/mongod/data/ --upgrade
```

Then, you will need to drop any existing `2dsphere` or `text` indexes using `db.collection.dropIndex()` (page 29), for example:

```
db.records.dropIndex( { loc: "2dsphere" } )
db.records.dropIndex( "records_text" )
```

> **Warning:** *`--upgrade`* will run `repairDatabase` (page 341) on any database where you have created a `2dsphere` or `text` index, which will rebuild *all* indexes.

Troubleshooting Upgrade/Downgrade Operations If you do not use *`--upgrade`*, when you attempt to start a 2.2 `mongod` (page 583) and you have created a `2dsphere` or `text` index, `mongod` (page 583) will return the following message:

```
'need to upgrade database index_plugin_upgrade with pdfile version 4.6, new version: 4.5 Not upgradi
```

While running 2.4, to check the data file version of a MongoDB database, use the following operation in the shell:

```
db.getSiblingDB('<databasename>').stats().dataFileVersion
```

The major data file [427] version for both 2.2 and 2.4 is `4`, the minor data file version for 2.2 is `5` and the minor data file version for 2.4 is `6` **after** you create a `2dsphere` or `text` index.

Compatibility and Index Type Changes in MongoDB 2.4 In 2.4 MongoDB includes two new features related to indexes that users upgrading to version 2.4 must consider, particularly with regard to possible downgrade paths. For more information on downgrades, see *Downgrade MongoDB from 2.4 to Previous Versions* (page 791).

New Index Types In 2.4 MongoDB adds two new index types: `2dsphere` and `text`. These index types do not exist in 2.2, and for each database, creating a `2dsphere` or `text` index, will upgrade the data-file version and make that database incompatible with 2.2.

If you intend to downgrade, you should always drop all `2dsphere` and `text` indexes before moving to 2.2.

You can use the *downgrade procedure* (page 791) to downgrade these databases and run 2.2 if needed, however this will run a full database repair (as with `repairDatabase` (page 341)) for all affected databases.

[427] The data file version (i.e. `pdfile version`) is independent and unrelated to the release version of MongoDB.

Index Type Validation In MongoDB 2.2 and earlier you could specify invalid index types that did not exist. In these situations, MongoDB would create an ascending (e.g. `1`) index. Invalid indexes include index types specified by strings that do not refer to an existing index type, and all numbers other than `1` and `-1`. [428]

In 2.4, creating any invalid index will result in an error. Furthermore, you cannot create a `2dsphere` or `text` index on a collection if its containing database has any invalid index types. [1]

Example

If you attempt to add an invalid index in MongoDB 2.4, as in the following:

```
db.coll.ensureIndex( { field: "1" } )
```

MongoDB will return the following error document:

```
{
  "err" : "Unknown index plugin '1' in index { field: \"1\" }"
  "code": 16734,
  "n": <number>,
  "connectionId": <number>,
  "ok": 1
}
```

See *Upgrade MongoDB to 2.4* (page 785) for full upgrade instructions.

Other Resources

- MongoDB Downloads[429].
- All JIRA issues resolved in 2.4[430].
- All Backwards incompatible changes[431].
- All Third Party License Notices[432].

7.3.2 Release Notes for MongoDB 2.2

Upgrading

MongoDB 2.2 is a production release series and succeeds the 2.0 production release series.

MongoDB 2.0 data files are compatible with 2.2-series binaries without any special migration process. However, always perform the upgrade process for replica sets and sharded clusters using the procedures that follow.

Synopsis

- `mongod` (page 583), 2.2 is a drop-in replacement for 2.0 and 1.8.

[428] In 2.4, indexes that specify a type of `"1"` or `"-1"` (the strings `"1"` and `"-1"`) will continue to exist, despite a warning on start-up. **However**, a *secondary* in a replica set cannot complete an initial sync from a primary that has a `"1"` or `"-1"` index. Avoid all indexes with invalid types.

[429] http://mongodb.org/downloads

[430] https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.3.2%22,+%222.3.1%22,+%222
rc0%22,+%222.4.0-rc1%22,+%222.4.0-rc2%22,+%222.4.0-rc3%22%29

[431] https://jira.mongodb.org/issues/?jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(%222.3.2%22%2C%20%222.3.1%22%2C%20%222.3.0%2
rc0%22%2C%20%222.4.0-rc1%22%2C%20%222.4.0-rc2%22%2C%20%222.4.0-rc3%22)%20AND%20%22Backwards%20Compatibility%22%20in%20(%22Major%

[432] https://github.com/mongodb/mongo/blob/v2.4/distsrc/THIRD-PARTY-NOTICES

- Check your `driver` documentation for information regarding required compatibility upgrades, and always run the recent release of your driver.

  Typically, only users running with authentication, will need to upgrade drivers before continuing with the upgrade to 2.2.

- For all deployments using authentication, upgrade the drivers (i.e. client libraries), before upgrading the `mongod` (page 583) instance or instances.

- For all upgrades of sharded clusters:

  – turn off the balancer during the upgrade process. See the *sharding-balancing-disable-temporarily* section for more information.

  – upgrade all `mongos` (page 601) instances before upgrading any `mongod` (page 583) instances.

Other than the above restrictions, 2.2 processes can interoperate with 2.0 and 1.8 tools and processes. You can safely upgrade the `mongod` (page 583) and `mongos` (page 601) 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[433].
2. Shutdown your `mongod` (page 583) instance. Replace the existing binary with the 2.2 `mongod` (page 583) binary and restart MongoDB.

Upgrading a Replica Set

You can upgrade to 2.2 by performing a "rolling" upgrade of the set by upgrading the members individually while the other members are available to minimize downtime. Use the following procedure:

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` (page 583) and replacing the 2.0 binary with the 2.2 binary. After upgrading a `mongod` (page 583) 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 179) in the `mongo` (page 610) shell.
2. Use the `mongo` (page 610) shell method `rs.stepDown()` (page 179) to step down the *primary* to allow the normal *failover* procedure. `rs.stepDown()` (page 179) 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 179), shut down the previous primary and replace `mongod` (page 583) 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.*

[433] http://downloads.mongodb.org/

- Upgrade all `mongos` (page 601) instances *first*, in any order.
- Upgrade all of the `mongod` (page 583) config server instances using the *stand alone* (page 794) 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 794) 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[434] for more information.

Changes

Major Features

Aggregation Framework The aggregation framework makes it possible to do aggregation operations without needing to use *map-reduce*. The `aggregate` (page 210) command exposes the aggregation framework, and the `aggregate()` (page 22) helper in the `mongo` (page 610) shell provides an interface to these operations. Consider the following resources for background on the aggregation framework and its use:

- Documentation: `http://docs.mongodb.org/manual/core/aggregation`
- Reference: *Aggregation Reference* (page 564)
- Examples: `http://docs.mongodb.org/manual/applications/aggregation`

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[435]
2. Improved Yielding on Page Faults[436]
3. Improved Page Fault Detection on Windows[437]

To reflect these changes, MongoDB now provides changed and improved reporting for concurrency and use, see *locks* (page 368) and *recordStats* (page 380) in *server status* (page 366) and see `db.currentOp()` (page 105), *mongotop* (page 664), and *mongostat* (page 657).

[434] https://jira.mongodb.org/browse/SERVER-6902
[435] https://jira.mongodb.org/browse/SERVER-4328
[436] https://jira.mongodb.org/browse/SERVER-3357
[437] https://jira.mongodb.org/browse/SERVER-4538

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` and *write concern*. For example, shard tagging can pin all "USA" data to one or more logical shards, while replica set tagging can control which `mongod` (page 583) instances (e.g. "`production`" or "`reporting`") the application uses to service requests.

See the documentation for the following helpers in the `mongo` (page 610) shell that support tagged sharding configuration:

- `sh.addShardTag()` (page 184)
- `sh.addTagRange()` (page 184)
- `sh.removeShardTag()` (page 188)

Also, see `http://docs.mongodb.org/manual/core/tag-aware-sharding` and `http://docs.mongodb.org/manual/tutorial/administer-shard-tags`.

Fully Supported Read Preference Semantics All MongoDB clients and drivers now support full `read preferences`, including consistent support for a full range of *read preference modes* and *tag sets*. This support extends to the `mongos` (page 601) and applies identically to single replica sets and to the replica sets for each shard in a *sharded cluster*.

Additional read preference support now exists in the `mongo` (page 610) shell using the `readPref()` (page 93) cursor method.

Compatibility Changes

Authentication Changes MongoDB 2.2 provides more reliable and robust support for authentication clients, including drivers and `mongos` (page 601) 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 794).

`findAndModify` Returns Null Value for Upserts that Perform Inserts In version 2.2, for *upsert* that perform inserts with the `new` option set to `false`, `findAndModify` (page 239) commands will now return the following output:

```
{ 'ok': 1.0, 'value': null }
```

In the `mongo` (page 610) shell, upsert `findAndModify` (page 239) 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[438] for more information.

[438] https://jira.mongodb.org/browse/SERVER-6226

`mongodump` 2.2 Output Incompatible with Pre-2.2 `mongorestore` If you use the `mongodump` (page 622) tool from the 2.2 distribution to create a dump of a database, you must use a 2.2 (or later) version of `mongorestore` (page 628) to restore that dump.

See: SERVER-6961[439] for more information.

`ObjectId().toString()` Returns String Literal `ObjectId("...")` In version 2.2, the `toString()` (page 200) method returns the string representation of the *ObjectId()* object and has the format `ObjectId("...")`.

Consider the following example that calls the `toString()` (page 200) 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*, which holds the hexadecimal string value in both versions.

`ObjectId().valueOf()` Returns hexadecimal string In version 2.2, the `valueOf()` (page 200) method returns the value of the *ObjectId()* object as a lowercase hexadecimal string.

Consider the following example that calls the `valueOf()` (page 200) 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* 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.

[439] https://jira.mongodb.org/browse/SERVER-6961

Collections names may have any other valid UTF-8 string.

See the SERVER-4442[440] and the *faq-restrictions-on-collection-names* FAQ item.

Restrictions on Database Names for Windows Database names running on Windows can no longer contain the following characters:

```
/\. "*<>:|?
```

The names of the data files include the database name. If you attempt to upgrade a database instance with one or more of these characters, `mongod` (page 583) will refuse to start.

Change the name of these databases before upgrading. See SERVER-4584[441] and SERVER-6729[442] 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[443] for more information.

New `$elemMatch` Projection Operator The `$elemMatch` (page 446) operator allows applications to narrow the data returned from queries so that the query operation will only return the first matching element in an array. See the `$elemMatch` (page 446) reference and the SERVER-2238[444] and SERVER-828[445] 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[446] for more information.

Service Support for `mongos.exe` You may now run `mongos.exe` (page 619) instances as a Windows Service. See the `mongos.exe` (page 619) reference and *tutorial-mongod-as-windows-service* and SERVER-1589[447] for more information.

Log Rotate Command Support MongoDB for Windows now supports log rotation by way of the `logRotate` (page 339) database command. See SERVER-2612[448] for more information.

New Build Using SlimReadWrite Locks for Windows Concurrency Labeled "2008+" on the Downloads Page[449], 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[450] for more information.

[440] https://jira.mongodb.org/browse/SERVER-4442
[441] https://jira.mongodb.org/browse/SERVER-4584
[442] https://jira.mongodb.org/browse/SERVER-6729
[443] https://jira.mongodb.org/browse/SERVER-5516
[444] https://jira.mongodb.org/browse/SERVER-2238
[445] https://jira.mongodb.org/browse/SERVER-828
[446] https://jira.mongodb.org/browse/SERVER-5648
[447] https://jira.mongodb.org/browse/SERVER-1589
[448] https://jira.mongodb.org/browse/SERVER-2612
[449] http://www.mongodb.org/downloads
[450] https://jira.mongodb.org/browse/SERVER-3844

Tool Improvements

Index Definitions Handled by `mongodump` and `mongorestore` When you specify the `--collection` option to `mongodump` (page 622), `mongodump` (page 622) will now backup the definitions for all indexes that exist on the source database. When you attempt to restore this backup with `mongorestore` (page 628), the target `mongod` (page 583) will rebuild all indexes. See SERVER-808[451] for more information.

`mongorestore` (page 628) now includes the `--noIndexRestore` option to provide the preceding behavior. Use `--noIndexRestore` to prevent `mongorestore` (page 628) from building previous indexes.

`mongooplog` for Replaying Oplogs The `mongooplog` (page 637) tool makes it possible to pull *oplog* entries from `mongod` (page 583) instance and apply them to another `mongod` (page 583) instance. You can use `mongooplog` (page 637) to achieve point-in-time backup of a MongoDB data set. See the SERVER-3873[452] case and the `mongooplog` (page 637) reference.

Authentication Support for `mongotop` and `mongostat` `mongotop` (page 664) and `mongostat` (page 657) now contain support for username/password authentication. See SERVER-3875[453] and SERVER-3871[454] for more information regarding this change. Also consider the documentation of the following options for additional information:

- `mongotop --username`
- `mongotop --password`
- `mongostat --username`
- `mongostat --password`

Write Concern Support for `mongoimport` and `mongorestore` `mongoimport` (page 642) 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[455] for more information.

In `mongorestore` (page 628), the `--w` option provides support for configurable write concern.

`mongodump` Support for Reading from Secondaries You can now run `mongodump` (page 622) when connected to a *secondary* member of a *replica set*. See SERVER-3854[456] for more information.

`mongoimport` Support for full 16MB Documents Previously, `mongoimport` (page 642) would only import documents that were less than 4 megabytes in size. This issue is now corrected, and you may use `mongoimport` (page 642) to import documents that are at least 16 megabytes ins size. See SERVER-4593[457] for more information.

`Timestamp()` Extended JSON format MongoDB extended JSON now includes a new `Timestamp()` type to represent the Timestamp type that MongoDB uses for timestamps in the *oplog* among other contexts.

This permits tools like `mongooplog` (page 637) and `mongodump` (page 622) to query for specific timestamps. Consider the following `mongodump` (page 622) operation:

[451] https://jira.mongodb.org/browse/SERVER-808
[452] https://jira.mongodb.org/browse/SERVER-3873
[453] https://jira.mongodb.org/browse/SERVER-3875
[454] https://jira.mongodb.org/browse/SERVER-3871
[455] https://jira.mongodb.org/browse/SERVER-3937
[456] https://jira.mongodb.org/browse/SERVER-3854
[457] https://jira.mongodb.org/browse/SERVER-4593

```
mongodump --db local --collection oplog.rs --query '{"ts":{"$gt":{"$timestamp" : {"t": 1344969612000,
```

See SERVER-3483[458] for more information.

Shell Improvements

Improved Shell User Interface 2.2 includes a number of changes that improve the overall quality and consistency of the user interface for the `mongo` (page 610) shell:

- Full Unicode support.
- Bash-like line editing features. See SERVER-4312[459] for more information.
- Multi-line command support in shell history. See SERVER-3470[460] for more information.
- Windows support for the `edit` command. See SERVER-3998[461] for more information.

Helper to load Server-Side Functions The `db.loadServerScripts()` (page 120) loads the contents of the current database's `system.js` collection into the current `mongo` (page 610) shell session. See SERVER-1651[462] for more information.

Support for Bulk Inserts If you pass an array of *documents* to the `insert()` (page 55) method, the `mongo` (page 610) shell will now perform a bulk insert operation. See SERVER-3819[463] and SERVER-2395[464] for more information.

Note: For bulk inserts on sharded clusters, the `getLastError` (page 245) 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[465] case and the documentation of the `syslogFacility` run-time option or the *`mongod --syslog`* and *`mongos --syslog`* command line-options.

`touch` Command Added the `touch` (page 344) command to read the data and/or indexes from a collection into memory. See: SERVER-2023[466] and `touch` (page 344) 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[467] and `indexCounters` for more information.

[458] https://jira.mongodb.org/browse/SERVER-3483
[459] https://jira.mongodb.org/browse/SERVER-4312
[460] https://jira.mongodb.org/browse/SERVER-3470
[461] https://jira.mongodb.org/browse/SERVER-3998
[462] https://jira.mongodb.org/browse/SERVER-1651
[463] https://jira.mongodb.org/browse/SERVER-3819
[464] https://jira.mongodb.org/browse/SERVER-2395
[465] https://jira.mongodb.org/browse/SERVER-2957
[466] https://jira.mongodb.org/browse/SERVER-2023
[467] https://jira.mongodb.org/browse/SERVER-5784

Padding Specifiable on `compact` Command See the documentation of the `compact` (page 322) and the SERVER-4018[468] 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[469] and SERVER-5172[470] 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[471] and SERVER-4683[472]. For more information about TCMalloc, see the documentation of TCMalloc[473] itself.

Replication

Improved Logging for Replica Set Lag When *secondary* members of a replica set fall behind in replication, `mongod` (page 583) 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[474] for more information.

Replica Set Members can Sync from Specific Members The new `replSetSyncFrom` (page 302) command and new `rs.syncFrom()` (page 179) helper in the `mongo` (page 610) 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 302) 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[475] for more information.

New Option To Configure Index Pre-Fetching during Replication By default, when replicating options, *secondaries* will pre-fetch *indexes* associated with a query to improve replication throughput in most cases. The `replication.secondaryIndexPrefetch` setting and *`--replIndexPrefetch`* option allow administrators to disable this feature or allow the `mongod` (page 583) to pre-fetch only the index on the `_id` field. See SERVER-6718[476] for more information.

[468] https://jira.mongodb.org/browse/SERVER-4018
[469] https://jira.mongodb.org/browse/SERVER-3829
[470] https://jira.mongodb.org/browse/SERVER-5172
[471] https://jira.mongodb.org/browse/SERVER-188
[472] https://jira.mongodb.org/browse/SERVER-4683
[473] http://goog-perftools.sourceforge.net/doc/tcmalloc.html
[474] https://jira.mongodb.org/browse/SERVER-3575
[475] https://jira.mongodb.org/browse/SERVER-4160
[476] https://jira.mongodb.org/browse/SERVER-6718

Map Reduce Improvements

In 2.2 Map Reduce received the following improvements:

- Improved support for sharded MapReduce[477], and
- MapReduce will retry jobs following a config error[478].

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 *sharding-shard-key-indexes* documentation and SERVER-1506[479] for more information.

Migration Thresholds Modified The *migration thresholds* have changed in 2.2 to permit more even distribution of *chunks* in collections that have smaller quantities of data. See the *sharding-migration-thresholds* documentation for more information.

Licensing Changes

Added License notice for Google Perftools (TCMalloc Utility). See the License Notice[480] and the SERVER-4683[481] for more information.

Resources

- MongoDB Downloads[482].
- All JIRA issues resolved in 2.2[483].
- All backwards incompatible changes[484].
- All third party license notices[485].
- What's New in MongoDB 2.2 Online Conference[486].

7.3.3 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.

[477] https://jira.mongodb.org/browse/SERVER-4521
[478] https://jira.mongodb.org/browse/SERVER-4158
[479] https://jira.mongodb.org/browse/SERVER-1506
[480] https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES#L231
[481] https://jira.mongodb.org/browse/SERVER-4683
[482] http://mongodb.org/downloads
[483] https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.1.0%22%2C+%222.1.1%22%2
rc0%22%2C+%222.2.0-rc1%22%2C+%222.2.0-rc2%22%29+ORDER+BY+component+ASC%2C+key+DESC
[484] https://jira.mongodb.org/issues/?filter=11225&jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(10483%2C%2010893%2C%2010894%2C%2
[485] https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES
[486] http://www.mongodb.com/events/webinar/mongodb-online-conference-sept

Preparation

Read through all release notes before upgrading, and ensure that no changes will affect your deployment.

If you create new indexes in 2.0, then downgrading to 1.8 is possible but you must reindex the new collections.

`mongoimport` (page 642) and `mongoexport` (page 649) 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[487].

`Journaling` is **enabled by default** in 2.0 for 64-bit builds. If you still prefer to run without journaling, start `mongod` (page 583) with the `--nojournal` run-time option. Otherwise, MongoDB creates journal files during startup. The first time you start `mongod` (page 583) with journaling, you will see a delay as `mongod` (page 583) creates new files. In addition, you may see reduced write throughput.

2.0 `mongod` (page 583) instances are interoperable with 1.8 `mongod` (page 583) instances; however, for best results, upgrade your deployments using the following procedures:

Upgrading a Standalone `mongod`

1. Download the v2.0.x binaries from the MongoDB Download Page[488].
2. Shutdown your `mongod` (page 583) instance. Replace the existing binary with the 2.0.x `mongod` (page 583) binary and restart MongoDB.

Upgrading a Replica Set

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` (page 583) and replacing the 1.8 binary with the 2.0.x binary from the MongoDB Download Page[489].
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* in your application code to confirm that each update reaches multiple servers.
3. Use the `rs.stepDown()` (page 179) to step down the primary to allow the normal *failover* procedure.

   `rs.stepDown()` (page 179) and `replSetStepDown` (page 301) provide for shorter and more consistent failover procedures than simply shutting down the primary directly.

   When the primary has stepped down, shut down its instance and upgrade by replacing the `mongod` (page 583) binary with the 2.0.x binary.

Upgrading a Sharded Cluster

1. Upgrade all *config server* instances *first*, in any order. Since config servers use two-phase commit, *shard* configuration metadata updates will halt until all are up and running.
2. Upgrade `mongos` (page 601) routers in any order.

[487] https://jira.mongodb.org/browse/SERVER-1097
[488] http://downloads.mongodb.org/
[489] http://downloads.mongodb.org/

Changes

Compact Command

A `compact` (page 322) 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[490] for more information.

The specific operations yield in 2.0 are:

- Updates by `_id`
- Removes
- Long cursor iterations

Default Stack Size

MongoDB 2.0 reduces the default stack size. This change can reduce total memory usage when there are many (e.g., 1000+) client connections, as there is a thread per connection. While portions of a thread's stack can be swapped out if unused, some operating systems do this slowly enough that it might be an issue. The default stack size is lesser of the system setting or 1MB.

Index Performance Enhancements

v2.0 includes significant improvements to the `index`. 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[491].
- The `repairDatabase` (page 341) command converts indexes to a 2.0 indexes.

To convert all indexes for a given collection to the *2.0 type* (page 804), invoke the `compact` (page 322) 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*.

[490] https://jira.mongodb.org/browse/SERVER-2563
[491] https://jira.mongodb.org/browse/SERVER-3866

Replica Sets

Hidden Nodes in Sharded Clusters In 2.0, `mongos` (page 601) instances can now determine when a member of a replica set becomes "hidden" without requiring a restart. In 1.8, `mongos` (page 601) if you reconfigured a member as hidden, you *had* to restart `mongos` (page 601) to prevent queries from reaching the hidden member.

Priorities Each *replica set* member can now have a priority value consisting of a floating-point from 0 to 1000, inclusive. Priorities let you control which member of the set you prefer to have as *primary* the member with the highest priority that can see a majority of the set will be elected primary.

For example, suppose you have a replica set with three members, `A`, `B`, and `C`, and suppose that their priorities are set as follows:

- `A`'s priority is `2`.
- `B`'s priority is `3`.
- `C`'s priority is `1`.

During normal operation, the set will always chose `B` as primary. If `B` becomes unavailable, the set will elect `A` as primary.

For more information, see the `priority` documentation.

Data-Center Awareness You can now "tag" *replica set* members to indicate their location. You can use these tags to design custom *write rules* across data centers, racks, specific servers, or any other architecture choice.

For example, an administrator can define rules such as "very important write" or `customerData` or "audit-trail" to replicate to certain servers, racks, data centers, etc. Then in the application code, the developer would say:

```
db.foo.insert(doc, {w : "very important write"})
```

which would succeed if it fulfilled the conditions the DBA defined for "very important write".

For more information, see `/data-center-awareness`.

Drivers may also support tag-aware reads. Instead of specifying `slaveOk`, you specify `slaveOk` with tags indicating which data-centers to read from. For details, see the Drivers[492] 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 `http://docs.mongodb.org/manual/core/write-concern`.

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 `http://docs.mongodb.org/manual/tutorial/reconfigure-replica-set-with-unava`

Primary Checks for a Caught up Secondary before Stepping Down To minimize time without a *primary*, the `rs.stepDown()` (page 179) 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 `http://docs.mongodb.org/manual/tutorial/force-member-to-be-primary`.

[492] http://docs.mongodb.org/ecosystem/drivers

Extended Shutdown on the Primary to Minimize Interruption When you call the `shutdown` (page 344) command, the *primary* will refuse to shut down unless there is a *secondary* whose optime is within 10 seconds of the primary. If such a secondary isn't available, the primary will step down and wait up to a minute for the secondary to be fully caught up before shutting down.

Note that to get this behavior, you must issue the `shutdown` (page 344) command explicitly; sending a signal to the process will not trigger this behavior.

You can also force the primary to shut down, even without an up-to-date secondary available.

Maintenance Mode When `repair` or `compact` (page 322) 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 *geospatial-indexes-multi-location*.

Polygon searches Polygonal `$within` (page 427) queries are also now supported for simple polygon shapes. For details, see the `$within` (page 427) operator documentation.

Journaling Enhancements

- Journaling is now enabled by default for 64-bit platforms. Use the `--nojournal` command line option to disable it.
- The journal is now compressed for faster commits to disk.
- A new `--journalCommitInterval` run-time option exists for specifying your own group commit interval. The default settings do not change.
- A new `{ getLastError: { j: true } }` (page 245) 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` (page 245) command will report whether any inserts have failed, not just the last one. If multiple errors occur, the client will only receive the most recent `getLastError` (page 245) results.

See OP_INSERT[493].

Note: For bulk inserts on sharded clusters, the `getLastError` (page 245) command alone is insufficient to verify success. Applications should must verify the success of bulk inserts in application logic.

[493] http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol#op-insert

Map Reduce

Output to a Sharded Collection Using the new `sharded` flag, it is possible to send the result of a map/reduce to a sharded collection. Combined with the `reduce` or `merge` flags, it is possible to keep adding data to very large collections from map/reduce jobs.

For more information, see `http://docs.mongodb.org/manual/core/map-reduce/` and the `mapReduce` (page 220) reference.

Performance Improvements Map/reduce performance will benefit from the following:

- Larger in-memory buffer sizes, reducing the amount of disk I/O needed during a job
- Larger javascript heap size, allowing for larger objects and less GC
- Supports pure JavaScript execution with the `jsMode` flag. See the `mapReduce` (page 220) reference.

New Querying Features

Additional regex options: `s` Allows the dot (`.`) to match all characters including new lines. This is in addition to the currently supported `i`, `m` and `x`. See `$regex` (page 414).

$and A special boolean `$and` (page 405) query operator is now available.

Command Output Changes

The output of the `validate` (page 389) command and the documents in the `system.profile` collection have both been enhanced to return information as BSON objects with keys for each value rather than as free-form strings.

Shell Features

Custom Prompt You can define a custom prompt for the `mongo` (page 610) 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 *shell-use-a-custom-prompt*.

Default Shell Init Script On startup, the shell will check for a `.mongorc.js` file in the user's home directory. The shell will execute this file after connecting to the database and before displaying the prompt.

If you would like the shell not to run the `.mongorc.js` file automatically, start the shell with *`--norc`*.

For more information, see the `mongo` (page 610) reference.

Most Commands Require Authentication

In 2.0, when running with authentication (e.g. `authorization`) *all* database commands require authentication, *except* the following commands.

- `isMaster` (page 289)
- `authenticate` (page 263)
- `getnonce` (page 264)

- buildInfo (page 346)
- ping (page 365)
- isdbgrid (page 308)

Resources

- MongoDB Downloads[494]
- All JIRA Issues resolved in 2.0[495]
- All Backward Incompatible Changes[496]

7.3.4 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 808).
- The mapReduce (page 220) command has changed in 1.8, causing incompatibility with previous releases. mapReduce (page 220) no longer generates temporary collections (thus, keepTemp has been removed). Now, you must always supply a value for out. See the out field options in the mapReduce (page 220) 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[497].
2. Shutdown your mongod (page 583) instance.
3. Replace the existing binary with the 1.8.x mongod (page 583) 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.

[494] http://mongodb.org/downloads
[495] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=11002
[496] https://jira.mongodb.org/issues/?filter=11023&jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(10889%2C%2010886%2C%2010784%2C%2
[497] http://downloads.mongodb.org/

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[498].

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 174) 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" : [
```

[498] http://downloads.mongodb.org/

```
                "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[499].
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[500].

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 808).
   - If the shard is a single `mongod` (page 583) process, shut it down and then restart it with the 1.8.x binary from the MongoDB Download Page[501].
3. For each `mongos` (page 601):
   (a) Shut down the `mongos` (page 601) process.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page[502].
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[503].

[499] http://downloads.mongodb.org/
[500] http://downloads.mongodb.org/
[501] http://downloads.mongodb.org/
[502] http://downloads.mongodb.org/
[503] http://downloads.mongodb.org/

5. Turn on the balancer:

```
use config
db.settings.update({_id:"balancer"},{$set : {stopped:false}})
```

Returning to 1.6

If for any reason you must move back to 1.6, follow the steps above in reverse. Please be careful that you have not inserted any documents larger than 4MB while running on 1.8 (where the max size has increased to 16MB). If you have you will get errors when the server tries to read those documents.

Journaling Returning to 1.6 after using 1.8 `Journaling` works fine, as journaling does not change anything about the data file format. Suppose you are running 1.8.x with journaling enabled and you decide to switch back to 1.6. There are two scenarios:

- If you shut down cleanly with 1.8.x, just restart with the 1.6 mongod binary.
- If 1.8.x shut down uncleanly, start 1.8.x up again and let the journal files run to fix any damage (incomplete writes) that may have existed at the crash. Then shut down 1.8.x cleanly and restart with the 1.6 mongod binary.

Changes

Journaling

MongoDB now supports write-ahead `http://docs.mongodb.org/manual/core/journaling` to facilitate fast crash recovery and durability in the storage engine. With journaling enabled, a `mongod` (page 583) 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 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 enable MongoDB to answer queries entirely from the index when the query only selects fields that the index contains.

Incremental MapReduce Support

The `mapReduce` (page 220) command supports new options that enable incrementally updating existing *collections*. Previously, a MapReduce job could output either to a temporary collection or to a named permanent collection, which it would overwrite with new data.

You now have several options for the output of your MapReduce jobs:

- You can merge MapReduce output into an existing collection. Output from the Reduce phase will replace existing keys in the output collection if it already exists. Other keys will remain in the collection.
- You can now re-reduce your output with the contents of an existing collection. Each key output by the reduce phase will be reduced with the existing document in the output collection.

- You can replace the existing output collection with the new results of the MapReduce job (equivalent to setting a permanent output collection in previous releases)
- You can compute MapReduce inline and return results to the caller without persisting the results of the job. This is similar to the temporary collections generated in previous releases, except results are limited to 8MB.

For more information, see the `out` field options in the `mapReduce` (page 220) document.

Additional Changes and Enhancements

1.8.1

- Sharding migrate fix when moving larger chunks.
- Durability fix with background indexing.
- Fixed mongos concurrency issue with many incoming connections.

1.8.0

- All changes from 1.7.x series.

1.7.6

- Bug fixes.

1.7.5

- `Journaling`.
- Extent allocation improvements.
- Improved *replica set* connectivity for `mongos` (page 601).
- `getLastError` (page 245) improvements for *sharding*.

1.7.4

- `mongos` (page 601) routes `slaveOk` queries to *secondaries* in *replica sets*.
- New `mapReduce` (page 220) output options.
- *index-type-sparse*.

1.7.3

- Initial *covered index* support.
- Distinct can use data from indexes when possible.
- `mapReduce` (page 220) can merge or reduce results into an existing collection.
- `mongod` (page 583) tracks and `mongostat` (page 657) displays network usage. See *mongostat* (page 657).
- Sharding stability improvements.

1.7.2

- $rename (page 457) operator allows renaming of fields in a document.
- db.eval() (page 112) not to block.
- Geo queries with sharding.
- *mongostat --discover* option
- Chunk splitting enhancements.
- Replica sets network enhancements for servers behind a nat.

1.7.1

- Many sharding performance enhancements.
- Better support for $elemMatch (page 446) on primitives in embedded arrays.
- Query optimizer enhancements on range queries.
- Window service enhancements.
- Replica set setup improvements.
- $pull (page 467) 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[504], 1.8.0[505]
- 1.7.6[506], 1.7.5[507], 1.7.4[508], 1.7.3[509], 1.7.2[510], 1.7.1[511], 1.7.0[512]

[504] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/v09MbhEm62Y
[505] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/JeHQOnam6Qk
[506] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/3t6GNZ1qGYc
[507] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/S5R0Tx9wkEg
[508] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/9Om3Vuw-y9c
[509] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/DfNUrdbmflI
[510] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/df7mwK6Xixo
[511] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/HUR9zYtTpA8
[512] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/TUnJCg9161A

Resources

- MongoDB Downloads[513]
- All JIRA Issues resolved in 1.8[514]

7.3.5 Release Notes for MongoDB 1.6

Upgrading

MongoDB 1.6 is a drop-in replacement for 1.4. To upgrade, simply shutdown `mongod` (page 583) then restart with the new binaries.

Please note that you should upgrade to the latest version of whichever driver you're using. Certain drivers, including the Ruby driver, will require the upgrade, and all the drivers will provide extra features for connecting to replica sets.

Sharding

`http://docs.mongodb.org/manual/sharding` is now production-ready, making MongoDB horizontally scalable, with no single point of failure. A single instance of `mongod` (page 583) can now be upgraded to a distributed cluster with zero downtime when the need arises.

- `http://docs.mongodb.org/manual/sharding`
- `http://docs.mongodb.org/manual/tutorial/deploy-shard-cluster`
- `http://docs.mongodb.org/manual/tutorial/convert-replica-set-to-replicated-shard-cluste`

Replica Sets

`Replica sets`, 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.

- `http://docs.mongodb.org/manual/replication`
- `http://docs.mongodb.org/manual/tutorial/deploy-replica-set`
- `http://docs.mongodb.org/manual/tutorial/convert-standalone-to-replica-set`

Other Improvements

- The `w` option (and `wtimeout`) forces writes to be propagated to `n` servers before returning success (this works especially well with replica sets)
- *$or queries* (page 408)
- Improved concurrency
- *$slice* (page 450) 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

[513] http://mongodb.org/downloads
[514] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10172

- The `findAndModify` (page 239) 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[515]

1.5.x Release Notes

- 1.5.8[516]
- 1.5.7[517]
- 1.5.6[518]
- 1.5.5[519]
- 1.5.4[520]
- 1.5.3[521]
- 1.5.2[522]
- 1.5.1[523]
- 1.5.0[524]

You can see a full list of all changes on JIRA[525].

Thank you everyone for your support and suggestions!

7.3.6 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` (page 583), then restart with the new binaries. (Users upgrading from release 1.0 should review the *1.2 release notes* (page 817), in particular the instructions for upgrading the DB format.)

[515] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/06_QCC05Fpk
[516] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/uJfF1QN6Thk
[517] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/OYvz40RWs90
[518] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/4l0N2U_H0cQ
[519] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/oO749nvTARY
[520] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/380V_Ec_q1c
[521] https://groups.google.com/forum/?hl=en&fromgroups=#!topic/mongodb-user/hsUQL9CxTQw
[522] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/94EE3HVidAA
[523] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/7SBPQ2RSfdM
[524] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/VAhJcjDGTy0
[525] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10107

Release 1.4 includes the following improvements over release 1.2:

Core Server Enhancements

- `concurrency` improvements
- indexing memory improvements
- *background index creation*
- better detection of regular expressions so the index can be used in more cases

Replication and Sharding

- better handling for restarting slaves offline for a while
- fast new slaves from snapshots (`--fastsync`)
- configurable slave delay (`--slavedelay`)
- replication handles clock skew on master
- *$inc* (page 452) 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* (page 335) for backing up raw files
- option for separate directory per db (`--directoryperdb`)
- `http://localhost:28017/_status` to get serverStatus via http
- REST interface is off by default for security (`--rest` to enable)
- can rotate logs with a db command, *logRotate* (page 339)
- enhancements to *serverStatus* (page 366) command (db.serverStatus()) - counters and *replication lag* stats
- new *mongostat* (page 657) tool

Query Language Improvements

- *$all* (page 439) with regex
- *$not* (page 407)
- partial matching of array elements *$elemMatch* (page 446)
- $ operator for updating arrays
- *$addToSet* (page 464)
- *$unset* (page 461)
- *$pull* (page 467) supports object matching
- *$set* (page 459) with array indexes

Geo

- `2d geospatial search`
- geo *$center* (page 433) and *$box* (page 431) searches

7.3.7 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` (page 583) process
  - run `./mongod --upgrade`
  - start `mongod` (page 583) again
- use a slave
  - start a slave on a different port and data directory
  - when its synced, shut down the master, and start the new slave on the regular port.

Ask in the forums or IRC for more help.

Replication Changes

- There have been minor changes in replication. If you are upgrading a master/slave setup from <= 1.1.2 you have to update the slave first.

mongoimport

- `mongoimport json` has been removed and is replaced with *mongoimport* (page 642) 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* (page 409)

7.4 Other MongoDB Release Notes

7.4.1 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 [526]. This will allow your application to receive acknowledgment of all write operations.

   See the documentation of *Write Concern* 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 818). 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

[526] The drivers will call `getLastError` (page 245) 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`.