MongoDB Reference Manual

Release 3.2.3

MongoDB, Inc.

February 17, 2016

Contents

1 About MongoDB Documentation . . . 3
- 1.1 License . . . 3
- 1.2 Editions . . . 3
- 1.3 Version and Revisions . . . 4
- 1.4 Report an Issue or Make a Change Request . . . 4
- 1.5 Contribute to the Documentation . . . 5

2 Interfaces Reference . . . 19
- 2.1 `mongo` Shell Methods . . . 19
- 2.2 Database Commands . . . 303
- 2.3 Operators . . . 527
- 2.4 Aggregation Reference . . . 746

3 MongoDB and SQL Interface Comparisons . . . 759
- 3.1 SQL to MongoDB Mapping Chart . . . 759
- 3.2 SQL to Aggregation Mapping Chart . . . 765

4 Program and Tool Reference Pages . . . 769
- 4.1 MongoDB Package Components . . . 769

5 Internal Metadata . . . 885
- 5.1 Config Database . . . 885
- 5.2 The `local` Database . . . 890
- 5.3 System Collections . . . 892

6 General System Reference . . . 895
- 6.1 Configuration File Options . . . 895
- 6.2 MongoDB Server Parameters . . . 927
- 6.3 MongoDB Limits and Thresholds . . . 940
- 6.4 Explain Results . . . 946
- 6.5 Connection String URI Format . . . 953
- 6.6 MongoDB Extended JSON . . . 960
- 6.7 Log Messages . . . 964
- 6.8 Glossary . . . 967

7 Release Notes . . . 979
- 7.1 Current Stable Release . . . 979
- 7.2 Previous Stable Releases . . . 1013

This document contains all of the reference material from the `MongoDB Manual`, reflecting the 3.2.3 release. See the full manual, for complete documentation of MongoDB, it's operation, and use.

CHAPTER 1

About MongoDB Documentation

On this page

- License (page 3)
- Editions (page 3)
- Version and Revisions (page 4)
- Report an Issue or Make a Change Request (page 4)
- Contribute to the Documentation (page 5)

The MongoDB Manual[1] contains comprehensive documentation on MongoDB. This page describes the manual's licensing, editions, and versions, and describes how to make a change request and how to contribute to the manual.

1.1 License

1.2 Editions

In addition to the MongoDB Manual[3], you can also access this content in the following editions:

- PDF Format[4] (without reference).
- HTML tar.gz[5]
- ePub Format[6]

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

- MongoDB Reference Manual[7]

[1]http://docs.mongodb.org/manual/#
[2]http://creativecommons.org/licenses/by-nc-sa/3.0/us/
[3]http://docs.mongodb.org/manual/#
[4]http://docs.mongodb.org/master/MongoDB-manual.pdf
[5]http://docs.mongodb.org/master/manual.tar.gz
[6]http://docs.mongodb.org/master/MongoDB-manual.epub
[7]http://docs.mongodb.org/master/MongoDB-reference-manual.pdf

- MongoDB CRUD Operations[8]
- Data Models for MongoDB[9]
- MongoDB Data Aggregation[10]
- Replication and MongoDB[11]
- Sharding and MongoDB[12]
- MongoDB Administration[13]
- MongoDB Security[14]

MongoDB Reference documentation is also available as part of dash[15]. You can also access the MongoDB Man Pages[16] which are also distributed with the official MongoDB Packages.

1.3 Version and Revisions

This version of the manual reflects version 3.2 of MongoDB.

See the MongoDB Documentation Project Page[17] for an overview of all editions and output formats of the MongoDB Manual. You can see the full revision history and track ongoing improvements and additions for all versions of the manual from its GitHub repository[18].

This edition reflects "`master`" branch of the documentation as of the "`1ca3a112f7ebd786aeee4e44884418dad4653249`" revision. This branch is explicitly accessible via "https://docs.mongodb.org/master" and you can always reference the commit of the current manual in the release.txt[19] file.

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

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[20].

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

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.

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

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

- complete the MongoDB Contributor Agreement[21], and
- join the mongodb-translators[22] user group.

The mongodb-translators[23] user group exists to facilitate collaboration between translators and the documentation team at large. You can join the group without signing the Contributor Agreement, but you will not be allowed to contribute translations.

See also:

- *Contribute to the Documentation* (page 5)
- *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[24], which is one of the MongoDB project repositories on GitHub[25].

To contribute to the documentation, you can open a GitHub account[26], fork the mongodb/docs repository[27], make a change, and issue a pull request.

In order for the documentation team to accept your change, you must complete the MongoDB Contributor Agreement[28].

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

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

1.5.2 About the Documentation Process

The MongoDB Manual uses Sphinx[29], a sophisticated documentation engine built upon Python Docutils[30]. The original reStructured Text[31] files, as well as all necessary Sphinx extensions and build tools, are available in the same

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

repository as the documentation.

For more information on the MongoDB documentation process, see:

Style Guide and Documentation Conventions

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

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

2016-01-05: Standardizing on 'embedded document'

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 `https://docs.mongodb.org/manual/core/sharding.rst` and `https://docs.mongodb.org/manual/administration/sharding.rst`.
  - For tutorials, the full title of the document should be in the file name. For example, `https://docs.mongodb.org/manual/tutorial/replace-one-configuration-server-in-a-shar`
- 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: "Back up 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 `https://docs.mongodb.org/manual/tutorial/convert-replica-set-to-replicated-shard-`
  procedure if you have an existing replica set.

  Type this:

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

General Formulations

- Contractions are acceptable insofar as they are necessary to increase readability and flow. Avoid otherwise.
- Make lists grammatically correct.
  - Do not use a period after every item unless the list item completes the unfinished sentence before the list.
  - Use appropriate commas and conjunctions in the list items.
  - Typically begin a bulleted list with an introductory sentence or clause, with a colon or comma.
- The following terms are one word:
  - standalone
  - workflow
- Use "unavailable," "offline," or "unreachable" to refer to a `mongod` instance that cannot be accessed. Do not use the colloquialism "down."
- Always write out units (e.g. "megabytes") rather than using abbreviations (e.g. "MB".)

Structural Formulations

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

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

ReStructured Text and Typesetting

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

- Place footnotes and other references, if you use them, at the end of a section rather than the end of a file.

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

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

- As it makes sense, use the `.. code-block:: [language]` form to insert literal blocks into the text. While the double colon, `::`, is functional, the `.. code-block:: [language]` form makes the source easier to read and understand.

- For all mentions of referenced types (i.e. commands, operators, expressions, functions, statuses, etc.) use the reference types to ensure uniform formatting and cross-referencing.

Jargon and Common Terms

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

Database Systems and Processes

- To indicate the entire database system, use “MongoDB,” not mongo or Mongo.
- To indicate the database process or a server instance, use `mongod` or `mongos`. Refer to these as “processes” or “instances.” Reserve “database” for referring to a database structure, i.e., the structure that holds collections and refers to a group of files on disk.

Distributed System Terms

- Refer to partitioned systems as “sharded clusters.” Do not use shard clusters or sharded systems.
- Refer to configurations that run with replication as “replica sets” (or “master/slave deployments”) rather than “clusters” or other variants.

Data Structure Terms

- “document” refers to “rows” or “records” in a MongoDB database. Potential confusion with “JSON Documents.”

  Do not refer to documents as “objects,” because drivers (and MongoDB) do not preserve the order of fields when fetching data. If the order of objects matter, use an array.
- “field” refers to a “key” or “identifier” of data within a MongoDB document.
- “value” refers to the contents of a “field”.
- “embedded document” describes a nested document.

Other Terms

- Use `example.net` (and `.org` or `.com` if needed) for all examples and samples.
- Hyphenate “map-reduce” in order to avoid ambiguous reference to the command name. Do not camel-case.

Notes on Specific Features

- Geo-Location
  1. While MongoDB *is capable* of storing coordinates in embedded documents, in practice, users should only store coordinates in arrays. (See: DOCS-41[32].)

MongoDB Documentation Practices and Processes

This document provides an overview of the practices and processes.

Commits

When relevant, include a Jira case identifier in a commit message. Reference documentation cases when applicable, but feel free to reference other cases from jira.mongodb.org[33].

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

[32] https://jira.mongodb.org/browse/DOCS-41
[33] http://jira.mongodb.org/

For the sake of consistency, remove trailing whitespaces in the source file.

"Hard wrap" files to between 72 and 80 characters per-line.

Standards and Practices

- At least two people should vet all non-trivial changes to the documentation before publication. One of the reviewers should have significant technical experience with the material covered in the documentation.
- All development and editorial work should transpire on GitHub branches or forks that editors can then merge into the publication branches.

Collaboration

To propose a change to the documentation, do either of the following:

- Open a ticket in the documentation project[34] proposing the change. Someone on the documentation team will make the change and be in contact with you so that you can review the change.
- Using GitHub[35], fork the mongodb/docs repository[36], commit your changes, and issue a pull request. Someone on the documentation team will review and incorporate your change into the documentation.

Builds

Building the documentation is useful because Sphinx[37] and docutils can catch numerous errors in the format and syntax of the documentation. Additionally, having access to an example documentation as it *will* appear to the users is useful for providing more effective basis for the review process. Besides Sphinx, Pygments, and Python-Docutils, the documentation repository contains all requirements for building the documentation resource.

Talk to someone on the documentation team if you are having problems running builds yourself.

Publication

The makefile for this repository contains targets that automate the publication process. Use `make html` to publish a test build of the documentation in the `build/` directory of your repository. Use `make publish` to build the full contents of the manual from the current branch in the `../public-docs/` directory relative the docs repository.

Other targets include:

- `man` - builds UNIX Manual pages for all Mongodb utilities.
- `push` - builds and deploys the contents of the `../public-docs/`.
- `pdfs` - builds a PDF version of the manual (requires LaTeX dependencies.)

Branches

This section provides an overview of the git branches in the MongoDB documentation repository and their use.

[34]https://jira.mongodb.org/browse/DOCS
[35]https://github.com/
[36]https://github.com/mongodb/docs
[37]http://sphinx.pocoo.org/

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

Migration from Legacy Documentation

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

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

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

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

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

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

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

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

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

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

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

Review Process

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

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

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

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

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

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

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

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

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

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

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

- Leave feedback on Jira cases in the DOCS[38] project. These are better for more general changes that aren't necessarily tied to a specific line, or affect multiple files.

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

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

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

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

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

MongoDB Manual Organization

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

If you have questions, don't hesitate to open a ticket in the Documentation Jira Project[39] or contact the documentation team[40].

[38] http://jira.mongodb.org/browse/DOCS
[39] https://jira.mongodb.org/browse/DOCS
[40] docs@mongodb.com

Global Organization

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

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

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

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

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

Tools

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

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

MongoDB Documentation Build System

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

Getting Started

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

- Python
- Git
- Inkscape (Image generation.)

- LaTeX/PDF LaTeX (typically texlive; for building PDFs)
- Giza[41]

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

```
easy_install giza
```

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

To generate the images used in the documentation, download and install Inkscape[42].

Optional

To generate PDFs for the full production build, install a TeX distribution (for building the PDF.) If you do not have a LaTeX installation, use MacTeX[43]. This is **only** required to build PDFs.

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

```
pacman -S inkscape python2-pip
```

Then install the following Python packages:

```
pip2 install giza
```

Optional

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

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

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

```
apt-get install inkscape python-pip
```

Then install the following Python packages:

```
pip install giza
```

Optional

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

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

Setup and Configuration Clone the repository:

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

[41] https://pypi.python.org/pypi/giza
[42] http://inkscape.org/download/
[43] http://www.tug.org/mactex/2011/

Building the Documentation

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

```
make html
```

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

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

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

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

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

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

`html`; `latex`; `dirhtml`; `epub`; `texinfo`; `man`; `json` These are standard targets derived from the default Sphinx Makefile, with adjusted dependencies. Additionally, for all of these targets you can append `-nitpick` to increase Sphinx's verbosity, or `-clean` to remove all Sphinx build artifacts.

`latex` performs several additional post-processing steps on `.tex` output generated by Sphinx. This target will also compile PDFs using `pdflatex`.

`html` and `man` also generates a `.tar.gz` file of the build outputs for inclusion in the final releases.

If you have any questions, please feel free to open a Jira Case[44].

[44] https://jira.mongodb.org/browse/DOCS

CHAPTER 2

Interfaces Reference

2.1 `mongo` Shell Methods

On this page

- Collection (page 19)
- Cursor (page 133)
- Database (page 161)
- Query Plan Cache (page 204)
- Bulk Write Operation (page 209)
- User Management (page 228)
- Role Management (page 241)
- Replication (page 255)
- Sharding (page 266)
- Subprocess (page 285)
- Constructors (page 286)
- Connection (page 292)
- Native (page 297)

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 20)	Provides access to the `aggregation pipeline`.
`db.collection.bulkWrite()` (page 25)	Provides bulk write operation functionality.
`db.collection.count()` (page 33)	Wraps `count` (page 307) to return a count of the number of docur
`db.collection.copyTo()` (page 35)	Deprecated. Wraps `eval` (page 358) to copy data between collecti
`db.collection.createIndex()` (page 36)	Builds an index on a collection.
`db.collection.dataSize()` (page 39)	Returns the size of the collection. Wraps the `size` (page 475) fiel

Table 2.1 – continued from pre

Name	Description
db.collection.deleteOne() (page 40)	Deletes a single document in a collection.
db.collection.deleteMany() (page 42)	Deletes multiple documents in a collection.
db.collection.distinct() (page 44)	Returns an array of documents that have distinct values for the spe
db.collection.drop() (page 45)	Removes the specified collection from the database.
db.collection.dropIndex() (page 46)	Removes a specified index on a collection.
db.collection.dropIndexes() (page 47)	Removes all indexes on a collection.
db.collection.ensureIndex() (page 47)	Deprecated. Use db.collection.createIndex() (page 36
db.collection.explain() (page 48)	Returns information on the query execution of various methods.
db.collection.find() (page 51)	Performs a query on a collection and returns a cursor object.
db.collection.findAndModify() (page 57)	Atomically modifies and returns a single document.
db.collection.findOne() (page 62)	Performs a query and returns a single document.
db.collection.findOneAndDelete() (page 64)	Finds a single document and deletes it.
db.collection.findOneAndReplace() (page 66)	Finds a single document and replaces it.
db.collection.findOneAndUpdate() (page 69)	Finds a single document and updates it.
db.collection.getIndexes() (page 73)	Returns an array of documents that describe the existing indexes o
db.collection.getShardDistribution() (page 73)	For collections in sharded clusters, db.collection.getShar
db.collection.getShardVersion() (page 75)	Internal diagnostic method for shard cluster.
db.collection.group() (page 75)	Provides simple data aggregation function. Groups documents in a
db.collection.insert() (page 79)	Creates a new document in a collection.
db.collection.insertOne() (page 82)	Inserts a new document in a collection.
db.collection.insertMany() (page 85)	Inserts several new document in a collection.
db.collection.isCapped() (page 90)	Reports if a collection is a *capped collection*.
db.collection.mapReduce() (page 90)	Performs map-reduce style data aggregation.
db.collection.reIndex() (page 97)	Rebuilds all existing indexes on a collection.
db.collection.replaceOne() (page 98)	Replaces a single document in a collection.
db.collection.remove() (page 101)	Deletes documents from a collection.
db.collection.renameCollection() (page 104)	Changes the name of a collection.
db.collection.save() (page 105)	Provides a wrapper around an insert() (page 79) and update
db.collection.stats() (page 107)	Reports on the state of a collection. Provides a wrapper around the
db.collection.storageSize() (page 116)	Reports the total size used by the collection in bytes. Provides a w
db.collection.totalSize() (page 116)	Reports the total size of a collection, including the size of all docu
db.collection.totalIndexSize() (page 117)	Reports the total size used by the indexes on a collection. Provides
db.collection.update() (page 117)	Modifies a document in a collection.
db.collection.updateOne() (page 125)	Modifies a single document in a collection.
db.collection.updateMany() (page 128)	Modifies multiple documents in a collection.
db.collection.validate() (page 132)	Performs diagnostic operations on a collection.

db.collection.aggregate()

On this page

- Definition (page 20)
- Behavior (page 22)
- Examples (page 22)

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 630) 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 Optional. Additional options that `aggregate()` (page 20) passes to the `aggregate` (page 303) 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 Optional. Specifies to return the information on the processing of the pipeline. See *Return Information on Aggregation Pipeline Operation* (page 23) for an example.

New in version 2.6.

field boolean allowDiskUse Optional. Enables writing to temporary files. When set to `true`, aggregation operations can write data to the `_tmp` subdirectory in the `dbPath` (page 915) directory. See *Perform Large Sort Operation with External Sort* (page 23) for an example.

New in version 2.6.

field document cursor Optional. 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.

field boolean bypassDocumentValidation Optional. Available only if you specify the `$out` (page 656) aggregation operator.

Enables `db.collection.aggregate` (page 20) to bypass document validation during the operation. This lets you insert documents that do not meet the validation requirements.

New in version 3.2.

field document readConcern Optional. Specifies the *read concern*. The default level is `"local"`.

To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

To use a `read concern` level of `"majority"`, you cannot include the `$out` (page 656) stage.

New in version 3.2.

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 `$out` (page 656) operator, `aggregate()` (page 20) returns an empty cursor. See `$out` (page 656) for more information.

Changed in version 2.6: The `db.collection.aggregate()` (page 20) 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.

Behavior

Error Handling Changed in version 2.4: If an error occurs, the `aggregate()` (page 20) 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 303) command.

Cursor Behavior In the `mongo` (page 803) shell, if the cursor returned from the `db.collection.aggregate()` (page 20) is not assigned to a variable using the `var` keyword, then the `mongo` (page 803) shell automatically iterates the cursor up to 20 times. See `https://docs.mongodb.org/manual/core/cursors` for cursor behavior in the `mongo` (page 803) shell and `https://docs.mongodb.org/manual/tutorial/iterate-a-cursor` for handling cursors in the `mongo` (page 803) 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 142)
- `cursor.next()` (page 150)
- `cursor.toArray()` (page 161)
- `cursor.forEach()` (page 141)
- `cursor.map()` (page 144)
- `cursor.objsLeftInBatch()` (page 151)
- `cursor.itcount()` (page 143)
- `cursor.pretty()` (page 151)

See also:

For more information, see `https://docs.mongodb.org/manual/core/aggregation-pipeline`, *Aggregation Reference* (page 746), `https://docs.mongodb.org/manual/core/aggregation-pipeline-limits`, and `aggregate` (page 303).

Examples 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 803) shell iterates the returned cursor automatically to print the results. See `https://docs.mongodb.org/manual/tutorial/iterate-a-cursor` for handling cursors manually in the mongo (page 803) 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 803) shell iterates the returned cursor automatically to print the results. See `https://docs.mongodb.org/manual/tutorial/iterate-a-cursor` for handling cursors manually in the mongo (page 803) 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
                                   }
                                 )
```

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

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

```
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* operations as with other MongoDB cursors.

The `mongo` (page 803) shell iterates the returned cursor automatically to print the results. See `https://docs.mongodb.org/manual/tutorial/iterate-a-cursor` for handling cursors manually in the `mongo` (page 803) shell.

Override `readConcern` The following operation on a replica set specifies a `https://docs.mongodb.org/manual/reference/read-concern` of `"majority"` to read the most recent copy of the data confirmed as having been written to a majority of the nodes.

Note:

- To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

  Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

- To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

- To use a `read concern` level of `"majority"`, you cannot include the `$out` (page 656) stage.

- Regardless of the *read concern* level, the most recent data on a node may not reflect the most recent version of the data in the system.

```
db.restaurants.aggregate(
   [ { $match: { rating: { $lt: 5 } } } ],
   { readConcern: { level: "majority" } }
)
```

db.collection.bulkWrite()

On this page

- Definition (page 25)
- Behavior (page 25)
- Examples (page 28)

Definition

`db.collection.bulkWrite()`

New in version 3.2.

Performs multiple write operations with controls for order of execution.

`bulkWrite()` (page 25) has the following syntax:

```
db.collection.bulkWrite(
   [ <operation 1>, <operation 2>, ... ],
   {
      writeConcern : <document>,
      ordered : <boolean>
   }
)
```

param array operations An array of `bulkWrite()` (page 25) write operations.

Valid operations are:

- *insertOne* (page 26)
- *updateOne* (page 26)
- *updateMany* (page 26)
- *deleteOne* (page 27)
- *deleteMany* (page 27)
- *replaceOne* (page 26)

See *Write Operations* (page 26) for usage of each operation.

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

param boolean ordered Optional. A boolean specifying whether the `mongod` (page 770) instance should perform an ordered or unordered operation execution. Defaults to `true`.

See *Execution of Operations* (page 27)

Returns

- A boolean `acknowledged` as `true` if the operation ran with *write concern* or `false` if write concern was disabled.
- A count for each write operation.
- An array containing an `_id` for each successfully inserted or upserted documents.

Behavior `bulkWrite()` (page 25) takes an array of write operations and executes each of them. By default operations are executed in order. See *Execution of Operations* (page 27) for controlling the order of write operation execution.

Write Operations

insertOne Inserts a single document into the collection.

See `db.collection.insertOne()` (page 82).

```
db.collection.bulkWrite( [
   { insertOne : { "document" : <document> } }
] )
```

updateOne and updateMany `updateOne` updates a *single* document in the collection that matches the filter. If multiple documents match, `updateOne` will update the *first* matching document only. See `db.collection.updateOne()` (page 125).

```
db.collection.bulkWrite( [
   { updateOne :
      {
         "filter" : <document>,
         "update" : <document>,
         "upsert" : <boolean>
      }
   }
] )
```

`updateMany` updates *all* documents in the collection that match the filter. See `db.collection.updateMany()` (page 128).

```
db.collection.bulkWrite( [
   { updateMany :
      {
         "filter" : <document>,
         "update" : <document>,
         "upsert" : <boolean>
      }
   }
] )
```

Use *query selectors* (page 527) such as those used with `find()` (page 51) for the `filter` field.

Use *Update Operators* (page 594) such as `$set` (page 600), `$unset` (page 602), or `$rename` (page 598) for the `update` field.

By default, `upsert` is `false`.

replaceOne `replaceOne` replaces a *single* document in the collection that matches the filter. If multiple documents match, `replaceOne` will replace the *first* matching document only. See `db.collection.replaceOne()` (page 98).

```
db.collection.bulkWrite([
   { replaceOne :
      {
         "filter" : <document>,
         "replacement" : <document>,
         "upsert" : <boolean>
      }
   }
] )
```

Use *query selectors* (page 527) such as those used with `find()` (page 51) for the `filter` field.

The `replacement` field cannot contain *update operators* (page 594).

By default, `upsert` is `false`.

deleteOne and deleteMany `deleteOne` deletes a *single* document in the collection that match the filter. If multiple documents match, `deleteOne` will delete the *first* matching document only. See `db.collection.deleteOne()` (page 40).

```
db.collection.bulkWrite([
   { deleteOne :  { "filter" : <document> } }
] )
```

`deleteMany` deletes *all* documents in the collection that match the filter. See `db.collection.deleteMany()` (page 42).

```
db.collection.bulkWrite([
   { deleteMany :  { "filter" : <document> } }
] )
```

Use *query selectors* (page 527) such as those used with `find()` (page 51) for the `filter` field.

`_id` Field If the document does not specify an *_id* field, then `mongod` (page 770) adds the `_id` field and assign a unique `https://docs.mongodb.org/manual/reference/object-id` for the document before inserting or upserting it. Most drivers create an ObjectId and insert the `_id` field, but the `mongod` (page 770) 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.

Update or replace operations cannot specify an `_id` value that differs from the original document.

Execution of Operations The `ordered` parameter specifies whether `bulkWrite()` (page 25) will execute operations in order or not. By default, operations are executed in order.

The following code represents a `bulkWrite()` (page 25) with five operations.

```
db.collection.bulkWrite(
   [
      { insertOne : <document> },
      { updateOne : <document> },
      { updateMany : <document> },
      { replaceOne : <document> },
      { deleteOne : <document> },
      { deleteMany : <document> }
   ]
)
```

In the default `ordered : true` state, each operation will be executed in order, from the first operation `insertOne` to the last operation `deleteMany`.

If `ordered` is set to false, operations may be reordered by `mongod` (page 770) to increase performance. Applications should not depend on order of operation execution.

The following code represents an unordered `bulkWrite()` (page 25) with six operations:

```
db.collection.bulkWrite(
   [
      { insertOne : <document> },
      { updateOne : <document> },
      { updateMany : <document> },
      { replaceOne : <document> },
      { deleteOne : <document> },
      { deleteMany : <document> }
   ],
   { ordered : false }
)
```

With `ordered : false`, the results of the operation may vary. For example, the `deleteOne` or `deleteMany` may remove more or fewer documents depending on whether the run before or after the `insertOne`, `updateOne`, `updateMany`, or `replaceOne` operations.

Each group of operations can have at most `1000 operations` (page 945). If a group exceeds this `limit` (page 945), MongoDB will divide the group into smaller groups of 1000 or less. For example, if the queue consists of 2000 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.

Executing an `ordered` (page 212) list of operations on a sharded collection will generally be slower than executing an `unordered` (page 213) list since with an ordered list, each operation must wait for the previous operation to finish.

Capped Collections `bulkWrite()` (page 25) write operations have restrictions when used on a *capped collection*.

`updateOne` and `updateMany` throw a `WriteError` if the `update` criteria increases the size of the document being modified.

`replaceOne` throws a `WriteError` if the `replacement` document has a larger size than the original document.

`deleteOne` and `deleteMany` throw a `WriteError` if used on a capped collection.

Error Handling `bulkWrite()` (page 25) throws a `BulkWriteError` exception on errors.

Excluding `https://docs.mongodb.org/manual/reference/write-concern` errors, ordered operations stop after an error, while unordered operations continue to process any remaining write operations in the queue.

Write concern errors are displayed in the `writeConcernErrors` field, while all other errors are displayed in the `writeErrors` field. If an error is encountered, the number of successful write operations are displayed instead of the inserted `_id` values. Ordered operations display the single error encountered while unordered operations display each error in an array.

Examples

Bulk Write Operations The `characters` collection contains the following documents:

```
{ "_id" : 1, "char" : "Brisbane", "class" : "monk", "lvl" : 4 },
{ "_id" : 2, "char" : "Eldon", "class" : "alchemist", "lvl" : 3 },
{ "_id" : 3, "char" : "Meldane", "class" : "ranger", "lvl" : 3 }
```

The following `bulkWrite()` (page 25) performs multiple operations on the collection:

```
try {
   db.characters.bulkWrite(
      [
         { insertOne :
            {
               "document" :
```

```
               {
                  "_id" : 4, "char" : "Dithras", "class" : "barbarian", "lvl" : 4
               }
            }
         },
         { insertOne :
            {
               "document" :
               {
                  "_id" : 5, "char" : "Taeln", "class" : "fighter", "lvl" : 3
               }
            }
         },
         { updateOne :
            {
               "filter" : { "char" : "Eldon" },
               "update" : { $set : { "status" : "Critical Injury" } }
            }
         },
         { deleteOne :
            { "filter" : { "char" : "Brisbane"} }
         },
         { replaceOne :
            {
               "filter" : { "char" : "Meldane" },
               "replacement" : { "char" : "Tanys", "class" : "oracle", "lvl" : 4 }
            }
         }
      ]
   );
}
catch (e) {
   print(e);
}
```

The operation returns the following:

```
{
   "acknowledged" : true,
   "deletedCount" : 1,
   "insertedCount" : 2,
   "matchedCount" : 2,
   "upsertedCount" : 0,
   "insertedIds" : {
      "0" : 4,
      "1" : 5
   },
   "upsertedIds" : {

   }
}
```

If the _id value for the second of the insertOne operations were a duplicate of an existing _id, the following exception would be thrown:

```
BulkWriteError({
   "writeErrors" : [
      {
         "index" : 0,
         "code" : 11000,
         "errmsg" : "E11000 duplicate key error collection: guidebook.characters index: _id_ dup key
         "op" : {
            "_id" : 5,
            "char" : "Taeln"
         }
      }
   ],
   "writeConcernErrors" : [ ],
   "nInserted" : 1,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

Since `ordered` was true by default, only the first operation completes successfully. The rest are not executed. Running the `bulkWrite()` (page 25) with `ordered : false` would allow the remaining operations to complete despite the error.

Unordered Bulk Write The `characters` collection contains the following documents:

```
{ "_id" : 1, "char" : "Brisbane", "class" : "monk", "lvl" : 4 },
{ "_id" : 2, "char" : "Eldon", "class" : "alchemist", "lvl" : 3 },
{ "_id" : 3, "char" : "Meldane", "class" : "ranger", "lvl" : 3 }
```

The following `bulkWrite()` (page 25) performs multiple `unordered` operations on the `characters` collection. Note that one of the `insertOne` stages has a duplicate `_id` value:

```
try {
   db.characters.bulkWrite(
         [
            { insertOne :
               {
                  "document" :
                  {
                     "_id" : 4, "char" : "Dithras", "class" : "barbarian", "lvl" : 4
                  }
               }
            },
            { insertOne :
               {
                  "document" :
                     {
                        "_id" : 4, "char" : "Taeln", "class" : "fighter", "lvl" : 3
                     }
               }
            },
            { updateOne :
               {
                  "filter" : { "char" : "Eldon" },
                  "update" : { $set : { "status" : "Critical Injury" } }
               }
            },
            { deleteOne :
               { "filter" : { "char" : "Brisbane"} }
            },
            { replaceOne :
               {
                  "filter" : { "char" : "Meldane" },
                  "replacement" : { "char" : "Tanys", "class" : "oracle", "lvl" : 4 }
               }
            }
         ],
            { ordered : false }
      );
   }
   catch (e) {
   print(e);
}
```

The operation returns the following:

```
BulkWriteError({
   "writeErrors" : [
      {
         "index" : 0,
         "code" : 11000,
         "errmsg" : "E11000 duplicate key error collection: guidebook.characters index: _id_ dup key
         "op" : {
            "_id" : 4,
            "char" : "Taeln"
         }
      }
   ],
   "writeConcernErrors" : [ ],
   "nInserted" : 1,
   "nUpserted" : 0,
   "nMatched" : 2,
   "nModified" : 2,
   "nRemoved" : 1,
   "upserted" : [ ]
})
```

Since this was an `unordered` operation, the writes remaining in the queue were processed despite the exception.

Bulk Write with Write Concern The `enemies` collection contains the following documents:

```
{ "_id" : 1, "char" : "goblin", "rating" : 1, "encounter" : 0.24 },
{ "_id" : 2, "char" : "hobgoblin", "rating" : 1.5, "encounter" : 0.30 },
{ "_id" : 3, "char" : "ogre", "rating" : 3, "encounter" : 0.2 },
{ "_id" : 4, "char" : "ogre berserker" , "rating" : 3.5, "encounter" : 0.12}
```

The following `bulkWrite()` (page 25) performs multiple operations on the collection using a *write concern* value of `"majority"` and *timeout* value of 100 milliseconds:

```
try {
   db.enemies.bulkWrite(
      [
         { updateMany :
            {
               "filter" : { "rating" : { $gte : 3} },
               "update" : { $inc : { "encounter" : 0.1 } }
            },
         },
         { updateMany :
            {
               "filter" : { "rating" : { $lt : 2} },
               "update" : { $inc : { "encounter" : -0.25 } }
            },
         },
         { deleteMany : { "filter" : { "encounter" { $lt : 0 } } } },
         { insertOne :
            {
               "document" :
                  {
                     "_id" :5, "char" : "ogrekin" , "rating" : 2, "encounter" : 0.31
                  }
            }
         }
      ],
      { writeConcern : { w : "majority", wtimeout : 100 } }
   );
}
catch (e) {
   print(e);
}
```

If the total time required for all required nodes in the replica set to acknowledge the write operation is greater than `wtimeout`, the following `writeConcernError` is displayed when the `wtimeout` period has passed.

```
BulkWriteError({
   "writeErrors" : [ ],
   "writeConcernErrors" : [
      {
         "code" : 64,
         "errInfo" : {
            "wtimeout" : true
         },
         "errmsg" : "waiting for replication timed out"
      }
   ],
   "nInserted" : 1,
   "nUpserted" : 0,
   "nMatched" : 4,
   "nModified" : 4,
   "nRemoved" : 1,
   "upserted" : [ ]
   })
```

The result set shows the operations executed since `writeConcernErrors` errors are *not* an indicator that any write operations failed.

db.collection.count()

On this page

- Definition (page 33)
- Behavior (page 33)
- Examples (page 34)

Definition

`db.collection.count(query, options)`

Returns the count of documents that would match a `find()` (page 51) query. The `db.collection.count()` (page 33) method does not perform the `find()` (page 51) operation but instead counts and returns the number of results that match a query.

param document query The query selection criteria.

param document options Optional. Extra options for modifying the count.

The `options` document contains the following fields:

field integer limit Optional. The maximum number of documents to count.

field integer skip Optional. The number of documents to skip before counting.

field string, document hint Optional. An index name hint or specification for the query.

New in version 2.6.

field integer maxTimeMS Optional. The maximum amount of time to allow the query to run.

field string readConcern

Optional. Specifies the *read concern*. The default level is `"local"`.

To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the `--enableMajorityReadConcern` (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

To use a *read concern* level of `"majority"`, you must specify a nonempty `query` condition.

New in version 3.2.

`count()` (page 33) is equivalent to the `db.collection.find(query).count()` construct.

See also:

`cursor.count()` (page 137)

Behavior

Sharded Clusters On a sharded cluster, `db.collection.count()` (page 33) 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 `$group` (page 644) stage of the `db.collection.aggregate()` (page 20) method to `$sum` (page 729) 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 635) stage as well:

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

See *Perform a Count* (page 636) 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.

Unexpected Shutdown and Count For MongoDB instances using the `WiredTiger` storage engine, after an unclean shutdown, statistics on size and count may off by up to 1000 documents as reported by `collStats` (page 473), `dbStats` (page 481), `count` (page 307). To restore the correct statistics for the collection, run `validate` (page 485) on the collection.

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.copyTo()

On this page

- Definition (page 35)
- Behavior (page 35)
- Example (page 35)

Definition

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

Deprecated since version 3.0.

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 35). Providing such access is not recommended, but if your organization requires a user to run `db.collection.copyTo()` (page 35), 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 35) check field types to ensure that the operation does not remove type information from documents during the translation from *BSON* to *JSON*. The `db.collection.copyTo()` (page 35) method uses the `eval` (page 358) command internally. As a result, the `db.collection.copyTo()` (page 35) operation takes a global lock that blocks all other read and write operations until the `db.collection.copyTo()` (page 35) completes.

`copyTo()` (page 35) returns the number of documents copied. If the copy fails, it throws an exception.

Behavior Because `copyTo()` (page 35) uses `eval` (page 358) internally, the copy operations will block all other operations on the `mongod` (page 770) instance.

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

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

db.collection.createIndex()

On this page

- Definition (page 36)
- Options (page 36)
- Behaviors (page 38)
- Examples (page 39)
- Additional Information (page 39)

Definition

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

Creates indexes on collections.

Changed in version 3.2: Starting in MongoDB 3.2, MongoDB disallows the creation of *version 0* (page 1001) indexes. To upgrade existing version 0 indexes, see *Version 0 Indexes* (page 1001).

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 `https://docs.mongodb.org/manual/core/index-types` for more information.

param document options Optional. A document that contains a set of options that controls the creation of the index. See *Options* (page 36) 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:

Changed in version 3.0: The `dropDups` option is no longer available.

param boolean background Optional. 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 Optional. 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 Optional. 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 941).

param document partialFilterExpression Optional. If specified, the index only references documents that match the filter expression. See `https://docs.mongodb.org/manual/core/index-partial` for more information.

A filter expression can include:

- equality expressions (i.e. `field: value` or using the `$eq` (page 527) operator),
- `$exists: true` (page 538) expression,
- `$gt` (page 529), `$gte` (page 530), `$lt` (page 530), `$lte` (page 531) expressions,
- `$type` (page 540) expressions,
- `$and` (page 535) operator at the top-level only

You can specify a `partialFilterExpression` option for all MongoDB `index types`.

New in version 3.2.

param boolean sparse Optional. 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 `https://docs.mongodb.org/manual/core/index-sparse` for more information.

Changed in version 3.2: Starting in MongoDB 3.2, MongoDB provides the option to create *partial indexes*. Partial indexes offer a superset of the functionality of sparse indexes. If you are using MongoDB 3.2 or later, *partial indexes* should be preferred over sparse indexes.

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

param document storageEngine Optional. 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.

New in version 3.0.

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

param document weights Optional. 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 `https://docs.mongodb.org/manual/tutorial/control-results-of-text-search` to adjust the scores. The default value is `1`.

param string default_language Optional. 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 `https://docs.mongodb.org/manual/tutorial/specify-language-for-text-index` for more information and examples. The default value is `english`.

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

param number max Optional. 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 `createIndex()` (page 36) method has the behaviors described here.

- To add or change index options you must drop the index using the `dropIndex()` (page 46) method and issue another `createIndex()` (page 36) operation with the new options.

  If you create an index with one set of options, and then issue the `createIndex()` (page 36) method with the same index fields and different options without first dropping the index, `createIndex()` (page 36) will *not* rebuild the existing index with the new options.

- If you call multiple `createIndex()` (page 36) 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 36) 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.createIndex( { orderDate: 1 } )
```

If the `keys` document specifies more than one field, then `createIndex()` (page 36) 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.createIndex( { 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 156) operations using the index.

Additional Information

- Use `db.collection.createIndex()` (page 36) rather than `db.collection.ensureIndex()` (page 47) to create indexes.
- The `https://docs.mongodb.org/manual/indexes` section of this manual for full documentation of indexes and indexing in MongoDB.
- `db.collection.getIndexes()` (page 73) to view the specifications of existing indexes for a collection.
- `https://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.dataSize()

`db.collection.`**`dataSize`**`()`

Returns The size of the collection. This method provides a wrapper around the `size` (page 475) output of the `collStats` (page 473) (i.e. `db.collection.stats()` (page 107)) command.

db.collection.deleteOne()

On this page

- Definition (page 40)
- Behavior (page 40)
- Examples (page 40)

Definition

`db.collection.deleteOne()`

Removes a single document from a collection.

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

param document filter Specifies deletion criteria using *query operators* (page 527).

Specify an empty document `{ }` to delete the first document returned in the collection.

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

Returns

A document containing:

- A boolean `acknowledged` as `true` if the operation ran with *write concern* or `false` if write concern was disabled
- `deletedCount` containing the number of deleted documents

Behavior

Deletion Order `deleteOne` (page 40) deletes the first document that matches the filter. Use a field that is part of a *unique index* such as `_id` for precise deletions.

Capped Collections `deleteOne()` (page 40) throws a `WriteError` exception if used on a *capped collection*. To remove documents from a capped collection, use `db.collection.drop()` (page 45) instead.

Examples

Delete a Single Document The `orders` collection has documents with the following structure:

```
{
   _id: ObjectId("563237a41a4d68582c2509da"),
   stock: "Brent Crude Futures",
   qty: 250,
   type: "buy-limit",
   limit: 48.90
   creationts: ISODate("2015-11-01T12:30:15Z"),
   expiryts: ISODate("2015-11-01T12:35:15Z"),
   client: "Crude Traders Inc."
}
```

The following operation deletes the order with `_id: ObjectId("563237a41a4d68582c2509da")`:

```
try {
   db.orders.deleteOne( { "_id" : ObjectId("563237a41a4d68582c2509da") } );
}
catch (e) {
   print(e);
}
```

The operation returns:

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

The following operation deletes the first document with `expiryts` greater than `ISODate("2015-11-01T12:40:15Z")`

```
try {
   db.orders.deleteOne( { "expiryts" : { $lt: ISODate("2015-11-01T12:40:15Z") } } );
}
catch (e) {
   print(e);
}
```

The operation returns:

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

deleteOne() with Write Concern Given a three member replica set, the following operation specifies a `w` of `majority`, `wtimeout` of `100`:

```
try {
   db.orders.deleteOne(
       { "_id" : ObjectId("563237a41a4d68582c2509da") },
       { w : "majority", wtimeout : 100 }
   );
}
catch (e) {
   print (e);
}
```

If the acknowledgement takes longer than the `wtimeout` limit, the following exception is thrown:

```
WriteConcernError({
   "code" : 64,
   "errInfo" : {
      "wtimeout" : true
   },
   "errmsg" : "waiting for replication timed out"
})
```

See also:

To delete multiple documents, see `db.collection.deleteMany()` (page 42)

db.collection.deleteMany()

On this page

- Definition (page 42)
- Behavior (page 42)
- Examples (page 42)

Definition

`db.collection.deleteMany()`

Removes all documents that match the `filter` from a collection.

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

param document filter Specifies deletion criteria using *query operators* (page 527).

To delete all documents in a collection, pass in an empty document (`{ }`).

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

Returns

A document containing:

- A boolean `acknowledged` as `true` if the operation ran with *write concern* or `false` if write concern was disabled
- `deletedCount` containing the number of deleted documents

Behavior

Capped Collections `deleteMany()` (page 42) throws a `WriteError` exception if used on a *capped collection*. To remove all documents from a capped collection, use `db.collection.drop()` (page 45) instead.

Delete a Single Document To delete a single document, use `db.collection.deleteOne()` (page 40) instead.

Alternatively, use a field that is a part of a *unique index* such as `_id`.

Examples

Delete Multiple Documents The `orders` collection has documents with the following structure:

```
{
   _id: ObjectId("563237a41a4d68582c2509da"),
   stock: "Brent Crude Futures",
   qty: 250,
   type: "buy-limit",
   limit: 48.90
   creationts: ISODate("2015-11-01T12:30:15Z"),
   expiryts: ISODate("2015-11-01T12:35:15Z"),
   client: "Crude Traders Inc."
}
```

The following operation deletes all documents where `client : "Crude Traders Inc."`:

```
try {
   db.orders.deleteMany( { "client" : "Crude Traders Inc." } );
}
catch (e) {
   print (e);
}
```

The operation returns:

```
{ "acknowledged" : true, "deletedCount" : 10 }
```

The following operation deletes all documents where `stock : "Brent Crude Futures"` and `limit` is greater than `48.88`:

```
try {
   db.orders.deleteMany( { "stock" : "Brent Crude Futures", "limit" : { $gt : 48.88 } } );
}
catch (e) {
   print (e);
}
```

The operation returns:

```
{ "acknowledged" : true, "deletedCount" : 8 }
```

deleteMany() with Write Concern Given a three member replica set, the following operation specifies a `w` of `majority` and `wtimeout` of `100`:

```
try {
   db.orders.deleteMany(
       { "client" : "Crude Traders Inc." },
       { w : "majority", wtimeout : 100 }
   );
}
catch (e) {
   print (e);
}
```

If the acknowledgement takes longer than the `wtimeout` limit, the following exception is thrown:

```
WriteConcernError({
   "code" : 64,
   "errInfo" : {
      "wtimeout" : true
   },
   "errmsg" : "waiting for replication timed out"
})
```

db.collection.distinct()

On this page

- Definition (page 44)
- Behavior (page 44)
- Examples (page 44)

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 44) method provides a wrapper around the `distinct` (page 310) command. Results must not be larger than the maximum *BSON size* (page 940).

Behavior

Array Fields If the value of the specified `field` is an array, `db.collection.distinct()` (page 44) considers each element of the array as a separate value.

For instance, if a field has as its value `[ 1, [1], 1 ]`, then `db.collection.distinct()` (page 44) considers `1`, `[1]`, and `1` as separate values.

For an example, see *Return Distinct Values for an Array Field* (page 45).

Index Use When possible, `db.collection.distinct()` (page 44) operations can use indexes.

Indexes can also *cover* `db.collection.distinct()` (page 44) operations. See *covered-queries* for more information on queries covered by indexes.

Examples The examples use the `inventory` collection that contains the following documents:

```
{ "_id": 1, "dept": "A", "item": { "sku": "111", "color": "red" }, "sizes": [ "S", "M" ] }
{ "_id": 2, "dept": "A", "item": { "sku": "111", "color": "blue" }, "sizes": [ "M", "L" ] }
{ "_id": 3, "dept": "B", "item": { "sku": "222", "color": "blue" }, "sizes": "S" }
{ "_id": 4, "dept": "A", "item": { "sku": "333", "color": "black" }, "sizes": [ "S" ] }
```

Return Distinct Values for a Field The following example returns the distinct values for the field `dept` from all documents in the `inventory` collection:

```
db.inventory.distinct( "dept" )
```

The method returns the following array of distinct `dept` values:

```
[ "A", "B" ]
```

Return Distinct Values for an Embedded Field The following example returns the distinct values for the field `sku`, embedded in the `item` field, from all documents in the `inventory` collection:

```
db.inventory.distinct( "item.sku" )
```

The method returns the following array of distinct `sku` values:

```
[ "111", "222", "333" ]
```

See also:

document-dot-notation for information on accessing fields within embedded documents

Return Distinct Values for an Array Field The following example returns the distinct values for the field `sizes` from all documents in the `inventory` collection:

```
db.inventory.distinct( "sizes" )
```

The method returns the following array of distinct `sizes` values:

```
[ "M", "S", "L" ]
```

For information on `distinct()` (page 44) and array fields, see the *Behavior* (page 44) section.

Specify Query with `distinct` The following example returns the distinct values for the field `sku`, embedded in the `item` field, from the documents whose `dept` is equal to `"A"`:

```
db.inventory.distinct( "item.sku", { dept: "A" } )
```

The method returns the following array of distinct `sku` values:

```
[ "111", "333" ]
```

db.collection.drop()

On this page

- Definition (page 45)
- Behavior (page 46)
- Example (page 46)

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 439) command.

`db.collection.drop()` (page 45) has the form:

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

`db.collection.drop()` (page 45) 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()

On this page

- Definition (page 46)
- Example (page 46)

Definition

`db.collection.dropIndex(index)`

Drops or removes the specified index from a collection. The `db.collection.dropIndex()` (page 46) method provides a wrapper around the `dropIndexes` (page 450) command.

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

The `db.collection.dropIndex()` (page 46) 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. [2]

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 46) method, use the `db.collection.getIndexes()` (page 73) method.

Example Consider a `pets` collection. Calling the `getIndexes()` (page 73) method on the `pets` collection returns the following indexes:

```
[
   {  "v" : 1,
      "key" : { "_id" : 1 },
      "ns" : "test.pets",
      "name" : "_id_"
   },
   {
```

[2] When using a `mongo` (page 803) 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.

```
      "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"
   }
]
```

The single field index on the field `cat` has the user-specified name of `catIdx`[3] 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 47) as a method on a collection object.

db.collection.ensureIndex()

On this page

- Definition (page 47)
- Additional Information (page 47)

Definition

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

Deprecated since version 3.0.0: `db.collection.ensureIndex()` (page 47) is now an alias for `db.collection.createIndex()` (page 36).

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

Additional Information

- Use `db.collection.createIndex()` (page 36) rather than `db.collection.ensureIndex()` (page 47) to create new indexes.
- The `https://docs.mongodb.org/manual/indexes` section of this manual for full documentation of indexes and indexing in MongoDB.

[3] 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`.

- db.collection.getIndexes() (page 73) to view the specifications of existing indexes for a collection.

db.collection.explain()

On this page

- Description (page 48)
- Behavior (page 48)
- Examples (page 50)
- Output (page 51)

Description

`db.collection.explain()`

Changed in version 3.2: Adds support for db.collection.distinct() (page 44)

New in version 3.0.

Returns information on the query plan for the following operations: aggregate() (page 20); count() (page 33); distinct() (page 44); find() (page 51); group() (page 75); remove() (page 101); and update() (page 117) methods.

To use db.collection.explain() (page 48), append to db.collection.explain() (page 48) 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 50). For a list of methods available for use with db.collection.explain() (page 48), see *db.collection.explain().help()* (page 49).

The db.collection.explain() (page 48) method has the following parameter:

param string verbosity Optional. 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 140), MongoDB interprets `true` as `"allPlansExecution"` and `false` as `"queryPlanner"`.

aggregate() (page 20) ignores the verbosity parameter and executes in `queryPlanner` mode.

For more information on the modes, see *Verbosity Modes* (page 48).

Behavior

Verbosity Modes The behavior of db.collection.explain() (page 48) and the amount of information returned depend on the `verbosity` mode.

aggregate() (page 20) ignores the verbosity parameter and executes in `queryPlanner` mode.

`queryPlanner` Mode By default, `db.collection.explain()` (page 48) runs in `queryPlanner` verbosity mode.

MongoDB runs the `query optimizer` to choose the winning plan for the operation under evaluation. `db.collection.explain()` (page 48) returns the `queryPlanner` (page 948) 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 48) 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 48) returns the `queryPlanner` (page 948) and `executionStats` (page 949) information for the evaluated method. However, `executionStats` (page 949) 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 48) 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 48) returns the `queryPlanner` (page 948) and `executionStats` (page 949) information for the evaluated method. The `executionStats` (page 949) includes the *completed* query execution information for the *winning plan*.

If the query optimizer considered more than one plan, `executionStats` (page 949) 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 48) method wraps the `explain` (page 467) command and is the preferred way to run `explain` (page 467).

`db.collection.explain().find()` is similar to `db.collection.find().explain()` (page 140) 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 49).
- 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 803) shell, the `mongo` (page 803) 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 49).

`db.collection.explain().aggregate()` is equivalent to passing the *explain* (page 23) option to the `db.collection.aggregate()` (page 20) method.

`help()` To see the list of operations supported by `db.collection.explain()` (page 48), 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 48) 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 50).

Examples

`queryPlanner` Mode By default, `db.collection.explain()` (page 48) runs in `"queryPlanner"` verbosity mode.

The following example runs `db.collection.explain()` (page 48) in *"queryPlanner"* (page 49) verbosity mode to return the query planning information for the specified `count()` (page 33) operation:

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

`executionStats` Mode The following example runs `db.collection.explain()` (page 48) in *"executionStats"* (page 49) verbosity mode to return the query planning and execution information for the specified `find()` (page 51) operation:

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

`allPlansExecution` Mode The following example runs `db.collection.explain()` (page 48) in *"allPlansExecution"* (page 49) verbosity mode. The `db.collection.explain()` (page 48) returns the `queryPlanner` (page 948) and `executionStats` (page 949) for all considered plans for the specified `update()` (page 117) 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 51) method with `sort()` (page 156) and `hint()` (page 142) 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 803) 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 803) shell, the `mongo` (page 803) 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 48) operations can return information regarding:

- *queryPlanner* (page 947), which details the plan selected by the `query optimizer` and lists the rejected plans;
- *executionStats* (page 948), which details the execution of the winning plan and the rejected plans; and
- *serverInfo* (page 951), which provides information on the MongoDB instance.

The verbosity mode (i.e. `queryPlanner`, `executionStats`, `allPlansExecution`) determines whether the results include *executionStats* (page 948) and whether *executionStats* (page 948) includes data captured during *plan selection*.

For details on the output, see *Explain Results* (page 946).

For a mixed version sharded cluster with version 3.0 `mongos` (page 792) and at least one 2.6 `mongod` (page 770) shard, when you run `db.collection.explain()` (page 48) in a version 3.0 `mongo` (page 803) shell, `db.collection.explain()` (page 48) will retry with the `$explain` operator to return results in the 2.6 format.

db.collection.find()

On this page

- Definition (page 51)
- Behavior (page 51)
- Examples (page 52)

Definition

`db.collection.`**`find`**`(`*query*, *projection*`)`

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

param document query Optional. Specifies selection criteria using *query operators* (page 527). To return all documents in a collection, omit this parameter or pass an empty document (`{}`).

param document projection Optional. Specifies the fields to return using *projection operators* (page 588). 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 51) method "returns documents," the method is actually returning a cursor to the documents.

Behavior

Projection If `find()` (page 51) includes a `projection` argument, the matching documents contain only the `projection` fields and the `_id` field. You can optionally exclude the `_id` field.

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 51) 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*.

Cursor Handling Executing `db.collection.find()` (page 51) in the `mongo` (page 803) shell automatically iterates the cursor to display up to the first 20 documents. Type `it` to continue iteration.

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

Read Concern To specify the `read concern` for `db.collection.find()` (page 51), use the `cursor.readConcern()` (page 152) method.

Examples

Find All Documents in a Collection The `find()` (page 51) 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.

See also:

specify-multiple-criteria-for-array-elements

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 an embedded document 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 Embedded Documents

Query Exact Matches on Embedded Documents The following operation returns documents in the `bios collection` where the embedded document `name` is *exactly* `{ first: "Yukihiro", last: "Matsumoto" }`, including the order:

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

The `name` field must match the embedded 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 an Embedded Document The following operation returns documents in the `bios collection` where the embedded document `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 an embedded document:

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

The query matches the document where the `name` field contains an embedded document 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` embedded document 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 Embedded Documents The following operation queries the `bios collection` and returns the `last` field in the `name` embedded document 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 51) method returns a *cursor* to the results.

In the `mongo` (page 803) shell, if the returned cursor is not assigned to a variable using the `var` keyword, the cursor is automatically iterated to access up to the first 20 documents that match the query. You can set the `DBQuery.shellBatchSize` variable to change the number of automatically iterated documents.

To manually iterate over the results, assign the returned cursor to a variable with the `var` keyword, as shown in the following sections.

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 150) 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 141) to iterate the cursor and access the documents:

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

myCursor.forEach(printjson);
```

Modify the Cursor Behavior The `mongo` (page 803) shell and the `drivers` provide several cursor methods that call on the *cursor* returned by the `find()` (page 51) method to modify its behavior.

Order Documents in the Result Set The `sort()` (page 156) 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 156) corresponds to the `ORDER BY` statement in SQL.

Limit the Number of Documents to Return The `limit()` (page 144) 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 144) corresponds to the `LIMIT` statement in SQL.

Set the Starting Point of the Result Set The `skip()` (page 155) 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 statements chain cursor methods `limit()` (page 144) and `sort()` (page 156):

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

The two statements are equivalent; i.e. the order in which you chain the `limit()` (page 144) and the `sort()` (page 156) methods is not significant. Both statements return the first five documents, as determined by the ascending sort order on 'name'.

db.collection.findAndModify()

On this page

- Definition (page 57)
- Return Data (page 58)
- Behavior (page 59)
- Examples (page 60)

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 57) method is a shell helper around the `findAndModify` (page 348) command.

The `findAndModify()` (page 57) method has the following form:

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

The `db.collection.findAndModify()` (page 57) method takes a document parameter with the following embedded document fields:

param document query Optional. The selection criteria for the modification. The `query` field employs the same *query selectors* (page 527) as used in the `db.collection.find()` (page 51) method. Although the query may match multiple documents, `findAndModify()` (page 57) **will only select one document to modify**.

param document sort Optional. Determines which document the operation modifies if the query selects multiple documents. `findAndModify()` (page 57) 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 595) or `field: value` specifications to modify the selected document.

param boolean new Optional. When `true`, returns the modified document rather than the original. The `findAndModify()` (page 57) method ignores the `new` option for `remove` operations. The default is `false`.

param document fields Optional. 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 Optional. Used in conjunction with the `update` field.

When `true`, `findAndModify()` (page 57) creates a new document if no document matches the `query`, or if documents match the `query`, `findAndModify()` (page 57) performs an update. To avoid multiple upserts, ensure that the `query` fields are *uniquely indexed*.

The default is `false`.

param boolean bypassDocumentValidation Optional. Enables `db.collection.findAndModify` (page 57) to bypass document validation during the operation. This lets you update documents that do not meet the validation requirements.

New in version 3.2.

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

New in version 3.2.

Return Data For remove operations, if the query matches a document, `findAndModify()` (page 57) returns the removed document. If the query does not match a document to remove, `findAndModify()` (page 57) returns `null`.

For update operations, `findAndModify()` (page 57) returns one of the following:

- If the `new` parameter is not set or is `false`:
  - the pre-modification document if the query matches a document;
  - otherwise, `null`.
- If `new` is `true`:
  - the modified document if the query returns a match;

  – the inserted document if `upsert: true` and no document matches the query;
  – otherwise, `null`.

Changed in version 3.0: In previous versions, if for the update, `sort` is specified, and `upsert: true`, and the `new` option is not set or `new: false`, `db.collection.findAndModify()` (page 57) returns an empty document `{}` instead of `null`.

Behavior

Upsert and Unique Index When `findAndModify()` (page 57) 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 57) 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 57) successfully inserts a new document.
- Zero or more `findAndModify()` (page 57) methods update the newly inserted document.
- Zero or more `findAndModify()` (page 57) 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 348) in a *sharded* environment, the `query` **must** contain the *shard key* for all operations against the shard cluster for the *sharded* collections.

`findAndModify` (page 348) operations issued against `mongos` (page 792) instances for *non-sharded* collections function normally.

Document Validation The `db.collection.findAndModify()` (page 57) method adds support for the `bypassDocumentValidation` option, which lets you bypass *document validation* (page 991) when inserting or updating documents in a collection with validation rules.

Comparisons with the `update` Method When updating a document, `findAndModify()` (page 57) and the `update()` (page 117) method operate differently:

- By default, both operations modify a single document. However, the `update()` (page 117) method with its `multi` option can modify more than one document.
- If multiple documents match the update criteria, for `findAndModify()` (page 57), you can specify a `sort` to provide some measure of control on which document to update.

With the default behavior of the `update()` (page 117) method, you cannot specify which single document to update when multiple documents match.

- By default, `findAndModify()` (page 57) returns the pre-modified version of the document. To obtain the updated document, use the `new` option.

  The `update()` (page 117) method returns a `WriteResult` (page 289) object that contains the status of the operation. To return the updated document, use the `find()` (page 51) 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.

When modifying a *single* document, both `findAndModify()` (page 57) and the `update()` (page 117) method *atomically* update the document. See `https://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`.

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
}
```

See also:

https://docs.mongodb.org/manual/tutorial/perform-findAndModify-quorum-reads

db.collection.findOne()

On this page

- Definition (page 62)
- Examples (page 62)

Definition

`db.collection.findOne(`*query*, *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. If no document satisfies the query, the method returns null.

param document query Optional. Specifies query selection criteria using *query operators* (page 527).

param document projection Optional. Specifies the fields to return using *projection operators* (page 588). 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 62) 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 62) 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 51) method, the `findOne()` (page 62) 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 embedded document `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` embedded document, 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 62) 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.findOneAndDelete()

On this page

- Definition (page 64)
- Behavior (page 64)

Definition

`db.collection.`**`findOneAndDelete`**(*filter*, *options*)

New in version 3.2.

Deletes a single document based on the `filter` and `sort` criteria, returning the deleted document.

The `findOneAndDelete()` (page 64) method has the following form:

```
db.collection.findOneAndDelete(
   <filter>,
   {
     projection: <document>,
     sort: <document>,
     maxTimeMS: <number>,
   }
)
```

The `findOneAndDelete()` (page 64) method takes the following parameters:

param document filter The selection criteria for the update. The same *query selectors* (page 527) as in the `find()` (page 51) method are available.

Specify an empty document `{ }` to delete the first document returned in the collection.

param document projection Optional. A subset of fields to return.

To return all fields in the returned document, omit this parameter.

param document sort Optional. Specifies a sorting order for the documents matched by the `filter`.

See `cursor.sort()` (page 156).

param number maxTimeMS Optional. Specifies a time limit in milliseconds within which the operation must complete within. Throws an error if the limit is exceeded.

Returns Returns the deleted document.

Behavior `findOneAndDelete()` (page 64) deletes the first matching document in the collection that matches the `filter`. The `sort` parameter can be used to influence which document is updated.

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

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

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

- `1` or `true` to include the field. The method returns the `_id` field even if it is not explicitly stated in the projection parameter.
- `0` or `false` to exclude the field. This can be used on any field, including `_id`.

Examples

Delete A Document The `grades` collection contains documents similar to the following:

```
{ _id: 6305, name : "A. MacDyver", "assignment" : 5, "points" : 24 },
{ _id: 6308, name : "B. Batlock", "assignment" : 3, "points" : 22 },
{ _id: 6312, name : "M. Tagnum", "assignment" : 5, "points" : 30 },
{ _id: 6319, name : "R. Stiles", "assignment" : 2, "points" : 12 },
{ _id: 6322, name : "A. MacDyver", "assignment" : 2, "points" : 14 },
{ _id: 6234, name : "R. Stiles", "assignment" : 1, "points" : 10 }
```

The following operation finds the first document where `name : M. Tagnum` and deletes it:

```
db.scores.findOneAndDelete(
   { "name" : "M. Tagnum" }
)
```

The operation returns the *original* document that has been deleted:

```
{ _id: 6312, name: "M. Tagnum", "assignment" : 5, "points" : 30 }
```

Sort And Delete A Document The `grades` collection contains documents similar to the following:

```
{ _id: 6305, name : "A. MacDyver", "assignment" : 5, "points" : 24 },
{ _id: 6308, name : "B. Batlock", "assignment" : 3, "points" : 22 },
{ _id: 6312, name : "M. Tagnum", "assignment" : 5, "points" : 30 },
{ _id: 6319, name : "R. Stiles", "assignment" : 2, "points" : 12 },
{ _id: 6322, name : "A. MacDyver", "assignment" : 2, "points" : 14 },
{ _id: 6234, name : "R. Stiles", "assignment" : 1, "points" : 10 }
```

The following operation first finds all documents where `name : "A. MacDyver"`. It then sorts by `points` ascending before deleting the document with the lowest points value:

```
db.scores.findOneAndDelete(
   { "name" : "A. MacDyver" },
   { sort : { "points" : 1 } }
)
```

The operation returns the *original* document that has been deleted:

```
{ _id: 6322, name: "A. MacDyver", "assignment" : 2, "points" : 14 }
```

Projecting the Deleted Document The following operation uses projection to only return the `_id` and `assignment` fields in the returned document:

```
db.scores.findOneAndDelete(
   { "name" : "A. MacDyver" },
   { sort : { "points" : 1 }, projection: { "assignment" : 1 } }
)
```

The operation returns the *original* document with the `assignment` and `_id` fields:

```
{ _id: 6322, "assignment" : 2 }
```

Update Document with Time Limit The following operation sets a 5ms time limit to complete the deletion:

```
try {
   db.scores.findOneAndDelete(
      { "name" : "A. MacDyver" },
      { sort : { "points" : 1 }, maxTimeMS : 5 };
   );
}
catch(e){
   print(e);
}
```

If the operation exceeds the time limit, it returns:

```
Error: findAndModifyFailed failed: { "ok" : 0, "errmsg" : "operation exceeded time limit", "code" : 5
```

db.collection.findOneAndReplace()

On this page

- Definition (page 66)
- Behavior (page 67)

Definition

`db.collection.`**`findOneAndReplace`**(*filter*, *replacement*, *options*)

New in version 3.2.

Modifies and replaces a single document based on the `filter` and `sort` criteria.

The `findOneAndReplace()` (page 66) method has the following form:

```
db.collection.findOneAndReplace(
   <filter>,
   <replacement>,
   {
     projection: <document>,
     sort: <document>,
     maxTimeMS: <number>,
     upsert: <boolean>,
     returnNewDocument: <boolean>
   }
)
```

The `findOneAndReplace()` (page 66) method takes the following parameters:

param document filter The selection criteria for the update. The same *query selectors* (page 527) as in the `find()` (page 51) method are available.

Specify an empty document `{ }` to replace the first document returned in the collection.

param document replacement The replacement document.

Cannot contain *update operators* (page 594).

The `<replacement>` document cannot specify an `_id` value that differs from the replaced document.

param document projection Optional. A subset of fields to return.

To return all fields in the matching document, omit this parameter.

param document sort Optional. Specifies a sorting order for the documents matched by the `filter`.

See `cursor.sort()` (page 156).

param number maxTimeMS Optional. Specifies a time limit in milliseconds within which the operation must complete within. Throws an error if the limit is exceeded.

param boolean upsert Optional. When `true`, `findOneAndReplace()` (page 66) creates a new document if no document matches the `filter`. If a document matches the filter, the method performs a replacement.

The new document is created using the equality conditions from the `filter` with the `replacement` document.

Comparison conditions like `$gt` (page 529) or `$lt` (page 530) are ignored.

Returns `null` after inserting the new document, unless `returnNewDocument` is `true`.

Defaults to `false`.

param boolean returnNewDocument Optional. When `true`, returns the replacement document instead of the original document.

Defaults to `false`.

Returns Returns either the original document or, if `returnNewDocument: true`, the replacement document.

Behavior `findOneAndReplace()` (page 66) replaces the first matching document in the collection that matches the `filter`. The `sort` parameter can be used to influence which document is modified.

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

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

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

- `1` or `true` to include the field. The method returns the `_id` field even if it is not explicitly stated in the projection parameter.
- `0` or `false` to exclude the field. This can be used on any field, including `_id`.

Examples

Replace A Document The `scores` collection contains documents similar to the following:

```
{ "_id" : 1521, "team" : "Fearful Mallards", "score" : 25000 },
{ "_id" : 2231, "team" : "Tactful Mooses", "score" : 23500 },
{ "_id" : 4511, "team" : "Aquatic Ponies", "score" : 19250 },
{ "_id" : 5331, "team" : "Cuddly Zebras", "score" : 15235 },
{ "_id" : 3412, "team" : "Garrulous Bears", "score" : 22300 }
```

The following operation finds the first document with `score` less than `20000` and replaces it:

```
db.scores.findOneAndReplace(
   { "score" : { $lt : 20000 } },
   { "team" : "Observant Badgers", "score" : 20000 }
)
```

The operation returns the *original* document that has been replaced:

```
{ "_id" : 2512, "team" : "Aquatic Ponies", "score" : 19250 }
```

If `returnNewDocument` was true, the operation would return the replacement document instead.

Sort and Replace A Document The `scores` collection contains documents similar to the following:

```
{ "_id" : 1521, "team" : "Fearful Mallards", "score" : 25000 },
{ "_id" : 2231, "team" : "Tactful Mooses", "score" : 23500 },
{ "_id" : 4511, "team" : "Aquatic Ponies", "score" : 19250 },
{ "_id" : 5331, "team" : "Cuddly Zebras", "score" : 15235 },
{ "_id" : 3412, "team" : "Garrulous Bears", "score" : 22300 }
```

Sorting by `score` changes the result of the operation. The following operation sorts the result of the `filter` by `score` ascending, and replaces the lowest scoring document:

```
db.scores.findOneAndReplace(
   { "score" : { $lt : 20000 } },
   { "team" : "Observant Badgers", "score" : 20000 },
   { sort: { "score" : 1 } }
)
```

The operation returns the *original* document that has been replaced:

```
{ "_id" : 5112, "team" : "Cuddly Zebras", "score" : 15235 }
```

See *Replace A Document* (page 67) for the non-sorted result of this command.

Project the Returned Document The `scores` collection contains documents similar to the following:

```
{ "_id" : 1521, "team" : "Fearful Mallards", "score" : 25000 },
{ "_id" : 2231, "team" : "Tactful Mooses", "score" : 23500 },
{ "_id" : 4511, "team" : "Aquatic Ponies", "score" : 19250 },
{ "_id" : 5331, "team" : "Cuddly Zebras", "score" : 15235 },
{ "_id" : 3412, "team" : "Garrulous Bears", "score" : 22300 }
```

The following operation uses projection to only display the `team` field in the returned document:

```
db.scores.findOneAndReplace(
   { "score" : { $lt : 22250 } },
   { "team" : "Therapeutic Hamsters", "score" : 22250 },
   { sort : { "score" : 1 }, project: { "_id" : 0, "team" : 1 } }
)
```

The operation returns the *original* document with only the `team` field:

```
{ "team" : "Aquatic Ponies"}
```

Replace Document with Time Limit The following operation sets a 5ms time limit to complete:

```
try {
   db.scores.findOneAndReplace(
      { "score" : { $gt : 25000 } },
      { "team" : "Emphatic Rhinos", "score" : 25010 },
      { maxTimeMS: 5 }
   );
}
```

```
catch(e){
   print(e);
}
```

If the operation exceeds the time limit, it returns:

```
Error: findAndModifyFailed failed: { "ok" : 0, "errmsg" : "operation exceeded time limit", "code" : 5
```

Replace Document with Upsert The following operation uses the `upsert` field to insert the replacement document if nothing matches the `filter`:

```
try {
   db.scores.findOneAndReplace(
      { "team" : "Fortified Lobsters" },
      { "_id" : 6019, "team" : "Fortified Lobsters" , "score" : 32000},
      { upsert : true, returnNewDocument: true }
   );
}
catch (e){
   print(e);
}
```

The operation returns the following:

```
{
     "_id" : 6019,
     "team" : "Fortified Lobsters",
     "score" : 32000
}
```

If `returnNewDocument` was false, the operation would return `null` as there is no original document to return.

db.collection.findOneAndUpdate()

On this page

- Definition (page 69)
- Behavior (page 70)

Definition

`db.collection.`**`findOneAndUpdate`**(*filter*, *update*, *options*)

New in version 3.2.

Updates a single document based on the `filter` and `sort` criteria.

The `findOneAndUpdate()` (page 69) method has the following form:

```
db.collection.findOneAndUpdate(
   <filter>,
   <update>,
   {
     projection: <document>,
     sort: <document>,
     maxTimeMS: <number>,
     upsert: <boolean>,
     returnNewDocument: <boolean>
   }
)
```

The `findOneAndUpdate()` (page 69) method takes the following parameters:

param document filter The selection criteria for the update. The same *query selectors* (page 527) as in the `find()` (page 51) method are available.

Specify an empty document `{ }` to update the first document returned in the collection.

param document update The update document.

Must contain only *update operators* (page 594).

The `<replacement>` document cannot specify an `_id` value that differs from the replaced document.

param document projection Optional. A subset of fields to return.

To return all fields in the returned document, omit this parameter.

param document sort Optional. Specifies a sorting order for the documents matched by the `filter`.

See `cursor.sort()` (page 156).

param number maxTimeMS Optional. Specifies a time limit in milliseconds within which the operation must complete within. Throws an error if the limit is exceeded.

param boolean upsert Optional. When `true`, `findOneAndUpdate()` (page 69) creates a new document if no document matches the `filter`. If a document matches the filter, the method performs an update.

The new document is created using the equality conditions from the `filter` with the modifications from the `update` document.

Comparison conditions like `$gt` (page 529) or `$lt` (page 530) are ignored.

Returns `null` after inserting the new document, unless `returnNewDocument` is `true`.

Defaults to `false`.

param boolean returnNewDocument Optional. When `true`, returns the replacement document instead of the original document.

Defaults to `false`.

Returns Returns either the original document or, if `returnNewDocument: true`, the updated document.

Behavior `findOneAndUpdate()` (page 69) updates the first matching document in the collection that matches the `filter`. The `sort` parameter can be used to influence which document is updated.

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

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

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

- `1` or `true` to include the field. The method returns the `_id` field even if it is not explicitly stated in the projection parameter.

- 0 or false to exclude the field. This can be used on any field, including _id.

Examples

Replace A Document The grades collection contains documents similar to the following:

```
{ _id: 6305, name : "A. MacDyver", "assignment" : 5, "points" : 24 },
{ _id: 6308, name : "B. Batlock", "assignment" : 3, "points" : 22 },
{ _id: 6312, name : "M. Tagnum", "assignment" : 5, "points" : 30 },
{ _id: 6319, name : "R. Stiles", "assignment" : 2, "points" : 12 },
{ _id: 6322, name : "A. MacDyver", "assignment" : 2, "points" : 14 },
{ _id: 6234, name : "R. Stiles", "assignment" : 1, "points" : 10 }
```

The following operation finds the first document where name : R. Stiles and increments the score by 5:

```
db.scores.findOneAndUpdate(
   { "name" : "R. Stiles" },
   { $inc: { "points" : 5 } }
)
```

The operation returns the *original* document that has been replaced:

```
{ _id: 6319, name: "R. Stiles", "assignment" : 2, "points" : 12 }
```

If returnNewDocument was true, the operation would return the replacement document instead.

Sort And Update A Document The grades collection contains documents similar to the following:

```
{ _id: 6305, name : "A. MacDyver", "assignment" : 5, "points" : 24 },
{ _id: 6308, name : "B. Batlock", "assignment" : 3, "points" : 22 },
{ _id: 6312, name : "M. Tagnum", "assignment" : 5, "points" : 30 },
{ _id: 6319, name : "R. Stiles", "assignment" : 2, "points" : 12 },
{ _id: 6322, name : "A. MacDyver", "assignment" : 2, "points" : 14 },
{ _id: 6234, name : "R. Stiles", "assignment" : 1, "points" : 10 }
```

The following operation updates a document where name : "A. MacGyver". The operation sorts the matching documents by points ascending to update the matching document with the least points.

```
db.scores.findOneAndUpdate(
   { "name" : "A. MacDyver" },
   { $inc : { "points" : 5 } },
   { sort : { "points" : 1 } }
)
```

The operation returns the *original* document that has been replaced:

```
{ _id: 6322, name: "A. MacDyver", "assignment" : 2, "points" : 14 }
```

Project the Returned Document The following operation uses projection to only display the _id, points, and assignment fields in the returned document:

```
db.scores.findOneAndUpdate(
   { "name" : "A. MacDyver" },
   { $inc : { "points" : 5 } },
   { sort : { "points" : 1 }, projection: { "assignment" : 1, "points" : 1 } }
)
```

The operation returns the *original* document with only the `assignment` field:

```
{ "_id" : 6322, "assignment" : 2, "points" : 14 }
```

Update Document with Time Limit The following operation sets a 5ms time limit to complete the update:

```
try {
   db.scores.findOneAndUpdate(
      { "name" : "A. MacDyver" },
      { $inc : { "points" : 5 } },
      { sort: { "points" : 1 }, maxTimeMS : 5 };
   );
}
catch(e){
   print(e);
}
```

If the operation exceeds the time limit, it returns:

```
Error: findAndModifyFailed failed: { "ok" : 0, "errmsg" : "operation exceeded time limit", "code" : 5
```

Update Document with Upsert The following operation uses the `upsert` field to insert the update document if nothing matches the `filter`:

```
try {
db.scores.findOneAndUpdate(
   { "name" : "A.B. Abracus" },
   { $set: { "name" : "A.B. Abracus", "assignment" : 5}, $inc : { "points" : 5 } },
   { sort: { "points" : 1 }, returnNewDocument : true }
);
}
catch (e){
   print(e);
}
```

The operation returns the following:

```
{
     "_id" : 5239,
     "name" : "A.B. Abracus",
     "assignment" : 5,
   "points" : 5
}
```

If `returnNewDocument` was false, the operation would return `null` as there is no original document to return.

db.collection.getIndexes()

On this page

- Definition (page 73)
- Considerations (page 73)
- Output (page 73)

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 `db.collection.getIndexes()` (page 73) on a collection. For example:

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

Change `collection` to the name of the collection for which to return index information.

Considerations Changed in version 3.0.0.

For MongoDB 3.0 deployments using the *WiredTiger* storage engine, if you run `db.collection.getIndexes()` (page 73) from a version of the `mongo` (page 803) shell before 3.0 or a version of the driver prior to *3.0 compatible version* (page 1051), `db.collection.getIndexes()` (page 73) will return no data, even if there are existing indexes. For more information, see *WiredTiger and Driver Version Compatibility* (page 1047).

Output `db.collection.getIndexes()` (page 73) returns an array of documents that hold index information for the collection. Index information includes the keys and options used to create the index. For information on the keys and index options, see `db.collection.createIndex()` (page 36).

db.collection.getShardDistribution()

On this page

- Definition (page 73)
- Output (page 74)
- Example Output (page 74)

Definition

`db.collection.getShardDistribution()`

Returns

Prints the data distribution statistics for a *sharded* collection. You must call the `getShardDistribution()` (page 73) 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:

https://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()

On this page

- Definition (page 75)
- Behavior (page 76)
- Examples (page 77)

Recommended Alternatives

Because `db.collection.group()` (page 75) uses JavaScript, it is subject to a number of performance limitations. For most cases the `$group` (page 644) 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 75) method returns an array.

The `db.collection.group()` (page 75) 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 Optional. 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 75) processes all the documents in the collection for the group operation.

field function finalize Optional. A function that runs each item in the result set before `db.collection.group()` (page 75) 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 75) method is a shell wrapper for the `group` (page 313) command. However, the `db.collection.group()` (page 75) method takes the `keyf` field and the `reduce` field whereas the `group` (page 313) command takes the `$keyf` field and the `$reduce` field.

Behavior

Limits and Restrictions The `db.collection.group()` (page 75) 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 940).

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 318). Previous versions had a limit of 10,000 elements.

Prior to 2.4, the `db.collection.group()` (page 75) method took the `mongod` (page 770) 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 318), the `group` (page 313) command, and `$where` (page 558) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 803) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 318), `group` (page 313) commands, or `$where` (page 558) 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 318), the `group` (page 313) command, and `$where` (page 558) 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:

`https://docs.mongodb.org/manual/aggregation`

db.collection.insert()

On this page

- Definition (page 79)
- Behaviors (page 80)
- Examples (page 80)
- WriteResult (page 81)
- BulkWriteResult (page 82)

Definition

`db.collection.insert()`

Inserts a document or documents into a collection.

The `insert()` (page 79) 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 Optional. A document expressing the `write concern`. Omit to use the default write concern. See *Write Concern* (page 80).

New in version 2.6.

param boolean ordered Optional. 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 79) returns an object that contains the status of the operation.

Returns

- A *WriteResult* (page 81) object for single inserts.

- A *BulkWriteResult* (page 82) object for bulk inserts.

Behaviors

Write Concern Changed in version 2.6.

The `insert()` (page 79) method uses the `insert` (page 337) 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 79) 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 `https://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 770) 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 79) method creates the collection.

Insert a Document without Specifying an `_id` Field In the following example, the document passed to the `insert()` (page 79) method does not contain the `_id` field:

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

During the insert, `mongod` (page 770) will create the `_id` field and assign it a unique `https://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 79) 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 79) 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 770) 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 voting replica set members or the method times out after 5 seconds.

Changed in version 3.0: In previous versions, majority referred to the majority of all members of the replica set.

```
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 79) returns a WriteResult object.

Successful Results The insert() (page 79) returns a WriteResult (page 289) object that contains the status of the operation. Upon success, the WriteResult (page 289) object contains information on the number of documents inserted:

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

Write Concern Errors If the insert() (page 79) method encounters write concern errors, the results include the WriteResult.writeConcernError (page 290) 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 79) method encounters a non-write concern error, the results include the `WriteResult.writeError` (page 290) 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 79) returns a `BulkWriteResult()` (page 291) object. See `BulkWriteResult()` (page 291) for details.

db.collection.insertOne()

On this page

- Definition (page 82)
- Behaviors (page 83)
- Examples (page 83)

Definition

`db.collection.insertOne()`

New in version 3.2.

Inserts a document into a collection.

The `insertOne()` (page 82) method has the following syntax:

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

param document document A document to insert into the collection.

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

Returns

A document containing:

- A boolean `acknowledged` as `true` if the operation ran with *write concern* or `false` if write concern was disabled.
- A single element `insertedId` with the `_id` of the inserted document.

Behaviors

Collection Creation If the collection does not exist, then the `insertOne()` (page 82) method creates the collection.

`_id` Field If the document does not specify an *_id* field, then `mongod` (page 770) will add the `_id` field and assign a unique `https://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 770) 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.

Explainability `insertOne()` (page 82) is not compatible with `db.collection.explain()` (page 48).

Use `insert()` (page 79) instead.

Error Handling On error, `insertOne()` (page 82) throws either a `writeError` or `writeConcernError` exception.

Examples

Insert a Document without Specifying an `_id` Field In the following example, the document passed to the `insertOne()` (page 82) method does not contain the `_id` field:

```
try {
   db.products.insertOne( { item: "card", qty: 15 } );
};
catch (e) {
   print (e);
};
```

The operation returns the following document:

```
{
   "acknowledged" : true,
   "insertedIds" : [
      ObjectId("562a94d381cb9f1cd6eb0e1a"),
   ]
}
```

Because the documents did not include `_id`, `mongod` (page 770) creates and adds the `_id` field and assigns it a unique `https://docs.mongodb.org/manual/reference/object-id` value.

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 `insertOne()` (page 82) method includes the `_id` field. The value of `_id` must be unique within the collection to avoid duplicate key error.

```
try {
   db.products.insertOne( { _id: 10, item: "box", qty: 20 } );
}
catch (e) {
   print (e);
}
```

The operation returns the following:

```
{
   "acknowledged" : true,
   "insertedIds" : ObjectId("562a94d381cb9f1cd6eb0e1a")
}
```

Inserting an duplicate value for any key that is part of a *unique index*, such as `_id`, throws an exception. The following attempts to insert a document with a `_id` value that already exists:

```
try {
   db.products.insertOne( { _id: 10, "item" : "packing peanuts", "qty" : 200 } );
}
catch (e) {
   print (e);
}
```

Since `_id: 10` already exists, the following exception is thrown:

```
WriteError({
   "index" : 0,
   "code" : 11000,
   "errmsg" : "E11000 duplicate key error collection: inventory.products index: _id_ dup key: { : 1.
   "op" : {
      "_id" : 10,
      "item" : "packing peanuts",
      "qty" : 200
   }
})
```

Increasing Write Concern Given a three member replica set, the following operation specifies a `w` of `majority`, `wtimeout` of `100`:

```
try {
   db.products.insertOne(
       { "item": "envelopes", "qty": 100, type: "Self-Sealing" },
       { writeConcern: { w : "majority", wtimeout : 100 } };
   )
}
catch (e) {
   print (e);
}
```

If the acknowledgement takes longer than the `wtimeout` limit, the following exception is thrown:

```
WriteConcernError({
   "code" : 64,
   "errInfo" : {
      "wtimeout" : true
   },
   "errmsg" : "waiting for replication timed out"
})
```

See also:

To insert multiple documents, see `db.collection.insertMany()` (page 85)

db.collection.insertMany()

On this page

- Definition (page 85)
- Behaviors (page 85)
- Examples (page 86)

Definition

`db.collection.insertMany()`

New in version 3.2.

Inserts multiple documents into a collection.

The `insertMany()` (page 85) method has the following syntax:

```
db.collection.insertMany(
   { [ <document 1> , <document 2>, ... ] },
   {
      writeConcern: <document>,
      ordered: <boolean>
   }
)
```

param document document An array of documents to insert into the collection.

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

param boolean ordered Optional. A boolean specifying whether the `mongod` (page 770) instance should perform an ordered or unordered insert. Defaults to `true`.

Returns

A document containing:

- A boolean `acknowledged` as `true` if the operation ran with *write concern* or `false` if write concern was disabled
- An array of `_id` for each successfully inserted documents

Behaviors Given an array of documents, `insertMany()` (page 85) inserts each document in the array into the collection.

Execution of Operations By default documents are inserted in order.

If `ordered` is set to false, documents are inserted in an unordered format and may be reordered by `mongod` (page 770) to increase performance. Applications should not depend on ordering of inserts if using an unordered `insertMany()` (page 85).

Each group of operations can have at most `1000 operations` (page 945). If a group exceeds this `limit` (page 945), MongoDB will divide the group into smaller groups of 1000 or less. For example, if the queue consists of 2000 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.

Executing an `ordered` (page 212) list of operations on a sharded collection will generally be slower than executing an `unordered` (page 213) list since with an ordered list, each operation must wait for the previous operation to finish.

Collection Creation If the collection does not exist, then `insertMany()` (page 85) creates the collection on successful write.

_id Field If the document does not specify an *_id* field, then `mongod` (page 770) adds the `_id` field and assign a unique `https://docs.mongodb.org/manual/reference/object-id` for the document before insertManying. Most drivers create an ObjectId and insert the `_id` field, but the `mongod` (page 770) 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.

Explainability `insertMany()` (page 85) is not compatible with `db.collection.explain()` (page 48).

Use `insert()` (page 79) instead.

Error Handling Inserts throw a `BulkWriteError` exception.

Excluding `https://docs.mongodb.org/manual/reference/write-concern` errors, ordered operations stop after an error, while unordered operations continue to process any remaining write operations in the queue.

Write concern errors are displayed in the `writeConcernErrors` field, while all other errors are displayed in the `writeErrors` field. If an error is encountered, the number of successful write operations are displayed instead of a list of inserted _ids. Ordered operations display the single error encountered while unordered operations display each error in an array.

Examples The following examples insert documents into the `products` collection.

Insert Several Document without Specifying an _id Field The following example uses `db.collection.insertMany()` (page 85) to insert documents that do not contain the `_id` field:

```
try {
   db.products.insertMany( [
      { item: "card", qty: 15 },
      { item: "envelope", qty: 20 },
      { item: "stamps" , qty: 30 }
   ] );
}
catch (e) {
   print (e);
}
```

The operation returns the following document:

```
{
   "acknowledged" : true,
   "insertedIds" : [
      ObjectId("562a94d381cb9f1cd6eb0e1a"),
      ObjectId("562a94d381cb9f1cd6eb0e1b"),
      ObjectId("562a94d381cb9f1cd6eb0e1c")
   ]
}
```

Because the documents did not include `_id`, `mongod` (page 770) creates and adds the `_id` field for each document and assigns it a unique `https://docs.mongodb.org/manual/reference/object-id` value.

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 Several Document Specifying an `_id` Field The following example/operation uses `insertMany()` (page 85) to insert documents that include the `_id` field. The value of `_id` must be unique within the collection to avoid a duplicate key error.

```
try {
   db.products.insertMany( [
      { _id: 10, item: "large box", qty: 20 },
      { _id: 11, item: "small box", qty: 55 },
      { _id: 12, item: "medium box", qty: 30 }
   ] );
}
catch (e) {
   print (e);
}
```

The operation returns the following document:

```
{ "acknowledged" : true, "insertedIds" : [ 10, 11, 12 ] }
```

Inserting a duplicate value for any key that is part of a *unique index*, such as `_id`, throws an exception. The following attempts to insert a document with a `_id` value that already exists:

```
try {
   db.products.insertMany( [
      { _id: 13, item: "envelopes", qty: 60 },
      { _id: 13, item: "stamps", qty: 110 },
      { _id: 14, item: "packing tape", qty: 38 }
   ] );
}
catch (e) {
   print (e);
}
```

Since `_id: 13` already exists, the following exception is thrown:

```
BulkWriteError({
   "writeErrors" : [
      {
         "index" : 0,
         "code" : 11000,
         "errmsg" : "E11000 duplicate key error collection: restaurant.test index: _id_ dup key: { :
         "op" : {
            "_id" : 13,
            "item" : "envelopes",
            "qty" : 60
         }
      }
   ],
   "writeConcernErrors" : [ ],
   "nInserted" : 0,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

Note that one document was inserted: The first document of `_id: 13` will insert successfully, but the second insert will fail. This will also stop additional documents left in the queue from being inserted.

With `ordered` to `false`, the insert operation would continue with any remaining documents.

Unordered Inserts The following attempts to insert multiple documents with `_id` field and `ordered: false`. The array of documents contains two documents with duplicate `_id` fields.

```
try {
   db.products.insertMany( [
      { _id: 10, item: "large box", qty: 20 },
      { _id: 11, item: "small box", qty: 55 },
      { _id: 11, item: "medium box", qty: 30 },
      { _id: 12, item: "envelope", qty: 100},
      { _id: 13, item: "stamps", qty: 125 },
      { _id: 13, item: "tape", qty: 20},
      { _id: 14, item: "bubble wrap", qty: 30}
   ], { ordered: false } );
}
catch (e) {
   print (e);
}
```

The operation throws the following exception:

```
BulkWriteError({
   "writeErrors" : [
      {
         "index" : 2,
         "code" : 11000,
         "errmsg" : "E11000 duplicate key error collection: inventory.products index: _id_ dup key:
         "op" : {
            "_id" : 11,
            "item" : "medium box",
            "qty" : 30
         }
      },
      {
         "index" : 5,
         "code" : 11000,
         "errmsg" : "E11000 duplicate key error collection: inventory.products index: _id_ dup key:
         "op" : {
            "_id" : 13,
            "item" : "tape",
            "qty" : 20
         }
      }
   ],
   "writeConcernErrors" : [ ],
   "nInserted" : 5,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

While the document with `item: "medium box"` and `item: "tape"` failed to insert due to duplicate `_id` values, `nInserted` shows that the remaining 5 documents were inserted.

Using Write Concern Given a three member replica set, the following operation specifies a `w` of `majority` and `wtimeout` of `100`:

```
try {
   db.products.insertMany(
      [
         { _id: 10, item: "large box", qty: 20 },
         { _id: 11, item: "small box", qty: 55 },
         { _id: 12, item: "medium box", qty: 30 }
      ],
      { w: "majority", wtimeout: 100 }
   );
}
catch (e) {
   print (e);
}
```

If the primary and at least one secondary acknowledge each write operation within 100 milliseconds, it returns:

```
{
   "acknowledged" : true,
   "insertedIds" : [
      ObjectId("562a94d381cb9f1cd6eb0e1a"),
      ObjectId("562a94d381cb9f1cd6eb0e1b"),
      ObjectId("562a94d381cb9f1cd6eb0e1c")
   ]
}
```

If the total time required for all required nodes in the replica set to acknowledge the write operation is greater than `wtimeout`, the following `writeConcernError` is displayed when the `wtimeout` period has passed.

This operation returns:

```
WriteConcernError({
   "code" : 64,
   "errInfo" : {
      "wtimeout" : true
   },
   "errmsg" : "waiting for replication timed out"
})
```

db.collection.isCapped()

`db.collection.isCapped()`

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

See also:

https://docs.mongodb.org/manual/core/capped-collections

db.collection.mapReduce()

On this page

- Requirements for the `map` Function (page 92)
- Requirements for the `reduce` Function (page 93)
- `out` Options (page 93)
- Requirements for the `finalize` Function (page 94)
- Map-Reduce Examples (page 95)
- Output (page 97)
- Additional Information (page 97)

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

The `db.collection.mapReduce()` (page 90) method provides a wrapper around the `mapReduce` (page 318) command.

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

`db.collection.mapReduce()` (page 90) takes the following parameters:

param 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 92) for more information.

param 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 93) for more information.

param document options A document that specifies additional parameters to `db.collection.mapReduce()` (page 90).

field boolean bypassDocumentValidation Optional. Enables `mapReduce` (page 318) to bypass document validation during the operation. This lets you insert documents that do not meet the validation requirements.

New in version 3.2.

The following table describes additional arguments that `db.collection.mapReduce()` (page 90) 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 93) for more information.

field document query Specifies the selection criteria using *query operators* (page 527) 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 function finalize Optional. Follows the `reduce` method and modifies the output.

See *Requirements for the finalize Function* (page 94) 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 318), the `group` (page 313) command, and `$where` (page 558) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 803) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 318), `group` (page 313) commands, or `$where` (page 558) 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 318), the `group` (page 313) command, and `$where` (page 558) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	
`args`	`assert()`	`Map()`
`MaxKey`	`BinData()`	`MD5()`
`MinKey`	`DBPointer()`	`NumberInt()`
	`DBRef()`	`NumberLong()`
	`doassert()`	`ObjectId()`
	`emit()`	`print()`
	`gc()`	`printjson()`
	`HexData()`	`printjsononeline()`
	`hex_md5()`	`sleep()`
	`isNumber()`	`Timestamp()`
	`isObject()`	`tojson()`
	`ISODate()`	`tojsononeline()`
	`isString()`	`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 940).
- 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 940). 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 940).

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 90) method is identical to that of the `mapReduce` (page 318) command. See the *Output* (page 325) section of the `mapReduce` (page 318) command for information on the `db.collection.mapReduce()` (page 90) output.

Additional Information

- `https://docs.mongodb.org/manual/tutorial/troubleshoot-map-function`
- `https://docs.mongodb.org/manual/tutorial/troubleshoot-reduce-function`
- `mapReduce` (page 318) command
- `https://docs.mongodb.org/manual/aggregation`
- `Map-Reduce`
- `https://docs.mongodb.org/manual/tutorial/perform-incremental-map-reduce`

db.collection.reIndex()

On this page

- Behavior (page 98)

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

The `db.collection.reIndex()` (page 97) 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 97) 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 97) will not propagate from the *primary* to *secondaries*. `db.collection.reIndex()` (page 97) will only affect a single `mongod` (page 770) instance.

Important: `db.collection.reIndex()` (page 97) will rebuild indexes in the *background if the index was originally specified with this option*. However, `db.collection.reIndex()` (page 97) 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 454) and `repairDatabase` (page 462) commands as well as the `db.collection.reIndex()` (page 97) 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 462) command, from continuing with the remainder of the process.

See

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

db.collection.replaceOne()

On this page

- Definition (page 98)
- Behavior (page 99)
- Examples (page 99)

Definition

`db.collection.replaceOne(`*filter*, *replacement*, *options*`)`

New in version 3.2.

Replaces a single document within the collection based on the filter.

The `replaceOne()` (page 98) method has the following form:

```
db.collection.replaceOne(
   <filter>,
   <replacement>,
   {
     upsert: <boolean>,
     writeConcern: <document>
   }
)
```

The `replaceOne()` (page 98) method takes the following parameters:

param document filter The selection criteria for the update. The same *query selectors* (page 527) as in the `find()` (page 51) method are available.

Specify an empty document `{ }` to replace the first document returned in the collection.

param document replacement The replacement document.

Cannot contain *update operators* (page 594).

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

param boolean upsert Optional. When `true`, if no documents match the `filter`, a new document is inserted based on the `replacement` document.

Returns

A document containing:

- A boolean `acknowledged` as `true` if the operation ran with *write concern* or `false` if write concern was disabled
- `matchedCount` containing the number of matched documents
- `modifiedCount` containing the number of modified documents
- `upsertedId` containing the `_id` for the upserted document

Behavior `replaceOne()` (page 98) replaces the first matching document in the collection that matches the `filter`, using the `replacement` document.

If `upsert: true` and no documents match the `filter`, `replaceOne()` (page 98) creates a new document based on the `replacement` document. See *Replace with Upsert* (page 100).

Capped Collections `replaceOne()` (page 98) throws a `WriteError` if the replacement document has a larger size in `bytes` than the original document.

Examples

Replace The `restaurant` collection contains the following documents:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "Borough" : "Manhattan" },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "Borough" : "Queens", "violations" : "2" },
{ "_id" : 3, "name" : "Empire State Pub", "Borough" : "Brooklyn", "violations" : "0" }
```

The following operation replaces a single document where `name: "Central Perk Cafe"`:

```
try {
   db.inventory.replaceOne(
      { "name" : "Central Perk Cafe" },
      { "name" : "Central Pork Cafe", "Borough" : "Manhattan" }
   );
}
catch (e){
   print(e);
}
```

The operation returns:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

If no matches were found, the operation instead returns:

```
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }
```

Setting `upsert: true` would insert the document if no match was found. See *Replace with Upsert* (page 100)

Replace with Upsert The `restaurant` collection contains the following documents:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "Borough" : "Manhattan", "violations" : 3 },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "Borough" : "Queens", "violations" : "2" },
{ "_id" : 3, "name" : "Empire State Pub", "Borough" : "Brooklyn", "violations" : "0" }
```

The following operation attempts to replace the document with `name : "Pizza Rat's Pizzaria"`, with `upsert : true`:

```
try {
   db.restaurant.replaceOne(
      { "name" : "Pizza Rat's Pizzaria" },
      { "_id:" 4, "name" : "Pizza Rat's Pizzaria", "Borough" : "Manhattan", "violations" : 8 },
      { upsert: true }
   )
}
catch (e){
   print(e);
}
```

Since `upsert : true` the document is inserted based on the `replacement` document. The operation returns:

```
{
   "acknowledged" : true,
   "matchedCount" : 0,
   "modifiedCount" : 0,
   "upsertedId" : 4
}
```

The collection now contains the following documents:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "Borough" : "Manhattan", "violations" : 3 },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "Borough" : "Queens", "violations" : "2" },
{ "_id" : 3, "name" : "Empire State Pub", "Borough" : "Brooklyn", "violations" : "4" },
{ "_id" : 4, "name" : "Pizza Rat's Pizzaria", "Borough" : "Manhattan", "violations" : "8" }
```

Replace with Write Concern Given a three member replica set, the following operation specifies a `w` of `majority` and `wtimeout` of `100`:

```
try {
   db.restaurant.replaceOne(
       { "name" : "Pizza Rat's Pizzaria" },
       { "name" : "Pizza Rat's Pub", "Borough" : "Manhattan", "violations" : 3 },
       { w: "majority", wtimeout: 100 }
   );
}
catch (e) {
   print(e);
}
```

If the acknowledgement takes longer than the `wtimeout` limit, the following exception is thrown:

```
try {
   WriteConcernError({
      "code" : 64,
      "errInfo" : {
         "wtimeout" : true
      },
      "errmsg" : "waiting for replication timed out"
   });
}
catch {
   print(e);
}
```

db.collection.remove()

On this page

- Definition (page 101)
- Behavior (page 102)
- Examples (page 102)
- WriteResult (page 103)

Definition

`db.collection.remove()`

Removes documents from a collection.

The `db.collection.remove()` (page 101) method can have one of two syntaxes. The `remove()` (page 101) 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 527). 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 Optional. 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 Optional. A document expressing the write concern. Omit to use the default write concern. See *Write Concern* (page 102).

New in version 2.6.

Changed in version 2.6: The remove() (page 101) returns an object that contains the status of the operation.

Returns A *WriteResult* (page 103) object that contains the status of the operation.

Behavior

Write Concern Changed in version 2.6.

The remove() (page 101) method uses the delete (page 345) 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 101) 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 61) 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 629) 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 103) for an example.

Capped Collections You cannot use the remove() (page 101) method with a *capped collection*.

Sharded Collections All remove() (page 101) 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 101) 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 101) method.

Remove All Documents from a Collection To remove all documents in a collection, call the remove (page 101) 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 45) method.

To remove all documents from a collection, it may be more efficient to use the drop() (page 45) 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 101) 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 voting replica set members or the method times out after 5 seconds.

Changed in version 3.0: In previous versions, `majority` referred to the majority of all members of the replica set.

```
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 101) 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 101) returns a `WriteResult` (page 289) object that contains the status of the operation. Upon success, the `WriteResult` (page 289) object contains information on the number of documents removed:

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

See also:

`WriteResult.nRemoved` (page 290)

Write Concern Errors If the `remove()` (page 101) method encounters write concern errors, the results include the `WriteResult.writeConcernError` (page 290) field:

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

See also:

`WriteResult.hasWriteConcernError()` (page 291)

Errors Unrelated to Write Concern If the remove() (page 101) method encounters a non-write concern error, the results include WriteResult.writeError (page 290) field:

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

See also:

WriteResult.hasWriteError() (page 290)

db.collection.renameCollection()

On this page

- Definition (page 104)
- Example (page 104)
- Limitations (page 104)

Definition

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

Renames a collection. Provides a wrapper for the renameCollection (page 431) *database command*.

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

param boolean dropTarget Optional. If `true`, mongod (page 770) drops the *target* of renameCollection (page 431) prior to renaming the collection. The default value is `false`.

Example Call the db.collection.renameCollection() (page 104) 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 104) cannot move a collection between databases. Use renameCollection (page 431) for these rename operations.
- db.collection.renameCollection() (page 104) is not supported on sharded collections.

The db.collection.renameCollection() (page 104) method operates within a collection by changing the metadata associated with a given collection.

Refer to the documentation renameCollection (page 431) for additional warnings and messages.

Warning: The `db.collection.renameCollection()` (page 104) method and `renameCollection` (page 431) command will invalidate open cursors which interrupts queries that are currently returning data.

db.collection.save()

On this page

- Definition (page 105)
- Behavior (page 105)
- Examples (page 106)
- WriteResult (page 107)

Definition

`db.collection.save()`

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

The `save()` (page 105) 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 Optional. A document expressing the `write concern`. Omit to use the default write concern. See *Write Concern* (page 105).

New in version 2.6.

Changed in version 2.6: The `save()` (page 105) returns an object that contains the status of the operation.

Returns A *WriteResult* (page 107) object that contains the status of the operation.

Behavior

Write Concern Changed in version 2.6.

The `save()` (page 105) method uses either the `insert` (page 337) or the `update` (page 340) 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 105) method calls the `insert()` (page 79) method. During the operation, the `mongo` (page 803) shell will create an `https://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 770) will add the _id field and generate the ObjectId.

Update If the document contains an *_id* field, then the save() (page 105) method is equivalent to an update with the *upsert option* (page 118) 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 105) 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, the shell will create the _id field with a unique https://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 105) 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 118) 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 105) 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 voting replica set members or the method times out after 5 seconds.

Changed in version 3.0: In previous versions, majority referred to the majority of all members of the replica set.

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

WriteResult Changed in version 2.6.

The `save()` (page 105) returns a `WriteResult` (page 289) object that contains the status of the insert or update operation. See *WriteResult for insert* (page 81) and *WriteResult for update* (page 123) for details.

db.collection.stats()

On this page

- Definition (page 107)
- Behavior (page 108)
- Examples (page 108)

Definition

`db.collection.`**`stats`**`(`*scale | options*`)`

Returns statistics about the collection. The method includes the following parameters:

param number scale Optional. 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`.

Changed in version 3.0: Legacy parameter format. Mutually exclusive with `options` as a document.

param document options Optional. Alternative to `scale` parameter. Use the `options` document to specify options, including `scale`.

New in version 3.0.

The `options` document can contain the following fields and values:

field number scale Optional. 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`.

New in version 3.0.

field boolean indexDetails Optional. If `true`, `db.collection.stats()` (page 107) returns `index details` (page 477) in addition to the collection stats.

Only works for *WiredTiger* storage engine.

Defaults to `false`.

New in version 3.0.

field document indexDetailsKey Optional. If `indexDetails` is `true`, you can use `indexDetailsKey` to filter index details by specifying the index key specification. Only the index that exactly matches `indexDetailsKey` will be returned.

If no match is found, `indexDetails` (page 477) will display statistics for all indexes.

Use getIndexes() (page 73) to discover index keys. You cannot use indexDetailsKey with indexDetailsName.

New in version 3.0.

field string indexDetailsName Optional. If indexDetails is true, you can use indexDetailsName to filter index details by specifying the index name. Only the index name that exactly matches indexDetailsName will be returned.

If no match is found, indexDetails (page 477) will display statistics for all indexes.

Use getIndexes() (page 73) to discover index names. You cannot use indexDetailsName with indexDetailsField.

New in version 3.0.

Returns A *document* that contains statistics on the specified collection. See collStats (page 473) for a breakdown of the returned statistics.

The db.collection.stats() (page 107) method provides a wrapper around the database command collStats (page 473).

Behavior This method returns a JSON document with statistics related to the current mongod (page 770) instance. Unless otherwise specified by the key name, values related to size are displayed in bytes and can be overridden by scale.

Note: The scale factor rounds values to whole numbers.

Depending on the storage engine, the data returned may differ. For details on the fields, see *output details* (page 475).

Index Filter Behavior Filtering on indexDetails using either indexDetailsKey or indexDetailsName will only return a single matching index. If no exact match is found, indexDetails will show information on all indexes for the collection.

The indexDetailsKey field takes a document of the following form:

```
{ '<string>' : <value>, '<string>' : <value>, ... }
```

Where <string>> is the field that is indexed and <value> is either the direction of the index, or the special index type such as text or 2dsphere. See https://docs.mongodb.org/manual/core/index-types/ for the full list of index types.

Unexpected Shutdown and Count For MongoDB instances using the WiredTiger storage engine, after an unclean shutdown, statistics on size and count may off by up to 1000 documents as reported by collStats (page 473), dbStats (page 481), count (page 307). To restore the correct statistics for the collection, run validate (page 485) on the collection.

Examples

Note: You can find the collection data used for these examples in our Getting Started Guide[4]

Basic Stats Lookup The following operation returns stats on the restaurants collection:

[4]https://docs.mongodb.org/getting-started/shell/import-data/

```
db.restaurants.stats()
```

The operation returns:

```
{
    "ns" : "guidebook.restaurants",
   "count" : 25359,
   "size" : 10630398,
   "avgObjSize" : 419,
   "storageSize" : 4104192
   "capped" : false,
   "wiredTiger" : {
         "metadata" : {
            "formatVersion" : 1
         },
         "creationString" : "allocation_size=4KB,app_metadata=(formatVersion=1),block_allocation=best
         "type" : "file",
         "uri" : "statistics:table:collection-2-7253336746667145592",
         "LSM" : {
            "bloom filters in the LSM tree" : 0,
            "bloom filter false positives" : 0,
            "bloom filter hits" : 0,
            "bloom filter misses" : 0,
            "bloom filter pages evicted from cache" : 0,
            "bloom filter pages read into cache" : 0,
            "total size of bloom filters" : 0,
            "sleep for LSM checkpoint throttle" : 0,
            "chunks in the LSM tree" : 0,
            "highest merge generation in the LSM tree" : 0,
            "queries that could have benefited from a Bloom filter that did not exist" : 0,
            "sleep for LSM merge throttle" : 0
         },
         "block-manager" : {
            "file allocation unit size" : 4096,
            "blocks allocated" : 338,
            "checkpoint size" : 4096000,
            "allocations requiring file extension" : 338,
            "blocks freed" : 0,
            "file magic number" : 120897,
            "file major version number" : 1,
            "minor version number" : 0,
            "file bytes available for reuse" : 0,
            "file size in bytes" : 4104192
         },
         "btree" : {
            "btree checkpoint generation" : 15,
            "column-store variable-size deleted values" : 0,
            "column-store fixed-size leaf pages" : 0,
            "column-store internal pages" : 0,
            "column-store variable-size leaf pages" : 0,
            "pages rewritten by compaction" : 0,
            "number of key/value pairs" : 0,
            "fixed-record size" : 0,
            "maximum tree depth" : 3,
            "maximum internal page key size" : 368,
            "maximum internal page size" : 4096,
            "maximum leaf page key size" : 3276,
            "maximum leaf page size" : 32768,
            "maximum leaf page value size" : 67108864,
            "overflow pages" : 0,
            "row-store internal pages" : 0,
            "row-store leaf pages" : 0
         },
         "cache" : {
            "bytes read into cache" : 9309503,
            "bytes written from cache" : 10817368,
            "checkpoint blocked page eviction" : 0,
            "unmodified pages evicted" : 0,
            "page split during eviction deepened the tree" : 0,
            "modified pages evicted" : 1,
            "data source pages selected for eviction unable to be evicted" : 0,
            "hazard pointer blocked page eviction" : 0,
            "internal pages evicted" : 0,
            "pages split during eviction" : 1,
            "in-memory page splits" : 1,
            "overflow values cached in memory" : 0,
            "pages read into cache" : 287,
            "overflow pages read into cache" : 0,
            "pages written from cache" : 337
         },
         "compression" : {
            "raw compression call failed, no additional data available" : 0,
            "raw compression call failed, additional data available" : 0,
            "raw compression call succeeded" : 0,
            "compressed pages read" : 287,
            "compressed pages written" : 333,
            "page written failed to compress" : 0,
            "page written was too small to compress" : 4
         },
         "cursor" : {
            "create calls" : 1,
            "insert calls" : 25359,
            "bulk-loaded cursor-insert calls" : 0,
            "cursor-insert key and value bytes inserted" : 10697901,
            "next calls" : 76085,
            "prev calls" : 1,
            "remove calls" : 0,
            "cursor-remove key bytes removed" : 0,
            "reset calls" : 25959,
            "restarted searches" : 0,
            "search calls" : 0,
            "search near calls" : 594,
            "update calls" : 0,
            "cursor-update value bytes updated" : 0
         },
         "reconciliation" : {
            "dictionary matches" : 0,
            "internal page multi-block writes" : 1,
            "leaf page multi-block writes" : 2,
            "maximum blocks required for a page" : 47,
            "internal-page overflow keys" : 0,
            "leaf-page overflow keys" : 0,
            "overflow values written" : 0,
            "pages deleted" : 0,
            "page checksum matches" : 0,
            "page reconciliation calls" : 4,
            "page reconciliation calls for eviction" : 1,
            "leaf page key bytes discarded using prefix compression" : 0,
            "internal page key bytes discarded using suffix compression" : 333
         },
         "session" : {
            "object compaction" : 0,
            "open cursor count" : 1
         },
         "transaction" : {
            "update conflicts" : 0
         }
      },
      "nindexes" : 4,
      "totalIndexSize" : 626688,
      "indexSizes" : {
         "_id_" : 217088,
         "borough_1_cuisine_1" : 139264,
         "cuisine_1" : 131072,
         "borough_1_address.zipcode_1" : 139264
      },
      "ok" : 1
 }
```

As stats was not give a scale parameter, all size values are in `bytes`.

Stats Lookup With Scale The following operation changes the scale of data from `bytes` to `kilobytes` by specifying a `scale` of `1024`:

```
db.restaurants.stats( { scale : 1024 } )
```

The operation returns:

```
{
   "ns" : "guidebook.restaurants",
   "count" : 25359,
   "size" : 10381,
   "avgObjSize" : 419,
   "storageSize" : 4008,
   "capped" : false,
   "wiredTiger" : {
      ...
   },
   "nindexes" : 4,
   "totalIndexSize" : 612,
   "indexSizes" : {
      "_id_" : 212,
      "borough_1_cuisine_1" : 136,
      "cuisine_1" : 128,
      "borough_1_address.zipcode_1" : 136
   },
   "ok" : 1
}
```

Statistics Lookup With Index Details The following operation creates an `indexDetails` document that contains information related to each of the indexes within the collection:

```
db.restaurant.stats( { indexDetails : true } )
```

The operation returns:

```
{
   "ns" : "guidebook.restaurants",
   "count" : 25359,
   "size" : 10630398,
   "avgObjSize" : 419,
   "storageSize" : 4104192,
   "capped" : false,
   "wiredTiger" : {
      ...
   },
   "nindexes" : 4,
   "indexDetails" : {
      "_id_" : {
         "metadata" : {
            "formatVersion" : 6,
            "infoObj" : "{ \"v\" : 1, \"key\" : { \"_id\" : 1 }, \"name\" : \"_id_\", \"ns\" : \"blog
         },
         "creationString" : "allocation_size=4KB,app_metadata=(formatVersion=6,infoObj={ \"v\" : 1, \
         "type" : "file",
         "uri" : "statistics:table:index-3-7253336746667145592",
         "LSM" : {
            ...
         },
         "block-manager" : {
            ...
         },
         "btree" : {
            ...
         },
         "cache" : {
            ...
         },
         "compression" : {
            ...
         },
         "cursor" : {
            ...
         },
         "reconciliation" : {
            ...
         },
         "session" : {
            ...
         },
         "transaction" : {
            ...
         }
      },
      "borough_1_cuisine_1" : {
         "metadata" : {
            "formatVersion" : 6,
            "infoObj" : "{ \"v\" : 1, \"key\" : { \"borough\" : 1, \"cuisine\" : 1 }, \"name\" : \"bo
         },
         "creationString" : "allocation_size=4KB,app_metadata=(formatVersion=6,infoObj={ \"v\" : 1, \
         "type" : "file",
         "uri" : "statistics:table:index-4-7253336746667145592",
         "LSM" : {
            ...
         },
         "block-manager" : {
            ...
         },
         "btree" : {
            ...
         },
         "cache" : {
            ...
         },
         "compression" : {
            ...
         },
         "cursor" : {
            ...
         },
         "reconciliation" : {
            ...
         },
         "session" : {
            "object compaction" : 0,
            "open cursor count" : 0
         },
         "transaction" : {
            "update conflicts" : 0
         }
      },
      "cuisine_1" : {
         "metadata" : {
            "formatVersion" : 6,
            "infoObj" : "{ \"v\" : 1, \"key\" : { \"cuisine\" : 1 }, \"name\" : \"cuisine_1\", \"ns\
         },
         "creationString" : "allocation_size=4KB,app_metadata=(formatVersion=6,infoObj={ \"v\" : 1, \
         "type" : "file",
         "uri" : "statistics:table:index-5-7253336746667145592",
         "LSM" : {
            ...
         },
         "block-manager" : {
            ...
         },
         "btree" : {
            ...
         },
         "cache" : {
            ...
         },
         "compression" : {
            ...
         },
         "cursor" : {
            ...
         },
         "reconciliation" : {
            ...
         },
         "session" : {
            ...
         },
         "transaction" : {
            ...
         }
      },
      "borough_1_address.zipcode_1" : {
         "metadata" : {
            "formatVersion" : 6,
            "infoObj" : "{ \"v\" : 1, \"key\" : { \"borough\" : 1, \"address.zipcode\" : 1 }, \"name\
         },
         "creationString" : "allocation_size=4KB,app_metadata=(formatVersion=6,infoObj={ \"v\" : 1, \
         "type" : "file",
         "uri" : "statistics:table:index-6-7253336746667145592",
         "LSM" : {
            ...
         },
         "block-manager" : {
            ...
         },
         "btree" : {
            ...
         },
         "cache" : {
            ...
         },
         "compression" : {
            ...
         },
         "cursor" : {
            ...
         },
         "reconciliation" : {
            ...
         },
         "session" : {
            ...
         },
         "transaction" : {
            ...
         }
      }
   },
   "totalIndexSize" : 626688,
   "indexSizes" : {
      "_id_" : 217088,
      "borough_1_cuisine_1" : 139264,
      "cuisine_1" : 131072,
      "borough_1_address.zipcode_1" : 139264
   },
   "ok" : 1
}
```

Statistics Lookup With Filtered Index Details To filter the indexes in the `indexDetails` (page 477) field, you can either specify the index keys using the `indexDetailsKey` option or specify the index name using the `indexDetailsName`. To discover index keys and names for the collection, use `db.collection.getIndexes()` (page 73).

Given the following index:

```
{
   "ns" : "guidebook.restaurants",
   "v" : 1,
   "key" : {
      "borough" : 1,
      "cuisine" : 1
   },
   "name" : "borough_1_cuisine_1"
}
```

The following operation filters the `indexDetails` document to a single index as defined by the `indexDetailsKey` document.

```
db.restaurants.stats(
   {
      'indexDetails' : true,
      'indexDetailsKey' :
      {
         'borough' : 1,
         'cuisine' : 1
      }
   }
)
```

The following operation filters the `indexDetails` document to a single index as defined by the `indexDetailsName` document.

```
db.restaurants.stats(
   {
      'indexDetails' : true,
      'indexDetailsName' : 'borough_1_cuisine_1'
   }
)
```

Both operations will return the same output:

```
{
   "ns" : "blogs.restaurants",
   "count" : 25359,
   "size" : 10630398,
   "avgObjSize" : 419,
   "storageSize" : 4104192,
   "capped" : false,
   "wiredTiger" : {
      ...
   },
   "nindexes" : 4,
   "indexDetails" : {
      "borough_1_cuisine_1" : {
         "metadata" : {
            "formatVersion" : 6,
            "infoObj" : "{ \"v\" : 1, \"key\" : { \"borough\" : 1, \"cuisine\" : 1 }, \"name\" : \"bo
         },
         "creationString" : "allocation_size=4KB,app_metadata=(formatVersion=6,infoObj={ \"v\" : 1, \
         "type" : "file",
         "uri" : "statistics:table:index-4-7253336746667145592",
         "LSM" : {
            ...
         },
         "block-manager" : {
            ...
         },
         "btree" : {
            ...
         },
         "cache" : {
            ...
         },
         "compression" : {
            ...
         },
         "cursor" : {
            ...
         },
         "reconciliation" : {
            ...
         },
         "session" : {
            ...
         },
         "transaction" : {
            ...
         }
      }
   },
   "totalIndexSize" : 626688,
   "indexSizes" : {
      "_id_" : 217088,
      "borough_1_cuisine_1" : 139264,
      "cuisine_1" : 131072,
      "borough_1_address.zipcode_1" : 139264
   },
   "ok" : 1
}
```

For explanation of the output, see *output details* (page 475).

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 476) field of the `collStats` (page 473) (i.e. `db.collection.stats()` (page 107)) output.

db.collection.totalSize()

`db.collection.totalSize()`

Returns The total size in bytes of the data in the collection plus the size of every indexes on the collection.

db.collection.totalIndexSize()

`db.collection.totalIndexSize()`

Returns The total size of all indexes for the collection. This method provides a wrapper around the `totalIndexSize` (page 476) output of the `collStats` (page 473) (i.e. `db.collection.stats()` (page 107)) operation.

db.collection.update()

On this page

- Definition (page 117)
- Behavior (page 118)
- Examples (page 120)
- WriteResult (page 123)
- Additional Resources (page 124)

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 118).

By default, the `update()` (page 117) method updates a **single** document. Set the *Multi Parameter* (page 120) to update all documents that match the query criteria.

The `update()` (page 117) method has the following form:

Changed in version 2.6.

```
db.collection.update(
   <query>,
   <update>,
   {
     upsert: <boolean>,
     multi: <boolean>,
     writeConcern: <document>
   }
)
```

The `update()` (page 117) method takes the following parameters:

param document query The selection criteria for the update. The same *query selectors* (page 527) as in the `find()` (page 51) method are available.

Changed in version 3.0: When you execute an `update()` (page 117) with `upsert: true` and the query matches no existing document, MongoDB will refuse to insert a new document if the query specifies conditions on the `_id` field using *dot notation*.

For more information and an example, see *upsert:true with a Dotted _id Query* (page 119).

param document update The modifications to apply. For details see *Update Parameter* (page 118).

param boolean upsert Optional. 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 Optional. 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 120).

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern. See *Write Concern* (page 118).

New in version 2.6.

Changed in version 2.6: The `update()` (page 117) method returns an object that contains the status of the operation.

Returns A *WriteResult* (page 123) object that contains the status of the operation.

Behavior

Write Concern Changed in version 2.6.

The `update()` (page 117) method uses the `update` (page 340) 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 122) for an example.

Update Parameter The `update()` (page 117) 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 595) modifiers, such as those using the `$set` (page 600) modifier, then:

- The `<update>` document must contain *only update operator* (page 595) expressions.
- The `update()` (page 117) 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 117) method *replaces* the matching document with the `<update>` document. The `update()` (page 117) method *does not* replace the `_id` value. For an example, see *Replace All Fields* (page 121).
- `update()` (page 117) *cannot update multiple documents* (page 120).

Upsert Option

Upsert Behavior If `upsert` is `true` and no document matches the query criteria, `update()` (page 117) 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 595) 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 117) performs an update.

See also:

`$setOnInsert` (page 599)

Use Unique Indexes

> **Warning:** To avoid inserting the same document more than once, only use `upsert: true` if the `query` f
> 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 117) 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 117) 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.

`upsert:true` with a Dotted `_id` Query When you execute an `update()` (page 117) with `upsert: true` and the query matches no existing document, MongoDB will refuse to insert a new document if the query specifies conditions on the `_id` field using *dot notation*.

This restriction ensures that the order of fields embedded in the `_id` document is well-defined and not bound to the order specified in the query

If you attempt to insert a document in this way, MongoDB will raise an error.

For example, consider the following update operation. Since the update operation specifies `upsert:true` and the query specifies conditions on the `_id` field using dot notation, then the update will result in an error when constructing the document to insert.

```
db.collection.update( { "_id.name": "Robert Frost", "_id.uid": 0 },
   { "categories": ["poet", "playwright"] },
   { upsert: true } )
```

Multi Parameter If `multi` is set to `true`, the `update()` (page 117) 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 629) 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 118) document contains *only* `field:value` expressions, then `update()` (page 117) *cannot* update multiple documents.

For an example, see *Update Multiple Documents* (page 122).

Sharded Collections All `update()` (page 117) 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 117) operations specifying `multi: false` in a sharded collection without the *shard key* *or* the `_id` field return an error.

See also:

`findAndModify()` (page 57)

Examples

Update Specific Fields To update specific fields in a document, use *update operators* (page 595) 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 `$inc` (page 595) operator to increment the `stock` field; and
- the `$set` (page 600) 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:

`$set` (page 600), `$inc` (page 595), *Update Operators* (page 594), *dot notation*

Remove Fields The following operation uses the `$unset` (page 602) operator to remove the `tags` field:

```
db.books.update( { _id: 1 }, { $unset: { tags: 1 } } )
```

See also:

`$unset` (page 602), `$rename` (page 598), *Update Operators* (page 594)

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 118) option to `true` so that `update()` (page 117) 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 118).

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 `$set` (page 600) operator will add the field with the specified value. See `$set` (page 600) 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 voting replica set members or the method times out after 5 seconds.

Changed in version 3.0: In previous versions, `majority` referred to the majority of all members of the replica set.

```
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 118).

WriteResult   Changed in version 2.6.

Successful Results The `update()` (page 117) method returns a `WriteResult` (page 289) object that contains the status of the operation. Upon success, the `WriteResult` (page 289) 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 289), `WriteResult.nUpserted` (page 289), `WriteResult.nModified` (page 289)

Write Concern Errors If the `update()` (page 117) method encounters write concern errors, the results include the `WriteResult.writeConcernError` (page 290) 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 291)

Errors Unrelated to Write Concern If the `update()` (page 117) method encounters a non-write concern error, the results include the `WriteResult.writeError` (page 290) 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 290)

Additional Resources

- Quick Reference Cards[5]

db.collection.updateOne()

[5]https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs

On this page

- Definition (page 125)
- Behavior (page 126)
- Examples (page 126)

Definition

`db.collection.`**`updateOne`**(*filter*, *update*, *options*)

New in version 3.2.

Updates a single document within the collection based on the filter.

The `updateOne()` (page 125) method has the following form:

```
db.collection.updateOne(
   <filter>,
   <update>,
   {
     upsert: <boolean>,
     writeConcern: <document>
   }
)
```

The `updateOne()` (page 125) method takes the following parameters:

param document filter The selection criteria for the update. The same *query selectors* (page 527) as in the `find()` (page 51) method are available.

Specify an empty document `{ }` to update the first document returned in the collection.

param document update The modifications to apply.

Use *Update Operators* (page 594) such as `$set` (page 600), `$unset` (page 602), or `$rename` (page 598).

Using the *update()* (page 118) pattern of `field: value` for the `update` parameter throws an error.

param boolean upsert Optional. When `true`, if no documents match the `filter`, a new document is created using the equality comparisons in `filter` with the modifications from `update`.

Comparison (page 527) operations from the `filter` will not be included in the new document.

If the `filter` only has comparison operations, then only the modifications from the `update` will be applied to the new document.

See *Update with Upsert* (page 126)

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

Returns

A document containing:

- A boolean `acknowledged` as `true` if the operation ran with *write concern* or `false` if write concern was disabled
- `matchedCount` containing the number of matched documents

- `modifiedCount` containing the number of modified documents
- `upsertedId` containing the `_id` for the upserted document

Behavior `updateOne()` (page 125) updates the first matching document in the collection that matches the `filter`, using the `update` instructions to apply modifications.

If `upsert: true` and no documents match the `filter`, `updateOne()` (page 125) creates a new document based on the `filter` criteria and `update` modifications. See *Update with Upsert* (page 126).

Capped Collection `updateOne()` (page 125) throws a `WriteError` exception if the `update` criteria increases the size of the first matching document in a *capped collection*.

Explainability `updateOne()` (page 125) is not compatible with `db.collection.explain()` (page 48).

Use `update()` (page 117) instead.

Examples

Update The `restaurant` collection contains the following documents:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "Borough" : "Manhattan" },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "Borough" : "Queens", "violations" : "2" },
{ "_id" : 3, "name" : "Empire State Pub", "Borough" : "Brooklyn", "violations" : "0" }
```

The following operation updates a single document where `name: "Central Perk Cafe"` with the `violations` field:

```
try {
   db.inventory.updateOne(
      { "name" : "Central Perk Cafe" },
      { $set: { "violations" : 3 } }
   );
}
catch (e) {
   print(e);
}
```

The operation returns:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

If no matches were found, the operation instead returns:

```
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }
```

Setting `upsert: true` would insert the document if no match was found. See *Update with Upsert* (page 126)

Update with Upsert The `restaurant` collection contains the following documents:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "Borough" : "Manhattan", "violations" : 3 },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "Borough" : "Queens", "violations" : "2" },
{ "_id" : 3, "name" : "Empire State Pub", "Borough" : "Brooklyn", "violations" : "0" }
```

The following operation attempts to update the document with `name : "Pizza Rat's Pizzaria"`, while `upsert: true`:

```
try {
   db.restaurant.updateOne(
      { "name" : "Pizza Rat's Pizzaria" },
      { $set: {"_id" : 4, "violations" : "7", "borough" : "Manhattan" } },
      { upsert: true }
   );
}
catch (e) {
   print(e);
}
```

Since `upsert:true` the document is `inserted` based on the `filter` and `update` criteria. The operation returns:

```
{
   "acknowledged" : true,
   "matchedCount" : 0,
   "modifiedCount" : 0,
   "upsertedId" : 4
}
```

The collection now contains the following documents:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "Borough" : "Manhattan", "violations" : 3 },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "Borough" : "Queens", "violations" : "2" },
{ "_id" : 3, "name" : "Empire State Pub", "Borough" : "Brooklyn", "violations" : "4" },
{ "_id" : 4, "name" : "Pizza Rat's Pizzaria", "Borough" : "Manhattan", "violations" : "7" }
```

The `name` field was filled in using the `filter` criteria, while the `update` operators were used to create the rest of the document.

The following operation updates the first document with `violations` that are greater than `10`:

```
try {
   db.restaurant.updateOne(
      { "violations" : { $gt: 10} },
      { $set: { "Closed" : true } },
      { upsert: true }
   );
}
catch {
   print(e);
}
```

The operation returns:

```
{
   "acknowledged" : true,
   "matchedCount" : 0,
   "modifiedCount" : 0,
   "upsertedId" : ObjectId("56310c3c0c5cbb6031cafaea")
}
```

The collection now contains the following documents:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "Borough" : "Manhattan", "violations" : 3 },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "Borough" : "Queens", "violations" : "2" },
{ "_id" : 3, "name" : "Empire State Pub", "Borough" : "Brooklyn", "violations" : "4" },
{ "_id" : 4, "name" : "Pizza Rat's Pizzaria", "Borough" : "Manhattan", "grade" : "7" }
{ "_id" : ObjectId("56310c3c0c5cbb6031cafaea"), "Closed" : true }
```

Since no documents matched the filter, and `upsert` was `true`, `updateOne` (page 125) inserted the document with a generated `_id` and the `update` criteria only.

Update with Write Concern Given a three member replica set, the following operation specifies a `w` of `majority`, `wtimeout` of `100`:

```
try {
   db.restaurant.updateOne(
       { "name" : "Pizza Rat's Pizzaria" },
       { $inc: { "violations" : 3}, $set: { "Closed" : true } },
       { w: "majority", wtimeout: 100 }
   );
}
catch {
   print(e);
}
```

If the primary and at least one secondary acknowledge each write operation within 100 milliseconds, it returns:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

If the acknowledgement takes longer than the `wtimeout` limit, the following exception is thrown:

```
WriteConcernError({
   "code" : 64,
   "errInfo" : {
      "wtimeout" : true
   },
   "errmsg" : "waiting for replication timed out"
}) :
```

See also:

To update multiple documents, see `db.collection.updateMany()` (page 128).

db.collection.updateMany()

On this page

- Definition (page 128)
- Behavior (page 129)
- Examples (page 130)

Definition

`db.collection.`**`updateMany`**`(`*filter*, *update*, *options*`)`

New in version 3.2.

Updates multiple documents within the collection based on the filter.

The `updateMany()` (page 128) method has the following form:

```
db.collection.updateMany(
   <filter>,
   <update>,
   {
     upsert: <boolean>,
     writeConcern: <document>
   }
)
```

The `updateMany()` (page 128) method takes the following parameters:

param document filter The selection criteria for the update. The same *query selectors* (page 527) as in the `find()` (page 51) method are available.

Specify an empty document `{ }` to update all documents in the collection.

param document update The modifications to apply.

Use *Update Operators* (page 594) such as `$set` (page 600), `$unset` (page 602), or `$rename` (page 598).

Using the *update()* (page 118) pattern of `field: value` for the `update` parameter throws an error.

param boolean upsert Optional. When `true`, if no documents match the `filter`, a new document is created using the equality comparisons in `filter` with the modifications from `update`.

Comparison (page 527) operations from the `filter` will not be included in the new document.

If the `filter` only has comparison operations, then only the modifications from the `update` will be applied to the new document.

See *Update Multiple Documents with Upsert* (page 130)

param document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

Returns

A document containing:

- A boolean `acknowledged` as `true` if the operation ran with *write concern* or `false` if write concern was disabled
- `matchedCount` containing the number of matched documents
- `modifiedCount` containing the number of modified documents
- `upsertedId` containing the `_id` for the upserted document

Behavior `updateMany()` (page 128) updates all matching documents in the collection that match the `filter`, using the `update` criteria to apply modifications.

If `upsert: true` and no documents match the `filter`, `updateMany()` (page 128) creates a new document based on the `filter` and `update` parameters. See *Update Multiple Documents with Upsert* (page 130).

Capped Collections `updateMany()` (page 128) throws a `WriteError` if the `update` criteria increases the size of any matching document in a *capped collection*.

Explainability `updateMany()` (page 128) is not compatible with `db.collection.explain()` (page 48).

Use `update()` (page 117) instead.

Examples

Update Multiple Documents The `restaurant` collection contains the following documents:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "violations" : 3 },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "violations" : "2" },
{ "_id" : 3, "name" : "Empire State Sub", "violations" : "5" },
{ "_id" : 4, "name" : "Pizza Rat's Pizzaria", "violations" : "8" }
```

The following operation updates all documents where `violations` are greater than `4` and `$set` (page 600) a flag for review:

```
try {
   db.inventory.updateMany(
      { violations: { $gt: 4 } },
      { $set: { "Review" : true } }
   );
}
catch (e) {
   print(e);
}
```

The operation returns:

```
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
```

The collection now contains the following documents:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "violations" : 3 },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "violations" : "2" },
{ "_id" : 3, "name" : "Empire State Sub", "violations" : "5", "Review" : true },
{ "_id" : 4, "name" : "Pizza Rat's Pizzaria", "violations" : "8", "Review" : true }
```

If no matches were found, the operation instead returns:

```
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }
```

Setting `upsert: true` would insert a document if no match was found.

Update Multiple Documents with Upsert The `inspectors` collection contains the following documents:

```
{ "_id" : 92412, "inspector" : "F. Drebin", "Sector" : 1, "Patrolling" : true },
{ "_id" : 92413, "inspector" : "J. Clouseau", "Sector" : 2, "Patrolling" : false },
{ "_id" : 92414, "inspector" : "J. Clouseau", "Sector" : 3, "Patrolling" : true },
{ "_id" : 92415, "inspector" : "R. Coltrane", "Sector" : 3, "Patrolling" : false }
```

The following operation updates all documents with `violations` that are greater than `10`:

```
try {
   db.inspectors.updateMany(
      { "inspector" : "J. Clouseau", "Sector" : 4 },
      { $set: { "Patrolling" : false } },
      { upsert: true }
   );
}
catch (e) {
   print(e);
}
```

The operation returns:

```
{
   "acknowledged" : true,
   "matchedCount" : 0,
   "modifiedCount" : 0,
   "upsertedId" : 92416
}
```

The collection now contains the following documents:

```
{ "_id" : 92412, "inspector" : "F. Drebin", "Sector" : 1, "Patrolling" : true },
{ "_id" : 92413, "inspector" : "J. Clouseau", "Sector" : 2, "Patrolling" : false },
{ "_id" : 92414, "inspector" : "J. Clouseau", "Sector" : 3, "Patrolling" : true },
{ "_id" : 92415, "inspector" : "R. Coltrane", "Sector" : 3, "Patrolling" : false },
{ "_id" : 92416, "inspector" : "J. Clouseau", "Sector" : 4, "Patrolling" : false }
```

No documents in the collection matched the `filter`, so a new document was created.

The following operation updates all documents with `Sector` greater than 4 for `inspector : "R. Coltrane"`:

```
try {
   db.inspectors.updateMany(
      { "Sector" : { $gt : 4 }, "inspector" : "R. Coltrane" },
      { $set: { "Patrolling" : false } },
      { upsert: true }
    );
}
catch (e) {
   print(e);
}
```

The operation returns:

```
{
   "acknowledged" : true,
   "matchedCount" : 0,
   "modifiedCount" : 0,
   "upsertedId" : 92417
}
```

The collection now contains the following documents:

```
{ "_id" : 92412, "inspector" : "F. Drebin", "Sector" : 1, "Patrolling" : true },
{ "_id" : 92413, "inspector" : "J. Clouseau", "Sector" : 2, "Patrolling" : false },
{ "_id" : 92414, "inspector" : "J. Clouseau", "Sector" : 3, "Patrolling" : true },
{ "_id" : 92415, "inspector" : "R. Coltrane", "Sector" : 3, "Patrolling" : false },
{ "_id" : 92416, "inspector" : "J. Clouseau", "Sector" : 4, "Patrolling" : false },
{ "_id" : 92417, "inspector" : "R. Coltrane", "Patrolling" : false }
```

Since no documents matched the filter, and `upsert` was `true`, `updateMany` (page 128) inserted the document with a generated `_id`, the equality operator from `filter`, and the `update` modifiers.

Update with Write Concern Given a three member replica set, the following operation specifies a `w` of `majority` and `wtimeout` of `100`:

```
try {
   db.restaurant.updateMany(
       { "name" : "Pizza Rat's Pizzaria" },
       { $inc: { "violations" : 3}, $set: { "Closed" : true } },
       { w: "majority", wtimeout: 100 }
   );
}
catch (e) {
   print(e);
}
```

If the acknowledgement takes longer than the `wtimeout` limit, the following exception is thrown:

```
WriteConcernError({
   "code" : 64,
   "errInfo" : {
      "wtimeout" : true
   },
   "errmsg" : "waiting for replication timed out"
}) :
undefined
```

The `wtimeout` error only indicates that the operation did not complete on time. The write operation itself can still succeed outside of the set time limit.

db.collection.validate()

On this page

- Description (page 132)

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 132) method has the following parameter:

param boolean full Optional. 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 132) 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 132) method is a wrapper around the `validate` (page 485) *database command*.

See also:

validate (page 485)

2.1.2 Cursor

Cursor Methods

These methods modify the way that the underlying query is executed.

Name	Description
`cursor.batchSize()` (page 135)	Controls the number of documents MongoDB will return to the client in a single network message.
`cursor.close()` (page 135)	Close a cursor and free associated server resources.
`cursor.comment()` (page 136)	Attaches a comment to the query to allow for traceability in the logs and the system.profile collection.
`cursor.count()` (page 137)	Modifies the cursor to return the number of documents in the result set rather than the documents themselves.
`cursor.explain()` (page 140)	Reports on the query execution plan for a cursor.
`cursor.forEach()` (page 141)	Applies a JavaScript function for every document in a cursor.
`cursor.hasNext()` (page 142)	Returns true if the cursor has documents and can be iterated.
`cursor.hint()` (page 142)	Forces MongoDB to use a specific index for a query.
`cursor.itcount()` (page 143)	Computes the total number of documents in the cursor client-side by fetching and iterating the result set.
`cursor.limit()` (page 144)	Constrains the size of a cursor's result set.
`cursor.map()` (page 144)	Applies a function to each document in a cursor and collects the return values in an array.
`cursor.maxScan()` (page 145)	Specifies the maximum number of items to scan; documents for collection scans, keys for index scans.
`cursor.maxTimeMS()` (page 146)	Specifies a cumulative time limit in milliseconds for processing operations on a cursor.
`cursor.max()` (page 147)	Specifies an exclusive upper index bound for a cursor. For use with `cursor.hint()` (page 142)
`cursor.min()` (page 148)	Specifies an inclusive lower index bound for a cursor. For use with `cursor.hint()` (page 142)
`cursor.next()` (page 150)	Returns the next document in a cursor.
`cursor.noCursorTimeout()` (page 150)	Instructs the server to avoid closing a cursor automatically after a period of inactivity.
`cursor.objsLeftInBatch()` (page 151)	Returns the number of documents left in the current cursor batch.
`cursor.pretty()` (page 151)	Configures the cursor to display results in an easy-to-read format.
`cursor.readConcern()` (page 152)	Specifies a *read concern* for a `find()` (page 51) operation.
`cursor.readPref()` (page 152)	Specifies a *read preference* to a cursor to control how the client directs queries to a *replica set*.
`cursor.returnKey()` (page 153)	Modifies the cursor to return index keys rather than the documents.
`cursor.showRecordId()` (page 154)	Adds an internal storage engine ID field to each document returned by the cursor.
`cursor.size()` (page 155)	Returns a count of the documents in the cursor after applying `skip()` (page 155) and `limit()` (page 144) methods.
`cursor.skip()` (page 155)	Returns a cursor that begins returning results only after passing or skipping a number of documents.
`cursor.snapshot()` (page 156)	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 156)	Returns results ordered according to a sort specification.
`cursor.tailable()` (page 160)	Marks the cursor as tailable. Only valid for cursors over capped collections.
`cursor.toArray()` (page 161)	Returns an array that contains all documents returned by the cursor.

cursor.batchSize()

On this page

- Definition (page 135)
- Example (page 135)

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 803) shell and most `drivers` return results as if MongoDB returned a single batch.

The `batchSize()` (page 135) 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 144) method.

Example The following example sets the batch size for the results of a query (i.e. `find()` (page 51)) to `10`. The `batchSize()` (page 135) method does not change the output in the `mongo` (page 803) shell, which, by default, iterates over the first 20 documents.

```
db.inventory.find().batchSize(10)
```

cursor.close()

Definition

`cursor.close()`

Instructs the server to close a *cursor* and free associated server resources. The server will automatically close cursors that have no remaining results, as well as cursors that have been idle for a period of time and lack the `cursor.noCursorTimeout()` (page 150) option.

The `close()` (page 135) method has the following prototype form:

```
db.collection.find(<query>).close()
```

cursor.comment()

On this page

- Definition (page 136)
- Behavior (page 136)
- Example (page 136)
- Output Examples (page 136)

Definition

`cursor.comment()`

New in version 3.2.

Adds a `comment` field to the query.

`cursor.comment()` (page 136) has the following syntax:

```
cursor.comment( <string> )
```

`comment()` (page 136) has the following parameter:

param string comment The comment to apply to the query.

Behavior `comment()` (page 136) associates a comment string with the find operation. This can make it easier to track a particular query in the following diagnostic outputs:

- The `system.profile` (page 893)
- The `QUERY` (page 965) *log* (page 964) component
- `db.currentOp()` (page 171)

See *configure log verbosity* (page 966) for the `mongod` (page 770) log, the `Database Profiler tutorial`, or the `db.currentOp()` (page 171) command.

Example The following operation attaches a comment to a query on the `restaurants` collection:

```
db.restaurants.find(
   { "borough" : "Manhattan" }
).comment( "Find all Manhattan restaurants" )
```

Output Examples

`system.profile` The following is an excerpt from the `system.profile` (page 893):

```
{
   "op" : "query",
   "ns" : "guidebook.restaurant",
   "query" : {
      "find" : "restaurant",
      "filter" : {
         "borough" : "Manhattan"
      },
      "comment" : "Find all Manhattan restaurants"
   },
   ...
}
```

`mongod` *log* The following is an excerpt from the `mongod` (page 770) log. It has been formatted for readability.

Important: The verbosity level for `QUERY` (page 965) must be greater than `0`. See *Configure Log Verbosity Levels* (page 966)

```
2015-11-23T13:09:16.202-0500 I COMMAND  [conn1]
   command guidebook.restaurant command: find {
      find: "restaurant",
      filter: { "borough" : "Manhattan" },
      comment: "Find all Manhattan restaurants"
   }
   ...
```

`db.currentOp()` Suppose the following operation is currently running on a `mongod` (page 770) instance:

```
db.restaurant.find(
   { "borough" : "Manhattan" }
).comment("Find all Manhattan restaurants")
```

Running the `db.currentOp()` (page 171) command returns the following:

```
{
   "inprog" : [
      {
         "desc" : "conn5",
         "threadId" : "0x10772e000",
         "connectionId" : 5,
         "client" : "127.0.0.1:62560",
         "active" : true,
         "opid" : 2249279,
         "secs_running" : 0,
         "microsecs_running" : NumberLong(258562),
         "op" : "query",
         "ns" : "guidebook.restaurant",
         "query" : {
            "find" : "restaurant",
            "filter" : { "borough" : "Manhattan" },
            "comment" : "Find all Manhattan restaurants"
         },
         ...
      },
      ...
   ],
   "ok" : 1
}
```

cursor.count()

On this page

- Definition (page 137)
- Behavior (page 138)
- Examples (page 139)

Definition

`cursor.count()`

Counts the number of documents referenced by a cursor. Append the `count()` (page 137) method to a `find()`

(page 51) 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 142) with `count()` (page 137). See *Specify the Index to Use* (page 139) for an example.

The `count()` (page 137) method has the following prototype form:

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

The `count()` (page 137) method has the following parameter:

param boolean applySkipLimit Optional. Specifies whether to consider the effects of the `cursor.skip()` (page 155) and `cursor.limit()` (page 144) methods in the count. By default, the `count()` (page 137) method ignores the effects of the `cursor.skip()` (page 155) and `cursor.limit()` (page 144). Set `applySkipLimit` to `true` to consider the effect of these methods.

MongoDB also provides an equivalent `db.collection.count()` (page 33) as an alternative to the `db.collection.find(<query>).count()` construct.

See also:

`cursor.size()` (page 155)

Behavior

Sharded Clusters On a sharded cluster, `count()` (page 137) 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 `$group` (page 644) stage of the `db.collection.aggregate()` (page 20) method to `$sum` (page 729) 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 635) stage as well:

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

See *Perform a Count* (page 636) 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 137) 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()

On this page

- Definition (page 140)
- Behavior (page 140)
- Example (page 141)
- Output (page 141)

Definition

`cursor.`**`explain`**`(`*verbosity*`)`

Changed in version 3.0: The parameter to the method and the output format have changed in 3.0.

Provides information on the query plan for the `db.collection.find()` (page 51) method.

The `explain()` (page 140) method has the following form:

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

The `explain()` (page 140) method has the following parameter:

param string verbose Optional. 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 140), MongoDB interprets `true` as `"allPlansExecution"` and `false` as `"queryPlanner"`.

For more information on the modes, see *Verbosity Modes* (page 140).

Changed in version 3.0.

The `explain()` (page 140) method returns a document with the query plan and, optionally, the execution statistics.

Behavior

Verbosity Modes The behavior of `cursor.explain()` (page 140) and the amount of information returned depend on the `verbosity` mode.

`queryPlanner` Mode By default, `cursor.explain()` (page 140) runs in `queryPlanner` verbosity mode.

MongoDB runs the `query optimizer` to choose the winning plan for the operation under evaluation. `cursor.explain()` (page 140) returns the `queryPlanner` (page 948) 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.

`cursor.explain()` (page 140) returns the `queryPlanner` (page 948) and `executionStats` (page 949) information for the evaluated method. However, `executionStats` (page 949) 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*.

`cursor.explain()` (page 140) returns the `queryPlanner` (page 948) and `executionStats` (page 949) information for the evaluated method. The `executionStats` (page 949) includes the *completed* query execution information for the *winning plan*.

If the query optimizer considered more than one plan, `executionStats` (page 949) 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 140) 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 49).
- 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 48) for more information.

Example The following example runs `cursor.explain()` (page 140) in *"executionStats"* (page 49) verbosity mode to return the query planning and execution information for the specified `db.collection.find()` (page 51) operation:

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

Output `cursor.explain()` (page 140) operations can return information regarding:

- *queryPlanner* (page 947), which details the plan selected by the `query optimizer` and lists the rejected plans;
- *executionStats* (page 948), which details the execution of the winning plan and the rejected plans; and
- *serverInfo* (page 951), which provides information on the MongoDB instance.

The verbosity mode (i.e. `queryPlanner`, `executionStats`, `allPlansExecution`) determines whether the results include *executionStats* (page 948) and whether *executionStats* (page 948) includes data captured during *plan selection*.

For details on the output, see *Explain Results* (page 946).

For a mixed version sharded cluster with version 3.0 `mongos` (page 792) and at least one 2.6 `mongod` (page 770) shard, when you run `cursor.explain()` (page 140) in a version 3.0 `mongo` (page 803) shell, `cursor.explain()` (page 140) will retry with the `$explain` operator to return results in the 2.6 format.

cursor.forEach()

On this page

- Description (page 141)
- Example (page 142)

Description

`cursor.forEach(`*function*`)`

Iterates the cursor to apply a JavaScript `function` to each document from the cursor.

The `forEach()` (page 141) method has the following prototype form:

```
db.collection.find().forEach(<function>)
```

The `forEach()` (page 141) 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 141) method on the cursor returned by `find()` (page 51) 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 144) for similar functionality.

cursor.hasNext()

`cursor.hasNext()`

Returns Boolean.

`cursor.hasNext()` (page 142) returns `true` if the cursor returned by the `db.collection.find()` (page 51) query can iterate further to return more documents.

cursor.hint()

On this page

- Definition (page 142)
- Behavior (page 142)
- Example (page 142)

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 73) to return the list of current indexes on a collection.

The `cursor.hint()` (page 142) 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.

You can also specify `{ $natural : 1 }` to force the query to perform a forwards collection scan, or `{ $natural : -1 }` for a reverse collection scan.

Behavior When an *index filter* exists for the query shape, MongoDB ignores the `hint()` (page 142).

You cannot use `hint()` (page 142) if the query includes a `$text` (page 549) 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" )
```

You can specify `{ $natural : 1 }` to force the query to perform a forwards collection scan:

```
db.users.find().hint( { $natural : 1 } )
```

You can also specify `{ $natural : -1 }` to force the query to perform a reverse collection scan:

```
db.users.find().hint( { $natural : -1 } )
```

See also:

- `https://docs.mongodb.org/manual/core/indexes-introduction`
- `https://docs.mongodb.org/manual/administration/indexes`
- `https://docs.mongodb.org/manual/core/query-plans`
- *index-filters*
- `$hint`

cursor.itcount()

On this page

- Definition (page 143)

Definition

`cursor.itcount()`

Counts the number of documents remaining in a cursor.

`itcount()` (page 143) is similar to `cursor.count()` (page 137), but actually executes the query on an existing iterator, exhausting its contents in the process.

The `itcount()` (page 143) method has the following prototype form:

```
db.collection.find(<query>).itcount()
```

See also:

`cursor.count()` (page 137)

cursor.limit()

On this page

- Definition (page 144)
- Behavior (page 144)

Definition

`cursor.limit()`

Use the `limit()` (page 144) method on a cursor to specify the maximum number of documents the cursor will return. `limit()` (page 144) is analogous to the `LIMIT` statement in a SQL database.

Note: You must apply `limit()` (page 144) to the cursor before retrieving any documents from the database.

Use `limit()` (page 144) to maximize performance and prevent MongoDB from returning more results than required for processing.

Behavior

Supported Values The behavior of `limit()` (page 144) is undefined for values less than -2^{31} and greater than 2^{31}.

Zero Value A `limit()` (page 144) value of 0 (i.e. `.limit(0)` (page 144)) is equivalent to setting no limit.

Negative Values A negative limit is similar to a positive limit but closes the cursor after returning a single *batch* of results. 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 specified limit. By passing a negative limit, the client indicates to the server that it will not ask for a subsequent batch via `getMore`.

cursor.map()

On this page

- Example (page 144)

`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 144) 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 141) for similar functionality.

cursor.maxScan()

On this page

- Definition (page 145)
- Behavior (page 145)
- Example (page 145)

Definition

`cursor.maxScan()`

New in version 3.2.

Specifies a maximum number of documents or index keys the query plan will scan. Once the limit is reached, the query terminates execution and returns the current batch of results.

`maxScan()` (page 145) has the following syntax:

```
cursor.maxScan( <maxScan> )
```

The method `cursor.maxScan` (page 145) has the following parameter:

param int, long, double maxScan The maximum number of documents or index keys that the query plan will scan.

Returns The *cursor* that `maxScan()` (page 145) is attached to with a modified result set based on the `maxScan` parameter. This allows for additional cursor modifiers to be chained.

Behavior For collection scans, `maxScan` is the maximum number of documents scanned before the query results are returned. For index scans, `maxScan` is the maximum number of index keys examined.

Using a value of `0` is equivalent to not using `cursor.maxScan()` (page 145).

Example Given the following data:

```
{ _id : 1, ts : 100, status : "OK" },
{ _id : 2, ts : 200, status : "OK" },
{ _id : 3, ts : 300, status : "WARN" },
{ _id : 4, ts : 400, status : "DANGER" },
{ _id : 5, ts : 500, status : "WARN" },
{ _id : 6, ts : 600, status : "OK" },
{ _id : 7, ts : 700, status : "OK" },
{ _id : 8, ts : 800, status : "WARN" },
{ _id : 9, ts : 900, status : "WARN" },
{ _id : 10, ts : 1000, status : "OK" }
```

Assuming this query were answered with a collection scan, the following limits the number of documents to scan to `5`:

```
db.collection.find ( { "status" : "OK" } ).maxScan(5)
```

The operation returns:

```
{ "_id" : 1, "ts" : 100, "status" : "OK" }
{ "_id" : 2, "ts" : 200, "status" : "OK" }
```

If this query were answered using an index scan on `{ status : 1 }`, the same operation returns the following:

```
{ "_id" : 1, "ts" : 100, "status" : "OK" }
{ "_id" : 2, "ts" : 200, "status" : "OK" }
{ "_id" : 6, "ts" : 600, "status" : "OK" }
{ "_id" : 7, "ts" : 700, "status" : "OK" }
```

cursor.maxTimeMS()

On this page

- Definition (page 146)
- Behaviors (page 146)
- Examples (page 146)

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 146) method has the following parameter:

param integer milliseconds Specifies a cumulative time limit in milliseconds for processing operations on the cursor.

Important: `maxTimeMS()` (page 146) is not related to the `NoCursorTimeout` query flag. `maxTimeMS()` (page 146) 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 192). 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.max()

On this page

- Definition (page 147)
- Behavior (page 147)
- Example (page 148)

Definition

`cursor.max()`

Specifies the *exclusive* upper bound for a specific index in order to constrain the results of `find()` (page 51). `max()` (page 147) provides a way to specify an upper bound on compound key indexes.

The `max()` (page 147) 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 142) method. Otherwise, `mongod` (page 770) 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 148).

`max()` (page 147) exists primarily to support the `mongos` (page 792) (sharding) process, and is a shell wrapper around the query modifier `$max`.

Behavior

Interaction with Index Selection Because `max()` (page 147) requires an index on a field, and forces the query to use this index, you may prefer the `$lt` (page 530) 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 147) with `min()` (page 148) to specify a range, the index bounds specified in `min()` (page 148) and `max()` (page 147) 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`.

Warning: If one of the two boundaries is not specified, 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 147) 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 142) method, it is ambiguous as to whether `mongod` (page 770) 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 147) limits the query to the documents that are below the index key bound of `price` equal to `1.99` and `min()` (page 148) 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" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
  ```

cursor.min()

On this page

- Definition (page 148)
- Behaviors (page 149)
- Example (page 149)

Definition

`cursor.min()`

Specifies the *inclusive* lower bound for a specific index in order to constrain the results of `find()` (page 51). `min()` (page 148) provides a way to specify lower bounds on compound key indexes.

The `min()` (page 148) 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 142) 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 147).

`min()` (page 148) exists primarily to support the `mongos` (page 792) process, and is a shell wrapper around the query modifier `$min`.

Behaviors

Interaction with Index Selection Because `min()` (page 148) requires an index on a field, and forces the query to use this index, you may prefer the `$gte` (page 530) 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 148) with `max()` (page 147) to specify a range, the index bounds specified in `min()` (page 148) and `max()` (page 147) 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`.

Warning: If one of the two boundaries is not specified, 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 148) 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 142) method, it is ambiguous as to whether `mongod` (page 770) 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 148) limits the query to the documents that are at or above the index key bound of `price` equal to `1.39` and `max()` (page 147) 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" : 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 51) method. See `cursor.hasNext()` (page 142) related functionality.

cursor.noCursorTimeout()

Definition

`cursor.noCursorTimeout()`

Instructs the server to avoid closing a cursor automatically after a period of inactivity.

The `noCursorTimeout()` (page 150) method has the following prototype form:

```
db.collection.find(<query>).noCursorTimeout()
```

cursor.objsLeftInBatch()

`cursor.objsLeftInBatch()`

`cursor.objsLeftInBatch()` (page 151) 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.pretty()

On this page

- Definition (page 151)
- Examples (page 151)

Definition

`cursor.pretty()`

Configures the cursor to display results in an easy-to-read format.

The `pretty()` (page 151) method has the following prototype form:

```
db.collection.find(<query>).pretty()
```

Examples

Consider the following document:

```
db.books.save({
    "_id" : ObjectId("54f612b6029b47909a90ce8d"),
    "title" : "A Tale of Two Cities",
    "text" : "It was the best of times, it was the worst of times, it was the age of wisdom, it was t
    "authorship" : "Charles Dickens"})
```

By default, `db.collection.find()` (page 51) returns data in a dense format:

```
db.books.find()
{ "_id" : ObjectId("54f612b6029b47909a90ce8d"), "title" : "A Tale of Two Cities", "text" : "It was th
```

By using `cursor.pretty()` (page 151) you can set the cursor to return data in a format that is easier for humans to parse:

```
db.books.find().pretty()
{
    "_id" : ObjectId("54f612b6029b47909a90ce8d"),
    "title" : "A Tale of Two Cities",
    "text" : "It was the best of times, it was the worst of times, it was the age of wisdom, it was t
    "authorship" : "Charles Dickens"
}
```

cursor.readConcern()

On this page

- Definition (page 152)

Definition

`cursor.readConcern(level)`

New in version 3.2.

Specify a *read concern* for the `db.collection.find()` (page 51) method.

The `readConcern()` (page 152) method has the following form:

```
db.collection.find().readConcern(<level>)
```

The `readConcern()` (page 152) method has the following parameter:

param string level Optional. One of the following *read concern* modes: `"local"` or `"majority"`

To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the `--enableMajorityReadConcern` (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

See also:

https://docs.mongodb.org/manual/reference/read-concern

cursor.readPref()

On this page

- Definition (page 152)

Definition

`cursor.readPref(mode, tagSet)`

Append `readPref()` (page 152) 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 Optional. 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 152) to the cursor before retrieving any documents from the database.

cursor.returnKey()

On this page

- Definition (page 153)
- Behavior (page 153)
- Example (page 153)

Definition

`cursor.returnKey()`

New in version 3.2.

Modifies the cursor to return index keys rather than the documents.

The `cursor.returnKey()` (page 153) has the following form:

```
cursor.returnKey()
```

Returns The *cursor* that `returnKey()` (page 153) is attached to with a modified result set. This allows for additional cursor modifiers to be chained.

Behavior If the query does not use an index to perform the read operation, the cursor returns empty documents.

Example The `restaurants` collection contains documents with the following schema:

```
{
   "_id" : ObjectId("564f3a35b385149fc7e3fab9"),
   "address" : {
      "building" : "2780",
      "coord" : [
         -73.98241999999999,
         40.579505
      ],
      "street" : "Stillwell Avenue",
      "zipcode" : "11224"
   },
   "borough" : "Brooklyn",
   "cuisine" : "American ",
   "grades" : [
      {
         "date" : ISODate("2014-06-10T00:00:00Z"),
         "grade" : "A",
         "score" : 5
      },
      {
         "date" : ISODate("2013-06-05T00:00:00Z"),
         "grade" : "A",
         "score" : 7
      }
   ],
   "name" : "Riviera Caterer",
   "restaurant_id" : "40356018"
}
```

The collection has two indexes in addition to the default `_id` index:

```
{
   "v" : 1,
   "key" : {
      "_id" : 1
   },
   "name" : "_id_",
   "ns" : "guidebook.restaurant"
},
{
   "v" : 1,
   "key" : {
      "cuisine" : 1
   },
   "name" : "cuisine_1",
   "ns" : "guidebook.restaurant"
},
{
   "v" : 1,
   "key" : {
      "_fts" : "text",
      "_ftsx" : 1
   },
   "name" : "name_text",
   "ns" : "guidebook.restaurant",
   "weights" : {
      "name" : 1
   },
   "default_language" : "english",
   "language_override" : "language",
   "textIndexVersion" : 3
}
```

The following code uses the `cursor.returnKey()` (page 153) method to return only the indexed fields used for executing the query:

```
var csr = db.restaurant.find( { "cuisine" : "Japanese" } )
csr.returnKey()
```

This returns the following:

```
{ "cuisine" : "Japanese" }
{ "cuisine" : "Japanese" }
{ "cuisine" : "Japanese" }
{ "cuisine" : "Japanese" }
...
```

cursor.showRecordId()

On this page

- Example (page 155)

`cursor.`**`showRecordId`**`()`

Changed in version 3.2: This method replaces the previous `cursor.showDiskLoc()`.

Modifies the output of a query by adding a field `$recordId` to matching documents. `$recordId` is the internal key which uniquely identifies a document in a collection. It has the form:

```
"$recordId": NumberLong(<int>)
```

Returns A modified cursor object that contains documents with appended information describing the internal record key.

Example The following operation appends the `showRecordId()` (page 154) method to the `db.collection.find()` (page 51) method in order to include storage engine record information in the matching documents:

```
db.collection.find( { a: 1 } ).showRecordId()
```

The operation returns the following documents, which include the `$recordId` field:

```
{
  "_id" : ObjectId("53908ccb18facd50a75bfbac"),
  "a" : 1,
  "b" : 1,
  "$recordId" : NumberLong(168112)
}
{
   "_id" : ObjectId("53908cd518facd50a75bfbad"),
   "a" : 1,
   "b" : 2,
   "$recordId" : NumberLong(168176)
}
```

You can *project* the added field `$recordId`, as in the following example:

```
db.collection.find( { a: 1 }, { $recordId: 1 } ).showRecordId()
```

This query returns only the `_id` field and the `$recordId` field in the matching documents:

```
{
  "_id" : ObjectId("53908ccb18facd50a75bfbac"),
  "$recordId" : NumberLong(168112)
}
{
   "_id" : ObjectId("53908cd518facd50a75bfbad"),
   "$recordId" : NumberLong(168176)
}
```

cursor.size()

`cursor.size()`

Returns A count of the number of documents that match the `db.collection.find()` (page 51) query after applying any `cursor.skip()` (page 155) and `cursor.limit()` (page 144) methods.

cursor.skip()

`cursor.skip()`

Call the `cursor.skip()` (page 155) 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 155) 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 155) 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 155) will become slower and more CPU intensive. With larger collections, `cursor.skip()` (page 155) 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 156) 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 156) to the cursor before retrieving any documents from the database.
- You can only use `snapshot()` (page 156) with **unsharded** collections.

The `snapshot()` (page 156) does not guarantee isolation from insertion or deletions.

The `snapshot()` (page 156) **cannot** be used with `sort()` (page 156) or `hint()` (page 142).

cursor.sort()

On this page

- Definition (page 156)
- Behaviors (page 157)
- Examples (page 158)
- Return in Natural Order (page 159)

Definition

`cursor.`**`sort`**`(`*sort*`)`

Specifies the order in which the query returns matching documents. You must apply `sort()` (page 156) to the cursor before retrieving any documents from the database.

The `sort()` (page 156) 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 157) or *sort on computed metadata* (page 158).

Behaviors

Result Ordering Unless you specify the `sort()` (page 156) method or use the `$near` (page 565) 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
11. Timestamp
12. Regular Expression
13. MaxKey (internal type)

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

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

The comparison treats a non-existent field as it would an empty BSON Object. As such, a sort on the `a` field in documents `{ }` and `{ a: null }` would treat the documents as equivalent in sort order.

With arrays, a less-than comparison or an ascending sort compares the smallest element of arrays, and a greater-than comparison or a descending sort compares the largest element of the arrays. As such, when comparing a field whose value is a single-element array (e.g. `[ 1 ]`) with non-array fields (e.g. `2`), the comparison is between `1` and `2`. A comparison of an empty array (e.g. `[ ]`) treats the empty array as less than `null` or a missing field.

MongoDB sorts `BinData` in the following order:

1. First, the length or size of the data.
2. Then, by the BSON one-byte subtype.
3. Finally, by the data, performing a byte-by-byte comparison.

Metadata Sort Specify in the sort parameter a new field name for the computed metadata and specify the `$meta` (page 592) 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 592) for details.

Restrictions When unable to obtain the sort order from an index, MongoDB will sort the results in memory, which requires that the result set being sorted is less than 32 megabytes.

When the sort operation consumes more than 32 megabytes, MongoDB returns an error. To avoid this error, either create an index supporting the sort operation (see *Sort and Index Use* (page 158)) or use `sort()` (page 156) in conjunction with `limit()` (page 144) (see *Limit Results* (page 158)).

Sort and Index Use The sort can sometimes be satisfied by scanning an index in order. If the query plan uses an index to provide the requested sort order, MongoDB does not perform an in-memory sorting of the result set. For more information, see `https://docs.mongodb.org/manual/tutorial/sort-results-with-indexes`.

Limit Results You can use `sort()` (page 156) in conjunction with `limit()` (page 144) to return the first (in terms of the sort order) `k` documents, where `k` is the specified limit.

If MongoDB cannot obtain the sort order via an index scan, then MongoDB uses a top-k sort algorithm. This algorithm buffers the first `k` results (or last, depending on the sort order) seen so far by the underlying index or collection access. If at any point the memory footprint of these `k` results exceeds 32 megabytes, the query will fail.

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 an embedded 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` 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.

Index Use Queries that include a sort by `$natural` order do **not** use indexes to fulfill the query predicate with the following exception: If the query predicate is an equality condition on the `_id` field `{ _id: <value> }`, then the query with the sort by `$natural` order can use the `_id` index.

MMAPv1 Typically, the natural order reflects insertion order with the following exception for the MMAPv1 storage engine. For the MMAPv1 storage engine, the natural order does not reflect insertion order if the documents relocate because of *document growth* or remove operations free up space which are then taken up by newly inserted documents.

Consider to following example which uses the MMAPv1 storage engine.

The following sequence of operations inserts documents into 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 } )
```

For MongoDB instances using MMAPv1, the documents return in the following natural order, which no longer reflects the insertion 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", "Goet
```

See also:

`$natural`

cursor.tailable()

On this page

- Definition (page 160)
- Behavior (page 161)

Definition

`cursor.`**`tailable`**`()`

New in version 3.2.

Marks the cursor as tailable.

For use against a *capped collection* only. Using `tailable` (page 160) against a non-capped collection will return an error.

`cursor.tailable()` (page 160) uses the following syntax:

```
cursor.tailable( isAwaitData : <boolean> )
```

`tailable()` (page 160) has the following parameter:

param boolean isAwaitData Optional. When `true`, enables `awaitData`.

`awaitData` is `false` by default.

Returns The *cursor* that `tailable()` (page 160) is attached to.

Behavior A tailable cursor performs a collection scan over a *capped collection*. It remains open even after reaching the end of the collection. Applications can continue to iterate the tailable cursor as new data is inserted into the collection.

If `awaitData` is set to `true`, when the cursor reaches the end of the capped collection, *MongoDB* blocks the query thread for a period of time waiting for new data to arrive. When new data is inserted into the capped collection, the blocked thread is signaled to wake up and return the next batch to the client.

See `https://docs.mongodb.org/manual/tutorial/create-tailable-cursor`

cursor.toArray()

`cursor.`**`toArray`**`()`

The `toArray()` (page 161) 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 161) to the cursor returned from the `find()` (page 51) 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 161).

2.1.3 Database

Database Methods

Name	Description
`db.cloneCollection()` (page 162)	Copies data directly between MongoDB instances. Wraps `cloneCollec`
`db.cloneDatabase()` (page 163)	Copies a database from a remote host to the current host. Wraps `clone` (p
`db.commandHelp()` (page 163)	Returns help information for a *database command*.
`db.copyDatabase()` (page 164)	Copies a database to another database on the current host. Wraps `copydb`
`db.createCollection()` (page 167)	Creates a new collection. Commonly used to create a capped collection.
`db.currentOp()` (page 171)	Reports the current in-progress operations.
`db.dropDatabase()` (page 177)	Removes the current database.
`db.eval()` (page 178)	Deprecated. Passes a JavaScript function to the `mongod` (page 770) instan
`db.fsyncLock()` (page 180)	Flushes writes to disk and locks the database to prevent write operations ar
`db.fsyncUnlock()` (page 181)	Allows writes to continue on a database locked with `db.fsyncLock()`
`db.getCollection()` (page 181)	Returns a collection object. Used to access collections with names that are
`db.getCollectionInfos()` (page 182)	Returns collection information for all collections in the current database.
`db.getCollectionNames()` (page 185)	Lists all collections in the current database.
`db.getLastError()` (page 185)	Checks and returns the status of the last operation. Wraps `getLastErro`
`db.getLastErrorObj()` (page 186)	Returns the status document for the last operation. Wraps `getLastErro`
`db.getLogComponents()` (page 187)	Returns the log message verbosity levels.
`db.getMongo()` (page 188)	Returns the `Mongo()` (page 295) connection object for the current connec
`db.getName()` (page 188)	Returns the name of the current database.
`db.getPrevError()` (page 188)	Returns a status document containing all errors since the last error reset. W

Table 2.2 – continued from previous page

Name	Description
`db.getProfilingLevel()` (page 188)	Returns the current profiling level for database operations.
`db.getProfilingStatus()` (page 189)	Returns a document that reflects the current profiling level and the profilin
`db.getReplicationInfo()` (page 189)	Returns a document with replication statistics.
`db.getSiblingDB()` (page 190)	Provides access to the specified database.
`db.help()` (page 190)	Displays descriptions of common `db` object methods.
`db.hostInfo()` (page 191)	Returns a document with information about the system MongoDB runs on
`db.isMaster()` (page 191)	Returns a document that reports the state of the replica set.
`db.killOp()` (page 192)	Terminates a specified operation.
`db.listCommands()` (page 192)	Displays a list of common database commands.
`db.loadServerScripts()` (page 192)	Loads all scripts in the `system.js` collection for the current database int
`db.logout()` (page 192)	Ends an authenticated session.
`db.printCollectionStats()` (page 193)	Prints statistics from every collection. Wraps `db.collection.stats`
`db.printReplicationInfo()` (page 193)	Prints a report of the status of the replica set from the perspective of the pr
`db.printShardingStatus()` (page 194)	Prints a report of the sharding configuration and the chunk ranges.
`db.printSlaveReplicationInfo()` (page 195)	Prints a report of the status of the replica set from the perspective of the se
`db.repairDatabase()` (page 195)	Runs a repair routine on the current database.
`db.resetError()` (page 196)	Resets the error message returned by `db.getPrevError()` (page 188)
`db.runCommand()` (page 196)	Runs a *database command* (page 303).
`db.serverBuildInfo()` (page 196)	Returns a document that displays the compilation parameters for the `mong`
`db.serverCmdLineOpts()` (page 196)	Returns a document with information about the runtime used to start the M
`db.serverStatus()` (page 197)	Returns a document that provides an overview of the state of the database p
`db.setLogLevel()` (page 197)	Sets a single log message verbosity level.
`db.setProfilingLevel()` (page 199)	Modifies the current level of database profiling.
`db.shutdownServer()` (page 199)	Shuts down the current `mongod` (page 770) or `mongos` (page 792) proces
`db.stats()` (page 199)	Returns a document that reports on the state of the current database.
`db.version()` (page 200)	Returns the version of the `mongod` (page 770) instance.
`db.upgradeCheck()` (page 200)	Performs a preliminary check for upgrade preparedness for a specific datab
`db.upgradeCheckAllDBs()` (page 202)	Performs a preliminary check for upgrade preparedness for all databases a

db.cloneCollection()

On this page

- Definition (page 162)
- Behavior (page 163)

Definition

`db.`**`cloneCollection`**(*from*, *collection*, *query*)

Copies data directly between MongoDB instances. The `db.cloneCollection()` (page 162) method wraps the `cloneCollection` (page 443) database command and accepts the following arguments:

param string from The address of the server to clone from.

param string collection The collection in the MongoDB instance that you want to copy. `db.cloneCollection()` (page 162) 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 443) directly.

param document query Optional. A standard query document that limits the documents copied as part of the `db.cloneCollection()` (page 162) operation. All *query selectors* (page 527) available to the `find()` (page 51) are available here.

Behavior `mongos` (page 792) does not support `db.cloneCollection()` (page 162).

Changed in version 3.0: If the given *namespace* already exists in the destination `mongod` (page 770) instance, `db.cloneCollection()` (page 162) will return an error.

db.cloneDatabase()

On this page

- Definition (page 163)
- Example (page 163)

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 442)." The `copydb` (page 433) 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()

On this page

- Description (page 163)

Description

`db.commandHelp(`*command*`)`

Displays help text for the specified *database command*. See the *Database Commands* (page 303).

The `db.commandHelp()` (page 163) method has the following parameter:

param string command The name of a *database command*.

db.copyDatabase()

On this page

- Definition (page 164)
- Behavior (page 164)
- Required Access (page 165)
- Example (page 166)

Definition

`db.copyDatabase`(*fromdb*, *todb*, *fromhost*, *username*, *password*, *mechanism*)

Changed in version 3.0: When authenticating to the `fromhost` instance, `db.copyDatabase()` (page 164) supports *MONGODB-CR* and *SCRAM-SHA-1* mechanisms to authenticate the `fromhost` user.

Copies a database either from one `mongod` (page 770) instance to the current `mongod` (page 770) instance or within the current `mongod` (page 770). `db.copyDatabase()` (page 164) wraps the `copydb` (page 433) command and takes the following arguments:

param string fromdb Name of the source database.

param string todb Name of the target database.

param string fromhost Optional. The hostname of the source `mongod` (page 770) instance. Omit to copy databases within the same `mongod` (page 770) instance.

param string username Optional. The name of the user on the `fromhost` MongoDB instance. The user authenticates to the `fromdb`.

For more information, see *Authentication to Source mongod Instance* (page 165).

param string password Optional. The password on the `fromhost` for authentication. The method does **not** transmit the password in plaintext.

For more information, see *Authentication to Source mongod Instance* (page 165).

param string mechanism Optional. The mechanism to authenticate the `username` and `password` on the `fromhost`. Specify either *MONGODB-CR* or *SCRAM-SHA-1*.

`db.copyDatabase` (page 164) defaults to *SCRAM-SHA-1* if the wire protocol version (`maxWireVersion` (page 410)) is greater than or equal to `3` (i.e. MongoDB versions 3.0 or greater). Otherwise, it defaults to *MONGODB-CR*.

Specify `MONGODB-CR` to authenticate to the version 2.6.x `fromhost` from a version 3.0 instance or greater. For an example, see *Copy Database from a mongod Instances that Enforce Authentication* (page 167).

New in version 3.0.

Behavior

Destination

- Run `db.copyDatabase()` (page 164) in the `admin` database of the destination `mongod` (page 770) instance, i.e. the instance receiving the copied data.
- `db.copyDatabase()` (page 164) creates the target database if it does not exist.

- `db.copyDatabase()` (page 164) requires enough free disk space on the host instance for the copied database. Use the `db.stats()` (page 199) operation to check the size of the database on the source `mongod` (page 770) instance.

Authentication to Source `mongod` Instance

- If copying from another `mongod` (page 770) instance (`fromhost`) that enforces `access control` (page 910), then you must authenticate to the `fromhost` instance by specifying the `username`, `password`, and optionally `mechanism`. The method does **not** transmit the password in plaintext.

  When authenticating to the `fromhost` instance, `db.copyDatabase()` (page 164) uses the `fromdb` as the *authentication database* for the specified user.

- When authenticating to the `fromhost` instance, `db.copyDatabase()` (page 164) supports *MONGODB-CR* and *SCRAM-SHA-1* mechanisms to authenticate the `fromhost` user.
  - To authenticate to a version 2.6 `fromhost`, you must specify `MONGODB-CR` as the authentication mechanism. See *Copy Database from a mongod Instances that Enforce Authentication* (page 167).
  - To copy from a version 3.0 `fromhost` to a version 2.6 instance, i.e. if running the method from a version 2.6 instance to copy from a version 3.0 `fromhost`, you can only authenticate to the `fromhost` as a `MONGODB-CR` user.

For more information on required access and authentication, see *Required Access* (page 165).

Concurrency

- `db.copyDatabase()` (page 164) and `clone` (page 442) 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 164) does not lock the destination server during its operation, so the copy will occasionally yield to allow other operations to complete.

Sharded Clusters

- Do not use `db.copyDatabase()` (page 164) from a `mongos` (page 792) instance.
- Do not use `db.copyDatabase()` (page 164) to copy databases that contain sharded collections.

Required Access Changed in version 2.6.

Source Database (`fromdb`) If the `mongod` (page 770) instance of the *source* database (`fromdb`) enforces `access control` (page 910), you must have proper authorization for the *source* database.

If copying from another `mongod` (page 770) instance (`fromhost`) that enforces `access control` (page 910), then you must authenticate to the `fromhost` instance by specifying the `username`, `password`, and optionally `mechanism`. The method does **not** transmit the password in plaintext.

When authenticating to the `fromhost` instance, `db.copyDatabase()` (page 164) uses the `fromdb` as the *authentication database* for the specified user.

When authenticating to the `fromhost` instance, `db.copyDatabase()` (page 164) supports *MONGODB-CR* and *SCRAM-SHA-1* mechanisms to authenticate the `fromhost` user.

- To authenticate to a version 2.6 `fromhost`, you must specify `MONGODB-CR` as the authentication mechanism. See *Copy Database from a mongod Instances that Enforce Authentication* (page 167).

- To copy from a version 3.0 `fromhost` to a version 2.6 instance, i.e. if running the method from a version 2.6 instance to copy from a version 3.0 `fromhost`, you can only authenticate to the `fromhost` as a `MONGODB-CR` user.

Source is non-`admin` Database Changed in version 3.0.

If the source database is a non-`admin` database, you must have privileges that specify `find`, `listCollections`, and `listIndexes` actions on the source database, and `find` action on the `system.js` collection in the source database.

```
{ resource: { db: "mySourceDB", collection: "" }, actions: [ "find", "listCollections", "listIndexes'
{ resource: { db: "mySourceDB", collection: "system.js" }, actions: [ "find" ] },
```

Source is `admin` Database Changed in version 3.0.

If the source database is the `admin` database, you must have privileges that specify `find`, `listCollections`, and `listIndexes` actions 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",  "listCollections", "listIndexes" ] 
{ 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" ] }
```

Target Database (`todb`) If the `mongod` (page 770) instance of the *target* database (`todb`) enforces `access control` (page 910), you must have proper authorization for the *target* database.

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" ] }
```

Example

Copy from the Same `mongod` Instance To copy within the same `mongod` (page 770) instance, omit the `fromhost`.

The following operation copies a database named `records` into a database named `archive_records`:

```
db.copyDatabase('records', 'archive_records')
```

Copy Database from a `mongod` Instances that Enforce Authentication If copying from another `mongod` (page 770) instance (`fromhost`) that enforces `access control` (page 910), then you must authenticate to the `fromhost` instance by specifying the `username`, `password`, and optionally `mechanism`. The method does **not** transmit the password in plaintext.

When authenticating to the `fromhost` instance, `db.copyDatabase()` (page 164) uses the `fromdb` as the *authentication database* for the specified user.

Changed in version 3.0: MongoDB 3.0 supports passing the authentication mechanism to use for the `fromhost`.

The following operation copies a database named `reporting` from a version 2.6 `mongod` (page 770) instance that runs on `example.net` and enforces access control.

```
db.copyDatabase(
   "reporting",
   "reporting_copy",
   "example.net",
   "reportUser",
   "abc123",
   "MONGODB-CR"
)
```

See also:

`clone` (page 442)

db.createCollection()

On this page

- Definition (page 167)
- Examples (page 170)

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 collections that use specific options. For example, you use `db.createCollection()` (page 167) to create a *capped collection*, or to create a new collection that uses `document validation`. `db.createCollection()` (page 167) is also used to pre-allocate space for an ordinary collection.

The `db.createCollection()` (page 167) method has the following prototype form:

Changed in version 3.2.

```
db.createCollection(<name>, { capped: <boolean>,
                              autoIndexId: <boolean>,
                              size: <number>,
                              max: <number>,
                              storageEngine: <document>,
                              validator: <document>,
                              validationLevel: <string>,
                              validationAction: <string>,
                              indexOptionDefaults: <document> } )
```

The `db.createCollection()` (page 167) method has the following parameters:

param string name The name of the collection to create.

param document options Optional. Configuration options for creating a capped collection or for preallocating space in a new collection.

The `options` document creates a capped collection, preallocates space in a new ordinary collection, or specifies `document validation` criteria. The `options` document contains the following fields:

field boolean capped Optional. 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 Optional. Specify `false` to disable the automatic creation of an index on the `_id` field.

Important: For replica sets, all collections must have `autoIndexId` set to `true`.

field number size Optional. Specify a maximum size in bytes for a capped collection. Once a capped collection reaches its maximum size, MongoDB removes the older documents to make space for the new documents. The `size` field is required for capped collections and ignored for other collections.

field number max Optional. The maximum number of documents allowed in the capped collection. The `size` limit takes precedence over this limit. If a capped collection reaches the `size` limit before it reaches the maximum number of documents, MongoDB removes old documents. If you prefer to use the `max` limit, ensure that the `size` limit, which is required for a capped collection, is sufficient to contain the maximum number of documents.

field boolean usePowerOf2Sizes Optional. Available for the MMAPv1 storage engine only.

Deprecated since version 3.0: For the MMAPv1 storage engine, all collections use the *power of 2 sizes allocation* unless the `noPadding` option is `true`. The `usePowerOf2Sizes` option does not affect the allocation strategy.

field boolean noPadding Optional. Available for the MMAPv1 storage engine only.

New in version 3.0: `noPadding` flag disables the *power of 2 sizes allocation* for the collection. With `noPadding` flag set to true, the allocation strategy does not include additional space to accommodate document growth, as such, document growth will result in new allocation. Use for collections with workloads that are insert-only or in-place updates (such as incrementing counters).

Defaults to `false`.

Warning: Do not set `noPadding` if the workload includes removes or any updates that may cause documents to grow. For more information, see *exact-fit-allocation*.

field document storageEngine Optional. Available for the WiredTiger storage engine only.

New in version 3.0.

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:

```
{ <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.

field document validator Optional. Allows users to specify `validation rules or expressions` for the collection. For more information, see `https://docs.mongodb.org/manual/core/document-validation`.

New in version 3.2.

The `validator` option takes a document that specifies the validation rules or expressions. You can specify the expressions using the same operators as the *query operators* (page 527) with the exception of `$geoNear`, `$near` (page 565), `$nearSphere` (page 567), `$text` (page 549), and `$where` (page 558).

Note:

- Validation occurs during updates and inserts. Existing documents do not undergo validation checks until modification.
- You cannot specify a validator for collections in the `admin`, `local`, and `config` databases.
- You cannot specify a validator for `system.*` collections.

field string validationLevel Optional. Determines how strictly MongoDB applies the validation rules to existing documents during an update.

New in version 3.2.

`validationLevel`	Description
`"off"`	No validation for inserts or updates.
`"strict"`	**Default** Apply validation rules to all inserts and all updates.
`"moderate"`	Apply validation rules to inserts and to updates on existing *valid* documents. Do not apply rules to updates on existing *invalid* documents.

field string validationAction Optional. Determines whether to `error` on invalid documents or just `warn` about the violations but allow invalid documents to be inserted.

New in version 3.2.

Important: Validation of documents only applies to those documents as determined by the `validationLevel`.

`validationAction`	Description
`"error"`	**Default** Documents must pass validation before the write occurs. Otherwise, the write operation fails.
`"warn"`	Documents do not have to pass validation. If the document fails validation, the write operation logs the validation failure.

field document indexOptionDefaults Optional. Allows users to specify a default configuration for indexes when creating a collection.

The `indexOptionDefaults` option accepts a `storageEngine` document, which 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.

New in version 3.2.

`db.createCollection()` (page 167) is a wrapper around the database command `create` (page 439).

Examples

Create a Capped Collection Capped collections have maximum size or document counts that prevent them from growing beyond maximum thresholds. All capped collections must specify a maximum size and may also specify a maximum document count. MongoDB removes older documents if a collection reaches the maximum size limit before it reaches the maximum document count. Consider the following example:

```
db.createCollection("log", { capped : true, size : 5242880, max : 5000 } )
```

This command creates a collection named `log` with a maximum size of 5 megabytes and a maximum of 5000 documents.

The following command simply pre-allocates a 2-gigabyte, uncapped collection named `people`:

```
db.createCollection("people", { size: 2147483648 } )
```

See `https://docs.mongodb.org/manual/core/capped-collections` for more information about capped collections.

Create a Collection with Document Validation New in version 3.2.

Collections with validation compare each inserted or updated document against the criteria specified in the `validator` option. Depending on the `validationLevel` and `validationAction`, MongoDB either returns a warning, or refuses to insert or update the document if it fails to meet the specified criteria.

The following example creates a `contacts` collection with a validator that specifies that inserted or updated documents should match at least one of three following conditions:

- the `phone` field is a string
- the `email` field matches the regular expression
- the `status` field is either `Unknown` or `Incomplete`.

```
db.createCollection( "contacts",
   {
      validator: { $or:
         [
            { phone: { $type: "string" } },
            { email: { $regex: /@mongodb\.com$/ } },
            { status: { $in: [ "Unknown", "Incomplete" ] } }
         ]
      }
   }
)
```

With the validator in place, the following insert operation fails validation:

```
db.contacts.insert( { name: "Amanda", status: "Updated" } )
```

The method returns the error in the `WriteResult`:

```
WriteResult({
   "nInserted" : 0,
   "writeError" : {
      "code" : 121,
      "errmsg" : "Document failed validation"
   }
})
```

For more information, see `https://docs.mongodb.org/manual/core/document-validation`. To view the validation specifications for a collection, use the `db.getCollectionInfos()` (page 182) method.

Specify Storage Engine Options New in version 3.0.

You can specify collection-specific storage engine configuration options when you create a collection with `db.createCollection()` (page 167). 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[6] for specific `wiredTiger` options.

db.currentOp()

On this page

- Definition (page 171)
- Behavior (page 172)
- Access Control (page 172)
- Examples (page 172)
- Output Example (page 173)
- Output Fields (page 174)

Definition

`db.currentOp()`

Returns a *document* that contains information on in-progress operations for the database instance.

`db.currentOp()` (page 171) method has the following form:

```
db.currentOp(<operations>)
```

The `db.currentOp()` (page 171) method can take the following *optional* argument:

param boolean or document operations Optional. 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 172) for details.

[6]http://source.wiredtiger.com/2.4.1/struct_w_t___s_e_s_s_i_o_n.html#a358ca4141d59c345f401c58501276bbb

Behavior If you pass in `true` to `db.currentOp()` (page 171), 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 171), the output returns information only for the current operations that match the query. You can query on the *Output Fields* (page 174). See *Examples* (page 172).

You can also specify `{ '$all': true }` query document to return information on all in-progress operations, including operations on idle connections and system operations. If the query document includes `'$all': true` as well as other query conditions, only the `'$all': true` applies.

Access Control On systems running with `authorization` (page 910), 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 171) 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.findandmodify": { $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 a prototype of `db.currentOp()` (page 171) output.

```
{
  "inprog": [
       {
         "desc" : <string>,
         "threadId" : <string>,
         "connectionId" : <number>,
         "opid" : <number>,
         "active" : <boolean>,
         "secs_running" : <NumberLong()>,
         "microsecs_running" : <number>,
         "op" : <string>,
         "ns" : <string>,
         "query" : <document>,
         "insert" : <document>,
         "planSummary": <string>,
         "client" : <string>,
         "msg": <string>,
         "progress" : {
             "done" : <number>,
             "total" : <number>
         },
         "killPending" : <boolean>,
         "numYields" : <number>,
         "locks" : {
             "Global" : <string>,
             "MMAPV1Journal" : <string>,
             "Database" : <string>,
             "Collection" : <string>,
             "Metadata" : <string>,
             "oplog" : <string>
         },
         "waitingForLock" : <boolean>,
         "lockStats" : {
             "Global": {
                "acquireCount": {
                   "r": <NumberLong>,
                   "w": <NumberLong>,
                   "R": <NumberLong>,
                   "W": <NumberLong>
                },
                "acquireWaitCount": {
                   "r": <NumberLong>,
                   "w": <NumberLong>,
                   "R": <NumberLong>,
                   "W": <NumberLong>
                },
                "timeAcquiringMicros" : {
                   "r" : NumberLong(0),
                   "w" : NumberLong(0),
                   "R" : NumberLong(0),
                   "W" : NumberLong(0)
                },
                "deadlockCount" : {
                   "r" : NumberLong(0),
                   "w" : NumberLong(0),
                   "R" : NumberLong(0),
                   "W" : NumberLong(0)
                }
             },
             "MMAPV1Journal": {
                ...
             },
             "Database" : {
                ...
             },
             ...
          }
       },
       ...
   ],
   "fsyncLock": <boolean>,
   "info": <string>
}
```

Output Fields

currentOp.**desc**

A description of the client. This string includes the connectionId (page 174).

currentOp.**threadId**

An identifier for the thread that handles the operation and its connection.

currentOp.**connectionId**

An identifier for the connection where the operation originated.

currentOp.**opid**

The identifier for the operation. You can pass this value to db.killOp() (page 192) in the mongo (page 803) shell to terminate the operation.

Warning: Terminate running operations with extreme caution. Only use db.killOp() (page 192) 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 idle, such as an idle connection or an internal thread that is currently idle. An operation can be active even if the operation has yielded to another operation.

Changed in version 3.0: For some inactive background threads, such as an inactive signalProcessingThread, MongoDB suppresses various empty fields.

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 174) 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 174) is `true`.

`currentOp.`**`op`**
A string that identifies the type of operation. The possible values are:

- `"none"`
- `"update"`
- `"insert"`
- `"query"`
- `"getmore"`
- `"remove"`
- `"killcursors"`

`"query"` operations include read operations as well as most commands such as the `createIndexes` (page 446) command and the `findandmodify` command.

Changed in version 3.0: Write operations that use the `insert` (page 337), `update` (page 340), and `delete` (page 345) commands respectively display `"insert"`, `"update"`, and `"delete"` for `op` (page 175). Previous versions include these write commands under `"query"` operations.

`currentOp.`**`ns`**
The *namespace* the operation targets. A namespace consists of the *database* name and the *collection* name concatenated with a dot (`.`); that is, `"<database>.<collection>"`.

`currentOp.`**`insert`**
Contains the document to be inserted for operations with `op` (page 175) value of `"insert"`. Only appears for operations with `op` (page 175) value `"insert"`.

Insert operations such as `db.collection.insert()` (page 79) that use the `insert` (page 337) command will have `op` (page 175) value of `"query"`.

`currentOp.`**`query`**
A document containing information on operations whose `op` (page 175) value is *not* `"insert"`. For instance, for a `db.collection.find()` (page 51) operation, the `query` (page 175) contains the query predicate.

`query` (page 175) does not appear for `op` (page 175) of `"insert"`. `query` (page 175) can also be an empty document.

For `"update"` (page 175) or `"remove"` (page 175) operations or for read operations categorized under `"query"` (page 175), the `query` (page 175) document contains the query predicate for the operations.

Changed in version 3.0.4: For `"getmore"` (page 175) operations on cursors returned from a `db.collection.find()` (page 51) or a `db.collection.aggregate()` (page 20), the `query` (page 175) field contains respectively the query predicate or the issued `aggregate` (page 303) command document. For details on the `aggregate` (page 303) command document, see the `aggregate` (page 303) reference page.

For other commands categorized under `"query"` (page 175), `query` (page 175) contains the issued command document. Refer to the specific command reference page for the details on the command document.

Changed in version 3.0: Previous versions categorized operations that used write commands under `op` (page 175) of `"query"` and returned the write command information (e.g. query predicate, update statement, and update options) in `query` (page 175) document.

`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.

`currentOp.`**`locks`**
Changed in version 3.0.

The `locks` (page 176) document reports the type and mode of locks the operation currently holds. The possible lock types are as follows:

Lock Type	Description
`Global`	Represents global lock.
`MMAPV1Journal`	Represents MMAPv1 storage engine specific lock to synchronize journal writes; for non-MMAPv1 storage engines, the mode for `MMAPV1Journal` is empty.
`Database`	Represents database lock.
`Collection`	Represents collection lock.
`Metadata`	Represents metadata lock.
`oplog`	Represents lock on the *oplog*.

The possible modes are as follows:

Lock Mode	Description
`R`	Represents Shared (S) lock.
`W`	Represents Exclusive (X) lock.
`r`	Represents Intent Shared (IS) lock.
`w`	Represents Intent Exclusive (IX) lock.

`currentOp.`**`waitingForLock`**
Returns a boolean value. `waitingForLock` (page 176) is `true` if the operation is waiting for a lock and `false` if the operation has the required lock.

`currentOp.`**`msg`**
The `msg` (page 176) 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 176) fields corresponds to the completion percentage in the `msg` (page 176) field. The `progress` (page 176) 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 176) 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.`**`fsyncLock`**

Specifies if database is currently locked for `fsync write/snapshot` (page 180).

Only appears if locked; i.e. if `fsyncLock` (page 177) is `true`.

`currentOp.`**`info`**

Information regarding how to unlock database from `db.fsyncLock()` (page 180). Only appears if `fsyncLock` (page 177) is `true`.

`currentOp.`**`lockStats`**

For each lock type and mode (see `currentOp.locks` (page 176) for descriptions of lock types and modes), returns the following information:

`currentOp.lockStats.`**`acquireCount`**

Number of times the operation acquired the lock in the specified mode.

`currentOp.lockStats.`**`acquireWaitCount`**

Number of times the operation had to wait for the `acquireCount` (page 177) lock acquisitions because the locks were held in a conflicting mode. `acquireWaitCount` (page 177) is less than or equal to `acquireCount` (page 177).

`currentOp.lockStats.`**`timeAcquiringMicros`**

Cumulative time in microseconds that the operation had to wait to acquire the locks.

`timeAcquiringMicros` (page 177) divided by `acquireWaitCount` (page 177) gives an approximate average wait time for the particular lock mode.

`currentOp.lockStats.`**`deadlockCount`**

Number of times the operation encountered deadlocks while waiting for lock acquisitions.

db.dropDatabase()

On this page

- Definition (page 177)
- Behavior (page 177)
- Example (page 178)

Definition

`db.`**`dropDatabase`**`()`

Removes the current database, deleting the associated data files.

Behavior The `db.dropDatabase()` (page 177) wraps the `dropDatabase` (page 437) command.

Warning: This command obtains a global write lock and will block other operations until it has completed.

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 378) command in the database you are deleting.

Example The following example in the `mongo` (page 803) shell uses the `use <database>` operation to switch the current database to the `temp` database and then uses the `db.dropDatabase()` (page 177) method to drops the `temp` database:

```
use temp
db.dropDatabase()
```

See also:

`dropDatabase` (page 437)

db.eval()

On this page

- Definition (page 178)
- Behavior (page 178)
- Examples (page 179)

Definition

`db.eval(function, arguments)`

Deprecated since version 3.0.

Provides the ability to run JavaScript code on the MongoDB server.

The helper `db.eval()` (page 178) in the `mongo` (page 803) shell wraps the `eval` (page 358) command. Therefore, the helper method shares the characteristics and behavior of the underlying command with *one exception*: `db.eval()` (page 178) method does not support the `nolock` option.

The method accepts the following parameters:

param function function A JavaScript function to execute.

param list arguments Optional. 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 178) takes a global write lock while evaluating the JavaScript function. As a result, `db.eval()` (page 178) blocks all other read and write operations to the database while the `db.eval()` (page 178) operation runs.

To prevent the taking of the global write lock while evaluating the JavaScript code, use the `eval` (page 358) *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 178) 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 178) with *sharded* collections. In general, you should avoid using `db.eval()` (page 178) in *sharded clusters*; nevertheless, it is possible to use `db.eval()` (page 178) 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 358). Providing such access is not recommended, but if your organization requires a user to run `eval` (page 358), 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 178) executed in a single thread.

Examples The following is an example of the `db.eval()` (page 178) 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 178). Otherwise, the `mongo` (page 803) shell's JavaScript interpreter evaluates functions entered directly into the shell.

If an error occurs, `db.eval()` (page 178) throws an exception. The following is an example of an invalid function that uses the variable `x` without declaring it as an argument:

```
db.eval( function() { return x + x; }, 3 );
```

The statement results in the following exception:

```
{
   "errmsg" : "exception: JavaScript execution failed: ReferenceError: x is not defined near '{ retur
   "code" : 16722,
   "ok" : 0
}
```

See also:

https://docs.mongodb.org/manual/core/server-side-javascript

db.fsyncLock()

On this page

- Definition (page 180)
- Behavior (page 180)

Definition

`db.fsyncLock()`

Forces the `mongod` (page 770) to flush all pending write operations to the disk and locks the *entire* `mongod` (page 770) instance to prevent additional writes until the user releases the lock with the `db.fsyncUnlock()` (page 181) command. `db.fsyncLock()` (page 180) is an administrative command.

This command provides a simple wrapper around a `fsync` (page 451) database command with the following syntax:

```
{ fsync: 1, lock: true }
```

This function locks the database and create a window for `backup operations`.

Behavior

Compatibility with WiredTiger Changed in version 3.2: Starting in MongoDB 3.2, `db.fsyncLock()` (page 180) can ensure that the data files do not change for MongoDB instances using either the MMAPv1 or the WiredTiger storage engine, thus providing consistency for the purposes of creating backups.

In previous MongoDB version, `db.fsyncLock()` (page 180) *cannot* guarantee a consistent set of files for low-level backups (e.g. via file copy `cp`, `scp`, `tar`) for WiredTiger.

Impact on Read Operations `db.fsyncLock()` (page 180) *may* block reads, including those necessary to verify authentication. Such reads are necessary to establish new connections to a `mongod` (page 770) that enforces authorization checks.

Connection When calling `db.fsyncLock()` (page 180), ensure that the connection is kept open to allow a subsequent call to `db.fsyncUnlock()` (page 181).

Closing the connection may make it difficult to release the lock.

db.fsyncUnlock()

On this page

- Definition (page 181)
- Wired Tiger Compatibility (page 181)

Definition

`db.fsyncUnlock()`

Unlocks a `mongod` (page 770) instance to allow writes and reverses the operation of a `db.fsyncLock()` (page 180) operation. Typically you will use `db.fsyncUnlock()` (page 181) following a database `backup operation`.

`db.fsyncUnlock()` (page 181) is an administrative operation.

Wired Tiger Compatibility Changed in version 3.2: Starting in MongoDB 3.2, `db.fsyncLock()` (page 180) can ensure that the data files do not change for MongoDB instances using either the MMAPv1 or the WiredTiger storage engine, thus providing consistency for the purposes of creating backups.

In previous MongoDB version, `db.fsyncLock()` (page 180) *cannot* guarantee a consistent set of files for low-level backups (e.g. via file copy `cp`, `scp`, `tar`) for WiredTiger.

db.getCollection()

On this page

- Definition (page 181)
- Behavior (page 181)
- Example (page 181)

Definition

`db.getCollection(name)`

Returns a *collection* object that is functionally equivalent to using the `db.<collectionName>` syntax. The method is useful for a collection whose name might interact with the shell itself, such as names that begin with _ or that match a *database shell method* (page 161).

The `db.getCollection()` (page 181) method has the following parameter:

param string name The name of the collection.

Behavior The `db.getCollection()` (page 181) object can access any *collection methods* (page 19).

The collection specified may or may not exist on the server. If the collection does not exist, MongoDB creates it implicitly as part of `write operations` like `insertOne()` (page 82).

Example The following example uses `db.getCollection()` (page 181) to access the `auth` collection and insert a document into it.

```
var authColl = db.getCollection("auth")

authColl.insertOne(
   {
       usrName : "John Doe",
       usrDept : "Sales",
       usrTitle : "Executive Account Manager",
       authLevel : 4,
       authDept : [ "Sales", "Customers"]
   }
)
```

This returns:

```
{
   "acknowledged" : true,
   "insertedId" : ObjectId("569525e144fe66d60b772763")
}
```

The previous example requires the use of `db.getCollection("auth")` (page 181) because of a name conflict with the database method `db.auth()` (page 229). Calling `db.auth` directly to perform an insert operation would reference the `db.auth()` (page 229) method and would error.

The following example attempts the same operation, but without using the `db.getCollection()` (page 181) method:

```
db.auth.insertOne(
   {
       usrName : "John Doe",
       usrDept : "Sales",
       usrTitle : "Executive Account Manager",
       authLevel : 4,
       authDept : [ "Sales", "Customers"]
   }
)
```

The operation errors as `db.auth()` method has no `insertOne` method.

See also:

Collection Methods (page 19)

db.getCollectionInfos()

On this page

- Definition (page 182)
- Example (page 183)

Definition

`db.getCollectionInfos()`

New in version 3.0.0.

Returns an array of documents with collection information, i.e. collection name and options, for the current database.

The `db.getCollectionInfos()` (page 182) helper wraps the `listCollections` (page 438) command.

Changed in version 3.2: MongoDB 3.2 added support for `document validation`. `db.getCollectionInfos()` (page 182) includes document validation information in the `options` document.

`db.getCollectionInfos()` (page 182) does not return `validationLevel` and `validationAction` unless they are explicitly set.

Example The following returns information for all collections in the `example` database:

```
use example
db.getCollectionInfos()
```

The method returns an array of documents that contain collection information:

```
[
   {
      "name" : "employees",
      "options" : {
         "flags" : 1,
         "validator" : {
            "$or" : [
               {
                  "phone" : {
                     "$exists" : true
                  }
               },
               {
                  "email" : {
                     "$exists" : true
                  }
               }
            ]
         }
      }
   },
   {
      "name" : "products",
      "options" : {
         "flags" : 1
      }
   },
   {
      "name" : "mylogs",
      "options" : {
         "capped" : true,
         "size" : 256
      }
   },
   {
      "name" : "restaurants",
      "options" : {
         "validator" : {
            "$and" : [
               {
                  "name" : {
                     "$exists" : true
                  }
               },
               {
                  "restaurant_id" : {
                     "$exists" : true
                  }
               }
            ]
         },
         "validationLevel" : "strict",
         "validationAction" : "error"
      }
   },
   {
      "name" : "system.indexes",
      "options" : {
      }
   }
]
```

To request collection information for a *specific* collection, specify the collection name when calling the method, as in the following:

```
use example
db.getCollectionInfos( { name: "restaurants" } )
```

The method returns an array with a single document that details the collection information for the `restaurants` collection in the `example` database.

```
[
   {
      "name" : "restaurants",
      "options" : {
         "validator" : {
            "$and" : [
               {
                  "name" : {
                     "$exists" : true
                  }
               },
               {
                  "restaurant_id" : {
                     "$exists" : true
                  }
               }
            ]
         },
         "validationLevel" : "strict",
         "validationAction" : "error"
      }
   }
]
```

db.getCollectionNames()

On this page

- Definition (page 185)
- Considerations (page 185)
- Example (page 185)

Definition

`db.getCollectionNames()`

Returns an array containing the names of all collections in the current database.

Considerations

Changed in version 3.0.0.

For MongoDB 3.0 deployments using the *WiredTiger* storage engine, if you run `db.getCollectionNames()` (page 185) from a version of the `mongo` (page 803) shell before 3.0 or a version of the driver prior to *3.0 compatible version* (page 1051), `db.getCollectionNames()` (page 185) will return no data, even if there are existing collections. For more information, see *WiredTiger and Driver Version Compatibility* (page 1047).

Example

The following returns the names of all collections in the `records` database:

```
use records
db.getCollectionNames()
```

The method returns the names of the collections in an array:

```
[ "employees", "products", "mylogs", "system.indexes" ]
```

db.getLastError()

On this page

- Definition (page 185)
- Behavior (page 186)
- Example (page 186)

Definition

`db.getLastError(<w>, <wtimeout>)`

Specifies the level of *write concern* for confirming the success of previous write operation issued over the same connection and returns the `error string` (page 355) for that operation.

When using `db.getLastError()` (page 185), clients must issue the `db.getLastError()` (page 185) on the same connection as the write operation they wish to confirm.

Changed in version 2.6: A new protocol for *write operations* (page 1095) integrates write concerns with the write operations, eliminating the need for a separate `db.getLastError()` (page 185). *Most write methods* (page 1102) now return the status of the write operation, including error information. In previous versions, clients typically used the `db.getLastError()` (page 185) in combination with a write operation to verify that the write succeeded.

The `db.getLastError()` (page 185) can accept the following parameters:

param int, string w Optional. The write concern's `w` value.

param int wtimeout Optional. The time limit in milliseconds.

Behavior The returned `error string` (page 355) provides error information on the previous write operation.

If the `db.getLastError()` (page 185) method itself encounters an error, such as an incorrect write concern value, the `db.getLastError()` (page 185) throws an exception.

Example The following example issues a `db.getLastError()` (page 185) operation that verifies that the preceding write operation, issued over the same connection, has propagated to at least two members of the replica set.

```
db.getLastError(2)
```

See also:

`getLastError` (page 355) and `https://docs.mongodb.org/manual/reference/write-concern` for all options, *Write Concern* for a conceptual overview, `https://docs.mongodb.org/manual/core/write-operations` for information about all write operations in MongoDB.

db.getLastErrorObj()

On this page

- Definition (page 186)
- Behavior (page 186)
- Example (page 187)

Definition

`db.getLastErrorObj()`

Specifies the level of *write concern* for confirming the success of previous write operation issued over the same connection and returns the *document* (page 355) for that operation.

When using `db.getLastErrorObj()` (page 186), clients must issue the `db.getLastErrorObj()` (page 186) on the same connection as the write operation they wish to confirm.

The `db.getLastErrorObj()` (page 186) is a `mongo` (page 803) shell wrapper around the `getLastError` (page 355) command.

Changed in version 2.6: A new protocol for *write operations* (page 1095) integrates write concerns with the write operations, eliminating the need for a separate `db.getLastErrorObj()` (page 186). *Most write methods* (page 1102) now return the status of the write operation, including error information. In previous versions, clients typically used the `db.getLastErrorObj()` (page 186) in combination with a write operation to verify that the write succeeded.

The `db.getLastErrorObj()` (page 186) can accept the following parameters:

param int, string key Optional. The write concern's `w` value.

param int wtimeout Optional. The time limit in milliseconds.

Behavior The returned *document* (page 355) provides error information on the previous write operation.

If the `db.getLastErrorObj()` (page 186) method itself encounters an error, such as an incorrect write concern value, the `db.getLastErrorObj()` (page 186) throws an exception.

For information on the returned document, see *getLastError command* (page 355).

Example The following example issues a `db.getLastErrorObj()` (page 186) operation that verifies that the preceding write operation, issued over the same connection, has propagated to at least two members of the replica set.

```
db.getLastErrorObj(2)
```

See also:

https://docs.mongodb.org/manual/reference/write-concern.

db.getLogComponents()

On this page

- Definition (page 187)
- Output (page 187)

Definition

`db.getLogComponents()`

New in version 3.0.

Returns the current verbosity settings. The verbosity settings determine the amount of *Log Messages* (page 964) that MongoDB produces for each *log message component* (page 964).

If a component inherits the verbosity level of its parent, `db.getLogComponents()` (page 187) displays `-1` for the component's verbosity.

Output The `db.getLogComponents()` (page 187) 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` (page 897) and `systemLog.component.<name>.verbosity` settings in the *configuration file* (page 895) or set the `logComponentVerbosity` (page 934) parameter using the `setParameter` (page 460) command or use the `db.setLogLevel()` (page 197) method. For examples, see *Configure Log Verbosity Levels* (page 966).

db.getMongo()

db.**getMongo**()

Returns The current database connection.

`db.getMongo()` (page 188) runs when the shell initiates. Use this command to test that the `mongo` (page 803) 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 357) (also `db.resetError()` (page 196)) command.

This method provides a wrapper around the `getPrevError` (page 357) command.

db.getProfilingLevel()

db.**getProfilingLevel**()

This method provides a wrapper around the database command "`profile` (page 484)" and returns the current profiling level.

Deprecated since version 1.8.4: Use `db.getProfilingStatus()` (page 189) for related functionality.

db.getProfilingStatus()

`db.getProfilingStatus()`

Returns The current `profile` (page 484) level and `slowOpThresholdMs` (page 921) setting.

db.getReplicationInfo()

On this page

- Definition (page 189)
- Output (page 189)

Definition

`db.getReplicationInfo()`

Returns A document with the status of the replica set, 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 287).

Only present if there are entries in the oplog.

db.getSiblingDB()

On this page

- Definition (page 190)
- Example (page 190)

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 190) as an alternative to the `use <database>` helper. This is particularly useful when writing scripts using the `mongo` (page 803) 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 32) 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 32) and an *example document* (page 62) 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 770) or `mongos` (page 792) runs on. Some of the returned fields are only included on some platforms.

`db.hostInfo()` (page 191) provides a helper in the `mongo` (page 803) shell around the `hostInfo` (page 490) The output of `db.hostInfo()` (page 191) 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 490) for full documentation of the output of `db.hostInfo()` (page 191).

db.isMaster()

`db.isMaster()`

Returns A document that describes the role of the `mongod` (page 770) instance.

If the `mongod` (page 770) is a member of a *replica set*, then the `ismaster` (page 409) and `secondary` (page 410) fields report if the instance is the *primary* or if it is a *secondary* member of the replica set.

See

`isMaster` (page 409) for the complete documentation of the output of `db.isMaster()` (page 191).

db.killOp()

On this page

- Description (page 192)

Description

db.**killOp**(*opid*)

Terminates an operation as specified by the operation ID. To find operations and their corresponding IDs, see db.currentOp() (page 171).

The db.killOp() (page 192) method has the following parameter:

param number opid An operation ID.

Warning: Terminate running operations with extreme caution. Only use db.killOp() (page 192) 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 303) document for a more extensive index of these options.

db.loadServerScripts()

db.**loadServerScripts**()

db.loadServerScripts() (page 192) loads all scripts in the system.js collection for the current database into the mongo (page 803) 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 558) clauses and mapReduce (page 318) 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 192) 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 192) 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 192) against this database in order to successfully log out.

Example

Use the `use <database-name>` helper in the interactive `mongo` (page 803) shell, or the following `db.getSiblingDB()` (page 190) in the interactive shell or in `mongo` (page 803) 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 192) to log out of database as in the following operation:

```
db.logout()
```

`db.logout()` (page 192) function provides a wrapper around the database command `logout` (page 371).

db.printCollectionStats()

`db.printCollectionStats()`

Provides a wrapper around the `db.collection.stats()` (page 107) method. Returns statistics from every collection separated by three hyphen characters.

Note: The `db.printCollectionStats()` (page 193) in the `mongo` (page 803) shell does **not** return *JSON*. Use `db.printCollectionStats()` (page 193) for manual inspection, and `db.collection.stats()` (page 107) in scripts.

See also:

collStats (page 472)

db.printReplicationInfo()

On this page

- Definition (page 193)
- Output Example (page 194)
- Output Fields (page 194)

Definition

`db.printReplicationInfo()`

Prints a formatted report of the replica set member's *oplog*. The displayed report formats the data returned by `db.getReplicationInfo()` (page 189). [7]

The output of `db.printReplicationInfo()` (page 193) is identical to that of `rs.printReplicationInfo()` (page 259).

Note: The `db.printReplicationInfo()` (page 193) in the `mongo` (page 803) shell does **not** return *JSON*. Use `db.printReplicationInfo()` (page 193) for manual inspection, and `db.getReplicationInfo()` (page 189) in scripts.

[7] If run on a slave of a `master-slave replication`, the method calls `db.printSlaveReplicationInfo()` (page 195). See `db.printSlaveReplicationInfo()` (page 195) for details.

Output Example The following example is a sample output from the db.printReplicationInfo() (page 193) 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 193) formats and prints the data returned by db.getReplicationInfo() (page 189):

configured oplog size Displays the db.getReplicationInfo.logSizeMB (page 189) value.

log length start to end Displays the db.getReplicationInfo.timeDiff (page 189) and db.getReplicationInfo.timeDiffHours (page 189) values.

oplog first event time Displays the db.getReplicationInfo.tFirst (page 189).

oplog last event time Displays the db.getReplicationInfo.tLast (page 189).

now Displays the db.getReplicationInfo.now (page 189).

See db.getReplicationInfo() (page 189) for description of the data.

db.printShardingStatus()

On this page

- Definition (page 194)

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 194) when connected to a mongos (page 792) instance.

The db.printShardingStatus() (page 194) method has the following parameter:

param boolean verbose Optional. If true, the method displays details of the document distribution across chunks when you have 20 or more chunks.

See *sh.status()* (page 279) for details of the output.

Note: The db.printShardingStatus() (page 194) in the mongo (page 803) shell does **not** return *JSON*. Use db.printShardingStatus() (page 194) for manual inspection, and *Config Database* (page 885) in scripts.

See also:

sh.status() (page 279)

db.printSlaveReplicationInfo()

On this page

- Definition (page 195)
- Output (page 195)

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 260).

Output The following is example output from the `db.printSlaveReplicationInfo()` (page 195) 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 195) in the `mongo` (page 803) shell does **not** return *JSON*. Use `db.printSlaveReplicationInfo()` (page 195) for manual inspection, and `rs.status()` (page 262) in scripts.

A *delayed member* may show as `0` seconds behind the primary when the inactivity period on the primary is greater than the `members[n].slaveDelay` value.

db.repairDatabase()

On this page

- Behavior (page 195)

`db.repairDatabase()`

`db.repairDatabase()` (page 195) provides a wrapper around the database command `repairDatabase` (page 462), and has the same effect as the run-time option *`mongod --repair`* option, limited to *only* the current database. See `repairDatabase` (page 462) for full documentation.

Behavior

Warning: During normal operations, only use the `repairDatabase` (page 462) command and wrappers including `db.repairDatabase()` (page 195) in the `mongo` (page 803) 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 462).

When using *journaling*, there is almost never any need to run `repairDatabase` (page 462). 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 195) 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 188) or `getPrevError` (page 357). Provides a wrapper around the `resetError` (page 357) command.

db.runCommand()

On this page

- Definition (page 196)
- Behavior (page 196)

Definition

`db.runCommand(`*command*`)`

Provides a helper to run specified *database commands* (page 303). 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 196) transforms the string into a document."

New in version 2.6: To specify a time limit in milliseconds, see `https://docs.mongodb.org/manual/tutorial/terminate-running-operations`.

Behavior `db.runCommand()` (page 196) 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 471) *database command*. `buildInfo` (page 471) returns a document that contains an overview of parameters used to compile this `mongod` (page 770) instance.

db.serverCmdLineOpts()

`db.serverCmdLineOpts()`

Wraps the `getCmdLineOpts` (page 483) *database command*.

Returns a document that reports on the arguments and configuration options used to start the `mongod` (page 770) or `mongos` (page 792) instance.

See *Configuration File Options* (page 895), *mongod* (page 769), and *mongos* (page 791) for additional information on available MongoDB runtime options.

db.serverStatus()

On this page

- Behavior (page 197)

`db.serverStatus()`

Returns a *document* that provides an overview of the database process's state.

Changed in version 3.0: `db.serverStatus()` (page 197) no longer outputs the `workingSet`, `indexCounters`, and `recordStats` sections.

This command provides a wrapper around the database command `serverStatus` (page 492).

Behavior By default, `db.serverStatus()` (page 197) excludes in its output *rangeDeleter* (page 501) information and some content in the *repl* (page 502) document.

To include fields that are excluded by default, specify the top-level field and set it to `1` in the command. To exclude fields that are included by default, specify the top-level field and set to `0` in the command.

For example, the following operation suppresses the `repl`, `metrics` and `locks` information in the output.

```
db.serverStatus( { repl: 0,  metrics: 0, locks: 0 } )
```

The following example includes *rangeDeleter* (page 501) and all *repl* (page 502) information in the output:

```
db.serverStatus( { rangeDeleter: 1, repl: 1 } )
```

See also:

serverStatus (page 492) for complete documentation of the output of this function.

db.setLogLevel()

On this page

- Definition (page 197)
- Behavior (page 198)
- Examples (page 198)

Definition

`db.setLogLevel()`

New in version 3.0.

Sets a single verbosity level for *log messages* (page 964).

`db.setLogLevel()` (page 197) has the following form:

```
db.setLogLevel(<level>, <component>)
```

`db.setLogLevel()` (page 197) 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* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

To inherit the verbosity level of the component's parent, you can also specify `-1`.

param string component Optional. 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` (page 899)
- `command` (page 899)
- `control` (page 899)
- `geo` (page 899)
- `index` (page 900)
- `network` (page 900)
- `query` (page 900)
- `replication` (page 900)
- `sharding` (page 900)
- `storage` (page 901)
- `storage.journal` (page 901)
- `write` (page 901)

Omit to specify the default verbosity level for all components.

Behavior `db.setLogLevel()` (page 197) sets a *single* verbosity level. To set multiple verbosity levels in a single operation, use either the `setParameter` (page 460) command to set the `logComponentVerbosity` (page 934) parameter. You can also specify the verbosity settings in the *configuration file* (page 895). See *Configure Log Verbosity Levels* (page 966) for examples.

Examples

Set Default Verbosity Level Omit the `<component>` parameter to set the default verbosity for all components; i.e. the `systemLog.verbosity` (page 897) 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` (page 901) to `2`:

```
db.setLogLevel(2, "storage.journal" )
```

db.setProfilingLevel()

On this page

- Definition (page 199)

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 484).

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 Optional. 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` (page 921) option to set the threshold for the profiler to consider a query "slow." Specify this value in milliseconds to override the default, 100 ms.

`mongod` (page 770) writes the output of the database profiler to the `system.profile` collection.

`mongod` (page 770) prints information about queries that take longer than the `slowOpThresholdMs` (page 921) to the log even when the database profiler is not active.

db.shutdownServer()

`db.shutdownServer()`

Shuts down the current `mongod` (page 770) or `mongos` (page 792) 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 465).

db.stats()

On this page

- Description (page 199)
- Behavior (page 200)
- Example (page 200)

Description

`db.stats(`*scale*`)`

Returns statistics that reflect the use state of a single *database*.

The `db.stats()` (page 199) method has the following parameter:

param number scale Optional. 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 481).

The `db.stats()` (page 199) method is a wrapper around the `dbStats` (page 481) database command.

Behavior For MongoDB instances using the `WiredTiger` storage engine, after an unclean shutdown, statistics on size and count may off by up to 1000 documents as reported by `collStats` (page 473), `dbStats` (page 481), `count` (page 307). To restore the correct statistics for the collection, run `validate` (page 485) on the collection.

Example The following example converts the returned values to kilobytes:

```
db.stats(1024)
```

Note: The scale factor rounds values to whole numbers.

db.version()

`db.version()`

Returns The version of the `mongod` (page 770) or `mongos` (page 792) instance.

db.upgradeCheck()

On this page

- Definition (page 200)
- Behavior (page 201)
- Required Access (page 201)
- Example (page 201)
- Error Output (page 201)
- Warning Output (page 202)

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 803) 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 1100),
- documents with `illegal field names` (page 946),
- 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 Optional. 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 1099) for details.

See also:

`db.upgradeCheckAllDBs()` (page 202)

Behavior `db.upgradeCheck()` (page 200) 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 792).
- For replica sets, run the command on the secondary members.

`db.upgradeCheck()` (page 200) can miss new data during the check when run on a live system with active write operations.

For index validation, `db.upgradeCheck()` (page 200) only supports the check of version `1` indexes and skips the check of version `0` indexes.

The `db.upgradeCheck()` (page 200) checks all of the data stored in the `mongod` (page 770) instance: the time to run `db.upgradeCheck()` (page 200) depends on the quantity of data stored by `mongod` (page 770).

Required Access On systems running with `authorization` (page 910), a user must have access that includes the `find` action on all collections, including the *system collections* (page 892).

Example The following example connects to a secondary running on `localhost` and runs `db.upgradeCheck()` (page 200) 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()

On this page

- Definition (page 202)
- Behavior (page 203)
- Required Access (page 203)
- Example (page 203)
- Error Output (page 203)
- Warning Output (page 204)

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 803) 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 1100),
- documents with `illegal field names` (page 946),
- 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 1099) for details.

See also:

`db.upgradeCheck()` (page 200)

Behavior db.upgradeCheckAllDBs() (page 202) 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 792).
- For replica sets, run the command on the secondary members.

db.upgradeCheckAllDBs() (page 202) can miss new data during the check when run on a live system with active write operations.

For index validation, db.upgradeCheckAllDBs() (page 202) only supports the check of version 1 indexes and skips the check of version 0 indexes.

The db.upgradeCheckAllDBs() (page 202) checks all of the data stored in the mongod (page 770) instance: the time to run db.upgradeCheckAllDBs() (page 202) depends on the quantity of data stored by mongod (page 770).

Required Access On systems running with authorization (page 910), 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 892).

You *must* run the db.upgradeCheckAllDBs() (page 202) operation in the admin database.

Example The following example connects to a secondary running on localhost and runs db.upgradeCheckAllDBs() (page 202) 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.

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 204) method.

Name	Description
`db.collection.getPlanCache()` (page 204)	Returns an interface to access the query plan cache object and associated PlanCache methods for a collection."
`PlanCache.help()` (page 205)	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 205)	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 207)	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 208)	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 209)	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()`.

db.collection.getPlanCache()

On this page

- Definition (page 204)
- Methods (page 204)

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 205)	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 205)	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 207)	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 208)	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 209)	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.help()

On this page

- Definition (page 205)

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 204) of a specific collection; i.e.

```
db.collection.getPlanCache().help()
```

See also:

`db.collection.getPlanCache()` (page 204)

PlanCache.listQueryShapes()

On this page

- Definition (page 205)
- Required Access (page 206)
- Example (page 206)

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 204) of a specific collection; i.e.

```
db.collection.getPlanCache().listQueryShapes()
```

Returns Array of *query shape* documents.

The method wraps the `planCacheListQueryShapes` (page 367) command.

Required Access On systems running with `authorization` (page 910), 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" : { }
  }
]
```

Note: Not all queries automatically place a query plan in the cache. `db.collection.getPlanCache().listQueryShapes()` returns an empty array if there are currently no query shapes with cached query plans.

See also:

- `db.collection.getPlanCache()` (page 204)
- `PlanCache.getPlansByQuery()` (page 207)
- `PlanCache.help()` (page 205)
- `planCacheListQueryShapes` (page 367)

PlanCache.getPlansByQuery()

On this page

- Definition (page 207)
- Required Access (page 207)
- Example (page 207)

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 204) of a specific collection; i.e.

```
db.collection.getPlanCache().getPlansByQuery( <query>, <projection>, <sort> )
```

The `PlanCache.getPlansByQuery()` (page 207) 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 Optional. The projection associated with the *query shape*. Required if specifying the `sort` parameter.

param document sort Optional. 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 205) method.

Required Access On systems running with `authorization` (page 910), 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 204)
- `PlanCache.listQueryShapes()` (page 205)

- PlanCache.help() (page 205)

PlanCache.clearPlansByQuery()

On this page

- Definition (page 208)
- Required Access (page 208)
- Example (page 208)

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 204) of a specific collection; i.e.

```
db.collection.getPlanCache().clearPlansByQuery( <query>, <projection>, <sort> )
```

The `PlanCache.clearPlansByQuery()` (page 208) 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 Optional. The projection associated with the *query shape*. Required if specifying the `sort` parameter.

param document sort Optional. The sort associated with the *query shape*.

To see the query shapes for which cached query plans exist, use the `PlanCache.listQueryShapes()` (page 205) method.

Required Access On systems running with `authorization` (page 910), 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 204)
- `PlanCache.listQueryShapes()` (page 205)

- PlanCache.clear() (page 209)

PlanCache.clear()

On this page

- Definition (page 209)
- Required Access (page 209)

Definition

`PlanCache.clear()`

Removes all cached query plans for a collection.

The method is only available from the `plan cache object` (page 204) 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` (page 910), a user must have access that includes the `planCacheWrite` action.

See also:

- `db.collection.getPlanCache()` (page 204)
- `PlanCache.clearPlansByQuery()` (page 208)

2.1.5 Bulk Write Operation

Bulk Operation Methods

New in version 2.6.

Name	Description
`Bulk()` (page 210)	Bulk operations builder.
`db.collection.initializeOrderedBulkOp()` (page 212)	Initializes a `Bulk()` (page 210) operations builder for an ordered list of operations.
`db.collection.initializeUnorderedBulkOp()` (page 213)	Initializes a `Bulk()` (page 210) operations builder for an unordered list of operations.
`Bulk.insert()` (page 214)	Adds an insert operation to a list of operations.
`Bulk.find()` (page 215)	Specifies the query condition for an update or a remove operation.
`Bulk.find.removeOne()` (page 216)	Adds a single document remove operation to a list of operations.
`Bulk.find.remove()` (page 217)	Adds a multiple document remove operation to a list of operations.
`Bulk.find.replaceOne()` (page 217)	Adds a single document replacement operation to a list of operations.
`Bulk.find.updateOne()` (page 218)	Adds a single document update operation to a list of operations.
`Bulk.find.update()` (page 220)	Adds a `multi` update operation to a list of operations.
`Bulk.find.upsert()` (page 221)	Specifies `upsert: true` for an update operation.
`Bulk.execute()` (page 223)	Executes a list of operations in bulk.
`Bulk.getOperations()` (page 226)	Returns an array of write operations executed in the `Bulk()` (page 210) operations object.
`Bulk.tojson()` (page 227)	Returns a JSON document that contains the number of operations and batches in the `Bulk()` (page 210) operations object.
`Bulk.toString()` (page 228)	Returns the `Bulk.tojson()` (page 227) results as a string.

Bulk()

On this page

- Description (page 210)
- Ordered and Unordered Bulk Operations (page 210)
- Methods (page 211)

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 212) or the `db.collection.initializeUnorderedBulkOp()` (page 213) 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 212) to create a builder for an ordered list of write commands.

When executing an `ordered` (page 212) list of operations, MongoDB groups the operations by the `operation type` (page 227) 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 945). If a group exceeds this `limit` (page 945), 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 226) *after* the execution.

Executing an `ordered` (page 212) list of operations on a sharded collection will generally be slower than executing an `unordered` (page 213) 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 213) to create a builder for an unordered list of write commands.

When executing an `unordered` (page 213) 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 213) bulk operations.

Each group of operations can have at most `1000 operations` (page 945). If a group exceeds this `limit` (page 945), 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 226) *after* the execution.

Methods The `Bulk()` (page 210) builder has the following methods:

Name	Description
`Bulk.insert()` (page 214)	Adds an insert operation to a list of operations.
`Bulk.find()` (page 215)	Specifies the query condition for an update or a remove operation.
`Bulk.find.removeOne()` (page 216)	Adds a single document remove operation to a list of operations.
`Bulk.find.remove()` (page 217)	Adds a multiple document remove operation to a list of operations.
`Bulk.find.replaceOne()` (page 217)	Adds a single document replacement operation to a list of operations.
`Bulk.find.updateOne()` (page 218)	Adds a single document update operation to a list of operations.
`Bulk.find.update()` (page 220)	Adds a `multi` update operation to a list of operations.
`Bulk.find.upsert()` (page 221)	Specifies `upsert: true` for an update operation.
`Bulk.execute()` (page 223)	Executes a list of operations in bulk.
`Bulk.getOperations()` (page 226)	Returns an array of write operations executed in the `Bulk()` (page 210) operations object.
`Bulk.tojson()` (page 227)	Returns a JSON document that contains the number of operations and batches in the `Bulk()` (page 210) operations object.
`Bulk.toString()` (page 228)	Returns the `Bulk.tojson()` (page 227) results as a string.

db.collection.initializeOrderedBulkOp()

On this page

- Definition (page 212)
- Behavior (page 212)
- Examples (page 213)

Definition

`db.collection.`**`initializeOrderedBulkOp`**`()`

Initializes and returns a new `Bulk()` (page 210) operations builder for a collection. The builder constructs an ordered list of write operations that MongoDB executes in bulk.

Returns new `Bulk()` (page 210) 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 212) list of operations, MongoDB groups the operations by the `operation type` (page 227) 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 945). If a group exceeds this `limit` (page 945), 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 226) *after* the execution.

Executing an `ordered` (page 212) list of operations on a sharded collection will generally be slower than executing an `unordered` (page 213) 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 210) 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 213)
- `Bulk.find()` (page 215)
- `Bulk.find.removeOne()` (page 216)
- `Bulk.execute()` (page 223)

db.collection.initializeUnorderedBulkOp()

On this page

- Definition (page 213)
- Behavior (page 213)
- Example (page 214)

Definition

`db.collection.`**`initializeUnorderedBulkOp`**`()`

New in version 2.6.

Initializes and returns a new `Bulk()` (page 210) 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 212) instead.

Execution of Operations When executing an `unordered` (page 213) 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 213) bulk operations.

Each group of operations can have at most `1000 operations` (page 945). If a group exceeds this `limit` (page 945), 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 226) *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 210) 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 212)
- `Bulk()` (page 210)
- `Bulk.insert()` (page 214)
- `Bulk.execute()` (page 223)

Bulk.insert()

On this page

- Description (page 214)
- Example (page 215)

Description

`Bulk.insert(<document>)`

New in version 2.6.

Adds an insert operation to a bulk operations list.

`Bulk.insert()` (page 214) 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 940).

Example The following initializes a `Bulk()` (page 210) 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 213)
- `db.collection.initializeOrderedBulkOp()` (page 212)
- `Bulk.execute()` (page 223)

Bulk.find()

On this page

- Description (page 215)
- Example (page 216)

Description

`Bulk.find(<query>)`

New in version 2.6.

Specifies a query condition for an update or a remove operation.

`Bulk.find()` (page 215) accepts the following parameter:

param document query Specifies a query condition using *Query Selectors* (page 527) 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 940).

With remove operations, the query document must be less than or equal to the `maximum BSON document size` (page 940).

Use `Bulk.find()` (page 215) with the following write operations:

- `Bulk.find.removeOne()` (page 216)
- `Bulk.find.remove()` (page 217)
- `Bulk.find.replaceOne()` (page 217)
- `Bulk.find.updateOne()` (page 218)
- `Bulk.find.update()` (page 220)

Example The following example initializes a `Bulk()` (page 210) 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 215) 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 213)
- `db.collection.initializeOrderedBulkOp()` (page 212)
- `Bulk.execute()` (page 223)

Bulk.find.removeOne()

On this page

- Description (page 216)
- Example (page 216)

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 215) method to specify the condition that determines which document to remove. The `Bulk.find.removeOne()` (page 216) limits the removal to one document. To remove multiple documents, see `Bulk.find.remove()` (page 217).

Example The following example initializes a `Bulk()` (page 210) operations builder for the `items` collection and adds two `Bulk.find.removeOne()` (page 216) 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 213)
- `db.collection.initializeOrderedBulkOp()` (page 212)
- `Bulk.find()` (page 215)
- `Bulk.find.remove()` (page 217)
- `Bulk.execute()` (page 223)
- *All Bulk Methods* (page 211)

Bulk.find.remove()

On this page

- Description (page 217)
- Example (page 217)

Description

`Bulk.find.remove()`

New in version 2.6.

Adds a remove operation to a bulk operations list. Use the `Bulk.find()` (page 215) method to specify the condition that determines which documents to remove. The `Bulk.find.remove()` (page 217) method removes all matching documents in the collection. To limit the remove to a single document, see `Bulk.find.removeOne()` (page 216).

Example The following example initializes a `Bulk()` (page 210) 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 213)
- `db.collection.initializeOrderedBulkOp()` (page 212)
- `Bulk.find()` (page 215)
- `Bulk.find.removeOne()` (page 216)
- `Bulk.execute()` (page 223)

Bulk.find.replaceOne()

On this page

- Description (page 217)
- Example (page 218)

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 215) method to specify the condition that determines which document to replace. The `Bulk.find.replaceOne()` (page 217) method limits the replacement to a single document.

`Bulk.find.replaceOne()` (page 217) 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 215) and the replacement document must be less than or equal to the `maximum BSON document size` (page 940).

To specify an *upsert* for this operation, see `Bulk.find.upsert()` (page 221).

Example The following example initializes a `Bulk()` (page 210) operations builder for the `items` collection, and adds various `replaceOne` (page 217) 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 213)
- `db.collection.initializeOrderedBulkOp()` (page 212)
- `Bulk.find()` (page 215)
- `Bulk.execute()` (page 223)
- *All Bulk Methods* (page 211)

Bulk.find.updateOne()

On this page

- Description (page 218)
- Behavior (page 219)
- Example (page 219)

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 215) method to specify the condition that determines which document to update. The `Bulk.find.updateOne()` (page 218) method limits the update or replacement to a single document. To update multiple documents, see `Bulk.find.update()` (page 220).

`Bulk.find.updateOne()` (page 218) 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 595) expressions. A replacement document contains only field and value pairs.

The sum of the associated `<query>` document from the `Bulk.find()` (page 215) 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 221).

Behavior

Update Specific Fields If the `<update>` document contains only *update operator* (page 595) expressions, as in:

```
{
  $set: { status: "D" },
  $inc: { points: 2 }
}
```

Then, `Bulk.find.updateOne()` (page 218) 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 218) *replaces* the matching document with the `<update>` document with the exception of the `_id` field. The `Bulk.find.updateOne()` (page 218) method *does not* replace the `_id` value.

Example The following example initializes a `Bulk()` (page 210) operations builder for the `items` collection, and adds various `updateOne` (page 218) 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 213)
- `db.collection.initializeOrderedBulkOp()` (page 212)
- `Bulk.find()` (page 215)
- `Bulk.find.update()` (page 220)
- `Bulk.execute()` (page 223)
- *All Bulk Methods* (page 211)

Bulk.find.update()

On this page

- Description (page 220)
- Example (page 220)

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 215) method to specify the condition that determines which documents to update. The `Bulk.find.update()` (page 220) method updates all matching documents. To specify a single document update, see `Bulk.find.updateOne()` (page 218).

`Bulk.find.update()` (page 220) accepts the following parameter:

param document update Specifies the fields to update. Only contains *update operator* (page 595) expressions.

The sum of the associated `<query>` document from the `Bulk.find()` (page 215) and the update document must be less than or equal to the `maximum BSON document size` (page 940).

To specify *upsert: true* for this operation, see `Bulk.find.upsert()` (page 221). With `Bulk.find.upsert()` (page 221), if no documents match the `Bulk.find()` (page 215) query condition, the update operation inserts only a single document.

Example The following example initializes a `Bulk()` (page 210) 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 213)
- `db.collection.initializeOrderedBulkOp()` (page 212)
- `Bulk.find()` (page 215)
- `Bulk.find.updateOne()` (page 218)
- `Bulk.execute()` (page 223)
- *All Bulk Methods* (page 211)

Bulk.find.upsert()

On this page

- Description (page 221)
- Behavior (page 221)

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 215) 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 221) with the following write operations:

- `Bulk.find.replaceOne()` (page 217)
- `Bulk.find.updateOne()` (page 218)
- `Bulk.find.update()` (page 220)

Behavior The following describe the insert behavior of various write operations when used in conjunction with `Bulk.find.upsert()` (page 221).

Insert for `Bulk.find.replaceOne()` The `Bulk.find.replaceOne()` (page 217) 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 221) 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 218) method accepts, as its parameter, an `<update>` document that contains only field and value pairs or only *update operator* (page 595) 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 221) 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 595) 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 221) option performs an insert, the update operation inserts a document with field and values from the `<query>` document of the `Bulk.find()` (page 215) 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 221) with the multiple document update method `Bulk.find.update()` (page 220), if no documents match the query condition, the update operation inserts a *single* document.

The `Bulk.find.update()` (page 220) method accepts, as its parameter, an `<update>` document that contains *only update operator* (page 595) 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 221) 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 215) 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 213)
- `db.collection.initializeOrderedBulkOp()` (page 212)
- `Bulk.find()` (page 215)
- `Bulk.execute()` (page 223)
- *All Bulk Methods* (page 211)

Bulk.execute()

On this page

- Description (page 223)
- Behavior (page 224)
- Examples (page 224)

Description

`Bulk.execute()`

New in version 2.6.

Executes the list of operations built by the `Bulk()` (page 210) operations builder.

`Bulk.execute()` (page 223) accepts the following parameter:

param document writeConcern Optional. `Write concern` document for the bulk operation as a whole. Omit to use default. For a standalone `mongod` (page 770) 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 225) for an example.

Returns A `BulkWriteResult` (page 291) object that contains the status of the operation.

After execution, you cannot re-execute the `Bulk()` (page 210) object without reinitializing. See `db.collection.initializeUnorderedBulkOp()` (page 213) and `db.collection.initializeOrderedBulkOp()` (page 212).

Behavior

Ordered Operations When executing an `ordered` (page 212) list of operations, MongoDB groups the operations by the `operation type` (page 227) 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 945). If a group exceeds this `limit` (page 945), 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 226) *after* the execution.

Executing an `ordered` (page 212) list of operations on a sharded collection will generally be slower than executing an `unordered` (page 213) list since with an ordered list, each operation must wait for the previous operation to finish.

Unordered Operations When executing an `unordered` (page 213) 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 213) bulk operations.

Each group of operations can have at most `1000 operations` (page 945). If a group exceeds this `limit` (page 945), 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 226) *after* the execution.

Examples

Execute Bulk Operations The following initializes a `Bulk()` (page 210) 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 291) object:

```
BulkWriteResult({
   "writeErrors" : [ ],
   "writeConcernErrors" : [ ],
   "nInserted" : 2,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

For details on the return object, see `BulkWriteResult()` (page 291). For details on the batches executed, see `Bulk.getOperations()` (page 226).

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 voting replica set members or the method times out after 5 seconds.

Changed in version 3.0: In previous versions, `majority` referred to the majority of all members of the replica set.

```
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 291) object:

```
BulkWriteResult({
   "writeErrors" : [ ],
   "writeConcernErrors" : [ ],
   "nInserted" : 2,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

See

`Bulk()` (page 210) for a listing of methods available for bulk operations.

Bulk.getOperations()

On this page

- Example (page 226)
- Returned Fields (page 227)

`Bulk.`**`getOperations`**`()`

New in version 2.6.

Returns an array of write operations executed through `Bulk.execute()` (page 223). 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 224).

Only use `Bulk.getOperations()` (page 226) after a `Bulk.execute()` (page 223). Calling `Bulk.getOperations()` (page 226) before you call `Bulk.execute()` (page 223) will result in an *incomplete* list.

Example The following initializes a `Bulk()` (page 210) operations builder on the `items` collection, adds a series of write operations, executes the operations, and then calls `getOperations()` (page 226) 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 226) 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 224)

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 227) 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 210) and `Bulk.execute()` (page 223).

Bulk.tojson()

On this page

- Example (page 227)

`Bulk.tojson()`
New in version 2.6.

Returns a JSON document that contains the number of operations and batches in the `Bulk()` (page 210) object.

Example The following initializes a `Bulk()` (page 210) operations builder on the `items` collection, adds a series of write operations, and calls `Bulk.tojson()` (page 227) 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 227) returns the following JSON document

{ “nInsertOps” : 2, “nUpdateOps” : 0, “nRemoveOps” : 1, “nBatches” : 2 }

See also:

`Bulk()` (page 210)

Bulk.toString()

On this page

- Example (page 228)

`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 210) object.

Example The following initializes a `Bulk()` (page 210) operations builder on the `items` collection, adds a series of write operations, and calls `Bulk.toString()` (page 228) 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 228) returns the following JSON document

{ “nInsertOps” : 2, “nUpdateOps” : 0, “nRemoveOps” : 1, “nBatches” : 2 }

See also:

`Bulk()` (page 210)

2.1.6 User Management

User Management Methods

Name	Description
`db.auth()` (page 229)	Authenticates a user to a database.
`db.createUser()` (page 230)	Creates a new user.
`db.updateUser()` (page 232)	Updates user data.
`db.changeUserPassword()` (page 234)	Changes an existing user’s password.
`db.removeUser()` (page 235)	Deprecated. Removes a user from a database.
`db.dropAllUsers()` (page 235)	Deletes all users associated with a database.
`db.dropUser()` (page 236)	Removes a single user.
`db.grantRolesToUser()` (page 237)	Grants a role and its privileges to a user.
`db.revokeRolesFromUser()` (page 238)	Removes a role from a user.
`db.getUser()` (page 239)	Returns information about the specified user.
`db.getUsers()` (page 240)	Returns information about all users associated with a database.

db.auth()

On this page

- Definition (page 229)

Definition

`db.auth()`

Allows a user to authenticate to the database from within the shell.

The `db.auth()` (page 229) method can accept either:

- the username and password.

```
db.auth( <username>, <password> )
```

- a user document that contains the username and password, and optionally, the authentication mechanism and a digest password flag.

```
db.auth( {
   user: <username>,
   pwd: <password>,
   mechanism: <authentication mechanism>,
   digestPassword: <boolean>
} )
```

param string username Specifies an existing username with access privileges for this database.

param string password Specifies the corresponding password.

param string mechanism Optional. Specifies the *authentication mechanism* (page 805) used. Defaults to either:

- `SCRAM-SHA-1` on new 3.0 installations and on 3.0 databases that have been *upgraded from 2.6 with authSchemaUpgrade* (page 1058); or
- `MONGODB-CR` otherwise.

Changed in version 3.0: In previous version, defaulted to `MONGODB-CR`.

For available mechanisms, see *authentication mechanisms* (page 805).

param boolean digestPassword Optional. Determines whether the server receives digested or undigested password. Set to false to specify undigested password. For use with `SASL/LDAP authentication` since the server must forward an undigested password to `saslauthd`.

Alternatively, you can use *`mongo --username`*, *`--password`*, and *`--authenticationMechanism`* to specify authentication credentials.

Note: The `mongo` (page 803) shell excludes all `db.auth()` (page 229) operations from the saved history.

Returns `db.auth()` (page 229) returns `0` when authentication is **not** successful, and `1` when the operation is successful.

db.createUser()

On this page

- Definition (page 230)
- Behavior (page 230)
- Required Access (page 231)
- Examples (page 231)

Definition

`db.createUser(user, writeConcern)`

Creates a new user for the database where the method runs. `db.createUser()` (page 230) returns a *duplicate user* error if the user already exists on the database.

The `db.createUser()` (page 230) method has the following syntax:

field document user The document with authentication and access information about the user to create.

field document writeConcern Optional. The level of `write concern` for the creation operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 230) on the `$external` database to create users who have credentials stored externally to MongoDB.

field document customData Optional. 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 230) 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 230) method wraps the `createUser` (page 373) command.

Behavior

Encryption `db.createUser()` (page 230) sends password to the MongoDB instance *without* encryption. To encrypt the password during transmission, use `TLS/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

- To create a new user in a database, you must have `createUser` *action* on that *database resource*.
- To grant roles to a user, you must have the `grantRole` *action* on the role's database.

Built-in roles `userAdmin` and `userAdminAnyDatabase` provide `createUser` and `grantRole` actions on their respective `resources`.

Examples The following `db.createUser()` (page 230) 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.updateUser()

On this page

- Definition (page 232)
- Behavior (page 233)
- Required Access (page 233)
- Example (page 233)

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 237) or `db.revokeRolesFromUser()` (page 238) methods.

The `db.updateUser()` (page 232) 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 232) 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.

param document writeConcern Optional. The level of `write concern` for the update operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 Optional. Any arbitrary information.

field array roles Optional. The roles granted to the user. An update to the `roles` array overrides the previous array's values.

field string pwd Optional. 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 232) 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 232) method wraps the `updateUser` (page 375) command.

Behavior `db.updateUser()` (page 232) sends password to the MongoDB instance *without* encryption. To encrypt the password during transmission, use `TLS/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 232) 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"
       }
   ]
}
```

db.changeUserPassword()

On this page

- Definition (page 234)
- Required Access (page 235)
- Example (page 235)

Definition

`db.changeUserPassword(`*username*, *password*`)`

Updates a user's password. Run the method in the database where the user is defined, i.e. the database you `created` (page 230) the user.

param string username Specifies an existing username with access privileges for this database.

param string password Specifies the corresponding password.

param string mechanism Optional. Specifies the *authentication mechanism* (page 805) used. Defaults to either:

- `SCRAM-SHA-1` on new 3.0 installations and on 3.0 databases that have been *upgraded from 2.6 with authSchemaUpgrade* (page 1058); or
- `MONGODB-CR` otherwise.

Changed in version 3.0: In previous version, defaulted to MONGODB-CR.

For available mechanisms, see *authentication mechanisms* (page 805).

param boolean digestPassword Optional. Determines whether the server receives digested or undigested password. Set to false to specify undigested password. For use with SASL/LDAP authentication since the server must forward an undigested password to saslauthd.

Required Access To modify the password of another user on a database, you must have the changeAnyPassword *action* on that database.

Example The following operation changes the password of the user named accountUser in the products database to SOh3TbYhx8ypJPxmt1oOfL:

```
use products
db.changeUserPassword("accountUser", "SOh3TbYhx8ypJPxmt1oOfL")
```

db.removeUser()

On this page

- Definition (page 235)

Deprecated since version 2.6: Use db.dropUser() (page 236) instead of db.removeUser() (page 235)

Definition

db.**removeUser**(*username*)

Removes the specified username from the database.

The db.removeUser() (page 235) method has the following parameter:

param string username The database username.

db.dropAllUsers()

On this page

- Definition (page 235)
- Required Access (page 236)
- Example (page 236)

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 Optional. The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) command.

The `db.dropAllUsers()` (page 235) method wraps the `dropAllUsersFromDatabase` (page 378) 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 235) 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()

On this page

- Definition (page 236)
- Required Access (page 236)
- Example (page 236)

Definition

`db.dropUser(`*username*, *writeConcern*`)`

Removes the user from the current database.

The `db.dropUser()` (page 236) method takes the following arguments:

param string username The name of the user to remove from the database.

param document writeConcern Optional. The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) command.

The `db.dropUser()` (page 236) method wraps the `dropUser` (page 377) command.

Before dropping a user who has the `userAdminAnyDatabase` role, ensure you have at least another user with user administration privileges.

Required Access You must have the `dropUser` *action* on a database to drop a user from that database.

Example The following `db.dropUser()` (page 236) operation drops the `reportUser1` user on the `products` database.

```
use products
db.dropUser("reportUser1", {w: "majority", wtimeout: 5000})
```

db.grantRolesToUser()

On this page

- Definition (page 237)
- Required Access (page 237)
- Example (page 237)

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.

param array roles An array of additional roles to grant to the user.

param document writeConcern Optional. The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 237) 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 237) method wraps the `grantRolesToUser` (page 378) 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.revokeRolesFromUser()

On this page

- Definition (page 238)
- Required Access (page 239)
- Example (page 239)

Definition

`db.`**`revokeRolesFromUser`**`()`

Removes a one or more roles from a user on the current database. The `db.revokeRolesFromUser()` (page 238) 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.

param array roles The roles to remove from the user.

param document writeConcern Optional. The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 238) 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 238) method wraps the `revokeRolesFromUser` (page 380) 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 238) 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.getUser()

On this page

- Definition (page 239)
- Required Access (page 240)
- Example (page 240)

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 239) method has the following parameter:

param string username The name of the user for which to retrieve information.

`db.getUser()` (page 239) wraps the `usersInfo` (page 381) command.

Required Access To view another user's information, you must have the `viewUser` *action* on the other user's database.

Users can view their 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()

On this page

- Definition (page 240)
- Required Access (page 240)

Definition

`db.getUsers()`

Returns information for all the users in the database.

`db.getUsers()` (page 240) wraps the `usersInfo` (page 381) command.

Required Access To view another user's information, you must have the `viewUser` *action* on the other user's database.

Users can view their own information.

2.1.7 Role Management

Role Management Methods

Name	Description
`db.createRole()` (page 241)	Creates a role and specifies its privileges.
`db.updateRole()` (page 243)	Updates a user-defined role.
`db.dropRole()` (page 245)	Deletes a user-defined role.
`db.dropAllRoles()` (page 246)	Deletes all user-defined roles associated with a database.
`db.grantPrivilegesToRole()` (page 247)	Assigns privileges to a user-defined role.
`db.revokePrivilegesFromRole()` (page 248)	Removes the specified privileges from a user-defined role.
`db.grantRolesToRole()` (page 250)	Specifies roles from which a user-defined role inherits privileges.
`db.revokeRolesFromRole()` (page 251)	Removes inherited roles from a role.
`db.getRole()` (page 253)	Returns information for the specified role.
`db.getRoles()` (page 254)	Returns information for all the user-defined roles in a database.

db.createRole()

On this page

- Definition (page 241)
- Behavior (page 242)
- Required Access (page 242)
- Example (page 242)

Definition

`db.createRole(`*role*, *writeConcern*`)`

Creates a role in a database. You can specify privileges for the role by explicitly listing the privileges or by having the role inherit privileges from other roles or both. The role applies to the database on which you run the method.

The `db.createRole()` (page 241) method takes the following arguments:

param document role A document containing the name of the role and the role definition.

param document writeConcern Optional. The level of `write concern` to apply to this operation. The `writeConcern` document uses the same fields as the `getLastError` (page 355) 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. For the syntax of a privilege, see the `privileges` array.

You must include the `privileges` field. Use an empty array to specify *no* privileges.

field array roles An array of roles from which this role inherits privileges.

You must include the `roles` field. Use an empty array to specify *no* roles to inherit from.

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 241) 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 241) method wraps the `createRole` (page 383) command.

Behavior Except for roles created in the `admin` database, a role can only include privileges that apply to its database and can only inherit from other roles in its database.

A role created in the `admin` database can include privileges that apply to the `admin` database, other databases or to the *cluster* resource, and can inherit from roles in other databases as well as the `admin` database.

The `db.createRole()` (page 241) method returns a *duplicate role* error if the role already exists in the database.

Required Access To create a role in a database, you 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 241) 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", "re
       { resource: { db: "", collection: "" }, actions: [ "find" ] }
```

```
      ],
      roles: [
        { role: "read", db: "admin" }
      ]
    },
    { w: "majority" , wtimeout: 5000 }
)
```

db.updateRole()

On this page

- Definition (page 243)
- Behavior (page 244)
- Required Access (page 244)
- Example (page 244)

Definition

`db.updateRole`(*rolename*, *update*, *writeConcern*)

Updates a *user-defined role*. The `db.updateRole()` (page 243) 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 250)
- `db.grantPrivilegesToRole()` (page 247)
- `db.revokeRolesFromRole()` (page 251)
- `db.revokePrivilegesFromRole()` (page 248)

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 243) 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.

param document writeConcern Optional. The level of `write concern` for the update operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 Optional. 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 Optional. 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 243) 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 243) method wraps the `updateRole` (page 385) command.

Behavior Except for roles created in the `admin` database, a role can only include privileges that apply to its database and can only inherit from other roles in its database.

A role created in the `admin` database can include privileges that apply to the `admin` database, other databases or to the *cluster* resource, and can inherit from roles in other databases as well as the `admin` database.

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 243) 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 395) command.

db.dropRole()

On this page

- Definition (page 245)
- Required Access (page 245)
- Example (page 245)

Definition

`db.dropRole(`*rolename*, *writeConcern*`)`

Deletes a *user-defined* role from the database on which you run the method.

The `db.dropRole()` (page 245) method takes the following arguments:

param string rolename The name of the *user-defined role* to remove from the database.

param document writeConcern Optional. The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) command.

The `db.dropRole()` (page 245) method wraps the `dropRole` (page 387) 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.dropAllRoles()

On this page

- Definition (page 246)
- Required Access (page 246)
- Example (page 246)

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 Optional. The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) command.

Returns The number of *user-defined* roles dropped.

The `db.dropAllRoles()` (page 246) method wraps the `dropAllRolesFromDatabase` (page 387) 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.grantPrivilegesToRole()

On this page

- Definition (page 247)
- Behavior (page 247)
- Required Access (page 247)
- Example (page 247)

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.

param array privileges The privileges to add to the role. For the format of a privilege, see `privileges`.

param document writeConcern Optional. The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 355) command.

The `grantPrivilegesToRole()` method can grant one or more privileges. Each `<privilege>` has the following syntax:

```
{ resource: { <resource> }, actions: [ "<action>", ... ] }
```

The `db.grantPrivilegesToRole()` (page 247) method wraps the `grantPrivilegesToRole` (page 388) command.

Behavior Except for roles created in the `admin` database, a role can only include privileges that apply to its database

A role created in the `admin` database can include privileges that apply to the `admin` database, other databases or to the *cluster* resource.

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 247) 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.js" },
      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 892). 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.js` (page 893).

db.revokePrivilegesFromRole()

On this page

- Definition (page 248)
- Behavior (page 248)
- Required Access (page 250)
- Example (page 250)

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.

param array privileges An array of privileges to remove from the role. See `privileges` for more information on the format of the privileges.

param document writeConcern Optional. The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 355) command.

The `db.revokePrivilegesFromRole()` (page 248) method wraps the `revokePrivilegesFromRole` (page 390) 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.grantRolesToRole()

On this page

- Definition (page 250)
- Behavior (page 251)
- Required Access (page 251)
- Example (page 251)

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.

param array roles An array of roles from which to inherit.

param document writeConcern Optional. The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 250) 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 250) method wraps the grantRolesToRole (page 392) 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.revokeRolesFromRole()

On this page

- Definition (page 251)
- Required Access (page 252)
- Example (page 252)

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.

param array roles The inherited roles to remove.

param document writeConcern Optional. The level of `write concern` to apply to this operation. The `writeConcern` document uses the same fields as the `getLastError` (page 355) 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 251) 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 251) method wraps the `revokeRolesFromRole` (page 393) 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 251) 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.getRole()

On this page

- Definition (page 253)
- Required Access (page 253)
- Examples (page 253)

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 253) 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 253) method takes the following arguments:

param string rolename The name of the role.

param document showPrivileges Optional. If `true`, returns the role's privileges. Pass this argument as a document: `{showPrivileges: true}`.

`db.getRole()` (page 253) wraps the `rolesInfo` (page 395) command.

Required Access To view a role's information, you must be either 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()

On this page

- Definition (page 254)
- Required Access (page 254)
- Example (page 254)

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 254) returns inheritance information for the database's *user-defined* roles.

To return more information, pass the `db.getRoles()` (page 254) a document with the following fields:

field integer rolesInfo Set this field to `1` to retrieve all user-defined roles.

field boolean showPrivileges Optional. 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 Optional. Set to true to display *built-in roles* as well as user-defined roles.

`db.getRoles()` (page 254) wraps the `rolesInfo` (page 395) command.

Required Access To view a role's information, you must be either 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
    }
)
```

2.1.8 Replication

Replication Methods

Name	Description
`rs.add()` (page 255)	Adds a member to a replica set.
`rs.addArb()` (page 256)	Adds an *arbiter* to a replica set.
`rs.conf()` (page 257)	Returns the replica set configuration document.
`rs.freeze()` (page 258)	Prevents the current member from seeking election as primary for a period of time.
`rs.help()` (page 258)	Returns basic help text for *replica set* functions.
`rs.initiate()` (page 258)	Initializes a new replica set.
`rs.printReplicationInfo()` (page 259)	Prints a report of the status of the replica set from the perspective of the primary.
`rs.printSlaveReplicationInfo()` (page 260)	Prints a report of the status of the replica set from the perspective of the secondaries.
`rs.reconfig()` (page 260)	Re-configures a replica set by applying a new replica set configuration object.
`rs.remove()` (page 262)	Remove a member from a replica set.
`rs.slaveOk()` (page 262)	Sets the `slaveOk` property for the current connection. Deprecated. Use `readPref()` (page 152) and `Mongo.setReadPref()` (page 294) to set *read preference*.
`rs.status()` (page 262)	Returns a document with information about the state of the replica set.
`rs.stepDown()` (page 263)	Causes the current *primary* to become a secondary which forces an *election*.
`rs.syncFrom()` (page 264)	Sets the member that this replica set member will sync from, overriding the default sync target selection logic.

rs.add()

On this page

- Definition (page 255)
- Behavior (page 256)
- Example (page 256)

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 256) for an example.

If a document, specify a replica set member configuration document as found in the `members` array. You must specify `members[n]._id` and the `members[n].host` fields in the member configuration document. See *Pass a Member Configuration Document to rs.add()* (page 256) for an example.

See `https://docs.mongodb.org/manual/reference/replica-configuration` document for full documentation of all replica set configuration options

param boolean arbiterOnly Optional. Applies only if the `<host>` value is a string. If `true`, the added host is an arbiter.

`rs.add()` (page 255) provides a wrapper around some of the functionality of the `replSetReconfig` (page 404) *database command* and the corresponding `mongo` (page 803) shell helper `rs.reconfig()` (page 260). See the `https://docs.mongodb.org/manual/reference/replica-configuration` document for full documentation of all replica set configuration options.

Behavior `rs.add()` (page 255) can, in some cases, trigger an election for primary which will disconnect the shell. In such cases, the `mongo` (page 803) shell displays an error even if the operation succeeds.

Example

Pass a Hostname String to `rs.add()` The following operation adds a `mongod` (page 770) 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 3.0.0: Previous versions required an `_id` field in the document you passed to `rs.add()` (page 255). After 3.0.0 you can omit the `_id` field in this document. `members[n]._id` describes the requirements for specifying `_id`.

The following operation adds a `mongod` (page 770) 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 `members[n].host` field in the member configuration document.

See the `https://docs.mongodb.org/manual/reference/replica-configuration` for the available replica set member configuration settings.

See `https://docs.mongodb.org/manual/administration/replica-sets` for more examples and information.

rs.addArb()

On this page

- Description (page 256)

Description

`rs.addArb(host)`

Adds a new *arbiter* to an existing replica set.

The `rs.addArb()` (page 256) 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.

rs.conf()

On this page

- Definition (page 257)
- Output Example (page 257)

Definition

`rs.conf()`

Returns a document that contains the current *replica set* configuration.

The method wraps the `replSetGetConfig` (page 411) command.

Output Example The following document provides a representation of a replica set configuration document. The configuration of your replica set may include only a subset of these settings:

```
{
  _id: <string>,
  version: <int>,
  protocolVersion: <number>,
  members: [
    {
      _id: <int>,
      host: <string>,
      arbiterOnly: <boolean>,
      buildIndexes: <boolean>,
      hidden: <boolean>,
      priority: <number>,
      tags: <document>,
      slaveDelay: <int>,
      votes: <number>
    },
    ...
  ],
  settings: {
    chainingAllowed : <boolean>,
    heartbeatIntervalMillis : <int>,
    heartbeatTimeoutSecs: <int>,
    electionTimeoutMillis : <int>,
    getLastErrorModes : <document>,
    getLastErrorDefaults : <document>
  }
}
```

For description of the configuration settings, see `https://docs.mongodb.org/manual/reference/replica-configura`

`rs.config()`

`rs.config()` (page 257) is an alias of `rs.conf()` (page 257).

rs.freeze()

On this page

- Description (page 258)

Description

`rs.freeze(seconds)`

Makes the current *replica set* member ineligible to become *primary* for the period specified.

The `rs.freeze()` (page 258) method has the following parameter:

param number seconds The duration the member is ineligible to become primary.

`rs.freeze()` (page 258) provides a wrapper around the *database command* `replSetFreeze` (page 399).

rs.help()

`rs.help()`

Returns a basic help text for all of the `replication` related shell functions.

rs.initiate()

On this page

- Description (page 258)
- Replica Set Configuration (page 258)

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 258) method has the following parameter:

param document configuration Optional. 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 258) method provides a wrapper around the "`replSetInitiate` (page 403)" *database command*.

Replica Set Configuration See `https://docs.mongodb.org/manual/administration/replica-set-member-co` and `https://docs.mongodb.org/manual/reference/replica-configuration` for examples of replica set configuration and invitation objects.

rs.printReplicationInfo()

On this page

- Definition (page 259)
- Output Example (page 259)
- Output Fields (page 259)

Definition

`rs.printReplicationInfo()`

New in version 2.6.

Prints a formatted report of the replica set member's *oplog*. The displayed report formats the data returned by `db.getReplicationInfo()` (page 189). [8] The output of `rs.printReplicationInfo()` (page 259) is identical to that of `db.printReplicationInfo()` (page 193).

Note: The `rs.printReplicationInfo()` (page 259) in the `mongo` (page 803) shell does **not** return *JSON*. Use `rs.printReplicationInfo()` (page 259) for manual inspection, and `db.getReplicationInfo()` (page 189) in scripts.

Output Example The following example is a sample output from the `rs.printReplicationInfo()` (page 259) 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 259) formats and prints the data returned by `db.getReplicationInfo()` (page 189):

configured oplog size Displays the `db.getReplicationInfo.logSizeMB` (page 189) value.

log length start to end Displays the `db.getReplicationInfo.timeDiff` (page 189) and `db.getReplicationInfo.timeDiffHours` (page 189) values.

oplog first event time Displays the `db.getReplicationInfo.tFirst` (page 189).

oplog last event time Displays the `db.getReplicationInfo.tLast` (page 189).

now Displays the `db.getReplicationInfo.now` (page 189).

See `db.getReplicationInfo()` (page 189) for description of the data.

rs.printSlaveReplicationInfo()

[8] If run on a slave of a `master-slave replication`, the method calls `db.printSlaveReplicationInfo()` (page 195). See `db.printSlaveReplicationInfo()` (page 195) for details.

On this page

- Definition (page 260)
- Output (page 260)

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 195).

Output The following is example output from the `rs.printSlaveReplicationInfo()` (page 260) 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 `members[n].slaveDelay` value.

rs.reconfig()

On this page

- Definition (page 260)
- Behavior (page 261)
- Examples (page 261)

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 Optional. 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.

To reconfigure an existing replica set, first retrieve the current configuration with `rs.conf()` (page 257), modify the configuration document as needed, and then pass the modified document to `rs.reconfig()` (page 260).

`rs.reconfig()` (page 260) provides a wrapper around the `replSetReconfig` (page 404) command.

The `force` parameter allows a reconfiguration command to be issued to a non-primary node.

Behavior The `rs.reconfig()` (page 260) shell method can trigger the current primary to step down in some situations. When the primary steps down, it forcibly closes all client connections. This is by design. Since it may take a period of time to elect a new primary, schedule reconfiguration changes during maintenance periods to minimize loss of write availability.

> **Warning:** Using `rs.reconfig()` (page 260) with `{ force: true }` can lead to *rollback* of committed writes. 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 `members[n].priority` of the second member. The operations are issued through a `mongo` (page 803) 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 257) 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 `members[n].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 `members[n]._id` field.
3. The last statement calls the `rs.reconfig()` (page 260) 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:

`rs.conf()` (page 257), *replSetGetConfig-output* and `https://docs.mongodb.org/manual/administration/replica`

rs.remove()

On this page

- Definition (page 262)

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 262) method has the following parameter:

param string hostname The hostname of a system in the replica set.

Note: Before running the `rs.remove()` (page 262) 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 262), 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 152) method for more fine-grained control over `read preference` in the `mongo` (page 803) 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 400) command. See the documentation of the command for a complete description of the *output* (page 400).

rs.stepDown()

On this page

- Description (page 263)
- Behavior (page 263)

Description

`rs.stepDown(`*stepDownSecs*, *secondaryCatchUpPeriodSecs*`)`

Forces the *primary* of the replica set to become a *secondary*, triggering an *election for primary*. The method steps down the primary for a specified number of seconds; during this period, the stepdown member is ineligible from becoming primary.

The method only steps down the primary if an `electable` secondary is up-to-date with the primary, waiting up to 10 seconds for a secondary to catch up.

The method is only valid against the primary and will error if run on a non-primary member.

The `rs.stepDown()` (page 263) method has the following parameters:

param number stepDownSecs The number of seconds to step down the primary, during which time the stepdown member is ineligible for becoming primary. If you specify a non-numeric value, the command uses `60` seconds.

The stepdown period starts from the time that the `mongod` (page 770) receives the command. The stepdown period must be greater than the `secondaryCatchUpPeriodSecs`.

param number secondaryCatchUpPeriodSecs Optional. The number of seconds that `mongod` will wait for an electable secondary to catch up to the primary.

When specified, `secondaryCatchUpPeriodSecs` overrides the default wait time of `10` seconds.

`rs.stepDown()` (page 263) provides a wrapper around the command `replSetStepDown` (page 405).

Behavior New in version 3.0.

Before stepping down, `rs.stepDown()` (page 263) will attempt to terminate long running user operations that would block the primary from stepping down, such as an index build, a write operation or a map-reduce job.

To avoid rollbacks, `rs.stepDown()` (page 263), by default, only steps down the primary if an electable secondary is completely caught up with the primary. The command will wait up to either 10 seconds or the `secondaryCatchUpPeriodSecs` for a secondary to catch up.

If no electable secondary meets this criterion by the waiting period, the primary does not step down and the method throws an exception.

Upon successful stepdown, `rs.stepDown()` (page 263) forces all clients currently connected to the database to disconnect. This helps ensure that the clients maintain an accurate view of the replica set.

Because the disconnect includes the connection used to run the command, you cannot retrieve the return status of the command if the command completes successfully; i.e. you can only retrieve the return status of the command if it errors. When running the command in a script, the script should account for this behavior.

Note: `rs.stepDown()` (page 263) blocks all writes to the primary while it runs.

rs.syncFrom()

rs.**syncFrom**()

New in version 2.2.

Provides a wrapper around the `replSetSyncFrom` (page 407), 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 407) for more details.

See `https://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 267)	Runs a *database command* against the admin database, like db.runCommand() (page 196), but can confirm that it is issued against a mongos (page 792).
sh.getBalancerLockDetails() (page 267)	Reports on the active balancer lock, if it exists.
sh._checkFullName() (page 267)	Tests a namespace to determine if its well formed.
sh._checkMongos() (page 268)	Tests to see if the mongo (page 803) shell is connected to a mongos (page 792) instance.
sh._lastMigration() (page 268)	Reports on the last *chunk* migration.
sh.addShard() (page 269)	Adds a *shard* to a sharded cluster.
sh.addShardTag() (page 270)	Associates a shard with a tag, to support `tag aware sharding`.
sh.addTagRange() (page 270)	Associates range of shard keys with a shard tag, to support `tag aware sharding`.
sh.removeTagRange() (page 271)	Removes an association between a range shard keys and a shard tag. Use to manage `tag aware sharding`.
sh.disableBalancing() (page 272)	Disable balancing on a single collection in a sharded database. Does not affect balancing of other collections in a sharded cluster.
sh.enableBalancing() (page 273)	Activates the sharded collection balancer process if previously disabled using sh.disableBalancing() (page 272).
sh.enableSharding() (page 273)	Enables sharding on a specific database.
sh.getBalancerHost() (page 273)	Returns the name of a mongos (page 792) that's responsible for the balancer process.
sh.getBalancerState() (page 274)	Returns a boolean to report if the *balancer* is currently enabled.
sh.help() (page 274)	Returns help text for the `sh` methods.
sh.isBalancerRunning() (page 274)	Returns a boolean to report if the balancer process is currently migrating chunks.
sh.moveChunk() (page 275)	Migrates a *chunk* in a *sharded cluster*.
sh.removeShardTag() (page 275)	Removes the association between a shard and a shard tag.
sh.setBalancerState() (page 276)	Enables or disables the *balancer* which migrates *chunks* between *shards*.
sh.shardCollection() (page 277)	Enables sharding for a collection.
sh.splitAt() (page 277)	Divides an existing *chunk* into two chunks using a specific value of the *shard key* as the dividing point.
sh.splitFind() (page 278)	Divides an existing *chunk* that contains a document matching a query into two approximately equal chunks.
sh.startBalancer() (page 278)	Enables the *balancer* and waits for balancing to start.
sh.status() (page 279)	Reports on the status of a *sharded cluster*, as db.printShardingStatus() (page 194).
sh.stopBalancer() (page 282)	Disables the *balancer* and waits for any in progress balancing rounds to complete.
sh.waitForBalancer() (page 283)	Internal. Waits for the balancer state to change.
sh.waitForBalancerOff() (page 283)	Internal. Waits until the balancer stops running.
sh.waitForDLock() (page 284)	Internal. Waits for a specified distributed *sharded cluster* lock.

sh._adminCommand()

On this page

- Definition (page 267)

Definition

`sh._adminCommand(command, checkMongos)`

Runs a database command against the admin database of a `mongos` (page 792) 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 792) instance.

See also:

`db.runCommand()` (page 196)

sh.getBalancerLockDetails()

`sh.getBalancerLockDetails()`

Reports on the active balancer lock, if it exists.

Returns `null` if lock document does not exist or no lock is not taken. Otherwise, returns the lock document.

Return type Document or `null`.

sh._checkFullName()

On this page

- Definition (page 267)

Definition

`sh._checkFullName(namespace)`

Verifies that a *namespace* name is well formed. If the namespace is well formed, the `sh._checkFullName()` (page 267) method exits *with no message*.

Throws If the namespace is not well formed, `sh._checkFullName()` (page 267) throws: "name needs to be fully qualified <db>.<collection>"

The `sh._checkFullName()` (page 267) 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 268) method throws an error message if the `mongo` (page 803) shell is not connected to a `mongos` (page 792) instance. Otherwise it exits (no return document or return code).

sh._lastMigration()

On this page

- Definition (page 268)
- Output (page 268)

Definition

`sh._lastMigration(namespace)`

Returns information on the last migration performed on the specified database or collection.

The `sh._lastMigration()` (page 268) method has the following parameter:

param string namespace The *namespace* of a database or collection within the current database.

Output The `sh._lastMigration()` (page 268) 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` embedded documents with the bounds of the migrated chunk.

sh.addShard()

On this page

- Definition (page 269)
- Considerations (page 269)
- Example (page 269)

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 792) instance.

The `sh.addShard()` (page 269) 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 269) 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 792) installed from official `.deb` and `.rpm` packages have the `bind_ip` configuration set to `127.0.0.1` by default.

The `sh.addShard()` (page 269) method is a helper for the `addShard` (page 414) command. The `addShard` (page 414) 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 `https://docs.mongodb.org/manual/core/sharding-balancing` for more information.

Changed in version 2.6: Chunk migrations can have an impact on disk space. Starting in MongoDB 2.6, the source shard automatically archives the migrated documents by default. For details, see *moveChunk-directory*.

Hidden Members

Important: You cannot include a `hidden member` in the seed list provided to `sh.addShard()` (page 269).

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()

On this page

- Definition (page 270)
- Example (page 270)

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 270) 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 270) when connected to a `mongos` (page 792) 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 270) and `sh.removeShardTag()` (page 275).

sh.addTagRange()

On this page

- Definition (page 270)
- Behavior (page 271)
- Example (page 271)

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 270) method. `sh.addTagRange()` (page 270) 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 270) 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 270) when connected to a `mongos` (page 792) 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 270) 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.removeTagRange()

On this page

- Definition (page 271)
- Example (page 272)

Definition

`sh.removeTagRange(namespace, minimum, maximum, tag)`

New in version 3.0.

Removes a range of shard key values to a shard tag created using the `sh.removeShardTag()` (page 275) method. `sh.removeTagRange()` (page 271) 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 275) to ensure that unused or out of date ranges are removed and hence chunks are balanced as required.

Only issue `sh.removeTagRange()` (page 271) when connected to a `mongos` (page 792) 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.disableBalancing()

On this page

- Description (page 272)

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 272) method has the following parameter:

param string namespace The *namespace* of the collection.

For more information on the balancing process, see `https://docs.mongodb.org/manual/tutorial/manage-shar` and *sharding-balancing*.

sh.enableBalancing()

On this page

- Description (page 273)

Description

`sh.enableBalancing(namespace)`

Enables the balancer for the specified namespace of the sharded collection.

The `sh.enableBalancing()` (page 273) method has the following parameter:

param string namespace The *namespace* of the collection.

Important: `sh.enableBalancing()` (page 273) 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 `https://docs.mongodb.org/manual/tutorial/manage-shar`
and *sharding-balancing*.

sh.enableSharding()

On this page

- Definition (page 273)

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 277).

The `sh.enableSharding()` (page 273) 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 277)

sh.getBalancerHost()

`sh.getBalancerHost()`

Returns String in form `hostname:port`

`sh.getBalancerHost()` (page 273) returns the name of the server that is running the balancer.

See also:

- `sh.enableBalancing()` (page 273)
- `sh.disableBalancing()` (page 272)
- `sh.getBalancerState()` (page 274)
- `sh.isBalancerRunning()` (page 274)
- `sh.setBalancerState()` (page 276)
- `sh.startBalancer()` (page 278)
- `sh.stopBalancer()` (page 282)
- `sh.waitForBalancer()` (page 283)

- `sh.waitForBalancerOff()` (page 283)

sh.getBalancerState()

`sh.getBalancerState()`

Returns boolean

`sh.getBalancerState()` (page 274) 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 274) to check the balancer's current state.

See also:

- `sh.enableBalancing()` (page 273)
- `sh.disableBalancing()` (page 272)
- `sh.getBalancerHost()` (page 273)
- `sh.isBalancerRunning()` (page 274)
- `sh.setBalancerState()` (page 276)
- `sh.startBalancer()` (page 278)
- `sh.stopBalancer()` (page 282)
- `sh.waitForBalancer()` (page 283)
- `sh.waitForBalancerOff()` (page 283)

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 274) to determine if the balancer is enabled or disabled.

See also:

- `sh.enableBalancing()` (page 273)
- `sh.disableBalancing()` (page 272)
- `sh.getBalancerHost()` (page 273)
- `sh.getBalancerState()` (page 274)
- `sh.setBalancerState()` (page 276)
- `sh.startBalancer()` (page 278)
- `sh.stopBalancer()` (page 282)

- `sh.waitForBalancer()` (page 283)
- `sh.waitForBalancerOff()` (page 283)

sh.moveChunk()

On this page

- Definition (page 275)
- Example (page 275)

Definition

`sh.moveChunk(namespace, query, destination)`

Moves the *chunk* that contains the document specified by the `query` to the `destination` shard. `sh.moveChunk()` (page 275) provides a wrapper around the `moveChunk` (page 427) 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 275) directly.

See also:

`moveChunk` (page 427), `sh.splitAt()` (page 277), `sh.splitFind()` (page 278), `https://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()

On this page

- Definition (page 275)

Definition

`sh.removeShardTag(shard, tag)`

New in version 2.2.

Removes the association between a tag and a shard. Only issue `sh.removeShardTag()` (page 275) when connected to a `mongos` (page 792) 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 270), `sh.addTagRange()` (page 270)

sh.setBalancerState()

On this page

- Description (page 276)

Description

`sh.setBalancerState(state)`

Enables or disables the *balancer*. Use `sh.getBalancerState()` (page 274) to determine if the balancer is currently enabled or disabled and `sh.isBalancerRunning()` (page 274) to check its current state.

The `sh.getBalancerState()` (page 274) 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 273)
- `sh.disableBalancing()` (page 272)
- `sh.getBalancerHost()` (page 273)
- `sh.getBalancerState()` (page 274)
- `sh.isBalancerRunning()` (page 274)
- `sh.startBalancer()` (page 278)
- `sh.stopBalancer()` (page 282)
- `sh.waitForBalancer()` (page 283)
- `sh.waitForBalancerOff()` (page 283)

sh.shardCollection()

On this page

- Definition (page 277)
- Considerations (page 277)
- Example (page 277)
- Additional Information (page 277)

Definition

`sh.shardCollection(namespace, key, unique)`

Shards a collection using the `key` as a the *shard key*. `sh.shardCollection()` (page 277) 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 422). Additionally, after `shardCollection` (page 422), 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 422) for additional options, `https://docs.mongodb.org/manual/sharding` and `https://docs.mongodb.org/manual/core/sharding-i` for an overview of sharding, `https://docs.mongodb.org/manual/tutorial/deploy-shard-cluster` for a tutorial, and *sharding-shard-key* for choosing a shard key.

sh.splitAt()

On this page

- Definition (page 277)
- Consideration (page 278)
- Behavior (page 278)

Definition

`sh.splitAt(namespace, query)`

Splits a chunk at the shard key value specified by the query.

The method takes the following arguments:

param string namespace The namespace (i.e. `<database>.<collection>`) of the sharded collection that contains the chunk to split.

param document query A query document that specifies the *shard key* value at which to split the chunk.

Consideration In most circumstances, you should leave chunk splitting to the automated processes within MongoDB. However, when initially deploying a *sharded cluster*, it may be beneficial to *pre-split* manually an empty collection using methods such as `sh.splitAt()` (page 277).

Behavior `sh.splitAt()` (page 277) splits the original chunk into two chunks. One chunk has a shard key range that starts with the original lower bound (inclusive) and ends at the specified shard key value (exclusive). The other chunk has a shard key range that starts with the specified shard key value (inclusive) as the lower bound and ends at the original upper bound (exclusive).

To split a chunk at its median point instead, see `sh.splitFind()` (page 278).

sh.splitFind()

On this page

- Definition (page 278)
- Consideration (page 278)

Definition

`sh.`**`splitFind`**`(`*namespace*, *query*`)`

Splits the chunk that contains the shard key value specified by the `query` at the chunk's median point. `sh.splitFind()` (page 278) creates two roughly equal chunks. To split a chunk at a specific point instead, see `sh.splitAt()` (page 277).

The method takes the following arguments:

param string namespace The namespace (i.e. `<database>.<collection>`) of the sharded collection that contains the chunk to split.

param document query A query document that specifies the *shard key* value that determines the chunk to split.

Consideration In most circumstances, you should leave chunk splitting to the automated processes within MongoDB. However, when initially deploying a *sharded cluster*, it may be beneficial to *pre-split* manually an empty collection using methods such as `sh.splitFind()` (page 278).

sh.startBalancer()

On this page

- Definition (page 278)

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 273)
- `sh.disableBalancing()` (page 272)
- `sh.getBalancerHost()` (page 273)
- `sh.getBalancerState()` (page 274)
- `sh.isBalancerRunning()` (page 274)
- `sh.setBalancerState()` (page 276)
- `sh.stopBalancer()` (page 282)
- `sh.waitForBalancer()` (page 283)
- `sh.waitForBalancerOff()` (page 283)

sh.status()

On this page

- Definition (page 279)
- Output Examples (page 279)
- Output Fields (page 281)

Definition

`sh.status()`

When run on a `mongos` (page 792) instance, 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 279) method has the following parameter:

param boolean verbose Optional. If `true`, the method displays details of the document distribution across chunks when you have 20 or more chunks.

See also:

`db.printShardingStatus()` (page 194)

Output Examples The *Sharding Version* (page 281) section displays information on the *config database*:

```
--- Sharding Status ---
  sharding version: {
   "_id" : <num>,
   "minCompatibleVersion" : <num>,
   "currentVersion" : <num>,
   "clusterId" : <ObjectId>
}
```

The *Shards* (page 281) 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 3.0.0: The *Balancer* (page 281) 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 282) 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 282) 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 [9], 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
   ...
```

[9] 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 279) 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 281) is an identifier for the version details.

`sh.status.sharding-version.minCompatibleVersion`
The `minCompatibleVersion` (page 281) is the minimum compatible version of the config server.

`sh.status.sharding-version.currentVersion`
The `currentVersion` (page 281) is the current version of the config server.

`sh.status.sharding-version.clusterId`
The `clusterId` (page 281) is the identification for the sharded cluster.

Shards

`sh.status.shards._id`
The `_id` (page 281) displays the name of the shard.

`sh.status.shards.host`
The `host` (page 281) displays the host location of the shard.

`sh.status.shards.tags`
The `tags` (page 281) displays all the tags for the shard. The field only displays if the shard has tags.

Balancer New in version 3.0.0: `sh.status()` (page 279) added the `balancer` field.

`sh.status.balancer.currently-enabled`
`currently-enabled` (page 281) indicates if the *balancer* is currently enabled on the sharded cluster.

`sh.status.balancer.currently-running`
`currently-running` (page 281) indicates whether the *balancer* is currently running, and therefore currently balancing the cluster.

If the *balancer* is running, `currently-running` (page 281) lists the process that holds the balancer lock, and the date and time that the process obtained the lock.

If there is an active balancer lock, `currently-running` (page 281) also reports the state of the balancer.

`sh.status.balancer.collections-with-active-migrations`
`collections-with-active-migrations` (page 281) 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 279) output.

`sh.status.balancer.failed-balancer-rounds-in-last-5-attempts`
`failed-balancer-rounds-in-last-5-attempts` (page 281) 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 281) lists the most recent balancer error message. If there have been no errors, this field will not appear in the `sh.status()` (page 279) output.

`sh.status.balancer.time-of-reported-error`
`time-of-reported-error` (page 281) 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 281) 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 281) displays `No recent migrations`.

`migration-results-for-the-last-24-hours` (page 281) includes *all* migrations, including those not initiated by the balancer.

Databases

`sh.status.databases.`**`_id`**

The `_id` (page 282) displays the name of the database.

`sh.status.databases.`**`partitioned`**

The `partitioned` (page 282) displays whether the database has sharding enabled. If `true`, the database has sharding enabled.

`sh.status.databases.`**`primary`**

The `primary` (page 282) displays the *primary shard* for the database.

Sharded Collection

`sh.status.databases.`**`shard-key`**

The `shard-key` (page 282) displays the shard key specification document.

`sh.status.databases.`**`chunks`**

The `chunks` (page 282) lists all the shards and the number of chunks that reside on each shard.

`sh.status.databases.`**`chunk-details`**

The `chunk-details` (page 282) 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 282) lists the details of the tags associated with a range of shard key values.

sh.stopBalancer()

On this page

- Definition (page 282)

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 273)
- `sh.disableBalancing()` (page 272)

- sh.getBalancerHost() (page 273)
- sh.getBalancerState() (page 274)
- sh.isBalancerRunning() (page 274)
- sh.setBalancerState() (page 276)
- sh.startBalancer() (page 278)
- sh.waitForBalancer() (page 283)
- sh.waitForBalancerOff() (page 283)

sh.waitForBalancer()

On this page

- Definition (page 283)

Definition

`sh.`**`waitForBalancer`**(*wait*, *timeout*, *interval*)

Waits for a change in the state of the balancer. `sh.waitForBalancer()` (page 283) is an internal method, which takes the following arguments:

param boolean wait Optional. 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()

On this page

- Definition (page 283)

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 273)
- sh.disableBalancing() (page 272)
- sh.getBalancerHost() (page 273)
- sh.getBalancerState() (page 274)
- sh.isBalancerRunning() (page 274)

- `sh.setBalancerState()` (page 276)
- `sh.startBalancer()` (page 278)
- `sh.stopBalancer()` (page 282)
- `sh.waitForBalancer()` (page 283)

sh.waitForDLock()

On this page

- Definition (page 284)

Definition

`sh.waitForDLock`(*lockname*, *wait*, *timeout*, *interval*)

Waits until the specified distributed lock changes state. `sh.waitForDLock()` (page 284) is an internal method that takes the following arguments:

param string lockname The name of the distributed lock.

param boolean wait Optional. 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()

On this page

- Definition (page 284)

Definition

`sh.waitForPingChange`(*activePings*, *timeout*, *interval*)

`sh.waitForPingChange()` (page 284) 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 889) 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 285)	For internal use.
`rawMongoProgramOutput()` (page 285)	For internal use.
`run()`	For internal use.
`runMongoProgram()` (page 285)	For internal use.
`runProgram()` (page 285)	For internal use.
`startMongoProgram()`	For internal use.
`stopMongoProgram()` (page 286)	For internal use.
`stopMongoProgramByPid()` (page 286)	For internal use.
`stopMongod()` (page 286)	For internal use.
`waitMongoProgramOnPort()` (page 286)	For internal use.
`waitProgram()` (page 286)	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
`Date()` (page 287)	Creates a date object. By default creates a date object including the current date.
`UUID()` (page 288)	Converts a 32-byte hexadecimal string to the UUID BSON subtype.
`ObjectId.getTimestamp()` (page 288)	Returns the timestamp portion of an *ObjectId*.
`ObjectId.toString()` (page 288)	Displays the string representation of an *ObjectId*.
`ObjectId.valueOf()` (page 289)	Displays the `str` attribute of an ObjectId as a hexadecimal string.
`WriteResult()` (page 289)	Wrapper around the result set from write methods.
`WriteResult.hasWriteError()` (page 290)	Returns a boolean specifying whether the results include `WriteResult.writeError` (page 290).
`WriteResult.hasWriteConcernError()` (page 291)	Returns a boolean specifying whether whether the results include `WriteResult.writeConcernError` (page 290).
`BulkWriteResult()` (page 291)	Wrapper around the result set from `Bulk.execute()` (page 223).

Date()

On this page

- Behavior (page 287)
- Examples (page 287)

Date()

Returns a date either as a string or as a *document-bson-type-date* object.

- `Date()` returns the current date as a string.
- `new Date()` returns the current date as a *document-bson-type-date* object. The `mongo` (page 803) shell wraps the *document-bson-type-date* object with the `ISODate` helper.
- `new Date("<YYYY-mm-dd>")` returns the specified date string (`"<YYYY-mm-dd>"`) as a *document-bson-type-date* object. The `mongo` (page 803) shell wraps the *document-bson-type-date* object with the `ISODate` helper.

Behavior Internally, *document-bson-type-date* objects are stored as a 64 bit integer representing the number of milliseconds since the Unix epoch (Jan 1, 1970), which results in a representable date range of about 290 millions years into the past and future.

Examples

Return Date as a String To return the date as a string, use the `Date()` method, as in the following example:

```
var myDateString = Date();
```

Return Date as `Date` Object The `mongo` (page 803) shell wrap objects of *document-bson-type-date* type with the `ISODate` helper; however, the objects remain of type *document-bson-type-date*.

The following example uses `new Date()` to return `Date` objects.

```
var myDate = new Date();
```

See also:

BSON Date, *mongo Shell Date*

UUID()

On this page

- Definition (page 288)
- Example (page 288)

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==")
```

ObjectId.getTimestamp()

ObjectId.**getTimestamp**()

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

In the following example, call the `getTimestamp()` (page 288) 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 288) returns the value of the ObjectId as a hexadecimal string.

In the following example, call the `toString()` (page 288) 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 289) returns the `ObjectId()` object.

In the following example, call the `valueOf()` (page 289) 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()
```

WriteResult()

On this page

- Definition (page 289)
- Properties (page 289)

Definition

`WriteResult``()`

A wrapper that contains the result status of the `mongo` (page 803) shell write methods.

See

`db.collection.insert()` (page 79), `db.collection.update()` (page 117), `db.collection.remove()` (page 101), and `db.collection.save()` (page 105).

Properties The `WriteResult` (page 289) has the following properties:

`WriteResult.`**`nInserted`**

The number of documents inserted, excluding `upserted` documents. See `WriteResult.nUpserted` (page 289) 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 600) expression updates the value to the current value, `nMatched` (page 289) can be greater than `nModified` (page 289).

`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 289) can be less than `nMatched` (page 289).

`WriteResult.`**`nUpserted`**
The number of documents inserted by an *upsert* (page 118).

`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.writeConcernError.`**`errmsg`**
A description of the error.

See also:

`WriteResult.hasWriteError()` (page 290), `WriteResult.hasWriteConcernError()` (page 291)

WriteResult.hasWriteError()

On this page

- Definition (page 290)

Definition

`WriteResult.`**`hasWriteError`**`()`
Returns `true` if the result of a `mongo` (page 803) shell write method has `WriteResult.writeError` (page 290). Otherwise, the method returns `false`.

See also:

`WriteResult()` (page 289)

WriteResult.hasWriteConcernError()

On this page

- Definition (page 291)

Definition

`WriteResult.`**`hasWriteConcernError`**`()`

Returns `true` if the result of a `mongo` (page 803) shell write method has `WriteResult.writeConcernError` (page 290). Otherwise, the method returns `false`.

See also:

`WriteResult()` (page 289)

BulkWriteResult()

On this page

- Properties (page 291)

`BulkWriteResult``()`

New in version 2.6.

A wrapper that contains the results of the `Bulk.execute()` (page 223) method.

Properties The `BulkWriteResult` (page 291) has the following properties:

`BulkWriteResult.`**`nInserted`**

The number of documents inserted using the `Bulk.insert()` (page 214) method. For documents inserted through operations with the `Bulk.find.upsert()` (page 221) option, see the `nUpserted` (page 291) 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 600) expression updates the value to the current value, `nMatched` (page 291) can be greater than `nModified` (page 291).

`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 291) can be less than `nMatched` (page 291). Inserted documents do not affect the number of `nModified` (page 291); refer to the `nInserted` (page 291) and `nUpserted` (page 291) fields instead.

`BulkWriteResult.`**`nRemoved`**

The number of documents removed.

`BulkWriteResult.`**`nUpserted`**

The number of documents inserted through operations with the `Bulk.find.upsert()` (page 221) option.

`BulkWriteResult.`**`upserted`**

An array of documents that contains information for each document inserted through operations with the `Bulk.find.upsert()` (page 221) 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 292) 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 226).

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.

2.1.12 Connection

Connection Methods

Name	Description
Mongo.getDB() (page 293)	Returns a database object.
Mongo.getReadPrefMode() (page 293)	Returns the current read preference mode for the MongoDB connection.
Mongo.getReadPrefTagSet() (page 294)	Returns the read preference tag set for the MongoDB connection.
Mongo.setReadPref() (page 294)	Sets the *read preference* for the MongoDB connection.
Mongo.setSlaveOk() (page 295)	Allows operations on the current connection to read from *secondary* members.
Mongo() (page 295)	Creates a new connection object.
connect()	Connects to a MongoDB instance and to a specified database on that instance.

Mongo.getDB()

On this page

- Description (page 293)
- Example (page 293)

Description

`Mongo.getDB(<database>)`

Provides access to database objects from the `mongo` (page 803) shell or from a JavaScript file.

The `Mongo.getDB()` (page 293) 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 295) and *connect()* (page 295)

Mongo.getReadPrefMode()

`Mongo.getReadPrefMode()`

Returns The current *read preference* mode for the `Mongo()` (page 188) connection object.

See `https://docs.mongodb.org/manual/core/read-preference` for an introduction to read preferences in MongoDB. Use `getReadPrefMode()` (page 293) 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:

`https://docs.mongodb.org/manual/core/read-preference`, `readPref()` (page 152), `setReadPref()` (page 294), and `getReadPrefTagSet()` (page 294).

Mongo.getReadPrefTagSet()

`Mongo.`**`getReadPrefTagSet`**`()`

Returns The current *read preference* tag set for the `Mongo()` (page 188) connection object.

See `https://docs.mongodb.org/manual/core/read-preference` for an introduction to read preferences and tag sets in MongoDB. Use `getReadPrefTagSet()` (page 294) 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:

`https://docs.mongodb.org/manual/core/read-preference`, `readPref()` (page 152), `setReadPref()` (page 294), and `getReadPrefTagSet()` (page 294).

Mongo.setReadPref()

On this page

- Definition (page 294)
- Examples (page 294)

Definition

`Mongo.`**`setReadPref`**`(`*mode*, *tagSet*`)`

Call the `setReadPref()` (page 294) method on a `Mongo` (page 188) 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 Optional. 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 803) 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 294), 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 294) 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 262).

See the `readPref()` (page 152) method for more fine-grained control over `read preference` in the `mongo` (page 803) shell.

Mongo()

On this page

- Description (page 295)
- Instantiation Options (page 295)

Description

`Mongo``(`*host*`)`

JavaScript constructor to instantiate a database connection from the `mongo` (page 803) shell or from a JavaScript file.

The `Mongo()` (page 295) method has the following parameter:

param string host Optional. 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 293) and `db.getMongo()` (page 188).

connect()

On this page

- Description (page 296)
- Example (page 296)

Description

`connect`(*url*, *user*, *password*)

Creates a connection to a MongoDB instance and returns the reference to the database. However, in most cases, use the `Mongo()` (page 295) object and its `getDB()` (page 293) method instead.

param string url Specifies the connection string. You can specify either:

- `<hostname>:<port>/<database>`
- `<hostname>/<database>`
- `<database>`

param string user Optional. Specifies an existing username with access privileges for this database. If `user` is specified, you must include the `password` parameter as well.

param string password Optional unless the `user` parameter is specified. Specifies the password for the `user`.

Example 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()` (page 295), `Mongo.getDB()` (page 293), `db.auth()` (page 229)

2.1.13 Native

Native Methods

Name	Description
`cat()`	Returns the contents of the specified file.
`version()`	Returns the current version of the `mongo` (page 803) shell instance.
`cd()`	Changes the current working directory to the specified path.
`sleep()`	Suspends the `mongo` (page 803) shell for a given period of time.
`copyDbpath()` (page 298)	Copies a local `dbPath` (page 915). For internal use.
`resetDbpath()` (page 299)	Removes a local `dbPath` (page 915). For internal use.
`fuzzFile()` (page 299)	For internal use to support testing.
`getHostName()` (page 299)	Returns the hostname of the system running the `mongo` (page 803) shell.
`getMemInfo()` (page 299)	Returns a document that reports the amount of memory used by the shell.
`hostname()`	Returns the hostname of the system running the shell.
`_isWindows()` (page 299)	Returns `true` if the shell runs on a Windows system; `false` if a Unix or Linux system.
`listFiles()` (page 300)	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 301)	The *md5* hash of the specified file.
`mkdir()`	Creates a directory at the specified path.
`pwd()`	Returns the current directory.
`quit()`	Exits the current shell session.
`_rand()` (page 302)	Returns a random number between `0` and `1`.
`removeFile()` (page 302)	Removes the specified file from the local file system.
`setVerboseShell()` (page 302)	Configures the `mongo` (page 803) shell to report operation timing.
`_srand()` (page 303)	For internal use.

cat()

On this page

- Definition (page 297)

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.

version()

`version`()

Returns The version of the `mongo` (page 803) 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.

cd()

On this page

- Definition (page 298)

Definition

`cd``(`*path*`)`

param string path A path on the file system local to the `mongo` (page 803) shell context.

`cd()` changes the directory context of the `mongo` (page 803) shell and has no effect on the MongoDB server.

sleep()

On this page

- Definition (page 298)
- Example (page 298)

Definition

`sleep``(`*ms*`)`

param integer ms A duration in milliseconds.

`sleep()` suspends a JavaScript execution context for a specified number of milliseconds.

Example Consider a low-priority bulk data import script. To avoid impacting other processes, you may suspend the shell after inserting each document, distributing the cost of insertion over a longer period of time.

The following example `mongo` (page 803) script will load a JSON file containing an array of documents, and save one element every 100 milliseconds.

```
JSON.parse(cat('users.json')).forEach(function(user) {
    db.users.save(user);
    sleep(100);
});
```

copyDbpath()

`copyDbpath``()`

For internal use.

resetDbpath()

resetDbpath()
For internal use.

fuzzFile()

On this page

- Description (page 299)

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 803) 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 803) shell process.

_isWindows()

_isWindows()

Returns boolean.

Returns "true" if the `mongo` (page 803) shell is running on a system that is Windows, or "false" if the shell is running on a Unix or Linux systems.

listFiles()

listFiles()

Returns an array, containing one document per object in the directory. This function operates in the context of the mongo (page 803) shell. The fields included in the documents are:

name
A string which contains the pathname of the object.

baseName
A string which contains the name of the object.

isDirectory
A boolean to indicate whether the object is a directory.

size
The size of the object in bytes. This field is only present for files.

load()

On this page

- Definition (page 300)
- Example (page 300)

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 803) 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()

On this page

- Description (page 301)

Description

`md5sumFile`(*filename*)

Returns a *md5* hash of the specified file.

The `md5sumFile()` (page 301) 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 803) shell.

mkdir()

On this page

- Description (page 301)

Description

`mkdir`(*path*)

Creates a directory at the specified path. This method creates the entire path specified if the enclosing directory or directories do not already exit.

This method is equivalent to **mkdir -p** with BSD or GNU utilities.

The `mkdir()` method has the following parameter:

param string path A path on the local filesystem.

pwd()

`pwd`()

Returns the current directory.

This function returns with output relative to the current shell session, and does not impact the server.

quit()

`quit`()

Exits the current shell session.

rand()

_rand()

Returns A random number between 0 and 1.

This function provides functionality similar to the Math.rand() function from the standard library.

removeFile()

On this page

- Description (page 302)

Description

removeFile(*filename*)

Removes the specified file from the local file system.

The removeFile() (page 302) method has the following parameter:

param string filename A filename or path to a local file.

setVerboseShell()

On this page

- Example (page 302)

setVerboseShell()

The setVerboseShell() (page 302) method configures the mongo (page 803) shell to print the duration of each operation.

setVerboseShell() (page 302) has the form:

```
setVerboseShell(true)
```

setVerboseShell() (page 302) takes one boolean parameter. Specify true or leave the parameter blank to activate the verbose shell. Specify false to deactivate.

Example The following example demonstrates the behavior of the verbose shell:

1. From the mongo (page 803) shell, set verbose shell to true:

```
setVerboseShell(true)
```

2. With verbose shell set to true, run db.collection.aggregate() (page 20):

```
db.restaurants.aggregate(
   [
      { $match: { "borough": "Queens", "cuisine": "Brazilian" } },
      { $group: { "_id": "$address.zipcode" , "count": { $sum: 1 } } }
   ]
);
```

3. In addition to returning the results of the operation, the mongo (page 803) shell now displays information about the duration of the operation:

```
{ "_id" : "11377", "count" : 1 }
{ "_id" : "11368", "count" : 1 }
{ "_id" : "11101", "count" : 2 }
{ "_id" : "11106", "count" : 3 }
{ "_id" : "11103", "count" : 1 }
Fetched 5 record(s) in 0ms
```

_srand()

_srand()

For internal use.

2.2 Database Commands

On this page

- User Commands (page 303)
- Database Operations (page 371)
- Internal Commands (page 518)
- Testing Commands (page 521)
- Auditing Commands (page 526)

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 803) shell helpers.

2.2.1 User Commands

Aggregation Commands

Aggregation Commands

Name	Description
aggregate (page 303)	Performs `aggregation tasks` such as group using the aggregation framework.
count (page 307)	Counts the number of documents in a collection.
distinct (page 310)	Displays the distinct values found for a specified key in a collection.
group (page 313)	Groups documents in a collection by the specified key and performs simple aggregation.
mapReduce (page 318)	Performs `map-reduce` aggregation for large data sets.

aggregate

On this page

- Example (page 305)

aggregate

Performs aggregation operation using the *aggregation pipeline* (page 630). The pipeline allows users to process data from a collection with a sequence of stage-based manipulations.

The command has following syntax:

Changed in version 3.2.

```
{
  aggregate: "<collection>",
  pipeline: [ <stage>, <...> ],
  explain: <boolean>,
  allowDiskUse: <boolean>,
  cursor: <document>,
  bypassDocumentValidation: <boolean>,
  readConcern: <document>
}
```

The `aggregate` (page 303) 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 630) that process and transform the document stream as part of the aggregation pipeline.

field boolean explain Optional. Specifies to return the information on the processing of the pipeline.

New in version 2.6.

field boolean allowDiskUse Optional. Enables writing to temporary files. When set to `true`, aggregation stages can write data to the `_tmp` subdirectory in the `dbPath` (page 915) directory.

New in version 2.6.

field document cursor Optional. Specify a document that contains options that control the creation of the cursor object.

New in version 2.6.

field boolean bypassDocumentValidation Optional. Available only if you specify the `$out` (page 656) aggregation operator.

Enables `aggregate` to bypass document validation during the operation. This lets you insert documents that do not meet the validation requirements.

New in version 3.2.

field document readConcern Optional. Specifies the *read concern*. The default level is `"local"`.

To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

To use a `read concern` level of `"majority"`, you cannot include the `$out` (page 656) stage.

New in version 3.2.

Changed in version 2.6: *aggregation pipeline* (page 630) introduces the `$out` (page 656) operator to allow `aggregate` (page 303) command to store results to a collection.

For more information about the aggregation pipeline `https://docs.mongodb.org/manual/core/aggregation-pipelir` *Aggregation Reference* (page 746), and `https://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 303) 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 803) shell, this operation can use the `aggregate()` (page 20) 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 20) helper always returns a cursor.

Changed in version 2.4: If an error occurs, the `aggregate()` (page 20) 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 303) 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 20) 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 20)

Aggregate Command Returns a Cursor

Note: Using the aggregate (page 303) command to return a cursor is a low-level operation, intended for authors of drivers. Most users should use the db.collection.aggregate() (page 20) helper provided in the mongo (page 803) shell or in their driver. In 2.6 and later, the aggregate() (page 20) 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* 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.

Override Default Read Concern To override the default read concern level of `"local"`, use the `readConcern` option.

The following operation on a replica set specifies a `read concern` of `"majority"` to read the most recent copy of the data confirmed as having been written to a majority of the nodes.

Important:

- To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

  Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

- To use a `read concern` level of `"majority"`, you cannot include the `$out` (page 656) stage.
- Regardless of the *read concern* level, the most recent data on a node may not reflect the most recent version of the data in the system.

```
db.runCommand(
   {
      aggregate: "orders",
      pipeline: [ { $match: { status: "A" } } ],
      readConcern: { level: "majority" }
   }
)
```

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

The `getMore` (page 354) command uses the `readConcern` level specified in the originating `aggregate` (page 303) command.

See also:

`db.collection.aggregate()` (page 20)

count

On this page

- Definition (page 307)
- Behavior (page 308)
- Examples (page 309)

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 307) has the following form:

```
{
  count: <collection-name>,
  query: <document>,
  limit: <integer>,
  skip: <integer>,
  hint: <hint>,
  readConcern: <document>
}
```

count (page 307) has the following fields:

field string count The name of the collection to count.

field document query Optional. A query that selects which documents to count in a collection.

field integer limit Optional. The maximum number of matching documents to return.

field integer skip Optional. The number of matching documents to skip before returning results.

field string, document hint Optional. The index to use. Specify either the index name as a string or the index specification document.

New in version 2.6.

field document readConcern Optional. Specifies the *read concern*. The default level is `"local"`.

To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the `--enableMajorityReadConcern` (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

To use a *read concern* level of `"majority"`, you must specify a nonempty `query` condition.

New in version 3.2.

MongoDB also provides the `count()` (page 137) and `db.collection.count()` (page 33) wrapper methods in the `mongo` (page 803) shell.

Behavior On a sharded cluster, `count` (page 307) 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 `$group` (page 644) stage of the `db.collection.aggregate()` (page 20) method to `$sum` (page 729) 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 635) stage as well:

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

See *Perform a Count* (page 636) for an example.

For MongoDB instances using the `WiredTiger` storage engine, after an unclean shutdown, statistics on size and count may off by up to 1000 documents as reported by `collStats` (page 473), `dbStats` (page 481), `count` (page 307). To restore the correct statistics for the collection, run `validate` (page 485) on the collection.

Examples The following sections provide examples of the `count` (page 307) 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 }
```

Override Default Read Concern To override the default read concern level of `"local"`, use the `readConcern` option.

The following operation on a replica set specifies a `https://docs.mongodb.org/manual/reference/read-concern` of `"majority"` to read the most recent copy of the data confirmed as having been written to a majority of the nodes.

Important:

- To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

  Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

- To use the `readConcern` level of `"majority"`, you must specify a nonempty `query` condition.

- Regardless of the *read concern* level, the most recent data on a node may not reflect the most recent version of the data in the system.

```
db.runCommand(
   {
     count: "restaurants",
     query: { rating: { $gte: 4 } },
     readConcern: { level: "majority" }
   }
)
```

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

distinct

On this page

- Definition (page 310)
- Behavior (page 311)
- Examples (page 311)

Definition

`distinct`

Finds the distinct values for a specified field across a single collection. `distinct` (page 310) returns a document that contains an array of the distinct values. The return document also contains an embedded document with query statistics and the query plan.

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 for which to return distinct values.

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

field document readConcern Optional. Specifies the *read concern*. The default level is `"local"`.

To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

New in version 3.2.

MongoDB also provides the shell wrapper method `db.collection.distinct()` (page 44) for the `distinct` (page 310) command. Additionally, many MongoDB *drivers* also provide a wrapper method. Refer to the specific driver documentation.

Behavior

Array Fields If the value of the specified `field` is an array, `distinct` (page 310) considers each element of the array as a separate value.

For instance, if a field has as its value `[ 1, [1], 1 ]`, then `distinct` (page 310) considers `1`, `[1]`, and `1` as separate values.

For an example, see *Return Distinct Values for an Array Field* (page 312).

Index Use When possible, `distinct` (page 310) operations can use indexes.

Indexes can also *cover* `distinct` (page 310) operations. See *covered-queries* for more information on queries covered by indexes.

Examples The examples use the `inventory` collection that contains the following documents:

```
{ "_id": 1, "dept": "A", "item": { "sku": "111", "color": "red" }, "sizes": [ "S", "M" ] }
{ "_id": 2, "dept": "A", "item": { "sku": "111", "color": "blue" }, "sizes": [ "M", "L" ] }
{ "_id": 3, "dept": "B", "item": { "sku": "222", "color": "blue" }, "sizes": "S" }
{ "_id": 4, "dept": "A", "item": { "sku": "333", "color": "black" }, "sizes": [ "S" ] }
```

Return Distinct Values for a Field The following example returns the distinct values for the field `dept` from all documents in the `inventory` collection:

```
db.runCommand ( { distinct: "inventory", key: "dept" } )
```

The command returns a document with a field named `values` that contains the distinct `dept` values:

```
{
   "values" : [ "A", "B" ],
   "stats" : { ... },
   "ok" : 1
}
```

Return Distinct Values for an Embedded Field The following example returns the distinct values for the field `sku`, embedded in the `item` field, from all documents in the `inventory` collection:

```
db.runCommand ( { distinct: "inventory", key: "item.sku" } )
```

The command returns a document with a field named `values` that contains the distinct `sku` values:

```
{
  "values" : [ "111", "222", "333" ],
  "stats" : { ... },
  "ok" : 1
}
```

See also:

document-dot-notation for information on accessing fields within embedded documents

Return Distinct Values for an Array Field The following example returns the distinct values for the field `sizes` from all documents in the `inventory` collection:

```
db.runCommand ( { distinct: "inventory", key: "sizes" } )
```

The command returns a document with a field named `values` that contains the distinct `sizes` values:

```
{
  "values" : [ "M", "S", "L" ],
  "stats" : { ... },
  "ok" : 1
}
```

For information on `distinct` (page 310) and array fields, see the *Behavior* (page 311) section.

Specify Query with `distinct` The following example returns the distinct values for the field `sku`, embedded in the `item` field, from the documents whose `dept` is equal to `"A"`:

```
db.runCommand ( { distinct: "inventory", key: "item.sku", query: { dept: "A"} } )
```

The command returns a document with a field named `values` that contains the distinct `sku` values:

```
{
  "values" : [ "111", "333" ],
  "stats" : { ... },
  "ok" : 1
}
```

Override Default Read Concern To override the default read concern level of `"local"`, use the `readConcern` option.

The following operation on a replica set specifies a `https://docs.mongodb.org/manual/reference/read-concern` of `"majority"` to read the most recent copy of the data confirmed as having been written to a majority of the nodes.

Note:

- To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

  Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

- Regardless of the *read concern* level, the most recent data on a node may not reflect the most recent version of the data in the system.

```
db.runCommand(
   {
     distinct: "restaurants",
     key: "rating",
     query: { cuisine: "italian" },
     readConcern: { level: "majority" }
   }
)
```

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

group

On this page

- Definition (page 313)
- Behavior (page 314)
- Examples (page 315)

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

field document cond Optional. The selection criteria to determine which documents in the collection to process. If you omit the `cond` field, `group` (page 313) processes all the documents in the collection for the group operation.

field function finalize Optional. A function that runs each item in the result set before `group` (page 313) 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 75). However, the `db.collection.group()` (page 75) method takes the `keyf` field and the `reduce` field whereas the `group` (page 313) command takes the `$keyf` field and the `$reduce` field.

Behavior

Limits and Restrictions The `group` (page 313) 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 940).

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 318). Previous versions had a limit of 10,000 elements.

Prior to 2.4, the `group` (page 313) command took the `mongod` (page 770) 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 318), the `group` (page 313) command, and `$where` (page 558) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 803) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 318), `group` (page 313) commands, or `$where` (page 558) 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 318), the `group` (page 313) command, and `$where` (page 558) operator expressions in MongoDB 2.4:

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

JavaScript in MongoDB

Although `group` (page 313) 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 75) 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:

https://docs.mongodb.org/manual/core/aggregation-pipeline

mapReduce

On this page

- Requirements for the map Function (page 320)
- Requirements for the reduce Function (page 321)
- Requirements for the finalize Function (page 322)
- out Options (page 322)
- Map-Reduce Examples (page 323)
- Output (page 325)
- Additional Information (page 327)

mapReduce

The `mapReduce` (page 318) command allows you to run *map-reduce* aggregation operations over a collection. The `mapReduce` (page 318) 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>,
                 bypassDocumentValidation: <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 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 320) for more information.

field 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 321) 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 322) for more information.

field document query Optional. Specifies the selection criteria using *query operators* (page 527) for determining the documents input to the `map` function.

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

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

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

See *Requirements for the finalize Function* (page 322) for more information.

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

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

field boolean bypassDocumentValidation Optional. Enables `mapReduce` (page 318) to bypass document validation during the operation. This lets you insert documents that do not meet the validation requirements.

New in version 3.2.

The following is a prototype usage of the `mapReduce` (page 318) 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 318) 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 318), the `group` (page 313) command, and `$where` (page 558) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 803) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 318), `group` (page 313) commands, or `$where` (page 558) 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 318), the `group` (page 313) command, and `$where` (page 558) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	
`args`	`assert()`	`Map()`
`MaxKey`	`BinData()`	`MD5()`
`MinKey`	`DBPointer()`	`NumberInt()`
	`DBRef()`	`NumberLong()`
	`doassert()`	`ObjectId()`
	`emit()`	`print()`
	`gc()`	`printjson()`
	`HexData()`	`printjsononeline()`
	`hex_md5()`	`sleep()`
	`isNumber()`	`Timestamp()`
	`isObject()`	`tojson()`
	`ISODate()`	`tojsononeline()`
	`isString()`	`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 940).
- 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 940). 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 940).

Map-Reduce Examples In the `mongo` (page 803) shell, the `db.collection.mapReduce()` (page 90) method is a wrapper around the `mapReduce` (page 318) command. The following examples use the `db.collection.mapReduce()` (page 90) 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 `https://docs.mongodb.org/manual/tutorial/perf`

Output The `mapReduce` (page 318) command adds support for the `bypassDocumentValidation` option, which lets you bypass *document validation* (page 991) when inserting or updating documents in a collection with validation rules.

If you set the *out* (page 322) parameter to write the results to a collection, the `mapReduce` (page 318) 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 322) parameter to output the results inline, the mapReduce (page 318) 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 322) did not specify the database name, or
- a document with both `db` and `collection` fields if *out* (page 322) 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 318) command.

mapReduce.counts.**input**

The number of documents the mapReduce (page 318) command called the `map` function.

mapReduce.counts.**emit**

The number of times the mapReduce (page 318) command called the `emit` function.

mapReduce.counts.**reduce**

The number of times the mapReduce (page 318) command called the `reduce` function.

mapReduce.counts.**output**

The number of output values produced.

mapReduce.**ok**

A value of `1` indicates the mapReduce (page 318) command ran successfully. A value of `0` indicates an error.

Additional Information

- `https://docs.mongodb.org/manual/tutorial/troubleshoot-map-function`
- `https://docs.mongodb.org/manual/tutorial/troubleshoot-reduce-function`
- `db.collection.mapReduce()` (page 90)
- `https://docs.mongodb.org/manual/aggregation`

For a detailed comparison of the different approaches, see *Aggregation Commands Comparison* (page 753).

Geospatial Commands

Geospatial Commands

Name	Description
`geoNear` (page 327)	Performs a geospatial query that returns the documents closest to a given point.
`geoSearch` (page 332)	Performs a geospatial query that uses MongoDB's *haystack index* functionality.

geoNear

On this page

- Definition (page 327)
- Considerations (page 328)
- Command Syntax (page 329)
- Behavior (page 329)
- Examples (page 329)
- Output (page 332)

Definition

`geoNear`

Returns documents in order of proximity to a specified point, from the nearest to farthest. `geoNear` (page 327) requires a geospatial index.

The `geoNear` (page 327) 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, *legacy coordinate pair* near:

The point for which to find the closest documents.

If using a `2dsphere` index, you can specify the point as either a GeoJSON point or legacy coordinate pair.

If using a `2d` index, specify the point as a legacy coordinate pair.

field boolean spherical Required if using a `2dsphere` index. Determines how MongoDB calculates the distance. The default value is `false`.

If `true`, then MongoDB uses spherical geometry to calculate distances in meters if the specified (`near`) point is a GeoJSON point and in radians if the specified (`near`) point is a legacy coordinate pair.

If `false`, then MongoDB uses 2d planar geometry to calculate distance between points.

If using a `2dsphere` index, `spherical` must be `true`.

field number limit Optional. The maximum number of documents to return. The default value is `100`. See also the `num` option.

field number num Optional. 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 Optional. 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 Optional. 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 Optional. 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 565) predicate in the `query` field of the `geoNear` (page 327) command.

field number distanceMultiplier Optional. 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 Optional. 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 an embedded document, use *dot notation*.

field boolean uniqueDocs Optional. 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 575) operator has no impact on results.

field document readConcern Optional. Specifies the *read concern*. The default level is `"local"`.

To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

New in version 3.2.

Considerations `geoNear` (page 327) requires a geospatial index. However, the `geoNear` (page 327) command requires that a collection have *at most* only one `2d index` and/or only one `2dsphere`.

You cannot specify a `$near` (page 565) predicate in the `query` field of the `geoNear` (page 327) command.

Command Syntax

`2dsphere` Index If using a `2dsphere` index, you can specify either a `GeoJSON` point or a legacy coordinate pair for the `near` value.

You must include `spherical: true` in the syntax.

With `spherical: true`, if you specify a GeoJSON point, MongoDB uses meters as the unit of measurement:

```
db.runCommand( {
   geoNear: <collection> ,
   near: { type: "Point" , coordinates: [ <coordinates> ] } ,
   spherical: true,
   ...
} )
```

With `spherical: true`, if you specify a legacy coordinate pair, MongoDB uses radians as the unit of measurement:

```
db.runCommand( {
   geoNear: <collection> ,
   near: [ <coordinates> ],
   spherical: true,
   ...
} )
```

`2d` Index To query a `2d` index, use the following syntax:

```
db.runCommand( {
   geoNear: <collection>,
   near : [ <coordinates> ],
   ...
} )
```

If you specify `spherical: true`, MongoDB uses spherical geometry to calculate distances in radians. Otherwise, MongoDB uses planar geometry to calculate distances between points.

Behavior `geoNear` (page 327) sorts documents by distance. If you also include a `sort()` (page 156) for the query, `sort()` (page 156) re-orders the matching documents, effectively overriding the sort operation already performed by `geoNear` (page 327). When using `sort()` (page 156) with geospatial queries, consider using `$geoWithin` (page 560) operator, which does not sort documents, instead of `geoNear` (page 327).

Because `geoNear` (page 327) 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.

The `geoNear` (page 327) command provides an alternative to the `$near` (page 565) operator. In addition to the functionality of `$near` (page 565), `geoNear` (page 327) returns additional diagnostic information.

Examples The following examples run the `geoNear` (page 327) command on the collection `places` that has a `2dsphere` index.

Specify a Query Condition The following `geoNear` (page 327) 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:

```
{
  "waitedMS" : NumberLong(0),
  "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:

```
{
  "waitedMS" : NumberLong(0),
  "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
}
```

Override Default Read Concern To override the default read concern level of `"local"`, use the `readConcern` option.

The following operation on a replica set specifies a `https://docs.mongodb.org/manual/reference/read-concern` of `"majority"` to read the most recent copy of the data confirmed as having been written to a majority of the nodes.

Note:

- To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

  Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

- Regardless of the *read concern* level, the most recent data on a node may not reflect the most recent version of the data in the system.

```
db.runCommand(
   {
      geoNear: "places",
      near: { type: "Point", coordinates: [ -73.9667, 40.78 ] },
      spherical: true,
      query: { category: "public" },
      readConcern: { level: "majority" }
   }
)
```

To ensure that a single thread can read its own writes, use "majority" read concern and "majority" write concern against the primary of the replica set.

Output The geoNear (page 327) command returns a document with the following fields:

geoNear.**results**
  An array with the results of the geoNear (page 327) 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 327) 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 327) 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 327) command and coordinates of the documents returned as results.

geoNear.stats.**maxDistance**
  The maximum distance between the coordinates defined in the geoNear (page 327) 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 327) search succeeded. A value of 0 indicates an error.

geoSearch

On this page

- Behavior (page 333)
- Examples (page 333)

geoSearch
  The geoSearch (page 332) 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.")

  The geoSearch (page 332) command accepts a *document* that contains the following fields.

  field string geoSearch The collection to query.

field document search Query to filter documents.

field array near Coordinates of a point.

field number maxDistance Optional. Maximum distance from the specified point.

field number limit Optional. Maximum number of documents to return.

field document readConcern Optional. Specifies the *read concern*. The default level is `"local"`.

To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

New in version 3.2.

Behavior Unless specified otherwise, the `geoSearch` (page 332) command limits results to 50 documents.

Important: `geoSearch` (page 332) is not supported for sharded clusters.

Examples Consider the following example:

```
{
   geoSearch : "places",
   near : [33, 33],
   maxDistance : 6,
   search : { type : "restaurant" },
   limit : 30
}
```

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.

Override Default Read Concern To override the default read concern level of `"local"`, use the `readConcern` option.

The following operation on a replica set specifies a `https://docs.mongodb.org/manual/reference/read-concern` of `"majority"` to read the most recent copy of the data confirmed as having been written to a majority of the nodes.

Note:

- To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

  Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

- Regardless of the *read concern* level, the most recent data on a node may not reflect the most recent version of the data in the system.

```
db.runCommand(
   {
      geoSearch: "places",
      near: [ -73.9667, 40.78 ],
      search : { type : "restaurant" },
      readConcern: { level: "majority" }
    }
)
```

To ensure that a single thread can read its own writes, use "majority" read concern and "majority" write concern against the primary of the replica set.

Query and Write Operation Commands

Query and Write Operation Commands

Name	Description
find (page 334)	Selects documents in a collection.
insert (page 337)	Inserts one or more documents.
update (page 340)	Updates one or more documents.
delete (page 345)	Deletes one or more documents.
findAndModify (page 348)	Returns and modifies a single document.
getMore (page 354)	Returns batches of documents currently pointed to by the cursor.
getLastError (page 355)	Returns the success status of the last operation.
getPrevError (page 357)	Returns status document containing all errors since the last resetError (page 357) command.
resetError (page 357)	Resets the last error status.
eval (page 358)	Deprecated. Runs a JavaScript function on the database server.
parallelCollectionScan (page 360)	Lets applications use multiple parallel cursors when reading documents from a collection.

find

On this page

- Definition (page 334)
- Examples (page 336)

Definition

find

New in version 3.2.

Executes a query and returns the first batch of results and the cursor id, from which the client can construct a cursor.

The find (page 334) command has the following form:

```
{
   "find": <string>,
   "filter": <document>,
   "sort": <document>,
   "projection": <document>,
   "hint": <document or string>,
   "skip": <int>,
   "limit": <int>,
   "batchSize": <int>,
   "singleBatch": <bool>,
   "comment": <string>,
   "maxScan": <int>,
   "maxTimeMS": <int>,
   "readConcern": <document>,
   "max": <document>,
   "min": <document>,
   "returnKey": <bool>,
   "showRecordId": <bool>,
   "snapshot": <bool>,
   "tailable": <bool>,
   "oplogReplay": <bool>,
   "noCursorTimeout": <bool>,
   "awaitData": <bool>,
   "allowPartialResults": <bool>
}
```

The command accepts the following fields:

field string find The name of the collection to query.

field document filter Optional. The query predicate. If unspecified, then all documents in the collection will match the predicate.

field document sort Optional. The sort specification for the ordering of the results.

field document projection Optional. The *projection specification* to determine which fields to include in the returned documents. See *projections* and *Projection Operators* (page 588).

field string or document hint Optional. Index specification. Specify either the index name as a string or the index key pattern. If specified, then the query system will only consider plans using the hinted index.

field Positive integer skip Optional. Number of documents to skip. Defaults to 0.

field Non-negative integer limit Optional. The maximum number of documents to return. If unspecified, then defaults to no limit. A limit of 0 is equivalent to setting no limit.

field non-negative integer batchSize Optional. The number of documents to return in the first batch. Defaults to 101. A batchSize of 0 means that the cursor will be established, but no documents will be returned in the first batch.

Unlike the previous wire protocol version, a batchSize of 1 for the `find` (page 334) command does not close the cursor.

field boolean singleBatch Optional. Determines whether to close the cursor after the first batch. Defaults to false.

field string comment Optional. A comment to attach to the query to help interpret and trace query `profile` (page 484) data.

field positive integer maxScan Optional. Maximum number of documents or index keys to scan when executing the query.

field positive integer maxTimeMS Optional. The cumulative time limit in milliseconds for processing operations on the cursor. MongoDB aborts the operation at the earliest following *interrupt point*.

field document readConcern Optional. Specifies the *read concern*. The default level is `"local"`.

To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the *`--enableMajorityReadConcern`* (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

Note: The `getMore` (page 354) command uses the `readConcern` level specified in the originating `find` command.

field document max Optional. The *exclusive* upper bound for a specific index. See `cursor.max()` (page 147) for details.

field document min Optional. The *inclusive* lower bound for a specific index. See `cursor.min()` (page 148) for details.

field boolean returnKey Optional. If true, returns only the index keys in the resulting documents. Default value is false. If returnKey is true and the `find` (page 334) command does not use an index, the returned documents will be empty.

field boolean showRecordId Optional. Determines whether to return the record identifier for each document. If true, adds a field $recordId to the returned documents.

field boolean snapshot Optional. Prevents the cursor from returning a document more than once because of an intervening write operation.

field boolean tailable Optional. Returns a *tailable cursor* for a capped collections.

field boolean awaitData Optional. Use in conjunction with the tailable option to block a `getMore` (page 354) command on the cursor temporarily if at the end of data rather than returning no data. After a timeout period, `find` (page 334) returns as normal.

field boolean oplogReplay Optional. Internal use for replica sets. To use oplogReplay, you must include the following condition in the filter:

```
{ ts: { $gte: <timestamp> } }
```

field boolean noCursorTimeout Optional. Prevents the server from timing out idle cursors after an inactivity period (10 minutes).

field boolean allowPartialResults Optional. For queries against a sharded collection, returns partial results from the `mongos` (page 792) if some shards are unavailable instead of throwing an error.

Examples

Specify a Sort and Limit The following command runs the `find` (page 334) command filtering on the `rating` field and the `cuisine` field. The command includes a `projection` to only return the following fields in the matching documents: `_id`, `name`, `rating`, and `address` fields.

The command sorts the documents in the result set by the `name` field and limits the result set to 5 documents.

```
db.runCommand(
   {
     find: "restaurants",
     filter: { rating: { $gte: 9 }, cuisine: "italian" },
     projection: { name: 1, rating: 1, address: 1 },
     sort: { name: 1 },
     limit: 5
   }
)
```

Override Default Read Concern To override the default read concern level of `"local"`, use the `readConcern` option.

The following operation on a replica set specifies a *read concern* of `"majority"` to read the most recent copy of the data confirmed as having been written to a majority of the nodes.

Note:

- To use a *read concern* level of `"majority"`, you must use the WiredTiger storage engine and start the `mongod` (page 770) instances with the `--enableMajorityReadConcern` (page 781) command line option (or the `replication.enableMajorityReadConcern` (page 922) setting if using a configuration file).

  Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

- Regardless of the *read concern* level, the most recent data on a node may not reflect the most recent version of the data in the system.

```
db.runCommand(
    {
      find: "restaurants",
      filter: { rating: { $lt: 5 } },
      readConcern: { level: "majority" }
    }
)
```

To ensure that a single thread can read its own writes, use `"majority"` read concern and `"majority"` write concern against the primary of the replica set.

The `getMore` (page 354) command uses the `readConcern` level specified in the originating `find` (page 334) command.

A `readConcern` can be specified for the `mongo` (page 803) shell method `db.collection.find()` (page 51) using the `cursor.readConcern` (page 152) method:

```
db.restaurants.find( { rating: { $lt: 5 } } ).readConcern("majority")
```

See also:

Driver Compatibility Changes (page 1002)

insert

On this page

- Definition (page 337)
- Behavior (page 338)
- Examples (page 338)
- Output (page 339)

Definition

insert

New in version 2.6.

The `insert` (page 337) 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> },
   bypassDocumentValidation: <boolean>
}
```

The `insert` (page 337) 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 Optional. 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 Optional. A document that expresses the `write concern` of the `insert` (page 337) command. Omit to use the default write concern.

field boolean bypassDocumentValidation Optional. Enables `insert` to bypass document validation during the operation. This lets you insert documents that do not meet the validation requirements.

New in version 3.2.

Returns A document that contains the status of the operation. See *Output* (page 339) 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 940).

The total number of documents in the `documents` array must be less than or equal to the `maximum bulk size`.

The `insert` (page 337) command adds support for the `bypassDocumentValidation` option, which lets you bypass *document validation* (page 991) when inserting or updating documents in a collection with validation rules.

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 339) 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 339) 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 339) 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 337) of a single document:

```
{ ok: 1, n: 1 }
```

The following is an example document returned for an `insert` (page 337) 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
      }
   ]
}
```

update

On this page

- Definition (page 340)
- Behavior (page 341)
- Examples (page 342)
- Output (page 343)

Definition

`update`

New in version 2.6.

The `update` (page 340) command modifies documents in a collection. A single `update` (page 340) command can contain multiple update statements. The update methods provided by the MongoDB drivers use this command internally.

The `update` (page 340) 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> },
   bypassDocumentValidation: <boolean>
}
```

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 Optional. 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 Optional. A document expressing the `write concern` of the `update` (page 340) command. Omit to use the default write concern.

field boolean bypassDocumentValidation Optional. Enables `update` to bypass document validation during the operation. This lets you update documents that do not meet the validation requirements.

New in version 3.2.

Each element of the `updates` array contains the following fields:

field document q The query that matches documents to update. Use the same *query selectors* (page 527) as used in the `find()` (page 51) method.

field document u The modifications to apply. For details, see *Behavior* (page 341).

field boolean upsert Optional. 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 Optional. 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 343) for details.

Behavior The `<update>` document can contain either all *update operator* (page 595) expressions or all `field:value` expressions.

Update Operator Expressions If the `<update>` document contains all *update operator* (page 595) expressions, as in:

```
{
  $set: { status: "D" },
  $inc: { quantity: 2 }
}
```

Then, the `update` (page 340) 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 340) command *replaces* the matching document with the update document. The `update` (page 340) command can only replace a *single* matching document; i.e. the `multi` field cannot be `true`. The `update` (page 340) 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 940).

The total number of update statements in the `updates` array must be less than or equal to the `maximum bulk size`.

Document Validation The `update` (page 340) command adds support for the `bypassDocumentValidation` option, which lets you bypass *document validation* (page 991) when inserting or updating documents in a collection with validation rules.

Examples

Update Specific Fields of One Document Use *update operators* (page 595) to update only the specified fields of a document.

For example, given a `users` collection, the following command uses the `$set` (page 600) and `$inc` (page 595) 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 343) for details.

```
{ "ok" : 1, "nModified" : 1, "n" : 1 }
```

Update Specific Fields of Multiple Documents Use *update operators* (page 595) 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 `$set` (page 600) and `$inc` (page 595) 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 343) 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 343) 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 600) expression updates the value to the current value, `n` (page 343) can be greater than `nModified` (page 343).

`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 343) can be less than `n` (page 343).

`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 343) 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 340) 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}."
      },
   ]
}
```

delete

On this page

- Definition (page 345)
- Behavior (page 345)
- Examples (page 345)
- Output (page 347)

Definition

delete

New in version 2.6.

The `delete` (page 345) command removes documents from a collection. A single `delete` (page 345) 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 345) 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 Optional. 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 Optional. A document expressing the `write concern` of the `delete` (page 345) command. Omit to use the default write concern.

Each element of the `deletes` array contains the following 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 347) 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 940).

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 347) 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 347) 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 347) 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 347) 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 347) 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 345) command:

```
{ ok: 1, n: 1 }
```

The following is an example document returned for a `delete` (page 345) command that encountered an error:

```
{
   "ok" : 1,
   "n" : 0,
   "writeErrors" : [
      {
         "index" : 0,
         "code" : 10101,
         "errmsg" : "can't remove from a capped collection: test.cappedLog"
      }
   ]
}
```

findAndModify

On this page

- Definition (page 348)
- Output (page 349)
- Behavior (page 349)
- Examples (page 351)

Definition

`findAndModify`

The `findAndModify` (page 348) 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>,
  bypassDocumentValidation: <boolean>,
  writeConcern: <document>
}
```

The `findAndModify` (page 348) command takes the following fields:

field document query Optional. The selection criteria for the modification. The `query` field employs the same *query selectors* (page 527) as used in the `db.collection.find()` (page 51) method. Although the query may match multiple documents, `findAndModify` (page 348) **will only select one document to modify**.

field document sort Optional. Determines which document the operation modifies if the query selects multiple documents. `findAndModify` (page 348) modifies the first document in the sort order specified by this argument.

field 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`.

field 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 595) or `field: value` specifications to modify the selected document.

field boolean new Optional. When `true`, returns the modified document rather than the original. The `findAndModify` (page 348) method ignores the `new` option for `remove` operations. The default is `false`.

field document fields Optional. 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*.

field boolean upsert Optional. Used in conjunction with the `update` field.

When `true`, `findAndModify` (page 348) creates a new document if no document matches the `query`, or if documents match the `query`, `findAndModify` (page 348) performs an update. To avoid multiple upserts, ensure that the `query` fields are *uniquely indexed*.

The default is `false`.

field boolean bypassDocumentValidation Optional. Enables `findAndModify` to bypass document validation during the operation. This lets you update documents that do not meet the validation requirements.

New in version 3.2.

field document writeConcern Optional. A document expressing the `write concern`. Omit to use the default write concern.

New in version 3.2.

field string findAndModify The collection against which to run the command.

Output The `findAndModify` (page 348) command returns a document with the following fields:

field document value Contains the command's returned value. See *value* (page 349) for details.

field document lastErrorObject Contains information about updated documents. See *lastErrorObject* (page 349) for details.

field number ok Contains the command's execution status. `1` on success, or `0` if an error occurred.

`lastErrorObject` The `lastErrorObject` embedded document contains the following fields:

field boolean updatedExisting Contains `true` if an `update` operation modified an existing document.

field document upserted Contains the *objectid* of the inserted document if an `update` operation with `upsert: true` resulted in a new document.

`value` For `remove` operations, `value` contains the removed document if the query matches a document. If the query does not match a document to remove, `value` contains `null`.

For `update` operations, the `value` embedded document contains the following:

- If the `new` parameter is not set or is `false`:
  - the pre-modification document if the query matches a document;
  - otherwise, `null`.
- If `new` is `true`:
  - the modified document if the query returns a match;
  - the inserted document if `upsert: true` and no document matches the query;
  - otherwise, `null`.

Changed in version 3.0: In previous versions, if for the update, `sort` is specified, and `upsert: true`, and the `new` option is not set or `new: false`, `findAndModify` (page 348) returns an empty document `{}` in the `value` field instead of `null`.

Behavior

Upsert and Unique Index When the findAndModify (page 348) 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 348) commands will exhibit one of the following behaviors:

- Exactly one findAndModify (page 348) successfully inserts a new document.
- Zero or more findAndModify (page 348) commands update the newly inserted document.
- Zero or more findAndModify (page 348) 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 348) in a *sharded* environment, the query must contain the *shard key* for all operations against the shard cluster. findAndModify (page 348) operations issued against mongos (page 792) instances for non-sharded collections function normally.

Document Validation The findAndModify (page 348) command adds support for the bypassDocumentValidation option, which lets you bypass *document validation* (page 991) when inserting or updating documents in a collection with validation rules.

Comparisons with the update Method When updating a document, findAndModify (page 348) and the update() (page 117) method operate differently:

- By default, both operations modify a single document. However, the update() (page 117) method with its multi option can modify more than one document.
- If multiple documents match the update criteria, for findAndModify (page 348), you can specify a sort to provide some measure of control on which document to update.

  With the default behavior of the update() (page 117) method, you cannot specify which single document to update when multiple documents match.
- By default, findAndModify (page 348) 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 117) method returns a WriteResult (page 289) object that contains the status of the operation. To return the updated document, use the find() (page 51) 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.

When modifying a *single* document, both findAndModify (page 348) and the update() (page 117) method *atomically* update the document. See https://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" : {
     "connectionId" : 1,
     "updatedExisting" : true,
     "n" : 1,
     "syncMillis" : 0,
     "writtenTo" : null,
     "err" : null,
     "ok" : 1
  },
  value" : {
    "_id" : ObjectId("54f62d2885e4be1f982b9c9c"),
    "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 803) shell and many *drivers* provide a `findAndModify()` (page 57) 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 57) shell helper method returns only the unmodified document, or if `new` is `true`, the modified document.

```
{
   "_id" : ObjectId("54f62d2885e4be1f982b9c9c"),
   "name" : "Tom",
   "state" : "active",
   "rating" : 100,
   "score" : 5
}
```

upsert: true The following `findAndModify` (page 348) 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 containing `null`.

```
{
  "value" : null,
  "lastErrorObject" : {
     "updatedExisting" : false,
     "n" : 1,
     "upserted" : ObjectId("54f62c8bc85d4472eadea26f")
  },
  "ok" : 1
}
```

Return New Document The following findAndModify (page 348) 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" : {
     "connectionId" : 1,
     "updatedExisting" : false,
     "upserted" : ObjectId("54f62bbfc85d4472eadea26d"),
     "n" : 1,
     "syncMillis" : 0,
     "writtenTo" : null,
     "err" : null,
     "ok" : 1
  },
  "value" : {
     "_id" : ObjectId("54f62bbfc85d4472eadea26d"),
     "name" : "Pascal",
     "rating" : 25,
     "state" : "active",
     "score" : 1
  },
  "ok" : 1
}
```

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

The method returns the deleted document:

```
{
  "lastErrorObject" : {
     "connectionId" : 1,
     "n" : 1,
     "syncMillis" : 0,
     "writtenTo" : null,
     "err" : null,
     "ok" : 1
  },
  "value" : {
     "_id" : ObjectId("54f62a6785e4be1f982b9c9b"),
     "name" : "XYZ123",
     "score" : 1,
     "state" : "active",
     "rating" : 3
  },
  "ok" : 1
}
```

See also:

https://docs.mongodb.org/manual/tutorial/perform-findAndModify-quorum-reads

getMore

On this page

- Definition (page 354)

Definition

getMore

New in version 3.2.

Use in conjunction with commands that return a cursor, e.g. `find` (page 334) and `aggregate` (page 303), to return subsequent batches of documents currently pointed to by the cursor.

The `getMore` (page 354) command has the following form:

```
{
   "getMore": <long>,
   "collection": <string>,
   "batchSize": <int>,
   "maxTimeMS": <int>
}
```

The command accepts the following fields:

field long getMore The cursor id.

field string collection The name of the collection over which the cursor is operating.

field non-negative integer batchSize Optional. The number of documents to return in the batch.

field positive integer maxTimeMS Optional. The maximum time period in milliseconds the `getMore()` (page 354) operation will block waiting for new data to be inserted into the capped collection.

Requires that the cursor on which this `getMore()` (page 354) is acting is an `awaitData` cursor. See the `awaitData` parameter for `find()` (page 334).

See also:

Driver Compatibility Changes (page 1002)

getLastError

On this page

- Definition (page 355)
- Output (page 355)
- Examples (page 357)

Definition

`getLastError`

Changed in version 2.6: A new protocol for *write operations* (page 1095) integrates write concerns with the write operations, eliminating the need for a separate `getLastError` (page 355). *Most write methods* (page 1102) now return the status of the write operation, including error information. In previous versions, clients typically used the `getLastError` (page 355) in combination with a write operation to verify that the write succeeded.

Returns the error status of the preceding write operation on the *current connection*.

`getLastError` (page 355) uses the following prototype form:

```
{ getLastError: 1 }
```

`getLastError` (page 355) 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 770) does not have journaling enabled, this option has no effect. If this option is enabled for a write operation, `mongod` (page 770) will wait *no more* than 1/3 of the current `commitIntervalMs` (page 916) 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 the voting replica set members.

Changed in version 3.0: In previous versions, `majority` referred to the majority of all members of the replica set.

If using `w`, you should also use `wtimeout`. Specifying a value for `w` without also providing a `wtimeout` may cause `getLastError` (page 355) to block indefinitely.

field integer wtimeout Optional. 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 355) command will return with an error status.

See also:

`https://docs.mongodb.org/manual/reference/write-concern`

Output

Each `getLastError()` command returns a document containing a subset of the fields listed below.

`getLastError.`**`ok`**

`ok` (page 355) is `true` when the `getLastError` (page 355) command completes successfully.

Note: A value of `true` does *not* indicate that the preceding operation did not produce an error.

`getLastError.`**`err`**

`err` (page 355) is `null` unless an error occurs. When there was an error with the preceding operation, `err` contains a string identifying the error.

`getLastError.`**`errmsg`**

New in version 2.6.

`errmsg` (page 355) contains the description of the error. `errmsg` (page 355) only appears if there was an error with the preceding operation.

`getLastError.`**`code`**

`code` (page 356) reports the preceding operation's error code. For description of the error, see `err` (page 355) and `errmsg` (page 355).

`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 356) is the *optime* timestamp in the *oplog* of the change.

`getLastError.`**`n`**

If the preceding operation was an update or a remove operation, but *not* a `findAndModify` (page 348) operation, `n` (page 356) reports the number of documents matched by the update or remove operation.

For a remove operation, the number of matched documents will equal the number removed.

For an update operation, if the update operation results in no change to the document, such as setting the value of the field to its current value, the number of matched documents may be smaller than the number of documents actually modified. If the update includes the `upsert:true` option and results in the creation of a new document, `n` (page 356) returns the number of documents inserted.

`n` (page 356) is `0` if reporting on an update or remove that occurs through a `findAndModify` (page 348) operation.

`getLastError.`**`syncMillis`**

`syncMillis` (page 356) is the number of milliseconds spent waiting for the write to disk operation (e.g. write to journal files).

`getLastError.`**`shards`**

When issued against a sharded cluster after a write operation, `shards` (page 356) identifies the shards targeted in the write operation. `shards` (page 356) 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 356) is only present if the write operation targets exactly one shard.

`getLastError.`**`updatedExisting`**

`updatedExisting` (page 356) 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 356) is the value of `_id` field of the document.

Changed in version 2.6: Earlier versions of MongoDB included `upserted` (page 356) 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 355).

See

`https://docs.mongodb.org/manual/reference/write-concern` for more information about `w` values.

---

getLastError.**wtimeout**

wtimeout (page 357) is true if the getLastError (page 355) timed out because of the wtimeout setting to getLastError (page 355).

getLastError.**waited**

If the preceding operation specified a timeout using the wtimeout setting to getLastError (page 355), then waited (page 357) reports the number of milliseconds getLastError (page 355) waited before timing out.

getLastError.**wtime**

getLastError.wtime (page 357) is the number of milliseconds spent waiting for the preceding operation to complete. If getLastError (page 355) timed out, wtime (page 357) and getLastError.waited are equal.

getLastError.**writtenTo**

If writing to a replica set, writtenTo (page 357) is an array that contains the hostname and port number of the members that confirmed the previous write operation, based on the value of the w field in the command.

Examples

Confirm Replication to Two Replica Set Members The following example ensures the preceding operation has replicated to two members (the primary and one other member). The command also specifies a timeout of 5000 milliseconds to ensure that the:dbcommand:*getLastError* command does not block forever if MongoDB cannot satisfy the requested write concern:

```
db.runCommand( { getLastError: 1, w: 2, wtimeout:5000 } )
```

Confirm Replication to a Majority of a Replica Set The following example ensures the write operation has replicated to a majority of the voting members of the replica set. The command also specifies a timeout of 5000 milliseconds to ensure that the:dbcommand:*getLastError* command does not block forever if MongoDB cannot satisfy the requested write concern:

```
db.runCommand( { getLastError: 1, w: "majority", wtimeout:5000 } )
```

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.

getPrevError

getPrevError

The getPrevError (page 357) command returns the errors since the last resetError (page 357) command.

See also:

db.getPrevError() (page 188)

resetError

resetError

The resetError (page 357) command resets the last error status.

See also:

db.resetError() (page 196)

eval

On this page

- Definition (page 358)
- Behavior (page 358)
- Example (page 359)

Definition

eval

Deprecated since version 3.0.

The `eval` (page 358) command evaluates JavaScript functions on the database server.

The `eval` (page 358) 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 Optional. An array of arguments to pass to the JavaScript function. Omit if the function does not take arguments.

field boolean nolock Optional. By default, `eval` (page 358) takes a global write lock before evaluating the JavaScript function. As a result, `eval` (page 358) blocks all other read and write operations to the database while the `eval` (page 358) operation runs. Set `nolock` to `true` on the `eval` (page 358) command to prevent the `eval` (page 358) 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 358) 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 358) takes a global write lock while evaluating the JavaScript function. As a result, `eval` (page 358) blocks all other read and write operations to the database while the `eval` (page 358) operation runs.

To prevent the taking of the global write lock while evaluating the JavaScript code, use the `eval` (page 358) 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 358) 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 358) with *sharded* collections. In general, you should avoid using `eval` (page 358) in *sharded clusters*; nevertheless, it is possible to use `eval` (page 358) 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 358). Providing such access is not recommended, but if your organization requires a user to run `eval` (page 358), 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 358) executed in a single thread.

Example The following example uses `eval` (page 358) 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 803) shell provides a helper method `db.eval()` (page 178) [10], 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 );
```

[10] The helper `db.eval()` (page 178) in the `mongo` (page 803) shell wraps the `eval` (page 358) command. Therefore, the helper method shares the characteristics and behavior of the underlying command with *one exception*: `db.eval()` (page 178) method does not support the `nolock` option.

If you want to use the server's interpreter, you must run `eval` (page 358). Otherwise, the `mongo` (page 803) shell's JavaScript interpreter evaluates functions entered directly into the shell.

If an error occurs, `eval` (page 358) 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 '{ retur
   "code" : 16722,
   "ok" : 0
}
```

See also:

https://docs.mongodb.org/manual/core/server-side-javascript

parallelCollectionScan	**On this page** • Example (page 361) • Output (page 361)

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 360) 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 360) 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.

field document readConcern Optional. Specifies the *read concern*. The default level is `"local"`.

To use a *read concern* level of "majority", you must use the WiredTiger storage engine and start the mongod (page 770) instances with the *--enableMajorityReadConcern* (page 781) command line option (or the replication.enableMajorityReadConcern (page 922) setting if using a configuration file).

Only replica sets using protocol version 1 support "majority" read concern. Replica sets running protocol version 0 do not support "majority" read concern.

To ensure that a single thread can read its own writes, use "majority" read concern and "majority" write concern against the primary of the replica set.

New in version 3.2.

parallelCollectionScan (page 360) is only available for mongod (page 770), and it cannot operate on a sharded cluster.

Example

Override Default Read Concern To override the default read concern level of "local", use the readConcern option.

The following operation on a replica set specifies a https://docs.mongodb.org/manual/reference/read-concern of "majority" to read the most recent copy of the data confirmed as having been written to a majority of the nodes.

Note:

- To use a *read concern* level of "majority", you must use the WiredTiger storage engine and start the mongod (page 770) instances with the *--enableMajorityReadConcern* (page 781) command line option (or the replication.enableMajorityReadConcern (page 922) setting if using a configuration file).

  Only replica sets using protocol version 1 support "majority" read concern. Replica sets running protocol version 0 do not support "majority" read concern.

- Regardless of the *read concern* level, the most recent data on a node may not reflect the most recent version of the data in the system.

```
db.runCommand(
   {
     parallelCollectionScan: "restaurants",
     numCursors: 5,
     readConcern: { level: "majority" }
   }
)
```

To ensure that a single thread can read its own writes, use "majority" read concern and "majority" write concern against the primary of the replica set.

Output The parallelCollectionScan (page 360) 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 360) command succeeded. A value of `0` indicates an error.

Query Plan Cache Commands

Query Plan Cache Commands

Name	Description
`planCacheListFilters` (page 362)	Lists the index filters for a collection.
`planCacheSetFilter` (page 364)	Sets an index filter for a collection.
`planCacheClearFilters` (page 366)	Clears index filter(s) for a collection.
`planCacheListQueryShapes` (page 367)	Displays the query shapes for which cached query plans exist.
`planCacheListPlans` (page 369)	Displays the cached query plans for the specified query shape.
`planCacheClear` (page 370)	Removes cached query plan(s) for a collection.

planCacheListFilters

On this page

- Definition (page 362)
- Required Access (page 363)
- Output (page 363)

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 362) command has the following field:

field string planCacheListFilters The name of the collection.

Returns Document listing the index filters. See *Output* (page 363).

Required Access A user must have access that includes the planCacheIndexFilter action.

Output The planCacheListFilters (page 362) 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 363) 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 363) predicate of { "type": "electronics", "status" : "A" } covers the following query predicates:

```
{ type: "food", status: "A" }
{ type: "utensil", status: "D" }
```

Together with the sort (page 363) and the projection (page 363), the query (page 363) 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 363) and the projection (page 363), the sort (page 363) 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 363) and the sort (page 363), the projection (page 363) 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 366), `planCacheSetFilter` (page 364)

planCacheSetFilter

On this page

- Definition (page 364)
- Required Access (page 365)
- Examples (page 365)

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 364) 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 Optional. 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 Optional. 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 366) command.

Required Access A user must have access that includes the `planCacheIndexFilter` action.

Examples

Set Filter on Query Shape Consisting of Predicate Only The following example creates an index filter on the `orders` collection such that for queries that consist 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" } )
```

To see whether MongoDB will apply an index filter for a query shape, check the `indexFilterSet` (page 948) field of either the `db.collection.explain()` (page 48) or the `cursor.explain()` (page 140) method.

Set Filter on Query Shape Consisting of Predicate, Projection, and Sort The following example creates an index filter for the `orders` collection. The filter applies to queries whose predicate is an equality match on the `item` field, where only the `quantity` field is projected and an ascending sort by `order_date` is specified.

```
db.runCommand(
   {
      planCacheSetFilter: "orders",
      query: { item: "ABC" },
      projection: { quantity: 1, _id: 0 },
      sort: { order_date: 1 },
      indexes: [
         { item: 1, order_date: 1 , quantity: 1 }
      ]
   }
)
```

For the query shape, the query optimizer will only consider indexed plans which use the index `{ item: 1, order_date: 1, quantity: 1 }`.

See also:

`planCacheClearFilters` (page 366), `planCacheListFilters` (page 362)

planCacheClearFilters

On this page

- Definition (page 366)
- Required Access (page 366)
- Examples (page 366)

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 366) 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 366) command has the following field:

field string planCacheClearFilters The name of the collection.

field document query Optional. 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 362).

field document sort Optional. The sort associated with the filter to remove, if any.

field document projection Optional. 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 362), `planCacheSetFilter` (page 364)

planCacheListQueryShapes

On this page

- Definition (page 367)
- Required Access (page 368)
- Example (page 368)

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 803) shell provides the wrapper `PlanCache.listQueryShapes()` (page 205) for this command.

The command has the following syntax:

```
db.runCommand(
   {
      planCacheListQueryShapes: <collection>
   }
)
```

The planCacheListQueryShapes (page 367) 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 (page 910), 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 205)

planCacheListPlans

On this page

- Definition (page 369)
- Required Access (page 369)
- Example (page 369)

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 803) shell provides the wrapper `PlanCache.getPlansByQuery()` (page 207) for this command.

The `planCacheListPlans` (page 369) command has the following syntax:

```
db.runCommand(
   {
     planCacheListPlans: <collection>,
     query: <query>,
     sort: <sort>,
     projection: <projection>
   }
```

The `planCacheListPlans` (page 369) 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 Optional. The projection associated with the *query shape*.

field document sort Optional. The sort associated with the *query shape*.

To see the query shapes for which cached query plans exist, use the `planCacheListQueryShapes` (page 367) command.

Required Access On systems running with `authorization` (page 910), 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 367)
- `PlanCache.getPlansByQuery()` (page 207)
- `PlanCache.listQueryShapes()` (page 205)

planCacheClear

On this page

- Definition (page 370)
- Required Access (page 370)
- Examples (page 370)

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 370) command has the following field:

field document query Optional. 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 Optional. The projection associated with the *query shape*.

field document sort Optional. The sort associated with the *query shape*.

To see the query shapes for which cached query plans exist, use the `planCacheListQueryShapes` (page 367) command.

Required Access On systems running with `authorization` (page 910), 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 208)
- `PlanCache.clear()` (page 209)

2.2.2 Database Operations

Authentication Commands

Authentication Commands

Name	Description
`logout` (page 371)	Terminates the current authenticated session.
`authenticate` (page 372)	Starts an authenticated session using a username and password.
`copydbgetnonce` (page 372)	This is an internal command to generate a one-time password for use with the `copydb` (page 433) command.
`getnonce` (page 372)	This is an internal command to generate a one-time password for authentication.
`authSchemaUpgrade` (page 372)	Supports the upgrade process for user data between version 2.4 and 2.6.

logout

`logout`

The `logout` (page 371) command terminates the current authenticated session:

```
{ logout: 1 }
```

Note: If you're not logged in and using authentication, `logout` (page 371) 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 371) 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 371) against this database in order to successfully log out.

Example

Use the `use <database-name>` helper in the interactive `mongo` (page 803) shell, or the following `db.getSiblingDB()` (page 190) in the interactive shell or in `mongo` (page 803) 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 371) to log out of database as in the following operation:

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

authenticate

`authenticate`

Clients use `authenticate` (page 372) to authenticate a connection. When using the shell, use the `db.auth()` (page 229) helper as follows:

```
db.auth( "username", "password" )
```

See

`db.auth()` (page 229) and `https://docs.mongodb.org/manual/security` for more information.

copydbgetnonce

`copydbgetnonce`

Client libraries use `copydbgetnonce` (page 372) to get a one-time password for use with the `copydb` (page 433) 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 372) to generate a one-time password for authentication.

authSchemaUpgrade

New in version 2.6.

`authSchemaUpgrade`

Changed in version 3.0.

`authSchemaUpgrade` (page 372) supports the upgrade process for existing systems that use *authentication* and *authorization* between:

- 2.4 and 2.6 (See *Upgrade User Authorization Data to 2.6 Format* (page 1115))
- 2.6 and 3.0 (See *Upgrade to SCRAM-SHA-1* (page 1058))

User Management Commands

User Management Commands

Name	Description
`createUser` (page 373)	Creates a new user.
`updateUser` (page 375)	Updates a user's data.
`dropUser` (page 377)	Removes a single user.
`dropAllUsersFromDatabase` (page 378)	Deletes all users associated with a database.
`grantRolesToUser` (page 378)	Grants a role and its privileges to a user.
`revokeRolesFromUser` (page 380)	Removes a role from a user.
`usersInfo` (page 381)	Returns information about the specified users.

createUser

On this page

- Definition (page 373)
- Behavior (page 374)
- Required Access (page 374)
- Example (page 374)

Definition

`createUser`

Creates a new user on the database where you run the command. The `createUser` (page 373) command returns a *duplicate user* error if the user exists. The `createUser` (page 373) command uses the following syntax:

```
{ createUser: "<name>",
  pwd: "<cleartext password>",
  customData: { <any information> },
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  writeConcern: { <write concern> }
}
```

`createUser` (page 373) 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 373) on the `$external` database to create users who have credentials stored externally to MongoDB.

field document customData Optional. 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 Optional. When `true`, the `mongod` (page 770) 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 Optional. The level of `write concern` for the creation operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 373) 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 373) sends password to the MongoDB instance in cleartext. To encrypt the password in transit, use `TLS/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

- To create a new user in a database, you must have `createUser` *action* on that *database resource*.
- To grant roles to a user, you must have the `grantRole` *action* on the role's database.

Built-in roles `userAdmin` and `userAdminAnyDatabase` provide `createUser` and `grantRole` actions on their respective `resources`.

Example The following `createUser` (page 373) 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 }
                                       } )
```

updateUser

On this page

- Definition (page 375)
- Behavior (page 376)
- Required Access (page 376)
- Example (page 376)

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 378) or `revokeRolesFromUser` (page 380) commands.

The `updateUser` (page 375) 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 Optional. The user's password.

field document customData Optional. Any arbitrary information.

field array roles Optional. The roles granted to the user. An update to the `roles` array overrides the previous array's values.

field boolean digestPassword Optional. When `true`, the `mongod` (page 770) 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 Optional. The level of `write concern` for the update operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 375) 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 375) sends the password to the MongoDB instance in cleartext. To encrypt the password in transit, use TLS/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 375) 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"
       }
   ]
}
```

dropUser

On this page

- Definition (page 377)
- Required Access (page 377)
- Example (page 377)

Definition

`dropUser`

Removes the user from the database on which you run the command. The `dropUser` (page 377) command has the following syntax:

```
{
  dropUser: "<user>",
  writeConcern: { <write concern> }
}
```

The `dropUser` (page 377) command document has the following fields:

field string dropUser The name of the user to delete. You must issue the `dropUser` (page 377) command while using the database where the user exists.

field document writeConcern Optional. The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) command.

Before dropping a user who has the `userAdminAnyDatabase` role, ensure you have at least another user with user administration privileges.

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 803) shell removes `reportUser1` from the `products` database:

```
use products
db.runCommand( {
   dropUser: "reportUser1",
   writeConcern: { w: "majority", wtimeout: 5000 }
} )
```

dropAllUsersFromDatabase

On this page

- Definition (page 378)
- Required Access (page 378)
- Example (page 378)

Definition

dropAllUsersFromDatabase

Removes all users from the database on which you run the command.

> **Warning:** The dropAllUsersFromDatabase (page 378) removes all users from the database.

The dropAllUsersFromDatabase (page 378) command has the following syntax:

```
{ dropAllUsersFromDatabase: 1,
  writeConcern: { <write concern> }
}
```

The dropAllUsersFromDatabase (page 378) document has the following fields:

field integer dropAllUsersFromDatabase Specify `1` to drop all the users from the current database.

field document writeConcern Optional. The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the getLastError (page 355) 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 803) 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 }
```

grantRolesToUser

On this page

- Definition (page 378)
- Required Access (page 379)
- Example (page 379)

Definition

grantRolesToUser

Grants additional roles to a user.

The grantRolesToUser (page 378) 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 Optional. The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 378) 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 378) 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

On this page

- Definition (page 380)
- Required Access (page 380)
- Example (page 380)

Definition

revokeRolesFromUser

Removes a one or more roles from a user on the database where the roles exist. The `revokeRolesFromUser` (page 380) 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 Optional. The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 380) 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 380) 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"
    }
]
```

usersInfo

On this page

- Definition (page 381)
- Required Access (page 381)
- Behavior (page 382)
- Examples (page 382)

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 382) for type and syntax.

field boolean showCredentials Optional. Set the field to true to display the user's password hash. By default, this field is false.

field boolean showPrivileges Optional. 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 381) 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 383)	Creates a role and specifies its privileges.
`updateRole` (page 385)	Updates a user-defined role.
`dropRole` (page 387)	Deletes the user-defined role.
`dropAllRolesFromDatabase` (page 387)	Deletes all user-defined roles from a database.
`grantPrivilegesToRole` (page 388)	Assigns privileges to a user-defined role.
`revokePrivilegesFromRole` (page 390)	Removes the specified privileges from a user-defined role.
`grantRolesToRole` (page 392)	Specifies roles from which a user-defined role inherits privileges.
`revokeRolesFromRole` (page 393)	Removes specified inherited roles from a user-defined role.
`rolesInfo` (page 395)	Returns information for the specified role or roles.
`invalidateUserCache` (page 398)	Flushes the in-memory cache of user information, including credentials and roles.

createRole

On this page

- Definition (page 383)
- Behavior (page 384)
- Required Access (page 384)
- Example (page 384)

Definition

`createRole`

Creates a role and specifies its *privileges*. The role applies to the database on which you run the command. The `createRole` (page 383) command returns a *duplicate role* error if the role already exists in the database.

The `createRole` (page 383) 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 383) 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. For the syntax of a privilege, see the privileges array.

You must include the privileges field. Use an empty array to specify *no* privileges.

field array roles An array of roles from which this role inherits privileges.

You must include the roles field. Use an empty array to specify *no* roles to inherit from.

field document writeConcern Optional. The level of write concern to apply to this operation. The writeConcern document uses the same fields as the getLastError (page 355) 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 383) 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, you 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 383) 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", "remo
    { resource: { db: "", collection: "" }, actions: [ "find" ] }
  ],
  roles: [
    { role: "read", db: "admin" }
  ],
  writeConcern: { w: "majority" , wtimeout: 5000 }
})
```

updateRole

On this page

- Definition (page 385)
- Behavior (page 386)
- Required Access (page 386)
- Example (page 386)

Definition

updateRole

Updates a *user-defined role*. The updateRole (page 385) 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 392)
- grantPrivilegesToRole (page 388)
- revokeRolesFromRole (page 393)
- revokePrivilegesFromRole (page 390)

Warning: An update to the privileges or roles array completely replaces the previous array's values.

The updateRole (page 385) 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 385) command has the following fields:

field string updateRole The name of the *user-defined role* role to update.

field array privileges Optional. 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 Optional. 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 Optional. The level of write concern for the update operation. The writeConcern document takes the same fields as the getLastError (page 355) 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 385) 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 385) 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 395) command.

dropRole

On this page

- Definition (page 387)
- Required Access (page 387)
- Example (page 387)

Definition

`dropRole`

Deletes a *user-defined* role from the database on which you run the command.

The `dropRole` (page 387) command uses the following syntax:

```
{
  dropRole: "<role>",
  writeConcern: { <write concern> }
}
```

The `dropRole` (page 387) command has the following fields:

field string dropRole The name of the *user-defined role* to remove from the database.

field document writeConcern Optional. The level of `write concern` for the removal operation. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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" }
   }
)
```

dropAllRolesFromDatabase

On this page

- Definition (page 387)
- Required Access (page 388)
- Example (page 388)

Definition

`dropAllRolesFromDatabase`

Deletes all *user-defined* roles on the database where you run the command.

Warning: The `dropAllRolesFromDatabase` (page 387) removes *all user-defined* roles from the database.

The dropAllRolesFromDatabase (page 387) 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 Optional. The level of write concern for the removal operation. The writeConcern document takes the same fields as the getLastError (page 355) 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 }
```

grantPrivilegesToRole

On this page

- Definition (page 388)
- Behavior (page 389)
- Required Access (page 389)
- Example (page 389)

Definition

grantPrivilegesToRole

Assigns additional *privileges* to a *user-defined* role defined on the database on which the command is run. The grantPrivilegesToRole (page 388) command uses the following syntax:

```
{
  grantPrivilegesToRole: "<role>",
  privileges: [
      {
        resource: { <resource> }, actions: [ "<action>", ... ]
      },
      ...
  ],
  writeConcern: { <write concern> }
}
```

The grantPrivilegesToRole (page 388) 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 Optional. The level of write concern for the modification. The writeConcern document takes the same fields as the getLastError (page 355) 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 388) 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.js" }, 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 queries on *system collections* (page 892), such as the system.js (page 893) collection. To grant access to these system collections, explicitly provision access in the privileges array. See https://docs.mongodb.org/manual/reference/resource-document.

The second privilege explicitly allows the find action on system.js (page 893) collections on all databases.

revokePrivilegesFromRole

On this page

- Definition (page 390)
- Behavior (page 390)
- Required Access (page 391)
- Example (page 391)

Definition

revokePrivilegesFromRole

Removes the specified privileges from the *user-defined* role on the database where the command is run. The revokePrivilegesFromRole (page 390) command has the following syntax:

```
{
  revokePrivilegesFromRole: "<role>",
  privileges:
      [
        { resource: { <resource> }, actions: [ "<action>", ... ] },
        ...
      ],
  writeConcern: <write concern document>
}
```

The revokePrivilegesFromRole (page 390) 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 Optional. The level of write concern for the modification. The writeConcern document takes the same fields as the getLastError (page 355) 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" }
   }
)
```

grantRolesToRole

On this page

- Definition (page 392)
- Behavior (page 393)
- Required Access (page 393)
- Example (page 393)

Definition

`grantRolesToRole`

Grants roles to a *user-defined role*.

The `grantRolesToRole` (page 392) command affects roles on the database where the command runs. `grantRolesToRole` (page 392) has the following syntax:

```
{ grantRolesToRole: "<role>",
  roles: [
              { role: "<role>", db: "<database>" },
              ...
          ],
  writeConcern: { <write concern> }
}
```

The `grantRolesToRole` (page 392) 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 Optional. The level of `write concern` for the modification. The `writeConcern` document takes the same fields as the `getLastError` (page 355) 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 392) 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 392) 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 }
   }
)
```

revokeRolesFromRole

On this page

- Definition (page 393)
- Required Access (page 394)
- Example (page 394)

Definition

`revokeRolesFromRole`

Removes the specified inherited roles from a role. The `revokeRolesFromRole` (page 393) 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 Optional. The level of `write concern` to apply to this operation. The `writeConcern` document uses the same fields as the `getLastError` (page 355) 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 393) 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 393) 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

On this page

- Definition (page 395)
- Behavior (page 395)
- Required Access (page 396)
- Output (page 396)
- Examples (page 396)

Definition

`rolesInfo`

Returns inheritance and privilege information for specified roles, including both *user-defined roles* and *built-in roles*.

The `rolesInfo` (page 395) 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 395).

field boolean showPrivileges Optional. 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 Optional. 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 either 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 396) array as well as the roles from which the roles in the `rolesInfo.roles` (page 396) 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 396) field. To include the field, specify `showPrivileges: true` when running the `rolesInfo` (page 395) 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 396) field. To include the field, specify `showPrivileges: true` when running the `rolesInfo` (page 395) 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
    }
)
```

invalidateUserCache

On this page

- Definition (page 398)
- Required Access (page 398)

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` (page 930) parameter.

`invalidateUserCache` (page 398) 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.

Replication Commands

Replication Commands

Name	Description
`replSetFreeze` (page 399)	Prevents the current member from seeking election as *primary* for a period of time.
`replSetGetStatus` (page 400)	Returns a document that reports on the status of the replica set.
`replSetInitiate` (page 403)	Initializes a new replica set.
`replSetMaintenance` (page 404)	Enables or disables a maintenance mode, which puts a *secondary* node in a `RECOVERING` state.
`replSetReconfig` (page 404)	Applies a new configuration to an existing replica set.
`replSetStepDown` (page 405)	Forces the current *primary* to *step down* and become a *secondary*, forcing an election.
`replSetSyncFrom` (page 407)	Explicitly override the default logic for selecting a member to replicate from.
`resync` (page 408)	Forces a `mongod` (page 770) to re-synchronize from the *master*. For master-slave replication only.
`applyOps` (page 408)	Internal command that applies *oplog* entries to the current data set.
`isMaster` (page 409)	Displays information about this member's role in the replica set, including whether it is the master.
`replSetGetConfig` (page 411)	Returns the replica set's configuration object.

replSetFreeze

`replSetFreeze`

The `replSetFreeze` (page 399) command prevents a replica set member from seeking election for the specified number of seconds. Use this command in conjunction with the `replSetStepDown` (page 405) command to make a different node in the replica set a primary.

The `replSetFreeze` (page 399) 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 770) process also unfreezes a replica set member.

`replSetFreeze` (page 399) is an administrative command, and you must issue it against the *admin database*.

replSetGetStatus

On this page

- Definition (page 400)
- Example (page 400)
- Output (page 401)

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 262) helper in the `mongo` (page 803) shell.

The `mongod` (page 770) must have replication enabled and be a member of a replica set for the for `replSetGetStatus` (page 400) 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("2015-11-19T15:22:32.597Z"),
    "myState" : 1,
    "term": NumberLong(1),
    "heartbeatIntervalMillis" : NumberLong(2000),
    "members" : [
        {
            "_id" : 0,
            "name" : "m1.example.net:27017",
            "health" : 1,
            "state" : 1,
            "stateStr" : "PRIMARY",
            "uptime" : 269,
            "optime" : {
                        "ts" : Timestamp(1447946550, 1),
                        "t" : NumberLong(1)
                     },
            "optimeDate" : ISODate("2015-11-19T15:22:30Z"),
            "infoMessage" : "could not find member to sync from",
            "electionTime" : Timestamp(1447946549, 1),
            "electionDate" : ISODate("2015-11-19T15:22:29Z"),
            "configVersion" : 1,
            "self" : true
        },
        {
            "_id" : 1,
            "name" : "m2.example.net:27017",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 13,
                     "optime" : {
                        "ts" : Timestamp(1447946539, 1),
                        "t" : NumberLong(-1)
                     },
            "optimeDate" : ISODate("2015-11-19T15:22:19Z"),
            "lastHeartbeat" : ISODate("2015-11-19T15:22:31.323Z"),
            "lastHeartbeatRecv" : ISODate("2015-11-19T15:22:32.045Z"),
            "pingMs" : NumberLong(0),
            "configVersion" : 1
        },
        {
            "_id" : 2,
            "name" : "m3.example.net:27017",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 13,
                     "optime" : {
                        "ts" : Timestamp(1447946539, 1),
                        "t" : NumberLong(-1)
                     },
            "optimeDate" : ISODate("2015-11-19T15:22:19Z"),
            "lastHeartbeat" : ISODate("2015-11-19T15:22:31.325Z"),
            "lastHeartbeatRecv" : ISODate("2015-11-19T15:22:31.971Z"),
            "pingMs" : NumberLong(0),
            "configVersion" : 1
        }
    ],
    "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 (page 922) setting. This is the same value as _id in rs.conf() (page 257).

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 to find the operational lag between the current host and the other hosts in the set.

replSetGetStatus.**myState**
The value of myState (page 401) is an integer between 0 and 10 that represents the replica state of the current member.

replSetGetStatus.**term**
New in version 3.2.

The election count for the replica set, as known to this replica set member. The term (page 401) is used by the distributed consensus algorithm to ensure correctness.

If using protocolVersion: 0, instead of protocolVersion: 1, returns -1.

replSetGetStatus.**heartbeatIntervalMillis**
New in version 3.2.

The frequency in milliseconds of the heartbeats.

`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. Its 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 262)). 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 402) 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 262) data.

`replSetGetStatus.members[n].`**`optime`**
Information regarding the last operation from the operation log that this member has applied.

Changed in version 3.2.

If using `protocolVersion: 1`, `optime` returns a document that contains:

- `ts`, the *Timestamp* of the last operation applied to this member of the replica set from the *oplog*.
- `t`, the `term` (page 401) in which the last applied operation was originally generated on the primary.

If using `protocolVersion: 0`, `optime` returns the *Timestamp* of the last operation applied to this member of the replica set from the *oplog*.

`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 402) 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 *Timestamp* from the operation log. See `https://docs.mongodb.org/manual/core/replica-set-elections` for more information about elections.

`replSetGetStatus.members[n].`**`electionDate`**
For the current primary, an *ISODate* formatted date string that reflects the election date. See `https://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 401) and `lastHeartBeatRecv` field to track latency between these members.

This value does not appear for the member that returns the `rs.status()` (page 262) 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 401) and `lastHeartBeat` field to track latency between these members.

`replSetGetStatus.members[n].`**`lastHeartbeatMessage`**

When the last heartbeat included an extra message, the `lastHeartbeatMessage` (page 403) 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 262) data.

`replSetGetStatus.members[n].`**`syncingTo`**

The `syncingTo` field is only present on the output of `rs.status()` (page 262) on *secondary* and recovering members, and holds the hostname of the member from which this instance is syncing.

`replSetGetStatus.members[n].`**`configVersion`**

New in version 3.0.

The `configVersion` value is the `replica set configuration version`.

replSetInitiate

`replSetInitiate`

The `replSetInitiate` (page 403) 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 258) 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:

https://docs.mongodb.org/manual/reference/replica-configuration, https://docs.mongodb.org/manual/administration/replica-sets, and *Replica Set Reconfiguration* (page 261).

replSetMaintenance

On this page

- Definition (page 404)
- Behavior (page 404)

Definition

replSetMaintenance

The replSetMaintenance (page 404) 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 404) 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 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 SECONDARY state.
- See https://docs.mongodb.org/manual/reference/replica-states/ for more information about replica set member states.

See https://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

On this page

- Behaviors (page 405)
- Additional Information (page 405)

replSetReconfig

The replSetReconfig (page 404) 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 404) with the shell's `rs.reconfig()` (page 260) method.

Behaviors Be aware of the following `replSetReconfig` (page 404) 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 404) 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.
- If you are reconfiguring a set that has 2 members where only one member votes, and you are adding a member, you will need to specify the force option.
- 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 404) 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 404) obtains a special mutually exclusive lock to prevent more than one `replSetReconfig` (page 404) operation from occurring at the same time.

Additional Information *replSetGetConfig-output*, `https://docs.mongodb.org/manual/reference/replica-confi` `rs.reconfig()` (page 260), and `rs.conf()` (page 257).

replSetStepDown

On this page

- Description (page 405)
- Behavior (page 406)
- Examples (page 406)

Description

`replSetStepDown`

Forces the *primary* of the replica set to become a *secondary*, triggering an *election for primary*. The command steps down the primary for a specified number of seconds; during this period, the stepdown member is ineligible from becoming primary.

By default, the command only steps down the primary if an `electable` secondary is up-to-date with the primary, waiting up to 10 seconds for a secondary to catch up.

The command is only valid against the primary and will error if run on a non-primary member. `replSetStepDown` (page 405) can only run in the `admin` database and has the following prototype form:

```
db.runCommand( {
    replSetStepDown: <seconds>,
    secondaryCatchUpPeriodSecs: <seconds>,
    force: <true|false>
} )
```

replSetStepDown (page 405) takes the following fields as arguments:

field number replSetStepDown The number of seconds to step down the primary, during which time the stepdown member is ineligible for becoming primary. If you specify a non-numeric value, the command uses 60 seconds.

The stepdown period starts from the time that the mongod (page 770) receives the command. The stepdown period must be greater than the secondaryCatchUpPeriodSecs.

field number secondaryCatchUpPeriodSecs Optional. The number of seconds that the mongod (page 770) will wait for an electable secondary to catch up to the primary.

When specified, secondaryCatchUpPeriodSecs overrides the default wait time of either 10 seconds or if force: true, 0 seconds.

field boolean force Optional. A boolean that determines whether the primary steps down if no electable and up-to-date secondary exists within the wait period.

If true, the primary steps down even if no suitable secondary member exists; this could lead to rollbacks if a secondary with replication lag becomes the new primary.

If false, the primary does not step down if no suitable secondary member exists and the command returns an error.

Defaults to false.

Behavior New in version 3.0.

Before stepping down, replSetStepDown (page 405) will attempt to terminate long running user operations that would block the primary from stepping down, such as an index build, a write operation or a map-reduce job.

To avoid rollbacks, replSetStepDown (page 405), by default, only steps down the primary if an electable secondary is completely caught up with the primary. The command will wait up to the secondaryCatchUpPeriodSecs for a secondary to catch up.

If no electable secondary meets this criterion by the waiting period, the primary does not step down and the command errors. However, you can override this behavior by including the force: true option.

Upon successful stepdown, replSetStepDown (page 405) forces all clients currently connected to the database to disconnect. This helps ensure that the clients maintain an accurate view of the replica set.

Because the disconnect includes the connection used to run the command, you cannot retrieve the return status of the command if the command completes successfully; i.e. you can only retrieve the return status of the command if it errors. When running the command in a script, the script should account for this behavior.

Note: replSetStepDown (page 405) blocks all writes to the primary while it runs.

Examples

Step Down with Default Options The following example, run on the current primary, attempts to step down the member for 120 seconds.

The operation will wait up to the default `10` seconds for a secondary to catch up. If no suitable secondary exists, the primary does not step down and the command errors.

Note: The command blocks all writes to the primary while it runs.

```
use admin
db.runCommand( { replSetStepDown: 120 } )
```

Specify Wait Time for Secondary Catch Up The following example, run on the current primary, attempts to step down the member for `120` seconds, waiting up to `15` seconds for an electable secondary to catch up. If no suitable secondary exists, the primary does not step down and the command errors.

Note: The command blocks all writes to the primary while it runs.

```
use admin
db.runCommand( { replSetStepDown: 120, secondaryCatchUpPeriodSecs: 15 } )
```

Specify Secondary Catch Up with Force Step Down The following example, run on the current primary, attempts to step down the member for `120` seconds, waiting up to `15` seconds for an electable secondary to catch up. Because of the `force:  true` option, the primary steps down even if no suitable secondary exists.

Note: The command blocks all writes to the primary while it runs.

```
use admin
db.runCommand( { replSetStepDown: 120, secondaryCatchUpPeriodSecs: 15, force: true } )
```

See also:

`rs.stepDown()` (page 263)

replSetSyncFrom

On this page

- Description (page 407)

Description

`replSetSyncFrom`

New in version 2.2.

Explicitly configures which host the current `mongod` (page 770) 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 407) command has the following form:

```
{ replSetSyncFrom: "hostname<:port>" }
```

The `replSetSyncFrom` (page 407) 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 407), see `https://docs.mongodb.org/manual/tutorial/configure-replica-set-secondary-sync-target`.

resync

resync

The resync (page 408) command forces an out-of-date slave mongod (page 770) instance to re-synchronize itself. Note that this command is relevant to master-slave replication only. It does not apply to replica sets.

Warning: This command obtains a global write lock and will block other operations until it has completed.

applyOps

On this page

- Definition (page 408)
- Behavior (page 409)

Definition

applyOps

Applies specified *oplog* entries to a mongod (page 770) instance. The applyOps (page 408) 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 408). Providing such access is not recommended, but if your organization requires a user to run applyOps (page 408), create a role that grants anyAction on *resource-anyresource*. Do not assign this role to any other user.

The applyOps (page 408) command has the following prototype form:

```
db.runCommand( { applyOps: [ <operations> ],
                 preCondition: [ { ns: <namespace>, q: <query>, res: <result> } ],
                 bypassDocumentValidation: <boolean> } )
```

The applyOps (page 408) command takes a document with the following fields:

field array applyOps The oplog entries to apply.

field array preCondition Optional. 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 boolean alwaysUpsert Optional. A flag that indicates whether to apply update operations in the oplog as *upserts*. 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.

field boolean bypassDocumentValidation Optional. Enables applyOps to bypass document validation during the operation. This lets you insert or update documents that do not meet the validation requirements.

New in version 3.2.

The preCondition array takes one or more documents with the following fields:

field string ns A *namespace*. If you use this field, applyOps (page 408) applies oplog entries only for the *collection* described by this namespace.

field string q Specifies the *query* that produces the results specified in the res field.

field 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

On this page

- Definition (page 409)
- Output (page 409)

Definition

isMaster

isMaster (page 409) returns a document that describes the role of the mongod (page 770) instance.

If the instance is a member of a replica set, then isMaster (page 409) 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 770) instance that is not a member of a replica set, isMaster (page 409) returns a subset of this information.

MongoDB *drivers* and *clients* use isMaster (page 409) to determine the state of the replica set members and to discover additional members of a *replica set*.

The db.isMaster() (page 191) method in the mongo (page 803) shell provides a wrapper around isMaster (page 409).

The command takes the following form:

```
{ isMaster: 1 }
```

See also:

db.isMaster() (page 191)

Output

All Instances The following isMaster (page 409) 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 792) instance, or a standalone mongod (page 770).

This field will be `false` if the instance is a *secondary* member of a replica set or if the member is an *arbiter* of a replica set.

isMaster.**maxBsonObjectSize**

The maximum permitted size of a *BSON* object in bytes for this mongod (page 770) 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 770) or `mongos` (page 792) instance is capable of using to communicate with clients.

Clients may use `minWireVersion` (page 409) to help negotiate compatibility with MongoDB.

`isMaster.`**`maxWireVersion`**

New in version 2.6.

The latest version of the wire protocol that this `mongod` (page 770) or `mongos` (page 792) instance is capable of using to communicate with clients.

Clients may use `maxWireVersion` (page 410) to help negotiate compatibility with MongoDB.

Sharded Instances `mongos` (page 792) instances add the following field to the `isMaster` (page 409) response document:

`isMaster.`**`msg`**

Contains the value `isdbgrid` when `isMaster` (page 409) returns from a `mongos` (page 792) instance.

Replica Sets `isMaster` (page 409) contains these fields when returned by a member of a replica set:

`isMaster.`**`setName`**

The name of the current :replica set.

`isMaster.`**`setVersion`**

New in version 2.6.

The current replica set config version.

`isMaster.`**`secondary`**

A boolean value that, when `true`, indicates if the `mongod` (page 770) 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 410) 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 `members[n].priority` of `0`.

This field only appears if there is at least one member with a `members[n].priority` of `0`.

Drivers use this array and the `isMaster.hosts` (page 410) 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 410) 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 410) field is only present for members with a `members[n].priority` of `0`.

`isMaster.`**`hidden`**

A boolean value that, when `true`, indicates that the current instance is *hidden*. The `hidden` (page 411) field is only present for hidden members.

`isMaster.`**`tags`**

A *tag set* document containing mappings of arbitrary keys and values. These documents describe replica set members in order to customize `write concern` and `read preference` and thereby allow configurable data center awareness.

This field is only present if there are tags assigned to the member. See `https://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 409).

`isMaster.`**`electionId`**

New in version 3.0.0.

A unique identifier for each election. Included only in the output of `isMaster` (page 409) for the *primary*. Used by clients to determine when elections occur.

replSetGetConfig New in version 3.0.0.

On this page

- Definition (page 411)
- Output Example (page 411)

Definition

`replSetGetConfig`

Returns a document that describes the current configuration of the *replica set*. To invoke the command directly, use the following operation:

```
db.runCommand( { replSetGetConfig: 1 } );
```

In the `mongo` (page 803) shell, you can access the data provided by `replSetGetConfig` (page 411) using the `rs.conf()` (page 257) method, as in the following:

```
rs.conf();
```

Output Example The following document provides a representation of a replica set configuration document. The configuration of your replica set may include only a subset of these settings:

```
{
  _id: <string>,
  version: <int>,
  protocolVersion: <number>,
  members: [
    {
      _id: <int>,
      host: <string>,
      arbiterOnly: <boolean>,
      buildIndexes: <boolean>,
      hidden: <boolean>,
      priority: <number>,
      tags: <document>,
      slaveDelay: <int>,
      votes: <number>
    },
    ...
  ],
  settings: {
    chainingAllowed : <boolean>,
    heartbeatIntervalMillis : <int>,
    heartbeatTimeoutSecs: <int>,
    electionTimeoutMillis : <int>,
    getLastErrorModes : <document>,
    getLastErrorDefaults : <document>
  }
}
```

For description of the configuration settings, see `https://docs.mongodb.org/manual/reference/replica-configura`

See also:

`rs.conf()` (page 257), `rs.reconfig()` (page 260)

See also:

`https://docs.mongodb.org/manual/replication` for more information regarding replication.

Sharding Commands

Sharding Commands

Name	Description
flushRouterConfig (page 413)	Forces an update to the cluster metadata cached by a mongos (page 792).
addShard (page 414)	Adds a *shard* to a *sharded cluster*.
cleanupOrphaned (page 415)	Removes orphaned data with shard key values outside of the ranges of the chunks owned by a shard.
checkShardingIndex (page 418)	Internal command that validates index on shard key.
enableSharding (page 418)	Enables sharding on a specific database.
listShards (page 419)	Returns a list of configured shards.
removeShard (page 419)	Starts the process of removing a shard from a sharded cluster.
getShardMap (page 420)	Internal command that reports on the state of a sharded cluster.
getShardVersion (page 420)	Internal command that returns the *config server* version.
mergeChunks (page 421)	Provides the ability to combine chunks on a single shard.
setShardVersion (page 422)	Internal command to sets the *config server* version.
shardCollection (page 422)	Enables the sharding functionality for a collection, allowing the collection to be sharded.
shardingState (page 423)	Reports whether the mongod (page 770) is a member of a sharded cluster.
unsetSharding (page 424)	Internal command that affects connections between instances in a MongoDB deployment.
split (page 424)	Creates a new *chunk*.
splitChunk (page 426)	Internal command to split chunk. Instead use the methods sh.splitFind() (page 278) and sh.splitAt() (page 277).
splitVector (page 427)	Internal command that determines split points.
medianKey (page 427)	Deprecated internal command. See splitVector (page 427).
moveChunk (page 427)	Internal command that migrates chunks between shards.
movePrimary (page 429)	Reassigns the *primary shard* when removing a shard from a sharded cluster.
isdbgrid (page 430)	Verifies that a process is a mongos (page 792).

flushRouterConfig

flushRouterConfig

flushRouterConfig (page 413) clears the current cluster information cached by a mongos (page 792) 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 792) process.

Warning: Do not modify the config data, except as explicitly documented. A config database cannot typically tolerate manual manipulation.

flushRouterConfig (page 413) is an administrative command that is only available for mongos (page 792) instances.

New in version 1.8.2.

addShard

On this page

- Definition (page 414)
- Considerations (page 414)
- Examples (page 415)

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 414) when connected to a mongos (page 792) 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 770) 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 Optional. 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 Optional. A name for the shard. If this is not specified, MongoDB automatically provides a unique name.

The addShard (page 414) command stores shard configuration information in the *config database*. Always run addShard (page 414) 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 510) 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 https://docs.mongodb.org/manual/core/sharding-balancing for more information.

Changed in version 2.6: Chunk migrations can have an impact on disk space. Starting in MongoDB 2.6, the source shard automatically archives the migrated documents by default. For details, see *moveChunk-directory*.

Hidden Members

Important: You cannot include a `hidden member` in the seed list provided to `addShard` (page 414).

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.

cleanupOrphaned

On this page

- Definition (page 415)
- Behavior (page 416)
- Required Access (page 417)
- Output (page 417)
- Examples (page 417)

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 415) examines both ranges for orphaned documents.

`cleanupOrphaned` (page 415) has the following syntax:

```
db.runCommand( {
   cleanupOrphaned: "<database>.<collection>",
   startingAtKey: <minimumShardKeyValue>,
   secondaryThrottle: <boolean>,
   writeConcern: <document>
} )
```

`cleanupOrphaned` (page 415) 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 Optional. 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 415) continues to examine the next ranges until it finds a range not owned by the shard. See *Determine Range* (page 416) for details.

field boolean secondaryThrottle Optional. 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 415) waits for all deletes to replicate to a majority of replica set members before returning.

field document writeConcern Optional. 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 415) in the `admin` database directly on the `mongod` (page 770) instance that is the primary replica set member of the shard. Do not run `cleanupOrphaned` (page 415) on a `mongos` (page 792) instance.

You do not need to disable the balancer before running `cleanupOrphaned` (page 415).

Performance `cleanupOrphaned` (page 415) scans the documents in the shard to determine whether the documents belong to the shard. As such, running `cleanupOrphaned` (page 415) can impact performance; however, performance will depend on the number of orphaned documents in the range.

To remove all orphaned documents in a shard, you can run the command in a loop (see *Remove All Orphaned Documents from a Shard* (page 418) for an example). If concerned about the performance impact of this operation, you may prefer to include a pause in-between iterations.

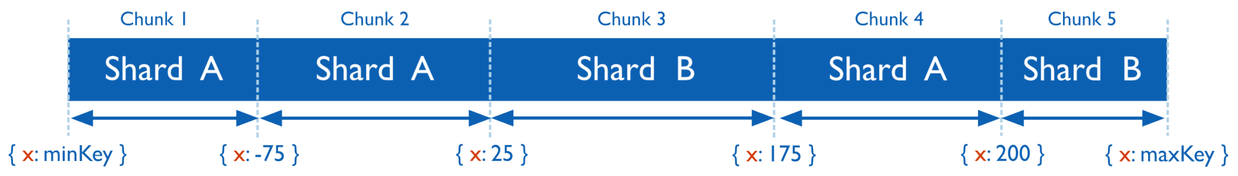
Alternatively, to mitigate the impact of `cleanupOrphaned` (page 415), you may prefer to run the command at off peak hours.

Determine Range The `cleanupOrphaned` (page 415) 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 415) 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 415) moves onto the next range until it finds a range for a chunk not owned by the shard.

The `cleanupOrphaned` (page 415) 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 415) command runs with `startingFromKey: { x: -70 }` or any other value belonging to range for `Chunk 1` or `Chunk 2`, the `cleanupOrphaned` (page 415) command examines the `Chunk 3` range of `{ x: 25 } --> { x: 175 }` to delete orphaned data.

If on `Shard B`, the `cleanupOrphaned` (page 415) command runs with the `startingFromKey: { x: -70 }` or any other value belonging to range for `Chunk 1`, the `cleanupOrphaned` (page 415) 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` (page 910), you must have `clusterAdmin` privileges to run `cleanupOrphaned` (page 415).

Output

Return Document Each `cleanupOrphaned` (page 415) 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 415) 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 415) times out.

A value of `0` could indicate either of two cases:

- `cleanupOrphaned` (page 415) found orphaned documents on the shard but could not delete them.
- `cleanupOrphaned` (page 415) 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 415) 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 415) command prints the number of deleted documents to the `mongod` (page 770) 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 415) 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 415) 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 415) examines documents from a single contiguous range of shard keys. To remove all orphaned documents from the shard, you can run `cleanupOrphaned` (page 415) in a loop, using the returned `stoppedAtKey` as the next `startingFromKey`, as in the following:

```
use admin
var nextKey = { };
var result;

while ( nextKey != null ) {
  result = db.runCommand( { cleanupOrphaned: "test.user", startingFromKey: nextKey } );

  if (result.ok != 1)
     print("Unable to complete at this time: failure or timeout.")

  printjson(result);

  nextKey = result.stoppedAtKey;
}
```

checkShardingIndex

`checkShardingIndex`

`checkShardingIndex` (page 418) is an internal command that supports the sharding functionality.

enableSharding

`enableSharding`

The `enableSharding` (page 418) 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 422) command to begin the process of distributing data among the shards.

listShards

On this page

- Definition (page 419)
- Example (page 419)

Definition

`listShards`

Returns a list of the configured shards. `listShards` (page 419) is only available for `mongos` (page 792) instances. You can only issue `listShards` (page 419) against the `admin` database.

The command takes the following form:

```
{ listShards: 1 }
```

Example The following example returns the list of shards:

```
db.getSiblingDB("admin").runCommand( { listShards: 1 } )
```

removeShard

On this page

- Behavior (page 419)
- Example (page 420)

`removeShard`

Removes a shard from a *sharded cluster*. When you run `removeShard` (page 419), 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 419) while connected to a `mongos` (page 792). Issue the command against the `admin` database or use the `sh._adminCommand()` (page 267) helper.

If you have `authorization` (page 910) 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 429) command and the `https://docs.mongodb.org/manual/tutorial/remove-shards-from-cluster` for more information.

Example From the mongo (page 803) shell, the removeShard (page 419) 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 419), 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 419) 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 194) to list the databases that you must move from the shard. Use the movePrimary (page 429) to move databases.

After removing all chunks and databases from the shard, you can issue removeShard (page 419) again see the following:

```
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "bristol01",
    "ok" : 1
}
```

getShardMap

getShardMap

getShardMap (page 420) is an internal command that supports the sharding functionality.

getShardVersion

getShardVersion

getShardVersion (page 420) is a command that supports sharding functionality and is not part of the stable client facing API.

mergeChunks

On this page

- Definition (page 421)
- Behavior (page 421)
- Return Messages (page 421)

Definition

mergeChunks

For a sharded collection, `mergeChunks` (page 421) combines contiguous *chunk* ranges on a shard into a single chunk. Issue the `mergeChunks` (page 421) command from a `mongos` (page 792) instance.

`mergeChunks` (page 421) 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 421) has the following form:

```
db.runCommand( { mergeChunks : <namespace> ,
                 bounds : [ { x: <minValue>, y: <minValue> },
                            { x: <maxValue>, y: <maxValue> } ] } )
```

The `mergeChunks` (page 421) 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 421) 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 421) returns an error if these conditions are not satisfied.

Return Messages

On success, `mergeChunks` (page 421) returns to following document:

```
{ "ok" : 1 }
```

Another Operation in Progress `mergeChunks` (page 421) returns the following error message if another meta-data operation is in progress on the `chunks` (page 887) collection:

```
errmsg: "The collection's metadata lock is already taken."
```

If another process, such as balancer process, changes metadata while mergeChunks (page 421) is running, you may see this error. You can retry the mergeChunks (page 421) operation without side effects.

Chunks on Different Shards If the input *chunks* are not on the same *shard*, mergeChunks (page 421) 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 421) returns an error similar to the following:

```
{
   "ok" : 0,
   "errmsg" : "could not merge chunks, collection test.users has more than 2 chunks between [{ userna
}
```

setShardVersion

setShardVersion

setShardVersion (page 422) is an internal command that supports sharding functionality.

shardCollection	**On this page** • Definition (page 422) • Considerations (page 423) • Example (page 423) • Additional Information (page 423)

Definition

shardCollection

Enables a collection for sharding and allows MongoDB to begin distributing data among shards. You must run enableSharding (page 418) on a database before running the shardCollection (page 422) command. shardCollection (page 422) has the following form:

```
{ shardCollection: "<database>.<collection>", key: <shardkey> }
```

shardCollection (page 422) 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 422) 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 422) command on the same collection at the same time.

MongoDB provides no method to deactivate sharding for a collection after calling `shardCollection` (page 422). Additionally, after `shardCollection` (page 422), 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 `https://docs.mongodb.org/manual/sharding`, `https://docs.mongodb.org/manual/core/sharding`, and `https://docs.mongodb.org/manual/tutorial/d`

shardingState

`shardingState`

`shardingState` (page 423) is an admin command that reports if `mongod` (page 770) is a member of a *sharded cluster*. `shardingState` (page 423) has the following prototype form:

```
{ shardingState: 1 }
```

For `shardingState` (page 423) to detect that a `mongod` (page 770) is a member of a sharded cluster, the `mongod` (page 770) must satisfy the following conditions:

1.the `mongod` (page 770) is a primary member of a replica set, and

2.the `mongod` (page 770) instance is a member of a sharded cluster.

If `shardingState` (page 423) detects that a `mongod` (page 770) is a member of a sharded cluster, `shardingState` (page 423) 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 423) will return the following document:

```
{ "note" : "from execCommand", "ok" : 0, "errmsg" : "not master" }
```

The response from `shardingState` (page 423) when used with a *config server* is:

```
{ "enabled": false, "ok": 1 }
```

Note: `mongos` (page 792) instances do not provide the `shardingState` (page 423).

Warning: This command obtains a write lock on the affected database and will block other operations until it has completed; however, the operation is typically short lived.

unsetSharding

`unsetSharding`

`unsetSharding` (page 424) is an internal command that supports sharding functionality.

split

On this page

- Definition (page 424)
- Considerations (page 425)
- Command Formats (page 425)
- Examples (page 425)
- Metadata Lock Error (page 426)

Definition

`split`

Splits a *chunk* in a *sharded cluster* into two chunks. The `mongos` (page 792) instance splits and manages chunks automatically, but for exceptional circumstances the `split` (page 424) command does allow administrators to manually create splits. See `https://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 424).

The `split` (page 424) command uses the following form:

```
db.adminCommand( { split: <database>.<collection>,
                   <find|middle|bounds> } )
```

The `split` (page 424) 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 424) requires one of the following options: `find`, `bounds`, or `middle`.

Considerations When used with either the `find` or the `bounds` option, the `split` (page 424) 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 424) command or use the `splitAt` command.

Command Formats To create a chunk split, connect to a `mongos` (page 792) 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 427), `sh.moveChunk()` (page 275), `sh.splitAt()` (page 277), and `sh.splitFind()` (page 278), which wrap the functionality of `split` (page 424).

Examples The following sections provide examples of the `split` (page 424) command.

Split a Chunk in Half

```
db.runCommand( { split : "test.people", find : { _id : 99 } } )
```

The `split` (page 424) command identifies the chunk in the `people` collection of the `test` database, that holds documents that match `{ _id : 99 }`. `split` (page 424) 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 424) 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 424) command identifies the chunk in the `people` collection of the `test` database, that would hold documents matching the query `{ _id : 99 }`. `split` (page 424) 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 279) to see the existing bounds of the shard keys.

Metadata Lock Error If another process in the `mongos` (page 792), such as a balancer process, changes metadata while `split` (page 424) 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 424) command.

splitChunk

On this page

- Definition (page 426)

Definition

`splitChunk`

An internal administrative command. To split chunks, use the `sh.splitFind()` (page 278) and `sh.splitAt()` (page 277) functions in the `mongo` (page 803) 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 427) and `sh.moveChunk()` (page 275).

The `splitChunk` (page 426) command takes a document with the following fields:

field string ns The complete *namespace* of the *chunk* to split.

field document keyPattern The *shard key*.

field document min The lower bound of the shard key for the chunk to split.

field document max The upper bound of the shard key for the chunk to split.

field string from The *shard* that owns the chunk to split.

field document splitKeys The split point for the chunk.

field document shardId The shard.

splitVector

`splitVector`

Is an internal command that supports meta-data operations in sharded clusters.

medianKey

`medianKey`

`medianKey` (page 427) is an internal command.

moveChunk

On this page

- Definition (page 427)
- Considerations (page 428)
- Behavior (page 428)

Definition

`moveChunk`

Internal administrative command. Moves *chunks* between *shards*. Issue the `moveChunk` (page 427) command via a `mongos` (page 792) 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 427) 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 Optional. 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 Optional. 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 Optional. Internal option for testing purposes. The default is `false`. If set to `true`, the delete phase of a `moveChunk` (page 427) 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 424), `sh.moveChunk()` (page 275), `sh.splitAt()` (page 277), and `sh.splitFind()` (page 278).

Considerations Only use the `moveChunk` (page 427) 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 `https://docs.mongodb.org/manual/tutorial/create-chunks-in-sharded-cluster` for more information.

Behavior

Indexes Changed in version 3.0.0: In previous versions, `moveChunk` (page 427) would build indexes as part of the migrations.

moveChunk (page 427) 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 427) returns the following error message if another metadata operation is in progress on the chunks (page 887) collection:

```
errmsg: "The collection's metadata lock is already taken."
```

If another process, such as a balancer process, changes meta data while moveChunk (page 427) is running, you may see this error. You may retry the moveChunk (page 427) operation without side effects.

movePrimary

On this page

- Considerations (page 429)
- Additional Information (page 429)

movePrimary

In a *sharded cluster*, movePrimary (page 429) reassigns the *primary shard* which holds all un-sharded collections in the database. movePrimary (page 429) 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 419) command.

movePrimary (page 429) is an administrative command that is only available for mongos (page 792) instances.

Considerations

Behavior Avoid accessing an un-sharded collection during migration. movePrimary (page 429) does not prevent reading and writing during its operation, and such actions yield undefined behavior.

movePrimary (page 429) may take significant time to complete, and you should plan for this unavailability.

movePrimary (page 429) 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 429) command to move un-sharded collections, you must either restart all mongos (page 792) instances, or use the flushRouterConfig (page 413) command on all mongos (page 792) instances before writing any data to the cluster. This action notifies the mongos (page 792) of the new shard for the database.

If you do not update the mongos (page 792) instances' metadata cache after using movePrimary (page 429), the mongos (page 792) may not write data to the correct shard. To recover, you must manually intervene.

Additional Information See `https://docs.mongodb.org/manual/tutorial/remove-shards-from-cluster` for a complete procedure.

isdbgrid

isdbgrid

This command verifies that a process is a mongos (page 792).

If you issue the isdbgrid (page 430) command when connected to a mongos (page 792), 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 430) command when connected to a mongod (page 770), MongoDB returns an error document. The isdbgrid (page 430) command is not available to mongod (page 770). The error document, however, also includes a line that reads `"isdbgrid" : 1`, just as in the document returned for a mongos (page 792). 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 409) command to determine connection to a mongos (page 792). When connected to a mongos (page 792), the isMaster (page 409) command returns a document that contains the string isdbgrid in the msg field.

See also:

https://docs.mongodb.org/manual/sharding for more information about MongoDB's sharding functionality.

Instance Administration Commands

Administration Commands

Name	Description
`renameCollection` (page 431)	Changes the name of an existing collection.
`copydb` (page 433)	Copies a database from a remote host to the current host.
`dropDatabase` (page 437)	Removes the current database.
`listCollections` (page 438)	Returns a list of collections in the current database.
`drop` (page 439)	Removes the specified collection from the database.
`create` (page 439)	Creates a collection and sets collection parameters.
`clone` (page 442)	Copies a database from a remote host to the current host.
`cloneCollection` (page 443)	Copies a collection from a remote host to the current host.
`cloneCollectionAsCapped` (page 444)	Copies a non-capped collection as a new *capped collection*.
`convertToCapped` (page 444)	Converts a non-capped collection to a capped collection.
`filemd5` (page 445)	Returns the *md5* hash for files stored using *GridFS*.
`createIndexes` (page 446)	Builds one or more indexes for a collection.
`listIndexes` (page 450)	Lists all indexes for a collection.
`dropIndexes` (page 450)	Removes indexes from a collection.
`fsync` (page 451)	Flushes pending writes to the storage layer and locks the database to allow backups.
`clean` (page 453)	Internal namespace administration command.
`connPoolSync` (page 453)	Internal command to flush connection pool.
`connectionStatus` (page 453)	Reports the authentication state for the current connection.
`compact` (page 454)	Defragments a collection and rebuilds the indexes.
`collMod` (page 457)	Add flags to collection to modify the behavior of MongoDB.
`reIndex` (page 460)	Rebuilds all indexes on a collection.
`setParameter` (page 460)	Modifies configuration options.
`getParameter` (page 461)	Retrieves configuration options.
`repairDatabase` (page 462)	Repairs any errors and inconsistencies with the data storage.
`repairCursor` (page 464)	Returns a cursor that iterates over all valid documents in a collection.
`touch` (page 464)	Loads documents and indexes from data storage to memory.
`shutdown` (page 465)	Shuts down the `mongod` (page 770) or `mongos` (page 792) process.
`logRotate` (page 465)	Rotates the MongoDB logs to prevent a single file from taking too much space.

renameCollection

On this page

- Definition (page 431)
- Behavior (page 432)
- Example (page 432)
- Exceptions (page 432)

Definition

`renameCollection`

Changes the name of an existing collection. Specify collection names to `renameCollection` (page 431) in the form of a complete *namespace* (`<database>.<collection>`).

Issue the `renameCollection` (page 431) 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 431) command copies the collection to the new database and drops the source collection.

field boolean dropTarget Optional. If `true`, `mongod` (page 770) will drop the `target` of `renameCollection` (page 431) prior to renaming the collection. The default value is `false`.

Behavior `renameCollection` (page 431) is suitable for production environments; *however*:

- `renameCollection` (page 431) blocks all database activity for the duration of the operation.
- `renameCollection` (page 431) is **not** compatible with sharded collections.
- `renameCollection` (page 431) fails if `target` is the name of an existing collection *and* you do not specify `dropTarget: true`.

Warning: If the `renameCollection` (page 431) operation does not complete, the `target` collection and indexes will not be usable and will require manual intervention to clean up.

Example The following example renames a collection named `orders` in the `test` database to `orders2014` in the `test` database.

Warning: This command obtains a global write lock and will block other operations until it has completed.

```
use admin
db.runCommand( { renameCollection: "test.orders", to: "test.orders2014" } )
```

The `mongo` (page 803) shell provides the `db.collection.renameCollection()` (page 104) helper for the command to rename collections within the *same* database. The following is equivalent to the previous example:

```
use test
db.orders.renameCollection( "orders2014" )
```

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.

copydb

On this page

- Definition (page 433)
- Behavior (page 433)
- Required Access (page 434)
- Generate `nonce` and `key` (page 435)
- Examples (page 436)

Definition

`copydb`

Copies a database either from one `mongod` (page 770) instance to the current `mongod` (page 770) instance or within the current `mongod` (page 770). Run `copydb` (page 433) 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 433) accepts the following options:

field string fromhost Optional. The hostname of the source `mongod` (page 770) instance. Omit to copy databases within the same `mongod` (page 770) instance.

field string fromdb Name of the source database.

field string todb Name of the target database.

field boolean slaveOk Optional. Set `slaveOK` to `true` to allow `copydb` (page 433) to copy data from secondary members as well as the primary. `fromhost` must also be set.

field string username Optional. The name of the user on the `fromhost` MongoDB instance. The user authenticates to the `fromdb`.

For more information, see *Authentication to Source mongod Instance* (page 434).

field string nonce Optional. A single use shared secret generated on the remote server, i.e. `fromhost`, using the `copydbgetnonce` (page 372) command.

For more information, *Generate nonce and key* (page 435).

field string key Optional. A hash of the password used for authentication. For more information, *Generate nonce and key* (page 435).

The `mongo` (page 803) shell provides the `db.copyDatabase()` (page 164) wrapper for the `copydb` (page 433) command.

When authenticating to the `fromhost` instance, `copydb` (page 433) uses *MONGODB-CR* mechanism to authenticate the `fromhost` user. To authenticate users with *SCRAM-SHA-1 mechanism*, use the `db.copyDatabase()` (page 164) method.

Behavior

Destination

- Run copydb (page 433) in the admin database of the destination mongod (page 770) instance, i.e. the instance receiving the copied data.
- copydb (page 433) creates the target database if it does not exist.
- copydb (page 433) requires enough free disk space on the host instance for the copied database. Use the db.stats() (page 199) operation to check the size of the database on the source mongod (page 770) instance.

Authentication to Source mongod Instance

- If copying from another mongod (page 770) instance (fromhost) that enforces access control (page 910), then you must authenticate to the fromhost instance by specifying the username, nonce, and key.

  When authenticating to the fromhost instance, copydb (page 433) uses the fromdb as the *authentication database* for the specified user.
- When authenticating to the fromhost instance, copydb (page 433) uses *MONGODB-CR* mechanism to authenticate the fromhost user. To authenticate users with *SCRAM-SHA-1 mechanism*, use the db.copyDatabase() (page 164) method.

For more information on required access and authentication, see *Required Access* (page 434).

Concurrency

- copydb (page 433) and clone (page 442) 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 433) does not lock the destination server during its operation, so the copy will occasionally yield to allow other operations to complete.

Replica Sets With *read preference* configured to set the slaveOk option to true, you may run copydb (page 433) on a *secondary* member of a *replica set*.

Sharded Clusters

- Do not use copydb (page 433) from a mongos (page 792) instance.
- Do not use copydb (page 433) to copy databases that contain sharded collections.

Required Access Changed in version 2.6.

If the mongod (page 770) instance of the *source* database (fromdb) enforces access control (page 910), you must have proper authorization for the *source* database.

If copying from another mongod (page 770) instance (fromhost) that enforces access control (page 910), then you must authenticate to the fromhost instance by specifying the username, nonce, and key.

When authenticating to the fromhost instance, copydb (page 433) uses the fromdb as the *authentication database* for the specified user.

When authenticating to the fromhost instance, copydb (page 433) uses *MONGODB-CR* mechanism to authenticate the fromhost user. To authenticate users with *SCRAM-SHA-1 mechanism*, use the db.copyDatabase() (page 164) method.

Source Database (fromdb)

Source is non-admin Database Changed in version 3.0.

If the source database is a non-admin database, you must have privileges that specify find, listCollections, and listIndexes actions on the source database, and find action on the system.js collection in the source database.

```
{ resource: { db: "mySourceDB", collection: "" }, actions: [ "find", "listCollections", "listIndexes
{ resource: { db: "mySourceDB", collection: "system.js" }, actions: [ "find" ] },
```

Source is admin Database Changed in version 3.0.

If the source database is the admin database, you must have privileges that specify find, listCollections, and listIndexes actions 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",  "listCollections", "listIndexes" ]
{ 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" ] }
```

Target Database (todb) If the mongod (page 770) instance of the *target* database (todb) enforces access control (page 910), you must have proper authorization for the *target* database.

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" ] }
```

Generate nonce and key If copying from another mongod (page 770) instance that enforces access control, then you must include a username, nonce, and key to authenticate to that instance as a user with proper access.

Tip

The db.copyDatabase() (page 164) handles the generation of the nonce and key.

nonce The nonce is a one-time password that you request from the remote server using the copydbgetnonce (page 372) command, as in the following:

```
use admin
mynonce = db.runCommand( { copydbgetnonce : 1, fromhost: <hostname> } ).nonce
```

If running the `copydbgetnonce` (page 372) command directly on the remote host, you can omit the `fromhost` field in the `copydbgetnonce` (page 372) command.

`key` The `key` is a hash generated as follows:

```
hex_md5(mynonce + username + hex_md5(username + ":mongo:" + password))
```

Examples

Copy from the Same `mongod` Instance 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 770) 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 770) instance:

```
use admin
db.runCommand({
   copydb: 1,
   fromdb: "test",
   todb: "records",
   fromhost: "example.net"
})
```

Copy Databases from a `mongod` Instances that Enforce Authentication If copying from another `mongod` (page 770) instance (`fromhost`) that enforces `access control` (page 910), then you must authenticate to the `fromhost` instance by specifying the `username`, `nonce`, and `key`.

When authenticating to the `fromhost` instance, `copydb` (page 433) uses the `fromdb` as the *authentication database* for the specified user.

Note: When authenticating to the `fromhost` instance, `copydb` (page 433) uses *MONGODB-CR* mechanism to authenticate the `fromhost` user. To authenticate users with *SCRAM-SHA-1 mechanism*, use the `db.copyDatabase()` (page 164) method.

The following command copies the `test` database from a `mongod` (page 770) instance that runs on the remote host `example.net` and enforces access control:

```
use admin
db.runCommand({
   copydb: 1,
   fromdb: "test",
   todb: "records",
   fromhost: "example.net",
   username: "reportingAdmin",
   nonce: "<nonce>",
   key: "<passwordhash>"
})
```

See also:

- `db.copyDatabase()` (page 164)
- `clone` (page 442) and `db.cloneDatabase()` (page 163)
- `https://docs.mongodb.org/manual/core/backups`

dropDatabase

On this page

- Definition (page 437)
- Behavior (page 437)
- Example (page 437)

Definition

`dropDatabase`

The `dropDatabase` (page 437) command drops the current database, deleting the associated data files.

The command has the following form:

```
{ dropDatabase: 1 }
```

The `mongo` (page 803) shell also provides the helper method `db.dropDatabase()` (page 177).

Behavior

Warning: This command obtains a global write lock and will block other operations until it has completed.

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 378) command in the database you are deleting.

Example

The following example in the `mongo` (page 803) shell uses the `use <database>` operation to switch the current database to the `temp` database and then uses the `dropDatabase` (page 437) command to drop the `temp` database:

```
use temp
db.runCommand( { dropDatabase: 1 } )
```

See also:

`dropAllUsersFromDatabase` (page 378)

listCollections

On this page

- Definition (page 438)
- Behavior (page 438)
- Output (page 439)
- Example (page 439)

New in version 3.0.0.

Definition

`listCollections`

Retrieve information, i.e. the name and options, about the collections in a database. Specifically, the command returns a document that contains information with which to create a cursor to the collection information. The `mongo` (page 803) shell provides the `db.getCollectionInfos()` (page 182) and the `db.getCollectionNames()` (page 185).

The command has the following form:

```
{ listCollections: 1, filter: <document> }
```

The `listCollections` (page 438) command can take the following optional field:

field document filter Optional. A query expression to filter the list of collections.

You can specify a query expression on the collection `name` and the collection options. For the available options, see *Behavior* (page 438).

Behavior

Use a filter to limit the results of `listCollections` (page 438). You can specify a `filter` on the collection `name` and the following collection options:

- `"options.capped"`
- `"options.autoIndexId"`
- `"options.size"`
- `"options.max"`
- `"options.flags"`
- `"options.storageEngine"`

The options correspond directly to the options available in `db.createCollection()` (page 167), with the exception of the `"options.flags"`. The `"options.flags"` corresponds to the `usePowerOf2Sizes` and the `noPadding` options in the `db.createCollection()` (page 167) method:

- `0` corresponds to `usePowerOf2Sizes` (page 458) flag set to `false` and `noPadding` (page 458) flag set to `false`.
- `1` corresponds to `usePowerOf2Sizes` (page 458) flag set to `true` and `noPadding` (page 458) flag set to `false`.
- `2` corresponds to `usePowerOf2Sizes` (page 458) flag set to `false` and `noPadding` (page 458) flag set to `true`.
- `3` corresponds to `usePowerOf2Sizes` (page 458) flag set to `true` and `noPadding` (page 458) flag set to `true`.

Note: MongoDB 3.0 ignores the `usePowerOf2Sizes` (page 458) flag. See `collMod` (page 457) and `db.createCollection()` (page 167) for more information.

For the descriptions on the options, see `db.createCollection()` (page 167).

Output

`listCollections.`**`cursor`**
A document that contains information with which to create a cursor to documents that contain collection names and options. The cursor information includes the cursor id, the full namespace for the command, as well as the first batch of results.

`listCollections.`**`ok`**
The return value for the command. A value of `1` indicates success.

Example

List All Collections The following example uses the `db.getCollectionInfos()` (page 182) helper to return information for all collections in the `records` database:

```
use records
db.getCollectionInfos();
```

See also:

`db.getCollectionInfos()` (page 182)

drop

`drop`
The `drop` (page 439) command removes an entire collection from a database. The command has following syntax:

```
{ drop: <collection_name> }
```

The `mongo` (page 803) shell provides the equivalent helper method `db.collection.drop()` (page 45).

This command also removes any indexes associated with the dropped collection.

Warning: This command obtains a write lock on the affected database and will block other operations until it has completed.

create

On this page

- Definition (page 439)
- Considerations (page 442)
- Examples (page 442)

Definition

create

Explicitly creates a collection. create (page 439) has the following form:

Changed in version 3.2.

```
{
  create: <collection_name>,
  capped: <true|false>,
  autoIndexId: <true|false>,
  size: <max_size>,
  max: <max_documents>,
  flags: <0|1|2|3>,
  storageEngine: <document>,
  validator: <document>,
  validationLevel: <string>,
  validationAction: <string>,
  indexOptionDefaults: <document>
}
```

create (page 439) has the following fields:

field string create The name of the new collection.

field boolean capped Optional. 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 Optional. Specify `false` to disable the automatic creation of an index on the `_id` field.

Important: For replica sets, all collections must have `autoIndexId` set to `true`.

field integer size Optional. Specify a maximum size in bytes for a capped collection. Once a capped collection reaches its maximum size, MongoDB removes the older documents to make space for the new documents. The `size` field is required for capped collections and ignored for other collections.

field integer max Optional. The maximum number of documents allowed in the capped collection. The `size` limit takes precedence over this limit. If a capped collection reaches the `size` limit before it reaches the maximum number of documents, MongoDB removes old documents. If you prefer to use the `max` limit, ensure that the `size` limit, which is required for a capped collection, is sufficient to contain the maximum number of documents.

field integer flags Optional. Available for the MMAPv1 storage engine only to set the usePowerOf2Sizes (page 458) and the noPadding (page 458) flags. To set, specify one of the following values:

- `0` corresponds to usePowerOf2Sizes (page 458) flag set to `false` and noPadding (page 458) flag set to `false`.
- `1` corresponds to usePowerOf2Sizes (page 458) flag set to `true` and noPadding (page 458) flag set to `false`.
- `2` corresponds to usePowerOf2Sizes (page 458) flag set to `false` and noPadding (page 458) flag set to `true`.
- `3` corresponds to usePowerOf2Sizes (page 458) flag set to `true` and noPadding (page 458) flag set to `true`.

Note: MongoDB 3.0 ignores the usePowerOf2Sizes (page 458) flag. See collMod (page 457) and db.createCollection() (page 167) for more information.

Defaults to `1`.

New in version 2.6.

Changed in version 3.0.0: Add support for setting the new `noPadding` (page 458) flag.

Warning: Do not set `noPadding` if the workload includes removes or any updates that may cause documents to grow. For more information, see *exact-fit-allocation*.

field document storageEngine Optional. Available for the WiredTiger storage engine only.

New in version 3.0.

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:

```
{ <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.

field document validator Optional. Allows users to specify `validation rules or expressions` for the collection. For more information, see `https://docs.mongodb.org/manual/core/document-validation`.

New in version 3.2.

The `validator` option takes a document that specifies the validation rules or expressions. You can specify the expressions using the same operators as the *query operators* (page 527) with the exception of `$geoNear`, `$near` (page 565), `$nearSphere` (page 567), `$text` (page 549), and `$where` (page 558).

Note:

- Validation occurs during updates and inserts. Existing documents do not undergo validation checks until modification.
- You cannot specify a validator for collections in the `admin`, `local`, and `config` databases.
- You cannot specify a validator for `system.*` collections.

field string validationLevel Optional. Determines how strictly MongoDB applies the validation rules to existing documents during an update.

New in version 3.2.

`validationLevel`	Description
`"off"`	No validation for inserts or updates.
`"strict"`	**Default** Apply validation rules to all inserts and all updates.
`"moderate"`	Apply validation rules to inserts and to updates on existing *valid* documents. Do not apply rules to updates on existing *invalid* documents.

field string validationAction Optional. Determines whether to `error` on invalid documents or just `warn` about the violations but allow invalid documents to be inserted.

New in version 3.2.

Important: Validation of documents only applies to those documents as determined by the `validationLevel`.

validationAction	Description
"error"	**Default** Documents must pass validation before the write occurs. Otherwise, the write operation fails.
"warn"	Documents do not have to pass validation. If the document fails validation, the write operation logs the validation failure.

field document indexOptionDefaults Optional. Allows users to specify a default configuration for indexes when creating a collection.

The `indexOptionDefaults` option accepts a `storageEngine` document, which 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.

New in version 3.2.

The `db.createCollection()` (page 167) method wraps the `create` (page 439) command.

Considerations The `create` (page 439) 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.

Examples

Create a Capped Collection 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 } )
```

Specify Storage Engine Options New in version 3.0.

You can specify collection-specific storage engine configuration options when you create a collection with `db.createCollection()` (page 167). Consider the following operation:

```
db.runCommand( {
    create: "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[11] for specific `wiredTiger` options.

clone

clone

The `clone` (page 442) command clones a database from a remote MongoDB instance to the current host. `clone` (page 442) copies the database on the remote instance with the same name as the current database. The command takes the following form:

[11] http://source.wiredtiger.com/2.4.1/struct_w_t___s_e_s_s_i_o_n.html#a358ca4141d59c345f401c58501276bbb

```
{ 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 442) can copy from a non-*primary* member of a *replica set*.
- `clone` (page 442) 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 442) on the **destination server**.
- The destination database will be locked periodically during the `clone` (page 442) operation. In other words, `clone` (page 442) will occasionally yield to allow other operations on the database to complete.

See `copydb` (page 433) for similar functionality with greater flexibility.

cloneCollection

On this page

- Definition (page 443)
- Behavior (page 443)
- Example (page 443)

Definition

`cloneCollection`

Copies a collection from a remote `mongod` (page 770) instance to the current `mongod` (page 770) instance. `cloneCollection` (page 443) creates a collection in a database with the same name as the remote collection's database. `cloneCollection` (page 443) takes the following form:

```
{ cloneCollection: "<namespace>",
  from: "<hostname>",
  query: { <query> }
}
```

`cloneCollection` (page 443) 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 The address of the server to clone from.

field document query Optional. A query that filters the documents in the source collection that `cloneCollection` (page 443) will copy to the current database.

Behavior

`mongos` (page 792) does not support `cloneCollection` (page 443).

Changed in version 3.0: If the given *namespace* already exists in the destination `mongod` (page 770) instance, `cloneCollection` (page 443) will return an error.

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 }`.

cloneCollectionAsCapped

On this page

- Definition (page 444)
- Behavior (page 444)

Definition

cloneCollectionAsCapped

The `cloneCollectionAsCapped` (page 444) 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 444) command.

Behavior If the `capped size` is less than the size of the source collection, then not all documents in the source collection will exist in the destination capped collection.

convertToCapped

On this page

- Example (page 445)

convertToCapped

The `convertToCapped` (page 444) 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 444) 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 444) 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 444) command creates the capped collection and imports the data.

– MongoDB drops the original collection.

– renameCollection (page 431) renames the new capped collection to the name of the original collection.

Note: MongoDB does not support the convertToCapped (page 444) command in a sharded cluster.

Warning: The convertToCapped (page 444) 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.

Example

Convert a Collection The following example uses a db.collection.save() (page 105) operation to create an events collection, and db.collection.stats() (page 107) to obtain information about the collection:

```
db.events.save( { click: 'button-1', time: new Date() } )
db.events.stats()
```

MongoDB will return the following:

```
{
        "ns" : "test.events",
        ...
        "capped" : false,
        ...
}
```

To convert the events collection into a capped collection and view the updated collection information, run the following commands:

```
db.runCommand( { convertToCapped: 'events', size: 8192 } )
db.events.stats()
```

MongoDB will return the following:

```
{
     "ns" : "test.events",
     ...
     "capped" : true,
     "max" : NumberLong("9223372036854775807"),
     "maxSize" : 8192,
     ...
}
```

The convertToCapped (page 444) 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 439)

filemd5

filemd5

The filemd5 (page 445) command returns the *md5* hash 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.

createIndexes

On this page

- Definition (page 446)
- Considerations (page 448)
- Behavior (page 449)
- Example (page 449)
- Output (page 449)

New in version 2.6.

Definition

createIndexes

Builds one or more indexes on a collection.

Changed in version 3.2: Starting in MongoDB 3.2, MongoDB disallows the creation of *version 0* (page 1001) indexes. To upgrade existing version 0 indexes, see *Version 0 Indexes* (page 1001).

The createIndexes (page 446) 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 446) 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:

Changed in version 3.0: The `dropDups` option is no longer available.

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 Optional. The *namespace* (i.e. `<database>.<collection>`) of the collection for which to create the index. If you omit `ns`, MongoDB generates the namespace.

field boolean background Optional. 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`.

field boolean unique Optional. 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.

field document partialFilterExpression Optional. If specified, the index only references documents that match the filter expression. See `https://docs.mongodb.org/manual/core/index-partial` for more information.

A filter expression can include:

- equality expressions (i.e. `field: value` or using the `$eq` (page 527) operator),
- `$exists: true` (page 538) expression,
- `$gt` (page 529), `$gte` (page 530), `$lt` (page 530), `$lte` (page 531) expressions,
- `$type` (page 540) expressions,
- `$and` (page 535) operator at the top-level only

You can specify a `partialFilterExpression` option for all MongoDB `index types`.

New in version 3.2.

field boolean sparse Optional. 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 `https://docs.mongodb.org/manual/core/index-sparse` for more information.

Changed in version 3.2: Starting in MongoDB 3.2, MongoDB provides the option to create *partial indexes*. Partial indexes offer a superset of the functionality of sparse indexes. If you are using MongoDB 3.2 or later, *partial indexes* should be preferred over sparse indexes.

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.

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

field document storageEngine Optional. 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.

New in version 3.0.

field document weights Optional. 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 `https://docs.mongodb.org/manual/tutorial/control-results-of-text-search` to adjust the scores. The default value is `1`.

field string default_language Optional. 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 `https://docs.mongodb.org/manual/tutorial/specify-language-for-text-index` for more information and examples. The default value is `english`.

field string language_override Optional. 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.

field integer textIndexVersion Optional. 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.

field integer 2dsphereIndexVersion Optional. 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.

field integer bits Optional. 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`.

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

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

field 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 941) limit.

Behavior

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

Multiple Index Builds Changed in version 3.0.0.

If you specify multiple indexes to the `createIndexes` (page 446) command, the operation only scans the collection once, and if at least one index is to be built in the foreground, the operation will build all the specified indexes in the foreground.

Index Options If you create an index with one set of options and then issue `createIndexes` (page 446) 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 46) before running the new `createIndexes` (page 446) 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 446) 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.

listIndexes

On this page

- Definition (page 450)
- Output (page 450)

New in version 3.0.0.

Definition

`listIndexes`
Returns information about the indexes on the specified collection. Specifically, the command returns a document that contains information with which to create a cursor to the index information. Index information includes the keys and options used to create the index. The `mongo` (page 803) shell provides the `db.collection.getIndexes()` (page 73) helper.

The command has the following form:

```
{ "listIndexes": "<collection-name>" }
```

field string listIndexes The name of the collection.

Output

`listIndexes.`**`cursor`**
A document that contains information with which to create a cursor to index information. The cursor information includes the cursor id, the full namespace for the command, as well as the first batch of results. Index information includes the keys and options used to create the index. For information on the keys and index options, see `db.collection.createIndex()` (page 36).

`listIndexes.`**`ok`**
The return value for the command. A value of `1` indicates success.

dropIndexes

`dropIndexes`
The `dropIndexes` (page 450) command drops one or all indexes from the current collection. To drop all indexes, issue the command like so:

```
{ dropIndexes: "collection", index: "*" }
```

To drop a single, issue the command by specifying the name of the index you want to drop. For example, to drop the index named `age_1`, use the following command:

```
{ dropIndexes: "collection", index: "age_1" }
```

The shell provides a useful command helper. Here's the equivalent command:

```
db.collection.dropIndex("age_1");
```

Warning: This command obtains a write lock on the affected database and will block other operations until it has completed.

fsync

On this page

- Definition (page 451)
- Considerations (page 451)
- Examples (page 452)

Definition

`fsync`

Forces the `mongod` (page 770) process to flush all pending writes from the storage layer to disk. Optionally, you can use `fsync` (page 451) to lock the `mongod` (page 770) 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` (page 916) interval, which is 60 seconds by default. Run `fsync` (page 451) when you want to flush writes to disk ahead of that interval.

The `fsync` (page 451) command has the following syntax:

```
{ fsync: 1, async: <Boolean>, lock: <Boolean> }
```

The `fsync` (page 451) command has the following fields:

field integer fsync Enter "1" to apply `fsync` (page 451).

field boolean async Optional. Runs `fsync` (page 451) asynchronously. By default, the `fsync` (page 451) operation is synchronous.

field boolean lock Optional. Locks `mongod` (page 770) instance and blocks all write operations.

Considerations

Wired Tiger Compatibility Changed in version 3.2: Starting in MongoDB 3.2, `fsync` (page 451) command with the `lock` option can ensure that the data files do not change for MongoDB instances using either the MMAPv1 or the WiredTiger storage engine, thus providing consistency for the purposes of creating backups.

In previous MongoDB version, `fsync` (page 451) command with the `lock` option *cannot* guarantee a consistent set of files for low-level backups (e.g. via file copy `cp`, `scp`, `tar`) for WiredTiger.

Impact on Larger Deployments An `fsync` (page 451) lock is only possible on *individual* `mongod` (page 770) instances of a sharded cluster, not on the entire cluster. To backup an entire sharded cluster, please see `https://docs.mongodb.org/manual/administration/backup-sharded-clusters` for more information.

Alternatives with Journaling If your `mongod` (page 770) has *journaling* enabled, consider using *another method* to create a back up of the data set.

Impact on Read Operations After `fsync` (page 451), with lock, runs on a `mongod` (page 770), all write operations will block until a subsequent unlock. Read operations *may* also block. As a result, `fsync` (page 451), with lock, is not a reliable mechanism for making a `mongod` (page 770) 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 770) that enforces authorization checks.

Warning: When calling `fsync` (page 451) with lock, ensure that the connection remains open to allow a subsequent call to `db.fsyncUnlock()` (page 181).
Closing the connection may make it difficult to release the lock.

Examples

Run Asynchronously The `fsync` (page 451) operation is synchronous by default To run `fsync` (page 451) asynchronously, use the `async` field set to `true`:

```
{ fsync: 1, async: true }
```

The operation returns immediately. To view the status of the `fsync` (page 451) operation, check the output of `db.currentOp()` (page 171).

Lock `mongod` Instance

Note: Changed in version 3.2: Starting in MongoDB 3.2, `fsync` (page 451) command with the `lock` option can ensure that the data files do not change for MongoDB instances using either the MMAPv1 or the WiredTiger storage engine, thus providing consistency for the purposes of creating backups.

In previous MongoDB version, `fsync` (page 451) command with the `lock` option *cannot* guarantee a consistent set of files for low-level backups (e.g. via file copy `cp`, `scp`, `tar`) for WiredTiger.

The primary use of `fsync` (page 451) is to lock the `mongod` (page 770) instance in order to back up the files within `mongod` (page 770)'s `dbPath` (page 915). The operation flushes all data to the storage layer and blocks all write operations until you unlock the `mongod` (page 770) 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 770) instance that has a `fsync` (page 451) lock. However, after the first write operation all subsequent read operations wait until you unlock the `mongod` (page 770) instance.

Unlock mongod Instance To unlock the mongod (page 770), use db.fsyncUnlock() (page 181):

```
db.fsyncUnlock();
```

Check Lock Status To check the state of the fsync lock, use db.currentOp() (page 171). Use the following JavaScript function in the shell to test if mongod (page 770) 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 803) shell session call it, with the following syntax:

```
serverIsLocked()
```

This function will return `true` if the mongod (page 770) instance is currently locked and `false` if the mongod (page 770) is not locked.

clean

clean

clean (page 453) is an internal command.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

connPoolSync

connPoolSync

connPoolSync (page 453) is an internal command.

connectionStatus

On this page

- Definition (page 453)
- Output (page 453)

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 connection, 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 `https://docs.mongodb.org/manual/reference/built-in-roles` and `https://docs.mongodb.org/manual/reference/privilege-actions` for more information.

`connectionStatus.authinfo.authenticatedUserRoles[n].`**`db`**
The database to which `role` (page 454) applies.

`connectionStatus.`**`ok`**
The return value for the command. A value of `1` indicates success.

compact

On this page

- Definition (page 454)
- Behavior (page 455)

Definition

`compact`
Rewrites and defragments all data and indexes in a collection. On *WiredTiger* databases, this command will release unneeded disk space to the operating system.

`compact` (page 454) has the following form:

```
{ compact: <collection name> }
```

`compact` (page 454) takes the following fields:

field string compact The name of the collection.

field boolean force Optional. If `true`, `compact` (page 454) can run on the *primary* in a *replica set*. If `false`, `compact` (page 454) returns an error when run on a primary, because the command blocks all other operations.

`compact` (page 454) blocks operations only for the database it is compacting.

field number paddingFactor Optional. *Applicable for the MMAPv1 storage engine only.* Specifies the padding to use (as a factor of the document size) during the `compact` (page 454) operation.

The `paddingFactor` does not affect the padding of subsequent record allocations after `compact` (page 454) completes. For more information, see *paddingFactor* (page 455).

field integer paddingBytes Optional. *Applicable for the MMAPv1 storage engine only.* Specifies the padding to use (in absolute number of bytes) during the `compact` (page 454) operation.

`paddingBytes` does not affect the padding of subsequent record allocations after `compact` (page 454) completes. For more information, see *paddingBytes* (page 455).

field boolean preservePadding Optional. *Applicable for the MMAPv1 storage engine only.* Specifies that the `compact` (page 454) process should leave document *padding* intact.

This option cannot be used with `paddingFactor` or `paddingBytes`.

New in version 2.6.

Warning: Always have an up-to-date backup before performing server maintenance such as the `compact` (page 454) operation.

paddingFactor

Note: Applicable for the MMAPv1 storage engine only; specifying `paddingFactor` has no effect when used with the WiredTiger storage engine.

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 454) 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 454) modifies existing documents, but does not set the padding factor for future documents.

paddingBytes

Note: Applicable for the MMAPv1 storage engine only; specifying `paddingBytes` has no effect when used with the WiredTiger storage engine.

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 1 kB, 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).

The following command uses the `paddingBytes` option to set the padding size to 100 bytes on the collection named by `<collection>`:

```
db.runCommand ( { compact: '<collection>', paddingBytes: 100 } )
```

Behavior

Blocking compact (page 454) only blocks operations for the database it is currently operating on. Only use compact (page 454) during scheduled maintenance periods.

You may view the intermediate progress either by viewing the mongod (page 770) log file or by running the db.currentOp() (page 171) in another shell instance.

Operation Termination If you terminate the operation with the db.killOp() (page 192) method or restart the server before the compact (page 454) operation has finished, be aware of the following:

- If you have journaling enabled, the data remains valid and usable, regardless of the state of the compact (page 454) operation. You may have to manually rebuild the indexes.
- If you do not have journaling enabled and the mongod (page 770) or compact (page 454) 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 454) has different impacts on available disk space depending on which storage engine is in use.

To see how the storage space changes for the collection, run the collStats (page 473) command before and after compaction.

WiredTiger On *WiredTiger*, compact (page 454) will rewrite the collection and indexes to minimize disk space by releasing unused disk space to the system. This is useful if you have removed a large amount of data from the collection, and do not plan to replace it.

MMAPv1 On *MMAPv1*, compact (page 454) defragments the collection's data files and recreates its indexes. Unused disk space is *not* released to the system, but instead retained for future data. If you wish to reclaim disk space from a MMAPv1 database, you should perform an *initial sync*.

compact (page 454) requires up to 2 gigabytes of additional disk space to run on MMAPv1 databases.

Size and Number of Data Files compact (page 454) 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 454) commands do not replicate to secondaries in a *replica set*.

- Compact each member separately.
- Ideally run compact (page 454) on a secondary. See option force:true above for information regarding compacting the primary.
- On secondaries, the 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 SECONDARY state.
- See https://docs.mongodb.org/manual/reference/replica-states/ for more information about replica set member states.

See https://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 454) only applies to mongod (page 770) instances. In a sharded environment, run compact (page 454) on each shard separately as a maintenance operation.

You cannot issue compact (page 454) against a mongos (page 792) instance.

Capped Collections It is not possible or necessary to compact *capped collections* because they lack padding and their documents cannot grow. As a result, they cannot become fragmented.

Index Building New in version 2.6.

mongod (page 770) rebuilds all indexes in parallel following the compact (page 454) operation.

collMod

On this page

- Definition (page 457)
- Flags (page 457)
- Examples (page 459)

Definition

`collMod`

New in version 2.2.

collMod (page 457) makes it possible to add flags to a collection to modify the behavior of MongoDB. Flags include usePowerOf2Sizes (page 458) and index (page 457). 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 476) field in the db.collection.stats() (page 107) output to check enabled collection flags.

Flags

TTL Collection Expiration Time

`index`

The index (page 457) 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 457) returns a document with fields `expireAfterSeconds_old` and `expireAfterSeconds_new` set to their respective values.

On failure, collMod (page 457) 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

Disable All Record Padding

noPadding

New in version 3.0.

noPadding (page 458) flag is available for the MMAPv1 storage engine only.

noPadding (page 458) disables record padding for the collection. Without padding, the record allocation size corresponds to the document size, and any updates that results in document growth will require a new allocation.

Warning: Only set noPadding (page 458) to true for collections whose workloads have *no* update operations that cause documents to grow, such as for collections with workloads that are insert-only. For more information, see *exact-fit-allocation*.

Powers of Two Record Allocation

usePowerOf2Sizes

Deprecated since version 3.0.

usePowerOf2Sizes (page 458) flag is available for the MMAPv1 storage engine only.

usePowerOf2Sizes (page 458) has no effect on the allocation strategy. MongoDB 3.0 uses the *power of 2 allocation* as the default record allocation strategy for MMAPv1.

To disable the *power of 2 allocation* for a collection, use the collMod (page 457) command with the noPadding (page 458) flag or the db.createCollection() (page 167) method with the noPadding option. For more information, see *MMAPv1 Record Allocation Behavior Changes* (page 1042).

Document Validation

validator

New in version 3.2.

validator (page 458) allows users to specify validation rules or expressions for a collection. For more information, see https://docs.mongodb.org/manual/core/document-validation.

The validator option takes a document that specifies the validation rules or expressions. You can specify the expressions using the same operators as the *query operators* (page 527) with the exception of : $geoNear, $near (page 565), $nearSphere (page 567), $text (page 549), and $where (page 558).

Note:

- Validation occurs during updates and inserts. Existing documents do not undergo validation checks until modification.
- You cannot specify a validator for collections in the admin, local, and config databases.
- You cannot specify a validator for system.* collections.

validationLevel

New in version 3.2.

The validationLevel (page 458) determines how strictly MongoDB applies the validation rules to existing documents during an update.

validationLevel	Description
"off"	No validation for inserts or updates.
"strict"	**Default** Apply validation rules to all inserts and all updates.
"moderate"	Apply validation rules to inserts and to updates on existing *valid* documents. Do not apply rules to updates on existing *invalid* documents.

validationAction

New in version 3.2.

The validationAction (page 459) option determines whether to error on invalid documents or just warn about the violations but allow invalid documents.

Important: Validation of documents only applies to those documents as determined by the validationLevel.

validationAction	Description
"error"	**Default** Documents must pass validation before the write occurs. Otherwise, the write operation fails.
"warn"	Documents do not have to pass validation. If the document fails validation, the write operation logs the validation failure.

To view the validation specifications for a collection, use the db.getCollectionInfos() (page 182) method.

Examples

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 }
```

Add Document Validation to an Existing Collection The following example adds a validator to a collection named contacts. The validator specifies that inserted or updated documents should meet at least one of the following conditions:

- the phone field is a string
- the email field matches the regular expression
- the status field is either Unknown or Incomplete.

The moderate validationLevel (page 458) specifies that only updates to existing *valid* documents will be checked against the validator, and the warn validationAction (page 459) means that the write operation will log a validation failure for any document that does not pass validation.

```
db.runCommand( { collMod: "contacts",
                 validator: { $or:
                    [
                       { phone: { $type: "string" } },
                       { email: { $regex: /@mongodb\.com$/ } },
                       { status: { $in: [ "Unknown", "Incomplete" ] } }
                    ]
                 },
                 validationLevel: "moderate",
                 validationAction: "warn"

} )
```

With the validator in place, the following insert operation fails validation:

```
db.contacts.insert( { name: "Amanda", status: "Updated" } )
```

The write operation logs the failure and succeeds:

```
2015-10-15T11:20:44.260-0400 W STORAGE  [conn3] Document would fail validation collection: example.co
```

For more information, see `https://docs.mongodb.org/manual/core/document-validation`.

reIndex

`reIndex`

The `reIndex` (page 460) 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 460) 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 460) using the following form:

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

Note: For replica sets, `reIndex` (page 460) will not propagate from the *primary* to *secondaries*. `reIndex` (page 460) will only affect a single `mongod` (page 770) instance.

Important: `reIndex` (page 460) will rebuild indexes in the *background if the index was originally specified with this option*. However, `reIndex` (page 460) will rebuild the `_id` index in the foreground, which takes the database's write lock.

See

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

setParameter

`setParameter`

`setParameter` (page 460) is an administrative command for modifying options normally set on the command line. You must issue the `setParameter` (page 460) command against the *admin database* in the form:

```
{ setParameter: 1, <option>: <value> }
```

Replace the `<option>` with one of the supported `setParameter` (page 460) options:

Indexing

- `failIndexKeyTooLong` (page 932)
- `notablescan` (page 932)

Logging

- `logComponentVerbosity` (page 934)
- `logLevel` (page 933)
- `logUserIds` (page 935)
- `quiet` (page 935)
- `traceExceptions` (page 936)

Replication

- `replApplyBatchSize` (page 937)
- `replIndexPrefetch` (page 937)

Security

- `auditAuthorizationSuccess` (page 939)
- `clusterAuthMode` (page 928)
- `scramIterationCount` (page 929)
- `sslMode` (page 929)

Sharding

- `userCacheInvalidationIntervalSecs` (page 930)

Storage

- `journalCommitInterval` (page 938)
- `syncdelay` (page 938)
- `wiredTigerEngineRuntimeConfigSetting`

JavaScript

- `disableJavaScriptJIT` (page 933)

getParameter

`getParameter`

`getParameter` (page 461) 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 460) for more about these parameters.

repairDatabase

On this page

- Definition (page 462)
- Behavior (page 463)
- Example (page 463)
- Using repairDatabase to Reclaim Disk Space (page 463)

Definition

repairDatabase

Checks and repairs errors and inconsistencies in data storage. repairDatabase (page 462) is analogous to a fsck command for file systems. Run the repairDatabase (page 462) command to ensure data integrity after the system experiences an unexpected system restart or crash, if:

1. The mongod (page 770) instance is not running with *journaling* enabled.

   When using *journaling*, there is almost never any need to run repairDatabase (page 462). 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 462) command and wrappers including db.repairDatabase() (page 195) in the mongo (page 803) 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 462).

repairDatabase (page 462) takes the following form:

```
{ repairDatabase: 1 }
```

repairDatabase (page 462) 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.

Changed in version 3.0: preserveClonedFilesOnFailure is only available with the mmapv1 storage engine.

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.

Changed in version 3.0: backupOriginalFiles is only available with the mmapv1 storage engine.

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 462) 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 462) you must run `repairDatabase` (page 462) 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 462) command is now available for secondary as well as primary members of replica sets.

The `repairDatabase` (page 462) command compacts all collections in the database. It is identical to running the `compact` (page 454) command on each collection individually.

`repairDatabase` (page 462) reduces the total size of the data files on disk. It also recreates all indexes in the database.

The time requirement for `repairDatabase` (page 462) depends on the size of the data set.

You may invoke `repairDatabase` (page 462) from multiple contexts:

- Use the `mongo` (page 803) shell to run the command, as above.
- Use the `db.repairDatabase()` (page 195) in the `mongo` (page 803) shell.
- Run `mongod` (page 770) directly from your system's shell. Make sure that `mongod` (page 770) isn't already running, and that you invoke `mongod` (page 770) 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` (page 915) 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

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

Using `repairDatabase` to Reclaim Disk Space You should not use `repairDatabase` (page 462) 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 462) is the appropriate and the only way to reclaim disk space.

repairCursor New in version 3.0.0.

`repairCursor`

Returns a cursor that iterates through all documents in a collection, omitting those that are not valid BSON. Used by `mongodump` (page 816) to provide the underlying functionality for the `--repair` option.

For internal use.

touch

On this page

- Considerations (page 464)

`touch`

New in version 2.2.

The `touch` (page 464) command loads data from the data storage layer into memory. `touch` (page 464) 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 770) will ideally be able to perform subsequent operations more efficiently. The `touch` (page 464) command has the following prototypical form:

```
{ touch: [collection], data: [boolean], index: [boolean] }
```

By default, `data` and `index` are false, and `touch` (page 464) 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 803) shell:

```
db.runCommand({ touch: "records", data: true, index: true })
```

`touch` (page 464) will not block read and write operations on a `mongod` (page 770), and can run on *secondary* members of replica sets.

Considerations

Performance Impact Using `touch` (page 464) to control or tweak what a `mongod` (page 770) 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 464) on a secondary, the secondary will enter a `RECOVERING` state to prevent clients from sending read operations during the `touch` (page 464) operation. When `touch` (page 464) finishes the secondary will automatically return to `SECONDARY` state. See `state` (page 402) for more information on replica set member states.

Storage Engines Changed in version 3.0.0.

If the current storage engine does not support `touch` (page 464), the `touch` (page 464) command will return an error.

The MMAPv1 storage engine supports `touch` (page 464).

The WiredTiger storage engine does *not* support `touch` (page 464).

shutdown

`shutdown`

The `shutdown` (page 465) command cleans up all database resources and then terminates the process. You must issue the `shutdown` (page 465) command against the *admin database* in the form:

```
{ shutdown: 1 }
```

Note: Run the `shutdown` (page 465) against the *admin database*. When using `shutdown` (page 465), 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 465) 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 803) shell helper syntax looks like this:

```
db.shutdownServer({timeoutSecs: 60});
```

logRotate

On this page

- Definition (page 465)
- Behavior (page 465)

Definition

`logRotate`

The `logRotate` (page 465) 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 465) command against the *admin database* in the form:

```
{ logRotate: 1 }
```

Note: Your `mongod` (page 770) 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 770) process. If your `mongod` (page 770) has a process ID of 2200, here's how to send the signal on Linux:

```
kill -SIGUSR1 2200
```

Behavior

Changed in version 3.0.0.

The `systemLog.logRotate` (page 898) setting or `--logRotate` (page 771) option specify `logRotate` (page 465)'s behavior.

When `systemLog.logRotate` (page 898) or `--logRotate` (page 771) are set to `rename`, `logRotate` (page 465) 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 465) creates a new log file with the same name as originally specified by the `systemLog.path` (page 897) setting to `mongod` (page 770) or `mongos` (page 792).

When `systemLog.logRotate` (page 898) or *`--logRotate`* (page 771) are set to `reopen`, `logRotate` (page 465) follows the typical Linux/Unix behavior, and simply closes the log file, and then reopens a log file with the same name. With `reopen`, `mongod` (page 770) expects that another process renames the file prior to the rotation, and that the reopen results in the creation of a new file.

Diagnostic Commands

Diagnostic Commands

Name	Description
`explain` (page 467)	Returns information on the execution of various operations.
`listDatabases` (page 469)	Returns a document that lists all databases and returns basic database statistics.
`dbHash` (page 470)	Internal command to support sharding.
`driverOIDTest` (page 470)	Internal command that converts an ObjectId to a string to support tests.
`listCommands` (page 470)	Lists all database commands provided by the current `mongod` (page 770) instance.
`availableQueryOptions` (page 471)	Internal command that reports on the capabilities of the current MongoDB instance.
`buildInfo` (page 471)	Displays statistics about the MongoDB build.
`collStats` (page 473)	Reports storage utilization statics for a specified collection.
`connPoolStats` (page 477)	Reports statistics on the outgoing connections from this MongoDB instance to other MongoDB instances in the deployment.
`shardConnPoolStats` (page 479)	Reports statistics on a `mongos` (page 792)‘s connection pool for client operations against shards.
`dbStats` (page 481)	Reports storage utilization statistics for the specified database.
`cursorInfo` (page 483)	Removed in MongoDB 3.2. Replaced with `metrics.cursor` (page 516).
`dataSize` (page 483)	Returns the data size for a range of data. For internal use.
`diagLogging` (page 483)	Provides a diagnostic logging. For internal use.
`getCmdLineOpts` (page 483)	Returns a document with the run-time arguments to the MongoDB instance and their parsed options.
`netstat` (page 484)	Internal command that reports on intra-deployment connectivity. Only available for `mongos` (page 792) instances.
`ping` (page 484)	Internal command that tests intra-deployment connectivity.
`profile` (page 484)	Interface for the *database profiler*.
`validate` (page 485)	Internal command that scans for a collection's data and indexes for correctness.
`top` (page 488)	Returns raw usage statistics for each database in the `mongod` (page 770) instance.
`whatsmyuri` (page 489)	Internal command that returns information on the current client.
`getLog` (page 489)	Returns recent log messages.
`hostInfo` (page 490)	Returns data that reflects the underlying host system.
`serverStatus` (page 492)	Returns a collection metrics on instance-wide resource utilization and status.
`features` (page 517)	Reports on features available in the current MongoDB instance.
`isSelf`	Internal command to support testing.

explain

On this page

- Definition (page 467)
- Behavior (page 467)
- Examples (page 468)
- Output (page 469)

Definition

explain

New in version 3.0.

The `explain` (page 467) command provides information on the execution of the following commands: `count` (page 307), `group` (page 313), `delete` (page 345), and `update` (page 340).

Although MongoDB provides the `explain` (page 467) command, the preferred method for running `explain` (page 467) is to use the `db.collection.explain()` (page 48) helper.

Note: The shell helpers(), `db.collection.explain()` (page 48) and `cursor.explain()` (page 140), are the only available mechanisms for explaining `db.collection.find()` (page 51) operations.

The `explain` (page 467) 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 307), `group` (page 313), `delete` (page 345), and `update` (page 340).

For `find()` operations, see `db.collection.explain()` (page 48).

field string verbosity Optional. A string specifying the mode in which to run `explain` (page 467). The mode affects the behavior of `explain` (page 467) and determines the amount of information to return.

Possible modes are: *"queryPlanner"* (page 467), *"executionStats"* (page 468), and *"allPlansExecution"* (page 468). For more information on the modes, see *explain behavior* (page 467).

By default, `explain` (page 467) runs in *"allPlansExecution"* (page 468) mode.

Behavior

The behavior of `explain` (page 467) 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 467) returns the `queryPlanner` (page 948) 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 467) returns information about the update or delete operations that *would* be performed, but does *not* apply the modifications to the database.

explain (page 467) returns the queryPlanner (page 948) and executionStats (page 949) information for the evaluated <command>. However, executionStats (page 949) does not provide query execution information for the rejected plans.

allPlansExecution Mode By default, explain (page 467) 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 467) returns information about the update or delete operations that *would* be performed, but does *not* apply the modifications to the database.

explain (page 467) returns the queryPlanner (page 948) and executionStats (page 949) information for the evaluated <command>. The executionStats (page 949) includes the *completed* query execution information for the *winning plan*.

If the query optimizer considered more than one plan, executionStats (page 949) 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 467) command runs in *"queryPlanner"* (page 467) verbosity mode to return the query planning information for a count (page 307) command:

```
db.runCommand(
   {
     explain: { count: "products", query: { quantity: { $gt: 50 } } },
     verbosity: "queryPlanner"
   }
)
```

executionStats Mode The following explain (page 467) operation runs in *"executionStats"* (page 468) verbosity mode to return the query planning and execution information for a count (page 307) command:

```
db.runCommand(
   {
      explain: { count: "products", query: { quantity: { $gt: 50 } } },
      verbosity: "executionStats"
   }
)
```

allPlansExecution Mode By default, explain (page 467) runs in *"allPlansExecution"* (page 468) verbosity mode. The following explain (page 467) command returns the queryPlanner (page 948) and executionStats (page 949) for all considered plans for an update (page 340) 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 467) operations can return information regarding:

- *queryPlanner* (page 947), which details the plan selected by the `query optimizer` and lists the rejected plans;
- *executionStats* (page 948), which details the execution of the winning plan and the rejected plans; and
- *serverInfo* (page 951), which provides information on the MongoDB instance.

The verbosity mode (i.e. `queryPlanner`, `executionStats`, `allPlansExecution`) determines whether the results include *executionStats* (page 948) and whether *executionStats* (page 948) includes data captured during *plan selection*.

For details on the output, see *Explain Results* (page 946).

listDatabases

On this page

- Definition (page 469)
- Output (page 469)

Definition

`listDatabases`

The `listDatabases` (page 469) command provides a list of all existing databases along with basic statistics about them:

```
{ listDatabases: 1 }
```

The value (e.g. `1`) does not affect the output of the command.

The `listDatabases` (page 469) must run against the `admin` database, as in the following example:

```
db.adminCommand( { listDatabases: 1 } )
```

Output The following is an example of a `listDatabases` (page 469) result:

```
{
   "databases" : [
      {
         "name" : "admin",
         "sizeOnDisk" : 83886080,
         "empty" : false
      },
      {
         "name" : "local",
         "sizeOnDisk" : 83886080,
         "empty" : false
      },
      {
         "name" : "test",
         "sizeOnDisk" : 83886080,
         "empty" : false
      }
   ],
   "totalSize" : 251658240,
   "totalSizeMb" : 240,
   "ok" : 1
}
```

listDatabases (page 469) returns a document that contains:

- A field named databases whose value is an array of documents, one 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.
  - For sharded clusters, a shards field that specifies the shards and the size in bytes of the database on disk for each shard.
- A field named totalSize whose value is the sum of all the sizeOnDisk fields.
- A field named totalSizeMb whose value is totalSize in megabytes.
- A field named ok whose value determines the success of the listDatabases (page 469) commands. 1 indicates success.

See also:

https://docs.mongodb.org/manual/tutorial/use-database-commands.

dbHash

dbHash

dbHash (page 470) is a command that supports *config servers* and is not part of the stable client facing API.

driverOIDTest

driverOIDTest

driverOIDTest (page 470) is an internal command.

listCommands

listCommands

The listCommands (page 470) command generates a list of all database commands implemented for the current mongod (page 770) or mongos (page 792) instance.

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

availableQueryOptions

availableQueryOptions

availableQueryOptions (page 471) is an internal command that is only available on mongos (page 792) instances.

buildInfo	**On this page** • Output (page 471)

buildInfo

buildInfo

The buildInfo (page 471) command is an administrative command which returns a build summary for the current mongod (page 770). buildInfo (page 471) has the following prototype form:

```
{ buildInfo: 1 }
```

In the mongo (page 803) shell, call buildInfo (page 471) in the following form:

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

Example

The output document of buildInfo (page 471) 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>
}
```

Output

buildInfo

The document returned by the buildInfo (page 471) command.

Supported These fields are stable and should provide consistent behavior.

buildInfo.**gitVersion**

The commit identifier that identifies the state of the code used to build the mongod (page 770).

buildInfo.**versionArray**

An array that conveys version information about the mongod (page 770) instance. See version for a more readable version of this string.

`document buildInfo.`**`version`**

A string that conveys version information about the `mongod` (page 770) instance. If you need to present version information to a human, this field is preferable to `versionArray` (page 471).

This string will take the format `<major>.<minor>.<patch>` in the case of a release, but development builds may contain additional information.

`document buildInfo.`**`storageEngines`**

New in version 3.2.

A list of `storage engines` avilable to the `mongod` (page 770) server.

`buildInfo.`**`javascriptEngine`**

Changed in version 3.2.

A string that reports the JavaScript engine used in the `mongod` (page 770) instance. By default, this is `mozjs` after version 3.2, and previously `V8`.

`buildInfo.`**`bits`**

A number that reflects the target processor architecture of the `mongod` (page 770) binary.

`buildInfo.`**`debug`**

A boolean. `true` when built with debugging options.

`buildInfo.`**`maxBsonObjectSize`**

A number that reports the `Maximum BSON Document Size` (page 940).

`buildInfo.`**`openssl`**

An embedded document describing the version of OpenSSL that `mongod` (page 770) was built with, as well as the version of OpenSSL that `mongod` (page 770) is currently using.

`buildInfo.`**`modules`**

A list of add-on modules that `mongod` (page 770) was built with. Possible values currently include “enterprise” and “rocksdb”.

Unstable These fields are for internal use only, and you should not expect their behavior or existence to remain consistent on any level.

`buildInfo.`**`sysInfo`**

Deprecated since version 3.2.

`buildInfo.sysInfo` (page 472) no longer contains useful information.

`buildInfo.`**`allocator`**

Changed in version 2.2.

The memory allocator that `mongod` (page 770) uses. By default this is `tcmalloc` after version 2.2, and `system` before 2.2.

`buildInfo.`**`buildEnvironment`**

An embedded document containing various debugging information about the `mongod` (page 770) build environment.

collStats

On this page

- Definition (page 473)
- Behavior (page 473)
- Example Output (page 473)
- Output (page 475)

Definition

collStats

The collStats (page 473) 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.

Note: The scale factor rounds values to whole numbers.

Behavior

Verbosity and MMAPv1 The verbose: true option increases reporting for the MMAPv1 storage engine.

Unexpected Shutdown and Count For MongoDB instances using the WiredTiger storage engine, after an unclean shutdown, statistics on size and count may off by up to 1000 documents as reported by collStats (page 473), dbStats (page 481), count (page 307). Run validate (page 485) on the collection to restore the correct statistics for the collection.

Example Output The following document provides a representation of the collStats (page 473) output. Depending on the configuration of your collection and the storage engine, the output fields may include a subset of the fields.

```
{
  "ns" : <string>,
  "count" : <number>,
  "size" : <number>,
  "avgObjSize" : <number>,
  "storageSize" : <number>,
  "capped" : <boolean>,
  "max" : <number>,
  "maxSize" :  <number>,
  "wiredTiger" : {
     "metadata" : {
        "formatVersion" : <num>
     },
     "creationString" : <string>
     "type" :  <string>,
     "uri" :  <string>,
     "LSM" : {
        "bloom filters in the LSM tree" : <number>,
        "bloom filter false positives" : <number>,
        "bloom filter hits" : <number>,
        "bloom filter misses" : <number>,
        "bloom filter pages evicted from cache" : <number>,
        "bloom filter pages read into cache" : <number>,
        "total size of bloom filters" : <number>,
        "sleep for LSM checkpoint throttle" : <number>,
        "chunks in the LSM tree" : <number>,
        "highest merge generation in the LSM tree" : <number>,
        "queries that could have benefited from a Bloom filter that did not exist" : <number>,
        "sleep for LSM merge throttle" : <number>
     },
     "block-manager" : {
        "file allocation unit size" : <number>,
        "blocks allocated" : <number>,
        "checkpoint size" : <number>,
        "allocations requiring file extension" : <number>,
        "blocks freed" : <number>,
        "file magic number" : <number>,
        "file major version number" : <number>,
        "minor version number" : <number>,
        "file bytes available for reuse" : <number>,
        "file size in bytes" : <number>
     },
     "btree" : {
        "btree checkpoint generation" : <number>,
        "column-store variable-size deleted values" : <number>,
        "column-store fixed-size leaf pages" : <number>,
        "column-store internal pages" : <number>,
        "column-store variable-size leaf pages" : <number>,
        "pages rewritten by compaction" : <number>,
        "number of key/value pairs" : <number>,
        "fixed-record size" : <number>,
        "maximum tree depth" : <number>,
        "maximum internal page key size" : <number>,
        "maximum internal page size" :<number>,
        "maximum leaf page key size" : <number>,
        "maximum leaf page size" : <number>,
        "maximum leaf page value size" : <number>,
        "overflow pages" : <number>,
        "row-store internal pages" : <number>,
        "row-store leaf pages" : <number>
     },
     "cache" : {
        "bytes read into cache" : <number>,
        "bytes written from cache" : <number>,
        "checkpoint blocked page eviction" : <number>,
        "unmodified pages evicted" : <number>,
        "page split during eviction deepened the tree" : <number>,
        "modified pages evicted" : <number>,
        "data source pages selected for eviction unable to be evicted" : <number>,
        "hazard pointer blocked page eviction" : <number>,
        "internal pages evicted" : <number>,
        "pages split during eviction" : <number>,
        "in-memory page splits" : <number>,
        "overflow values cached in memory" : <number>,
        "pages read into cache" : <number>,
        "overflow pages read into cache" : <number>,
        "pages written from cache" : 2
     },
     "compression" : {
        "raw compression call failed, no additional data available" : <number>,
        "raw compression call failed, additional data available" : <number>,
        "raw compression call succeeded" : <number>,
        "compressed pages read" : <number>,
        "compressed pages written" : <number>,
        "page written failed to compress" : <number>,
        "page written was too small to compress" : 1
     },
     "cursor" : {
        "create calls" : <number>,
        "insert calls" : <number>,
        "bulk-loaded cursor-insert calls" : <number>,
        "cursor-insert key and value bytes inserted" : <number>,
        "next calls" : <number>,
        "prev calls" : <number>,
        "remove calls" : <number>,
        "cursor-remove key bytes removed" : <number>,
        "reset calls" : <number>,
        "search calls" : <number>,
        "search near calls" : <number>,
        "update calls" : <number>,
        "cursor-update value bytes updated" : <number>
     },
     "reconciliation" : {
        "dictionary matches" : <number>,
        "internal page multi-block writes" : <number>,
        "leaf page multi-block writes" : <number>,
        "maximum blocks required for a page" : <number>,
        "internal-page overflow keys" : <number>,
        "leaf-page overflow keys" : <number>,
        "overflow values written" : <number>,
        "pages deleted" : <number>,
        "page checksum matches" : <number>,
        "page reconciliation calls" : <number>,
        "page reconciliation calls for eviction" : <number>,
        "leaf page key bytes discarded using prefix compression" : <number>,
        "internal page key bytes discarded using suffix compression" : <number>
     },
     "session" : {
        "object compaction" : <number>,
        "open cursor count" : <number>
     },
     "transaction" : {
        "update conflicts" : <number>
     }
  },
  "nindexes" : <number>,          // number of indexes
  "totalIndexSize" : <number>,    // total index size in bytes
  "indexSizes" : {                 // size of specific indexes in bytes
          "_id_" : <number>,
          "username" : <number>
  },
  // ...
  "ok" : <number>
}
```

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 in memory 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 475) does not include the size of any indexes associated with the collection, which the `totalIndexSize` (page 476) field reports.

The `scale` argument affects this value.

`collStats.`**`avgObjSize`**
The average size of an object in the collection. The `scale` argument does not affect this value.

`collStats.`**`storageSize`**
The total amount of storage allocated to this collection for *document* storage. The `scale` argument affects this value.

`storageSize` (page 476) does not include index size. See `totalIndexSize` (page 476) for index sizing.

For MMAPv1, `storageSize` (page 476) 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 770) 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 3.0.0: `paddingFactor` (page 476) is no longer used in 3.0.0, and remains hard coded to 1.0 for compatibility only.

`paddingFactor` (page 476) only appears when using the `mmapv1` storage engine.

`collStats.`**`userFlags`**
New in version 2.2.

A number that indicates the user-set flags on the collection. `userFlags` (page 476) only appears when using the `mmapv1` storage engine.

Changed in version 3.0.0: `userFlags` (page 476) reports on the `usePowerOf2Sizes` (page 458) and the `noPadding` (page 458) flags.

- `0` corresponds to `usePowerOf2Sizes` (page 458) flag set to `false` and `noPadding` (page 458) flag set to `false`.
- `1` corresponds to `usePowerOf2Sizes` (page 458) flag set to `true` and `noPadding` (page 458) flag set to `false`.
- `2` corresponds to `usePowerOf2Sizes` (page 458) flag set to `false` and `noPadding` (page 458) flag set to `true`.
- `3` corresponds to `usePowerOf2Sizes` (page 458) flag set to `true` and `noPadding` (page 458) flag set to `true`.

Note: MongoDB 3.0 ignores the `usePowerOf2Sizes` (page 458) flag. See `collMod` (page 457) and `db.createCollection()` (page 167) for more information.

`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 3.0.0.

`wiredTiger` (page 477) only appears when using the WiredTiger storage engine.

This document contains data reported directly by the WiredTiger engine and other data for internal diagnostic use.

`collStats`.**indexDetails**
New in version 3.0.0.

A document that reports data from the *WiredTiger* storage engine for each index in the collection. Other storage engines will return an empty document.

The fields in this document are the names of the indexes, while the values themselves are documents that contain statistics for the index provided by the storage engine. These statistics are for internal diagnostic use.

connPoolStats

On this page

- Definition (page 477)
- Output (page 477)

Definition

connPoolStats
The command `connPoolStats` (page 477) 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 477) only returns meaningful results for `mongos` (page 792) instances and for `mongod` (page 770) 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 embedded documents of the `hosts` (page 477) *document* report connections between the `mongos` (page 792) or `mongod` (page 770) instance and each component `mongod` (page 770) of the *sharded cluster*.

`connPoolStats.hosts.[host].`**`available`**

`available` (page 477) reports the total number of connections that the `mongos` (page 792) or `mongod` (page 770) could use to connect to this `mongod` (page 770).

`connPoolStats.hosts.[host].`**`created`**

`created` (page 478) reports the number of connections that this `mongos` (page 792) or `mongod` (page 770) has ever created for this host.

`connPoolStats.`**`replicaSets`**

`replicaSets` (page 478) is a *document* that contains *replica set* information for the *sharded cluster*.

`connPoolStats.replicaSets.`**`shard`**

The `shard` (page 478) *document* reports on each *shard* within the *sharded cluster*

`connPoolStats.replicaSets.[shard].`**`host`**

The `host` (page 478) 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 399) values.

`connPoolStats.replicaSets.[shard].host[n].`**`addr`**

`addr` (page 478) reports the address for the host in the *sharded cluster* in the format of "`[hostname]:[port]`".

`connPoolStats.replicaSets.[shard].host[n].`**`ok`**

`ok` (page 478) reports `false` when:

- the `mongos` (page 792) or `mongod` (page 770) cannot connect to instance.
- the `mongos` (page 792) or `mongod` (page 770) received a connection exception or error.

This field is for internal use.

`connPoolStats.replicaSets.[shard].host[n].`**`ismaster`**

`ismaster` (page 478) reports `true` if this `host` (page 478) is the *primary* member of the *replica set*.

`connPoolStats.replicaSets.[shard].host[n].`**`hidden`**

`hidden` (page 478) reports `true` if this `host` (page 478) is a *hidden member* of the *replica set*.

`connPoolStats.replicaSets.[shard].host[n].`**`secondary`**

`secondary` (page 478) reports `true` if this `host` (page 478) is a *secondary* member of the *replica set*.

`connPoolStats.replicaSets.[shard].host[n].`**`pingTimeMillis`**

`pingTimeMillis` (page 478) reports the ping time in milliseconds from the `mongos` (page 792) or `mongod` (page 770) to this `host` (page 478).

`connPoolStats.replicaSets.[shard].host[n].`**`tags`**

New in version 2.2.

`tags` (page 478) reports the `members[n].tags`, if this member of the set has tags configured.

`connPoolStats.replicaSets.[shard].`**`master`**

`master` (page 478) reports the ordinal identifier of the host in the `host` (page 478) array that is the *primary* of the *replica set*.

`connPoolStats.replicaSets.[shard].`**`nextSlave`**

Deprecated since version 2.2.

`nextSlave` (page 478) reports the *secondary* member that the `mongos` (page 792) will use to service the next request for this *replica set*.

`connPoolStats.`**`createdByType`**

`createdByType` (page 478) *document* reports the number of each type of connection that `mongos` (page 792) or `mongod` (page 770) has created in all connection pools.

mongos (page 792) connect to mongod (page 770) instances using one of three types of connections. The following embedded document reports the total number of connections by type.

connPoolStats.createdByType.**master**
master (page 479) reports the total number of connections to the *primary* member in each *cluster*.

connPoolStats.createdByType.**set**
set (page 479) reports the total number of connections to a *replica set* member.

connPoolStats.createdByType.**sync**
sync (page 479) reports the total number of *config database* connections.

connPoolStats.**totalAvailable**
totalAvailable (page 479) reports the running total of connections from the mongos (page 792) or mongod (page 770) to all mongod (page 770) instances in the *sharded cluster* available for use.

connPoolStats.**totalCreated**
totalCreated (page 479) reports the total number of connections ever created from the mongos (page 792) or mongod (page 770) to all mongod (page 770) instances in the *sharded cluster*.

connPoolStats.**numDBClientConnection**
numDBClientConnection (page 479) reports the total number of connections from the mongos (page 792) or mongod (page 770) to all of the mongod (page 770) instances in the *sharded cluster*.

connPoolStats.**numAScopedConnection**
numAScopedConnection (page 479) reports the number of exception safe connections created from mongos (page 792) or mongod (page 770) to all mongod (page 770) in the *sharded cluster*. The mongos (page 792) or mongod (page 770) releases these connections after receiving a socket exception from the mongod (page 770).

shardConnPoolStats

On this page

- Definition (page 479)
- Output (page 479)

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 479) command uses the following syntax:

```
{ shardConnPoolStats: 1 }
```

The sharded connection pool is specific to connections between members in a sharded cluster. The mongos (page 792) instances in a cluster use the connection pool to execute client reads and writes. The mongod (page 770) instances in a cluster use the pool when issuing mapReduce (page 318) 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 792).

`shardConnPoolStats.hosts.<host>.`**`created`**
The number of connections the host has ever created to connect to the `mongos` (page 792).

`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 399) 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 792) 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 `true`.

`shardConnPoolStats.replicaSets.<name>.host[n].`**`hidden`**
The host is a *hidden member* of the replica set if this is `true`.

`shardConnPoolStats.replicaSets.<name>.host[n].`**`secondary`**
The host is a *hidden member* of the replica set if this is `true`.

The host is a *secondary* member of the replica set if this is `true`.

`shardConnPoolStats.replicaSets.<name>.host[n].`**`pingTimeMillis`**
The latency, in milliseconds, from the `mongos` (page 792) to this member.

`shardConnPoolStats.replicaSets.<name>.host[n].`**`tags`**
A *tag set* document containing mappings of arbitrary keys and values. These documents describe replica set members in order to customize `write concern` and `read preference` and thereby allow configurable data center awareness.

This field is only present if there are tags assigned to the member. See `https://docs.mongodb.org/manual/tutorial/configure-replica-set-tag-sets` for more information.

`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 792) to the config servers, replica sets, and standalone `mongod` (page 770) instances in the cluster.

`shardConnPoolStats.`**`totalCreated`**
The number of connections the `mongos` (page 792) 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 792), this array field displays one document per incoming client thread. For a `mongod` (page 770), the array displays one entry per incoming sharded `mapReduce` (page 318) 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.

dbStats

On this page

- Definition (page 481)
- Behavior (page 481)
- Output (page 481)

Definition

`dbStats`

The `dbStats` (page 481) 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: Scaling rounds values to whole numbers.

In the `mongo` (page 803) shell, the `db.stats()` (page 199) function provides a wrapper around `dbStats` (page 481).

Behavior 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.

For MongoDB instances using the `WiredTiger` storage engine, after an unclean shutdown, statistics on size and count may off by up to 1000 documents as reported by `collStats` (page 473), `dbStats` (page 481), `count` (page 307). To restore the correct statistics for the collection, run `validate` (page 485) on the collection.

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 482) 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 482) 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 482) 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 482) 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 940)

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 3.0.0.

dbStats.extentFreeList.**num**

New in version 3.0.0.

Number of extents in the freelist. Only present when using the `mmapv1` storage engine.

dbStats.extentFreeList.**size**

New in version 3.0.0.

Total size of the extents on the freelist.

The `scale` argument affects this value. Only present when using the `mmapv1` storage engine.

cursorInfo

cursorInfo

Changed in version 3.2: Removed.

Use the `serverStatus` (page 492) command to return the `metrics.cursor` (page 516) information.

dataSize

dataSize

The `dataSize` (page 483) 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 483) depends on the amount of data in the collection.

diagLogging Deprecated since version 3.0.0.

diagLogging

`diagLogging` (page 483) 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.

getCmdLineOpts

getCmdLineOpts

The `getCmdLineOpts` (page 483) command returns a document containing command line options used to start the given `mongod` (page 770) or `mongos` (page 792):

```
{ 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 770) or `mongos` (page 792). 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 483):

```
{
        "argv" : [
               "/usr/bin/mongod",
               "--config",
               "/etc/mongod.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
}
```

netstat

`netstat`

`netstat` (page 484) is an internal command that is only available on `mongos` (page 792) instances.

ping

`ping`

The `ping` (page 484) 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 484) 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 770) writes the output of the database profiler to the `system.profile` collection.

mongod (page 770) records queries that take longer than the slowOpThresholdMs (page 921) to the server log even when the database profiler is not active.

See also:

Additional documentation regarding *Database Profiling*.

See also:

"db.getProfilingStatus() (page 189)" and "db.setProfilingLevel() (page 199)" provide wrappers around this functionality in the mongo (page 803) 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.

validate

On this page

- Definition (page 485)
- Output (page 485)

Definition

`validate`

The `validate` (page 485) 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 803) shell also provides a wrapper `db.collection.validate()` (page 132):

```
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 485) returns one instance of this document for every extent in the collection. This embedded 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 486) 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 486) embedded 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 486) 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 487) 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 487) returns data collected from a scan of all records. The value should be the same as `datasize` (page 486).

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 485) determines that an aspect of the collection is not valid. When `false`, see the `errors` (page 487) field for more information.

`validate.`**`errors`**
Typically empty; however, if the collection is not valid (i.e `valid` (page 487) 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 488) field has a value of `0`.

top

On this page

- Example (page 488)

`top`

`top` (page 488) is an administrative command that returns usage statistics for each collection. `top` (page 488) 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 488) command against the *admin database* in the form:

```
{ top: 1 }
```

Example At the `mongo` (page 803) shell prompt, use `top` (page 488) with the following evocation:

```
db.adminCommand("top")
```

Alternately you can use `top` (page 488) 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
                        }
                }
}
```

whatsmyuri

whatsmyuri

whatsmyuri (page 489) is an internal command.

getLog

getLog

The getLog (page 489) command returns a document with a log array that contains recent messages from the mongod (page 770) process log. The getLog (page 489) 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 770) is part of a *replica set*, getLog (page 489) 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 770) 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 803) shell connected to a replica set:

```
db.adminCommand({getLog: "*" })
{ "names" : [ "global", "rs", "startupWarnings" ], "ok" : 1 }
```

getLog (page 489) returns events from a RAM cache of the mongod (page 770) events and *does not* read log data from the log file.

hostInfo

On this page

- Output (page 490)

hostInfo

New in version 2.2.

Returns A document with information about the underlying system that the mongod (page 770) or mongos (page 792) runs on. Some of the returned fields are only included on some platforms.

You must run the hostInfo (page 490) command, which takes no arguments, against the admin database. Consider the following invocations of hostInfo (page 490):

```
db.hostInfo()
db.adminCommand( { "hostInfo" : 1 } )
```

In the mongo (page 803) shell you can use db.hostInfo() (page 191) as a helper to access hostInfo (page 490). The output of hostInfo (page 490) 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>
}
```

Output

hostInfo

The document returned by the hostInfo (page 490).

hostInfo.**system**

An embedded document providing information about the underlying environment of the system running the mongod (page 770) or mongos (page 792)

hostInfo.system.**currentTime**
A timestamp 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**
An embedded document that contains information about the operating system running the `mongod` (page 770) and `mongos` (page 792).

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**
An embedded document with extra information about the operating system and the underlying hardware. The content of the `extra` (page 491) embedded document depends on the operating system.

hostInfo.extra.**cpuString**
A string containing a human-readable description of the system's processor.

`cpuString` (page 491) only appears on OS X systems.

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 491) only appears on Linux systems.

hostInfo.extra.**kernelVersion**
The release of the Linux kernel in current use.

`kernelVersion` (page 491) only appears on Linux systems.

hostInfo.extra.**alwaysFullSync**
`alwaysFullSync` (page 491) only appears on OS X systems.

hostInfo.extra.**nfsAsync**
`nfsAsync` (page 491) 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.**physicalCores**
Reports the number of physical, non-HyperThreading, cores available on the system.

`physicalCores` (page 492) only appears on OS X systems.

hostInfo.extra.**numPages**
`numPages` (page 492) only appears on Linux systems.

hostInfo.extra.**maxOpenFiles**
Reports the current system limits on open file handles. See `https://docs.mongodb.org/manual/reference/ulimit`
for more information.

`maxOpenFiles` (page 492) only appears on Linux systems.

hostInfo.extra.**scheduler**
Reports the active I/O scheduler. `scheduler` (page 492) only appears on OS X systems.

serverStatus	**On this page**
	• Definition (page 492) • Behavior (page 492) • Output (page 493)

Definition

serverStatus
The `serverStatus` (page 492) command returns a document that provides an overview of the database's state. Monitoring applications can run this command at a regular interval to collection statistics about the instance.

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

The value (i.e. `1` above) does not affect the operation of the command. The `mongo` (page 803) shell provides the `db.serverStatus()` (page 197) wrapper for the command.

See also:

Much of the output of `serverStatus` (page 492) is also displayed dynamically by `mongostat` (page 858). See the *mongostat* (page 857) command for more information.

Behavior By default, `serverStatus` (page 492) excludes in its output *rangeDeleter* (page 501) information and some content in the *repl* (page 502) document.

To include fields that are excluded by default, specify the top-level field and set it to `1` in the command. To exclude fields that are included by default, specify the top-level field and set to `0` in the command.

For example, the following operation suppresses the `repl`, `metrics` and `locks` information in the output.

```
db.runCommand( { serverStatus: 1, repl: 0, metrics: 0, locks: 0 } )
```

The following example includes *rangeDeleter* (page 501) and all *repl* (page 502) information in the output:

```
db.runCommand( { serverStatus: 1, rangeDeleter: 1, repl: 1 } )
```

Output

Note: The output fields vary depending on the version of MongoDB, underlying operating system platform, the storage engine, and the kind of node, including `mongos` (page 792), `mongod` (page 770) or *replica set* member.

For the `serverStatus` (page 492) output specific to the version of your MongoDB, refer to the appropriate version of the MongoDB Manual.

Changed in version 3.0: `serverStatus` (page 492) no longer outputs the `workingSet`, `indexCounters`, and `recordStats` sections.

Instance Information

```
"host" : <string>,
"advisoryHostFQDNs" : <array>,
"version" : <string>,
"process" : <"mongod"|"mongos">,
"pid" : <num>,
"uptime" : <num>,
"uptimeMillis" : <num>,
"uptimeEstimate" : <num>,
"localTime" : ISODate(""),
```

`host`
The system's hostname. In Unix/Linux systems, this should be the same as the output of the `hostname` command.

`advisoryHostFQDNs`
New in version 3.2.

An array of the system's fully qualified domain names (FQDNs).

`version`
The MongoDB version of the current MongoDB process.

`process`
The current MongoDB process. Possible values are: `mongos` (page 792) or `mongod` (page 770).

`pid`
The process id number.

`uptime`
The number of seconds that the current MongoDB process has been active.

`uptimeMillis`
The number of milliseconds that the current MongoDB process has been active.

`uptimeEstimate`
The uptime in seconds as calculated from MongoDB's internal course-grained time keeping system.

`localTime`
The ISODate representing the current time, according to the server, in UTC.

asserts

```
"asserts" : {
   "regular" : <num>,
   "warning" : <num>,
   "msg" : <num>,
   "user" : <num>,
   "rollovers" : <num>
},
```

asserts
A document that reports on the number of assertions raised since the MongoDB process started. While assert errors are typically uncommon, if there are non-zero values for the `asserts`, you should check the log file for more information. In many cases, these errors are trivial, but are worth investigating.

asserts.**regular**
The number of regular assertions raised since the MongoDB process started. Check the log file for more information about these messages.

asserts.**warning**
The number of warnings raised since the MongoDB process started. Check the log file for more information about these warnings.

asserts.**msg**
The number of message assertions raised since the MongoDB process started. Check the log file for more information about these messages.

asserts.**user**
The number of "user asserts" that have occurred since the last time the MongogDB 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.

asserts.**rollovers**
The number of times that the rollover counters have rolled over since the last time the MongoDB 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 494) data structure.

backgroundFlushing

```
"backgroundFlushing" : {
   "flushes" : <num>,
   "total_ms" : <num>,
   "average_ms" : <num>,
   "last_ms" : <num>,
   "last_finished" : ISODate("...")
},
```

Note: `backgroundFlushing` information only appears for instances that use the `MMAPv1` storage engine.

backgroundFlushing
A document that reports on the `mongod` (page 770) process's periodic writes to disk. Consider these values if you have concerns about write performance and *journaling* (page 495).

backgroundFlushing.**flushes**
The number of times the database has flushed all writes to disk. This value will grow as database runs for longer periods of time.

backgroundFlushing.**total_ms**
The total number of milliseconds (ms) that the `mongod` (page 770) processes have spent writing (i.e. flushing)

data to disk. Because `total_ms` (page 494) is an absolute value, consider the `flushes` (page 494) and `average_ms` (page 495) values to provide context.

`backgroundFlushing.`**`average_ms`**
The average time in milliseconds for each flush to disk, calculated by dividing `total_ms` (page 494) by `flushes`.

The `average_ms` (page 495) is more likely to to represent a "normal" time as the value of the `flushes` (page 494) increases. However, abnormal data can skew this value. Use the `backgroundFlushing.last_ms` (page 495) to check that a high average is not skewed by transient historical issue or a random write distribution.

`backgroundFlushing.`**`last_ms`**
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 is in line with the historical data provided by `backgroundFlushing.average_ms` (page 495) and `backgroundFlushing.total_ms` (page 494).

`backgroundFlushing.`**`last_finished`**
The timestamp of the last completed flush operation in the *ISODate* format. If this value is more than a few minutes past 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 blocking write operations.

connections

```
"connections" : {
   "current" : <num>,
   "available" : <num>,
   "totalCreated" : NumberLong(<num>)
},
```

`connections`
A document that reports on the status of the connections. Use these values to assess the current load and capacity requirements of the server.

`connections.`**`current`**
The number of incoming connections from clients to the database server . This number includes the current shell session. Consider the value of `connections.available` (page 495) to add more context to this datum.

The value will include all incoming connections including any shell connections or connections from other servers, such as *replica set* members or `mongos` (page 792) instances.

`connections.`**`available`**
The number of unused incoming connections available. Consider this value in combination with the value of `connections.current` (page 495) to understand the connection load on the database, and the `https://docs.mongodb.org/manual/reference/ulimit` document for more information about system thresholds on available connections.

`connections.`**`totalCreated`**
Count of **all** incoming connections created to the server. This number includes connections that have since closed.

dur (Journaling)

```
"dur" : {
   "commits" : <num>,
   "journaledMB" : <num>,
   "writeToDataFilesMB" : <num>,
   "compression" : <num>,
   "commitsInWriteLock" : <num>,
   "earlyCommits" : <num>,
   "timeMs" : {
      "dt" : <num>,
      "prepLogBuffer" : <num>,
      "writeToJournal" : <num>,
      "writeToDataFiles" : <num>,
      "remapPrivateView" : <num>,
      "commits" : <num>,
      "commitsInWriteLock" : <num>
   }
},
```

Note: `dur` (page 496) (journaling) information only appears for `mongod` (page 770) instances that use the `MMAPv1` storage engine and have journaling enabled.

dur

A document that reports the `mongod` (page 770) instance's `journaling-related operations` and performance. MongoDB reports on this data based on 3 second intervals, collected between 3 and 6 seconds in the past.

dur.**commits**

The number of transactions written to the *journal* during the last *journal group commit interval*.

dur.**journaledMB**

The amount of data in megabytes (MB) written to *journal* during the last *journal group commit interval*.

dur.**writeToDataFilesMB**

The amount of data in megabytes (MB) written from *journal* to the data files during the last *journal group commit interval*.

dur.**compression**

The compression ratio of the data written to the *journal*:

```
( journaled_size_of_data / uncompressed_size_of_data )
```

dur.**commitsInWriteLock**

The 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.

dur.**earlyCommits**

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.

dur.**timeMS**

A document that reports on the performance of the `mongod` (page 770) instance during the various phases of journaling in the last *journal group commit interval*.

dur.timeMS.**dt**

The amount of time, in milliseconds, over which MongoDB collected the `dur.timeMS` (page 496) data. Use this field to provide context to the other `dur.timeMS` (page 496) field values.

dur.timeMS.**prepLogBuffer**

The amount of time, in milliseconds, spent preparing to write to the journal. Smaller values indicate better journal performance.

dur.timeMS.**writeToJournal**

The amount of time, in milliseconds, spent actually writing to the journal. File system speeds and device interfaces can affect performance.

dur.timeMS.**writeToDataFiles**
The amount of time, in milliseconds, spent writing to data files after journaling. File system speeds and device interfaces can affect performance.

dur.timeMS.**remapPrivateView**
The amount of time, in milliseconds, spent remapping copy-on-write memory mapped views. Smaller values indicate better journal performance.

dur.timeMS.**commits**
The amount of time, in milliseconds, spent for commits.

dur.timeMS.**commitsInWriteLock**
The amount of time, in milliseconds, spent for commits that occurred while a write lock was held.

extra_info

```
"extra_info" : {
   "note" : "fields vary by platform.",
   "heap_usage_bytes" : <num>,
   "page_faults" : <num>
},
```

extra_info
A document that provides additional information regarding the underlying system.

extra_info.**note**
A string with the text `"fields vary by platform."`

extra_info.**heap_usage_bytes**
The total size in bytes of heap space used by the database process. Available on Unix/Linux systems only.

extra_info.**page_faults**
The total number of page faults. The `extra_info.page_faults` (page 497) 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.

Windows draws a distinction between “hard” page faults involving disk I/O, and “soft” page faults that only require moving pages in memory. MongoDB counts both hard and soft page faults in this statistic.

globalLock

```
"globalLock" : {
   "totalTime" : <num>,
   "currentQueue" : {
      "total" : <num>,
      "readers" : <num>,
      "writers" : <num>
   },
   "activeClients" : {
      "total" : <num>,
      "readers" : <num>,
      "writers" : <num>
   }
},
```

globalLock
A document that reports on the database’s lock state.

Generally, the *locks* (page 498) document provides more detailed data on lock uses.

globalLock.**totalTime**
The time, in microseconds, since the database last started and created the globalLock (page 497). This is roughly equivalent to total server uptime.

globalLock.**currentQueue**
A document that provides information concerning the number of operations queued because of a lock.

globalLock.currentQueue.**total**
The total number of operations queued waiting for the lock (i.e., the sum of globalLock.currentQueue.readers (page 498) and globalLock.currentQueue.writers (page 498)).

A consistently small queue, particularly of shorter operations, should cause no concern. The globalLock.activeClients (page 498) readers and writers information provides contenxt for this data.

globalLock.currentQueue.**readers**
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.

globalLock.currentQueue.**writers**
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**
A document that provides information about the number of connected clients and the read and write operations performed by these clients.

Use this data to provide context for the globalLock.currentQueue (page 498) data.

globalLock.activeClients.**total**
The total number of active client connections to the database (i.e., the sum of globalLock.activeClients.readers (page 498) and globalLock.activeClients.writers (page 498)).

globalLock.activeClients.**readers**
The number of the active client connections performing read operations.

globalLock.activeClients.**writers**
The number of active client connections performing write operations.

locks

```
"locks" : {
   <type> : {
         "acquireCount" : {
            <mode> : NumberLong(<num>),
            ...
         },
         "acquireWaitCount" : {
            <mode> : NumberLong(<num>),
            ...
         },
         "timeAcquiringMicros" : {
            <mode> : NumberLong(<num>),
            ...
         },
         "deadlockCount" : {
            <mode> : NumberLong(<num>),
            ...
         }
   },
   ...
```

locks

Changed in version 3.0.

A document that reports for each lock `<type>`, data on lock `<modes>`.

The possible lock `<types>` are:

Lock Type	Description
`Global`	Represents global lock.
`MMAPV1Journal`	Represents MMAPv1 storage engine specific lock to synchronize journal writes; for non-MMAPv1 storage engines, the mode for `MMAPV1Journal` is empty.
`Database`	Represents database lock.
`Collection`	Represents collection lock.
`Metadata`	Represents metadata lock.
`oplog`	Represents lock on the *oplog*.

The possible `<modes>` are:

Lock Mode	Description
`R`	Represents Shared (S) lock.
`W`	Represents Exclusive (X) lock.
`r`	Represents Intent Shared (IS) lock.
`w`	Represents Intent Exclusive (IX) lock.

All values are of the `NumberLong()` type.

`locks.<type>.`**acquireCount**

Number of times the lock was acquired in the specified mode.

`locks.<type>.`**acquireWaitCount**

Number of times the `locks.acquireCount` lock acquisitions encountered waits because the locks were held in a conflicting mode.

`locks.<type>.`**timeAcquiringMicros**

Cumulative wait time in microseconds for the lock acquisitions.

`locks.timeAcquiringMicros` divided by `locks.acquireWaitCount` gives an approximate average wait time for the particular lock mode.

`locks.<type>.`**deadlockCount**

Number of times the lock acquisitions encountered deadlocks.

network

```
"network" : {
   "bytesIn" : <num>,
   "bytesOut" : <num>,
   "numRequests" : <num>
},
```

network

A document that reports data on MongoDB's network use.

`network.`**bytesIn**

The number of bytes that reflects the amount of network traffic received *by* this database. Use this value to ensure that network traffic sent to the `mongod` (page 770) process is consistent with expectations and overall inter-application traffic.

`network.`**`bytesOut`**
The number of bytes that reflects the amount of network traffic sent *from* this database. Use this value to ensure that network traffic sent by the `mongod` (page 770) process is consistent with expectations and overall inter-application traffic.

`network.`**`numRequests`**
The total number of distinct requests that the server has received. Use this value to provide context for the `network.bytesIn` (page 499) and `network.bytesOut` (page 499) values to ensure that MongoDB's network utilization is consistent with expectations and application use.

opcounters

```
"opcounters" : {
   "insert" : <num>,
   "query" : <num>,
   "update" : <num>,
   "delete" : <num>,
   "getmore" : <num>,
   "command" : <num>
},
```

`opcounters`
A document that reports on database operations by type since the `mongod` (page 770) instance last started.

These numbers will grow over time until next restart. Analyze these values over time to track database utilization.

Note: The data in `opcounters` (page 500) treats operations that affect multiple documents, such as bulk insert or multi-update operations, as a single operation. See `metrics.document` (page 513) for more granular document-level operation tracking.

Additionally, these values reflect received operations, and increment even when operations are not successful.

`opcounters.`**`insert`**
The total number of insert operations received since the `mongod` (page 770) instance last started.

`opcounters.`**`query`**
The total number of queries received since the `mongod` (page 770) instance last started.

`opcounters.`**`update`**
The total number of update operations received since the `mongod` (page 770) instance last started.

`opcounters.`**`delete`**
The total number of delete operations since the `mongod` (page 770) instance last started.

`opcounters.`**`getmore`**
The total number of "getmore" operations since the `mongod` (page 770) 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.

`opcounters.`**`command`**
The total number of commands issued to the database since the `mongod` (page 770) instance last started.

`opcounters.command` (page 500) counts all *commands* (page 303) **except** the write commands: `insert` (page 337), `update` (page 340), and `delete` (page 345).

opcountersRepl

```
"opcountersRepl" : {
   "insert" : <num>,
   "query" : <num>,
   "update" : <num>,
   "delete" : <num>,
   "getmore" : <num>,
   "command" : <num>
},
```

opcountersRepl

A document that reports on database replication operations by type since the mongod (page 770) instance last started.

These values only appear when the current host is a member of a replica set.

These values will differ from the opcounters (page 500) values because of how MongoDB serializes operations during replication. See https://docs.mongodb.org/manual/replication for more information on replication.

These numbers will grow over time in response to database use until next restart. Analyze these values over time to track database utilization.

opcountersRepl.**insert**

The total number of replicated insert operations since the mongod (page 770) instance last started.

opcountersRepl.**query**

The total number of replicated queries since the mongod (page 770) instance last started.

opcountersRepl.**update**

The total number of replicated update operations since the mongod (page 770) instance last started.

opcountersRepl.**delete**

The total number of replicated delete operations since the mongod (page 770) instance last started.

opcountersRepl.**getmore**

The total number of "getmore" operations since the mongod (page 770) 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.

opcountersRepl.**command**

The total number of replicated commands issued to the database since the mongod (page 770) instance last started.

rangeDeleter

```
"rangeDeleter" : {
   "lastDeleteStats" : [
      {
        "deletedDocs" : NumberLong(<num>),
        "queueStart" : <date>,
        "queueEnd" : <date>,
        "deleteStart" : <date>,
        "deleteEnd" : <date>,
        "waitForReplStart" : <date>,
        "waitForReplEnd" : <date>
      }
   ]
}
```

By default, `serverStatus` (page 492) does not include `rangeDeleter` (page 502) data in the output. To include the `rangeDeleter` (page 502) data, use one of the following commands:

```
db.serverStatus( { rangeDeleter: 1 } )
db.runCommand( { serverStatus: 1, rangeDeleter: 1 } )
```

rangeDeleter
New in version 3.0.

A document that reports on the work performed by the `cleanupOrphaned` (page 415) command and the cleanup phase of the `moveChunk` (page 427) command.

`rangeDeleter.`**lastDeleteStats**
An array of documents that each report on the last operations of migration cleanup operations. At most `rangeDeleter.lastDeleteStats` (page 502) will report data for the last 10 operations.

`rangeDeleter.lastDeleteStats[n].`**deletedDocs**
The number of documents deleted by migration cleanup operations.

`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 770) waits for open cursors to close on the namespace.

`rangeDeleter.lastDeleteStats[n].`**queueEnd**
A timestamp that reflects when the migration cleanup operation begins.

`rangeDeleter.lastDeleteStats[n].`**deleteStart**
A timestamp for the beginning of the delete process that is part of the migration cleanup operation.

`rangeDeleter.lastDeleteStats[n].`**deleteEnd**
A timestamp for the end of the delete process that is part of the migration cleanup operation.

`rangeDeleter.lastDeleteStats[n].`**waitForReplStart**
A timestamp that reflects when the migration cleanup operation began waiting for replication to process the delete operation.

`rangeDeleter.lastDeleteStats[n].`**waitForReplEnd**
A timestamp that reflects when the migration cleanup operation finished waiting for replication to process the delete operation.

repl

```
"repl" : {
   "hosts" : [
         <string>,
         <string>,
         <string>
   ],
   "setName" : <string>,
   "setVersion" : <num>,
   "ismaster" : <boolean>,
   "secondary" : <boolean>,
   "primary" : <hostname>,
   "me" : <hostname>,
   "electionId" : ObjectId(""),
   "rbid" : <num>,
   "replicationProgress" : [
         {
            "rid" : <ObjectId>,
            "optime" : { ts: <timestamp>, term: <num> },
            "host" : <hostname>,
            "memberId" : <num>
         },
         ...
   ]
}
```

repl

A document that reports on the replica set configuration. `repl` (page 503) only appear when the current host is a replica set. See `https://docs.mongodb.org/manual/replication` for more information on replication.

`repl.`**hosts**

An array of the current replica set members' hostname and port information (`"host:port"`).

`repl.`**setName**

A string with the name of the current replica set. This value reflects the `--replSet` command line argument, or `replSetName` (page 922) value in the configuration file.

`repl.`**ismaster**

A boolean that indicates whether the current node is the *primary* of the replica set.

`repl.`**secondary**

A boolean that indicates whether the current node is a *secondary* member of the replica set.

`repl.`**primary**

New in version 3.0.

The hostname and port information (`"host:port"`) of the current *primary* member of the replica set.

`repl.`**me**

New in version 3.0: The hostname and port information (`"host:port"`) for the current member of the replica set.

`repl.`**rbid**

New in version 3.0.

Rollback identifier. Used to determine if a rollback has happened for this `mongod` (page 770) instance.

`repl.`**replicationProgress**

Changed in version 3.2: Previously named `serverStatus.repl.slaves`.

New in version 3.0.

An array with one document for each 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 `serverStatus` (page 492), as in the following:

```
db.serverStatus({ "repl": 1 })
db.runCommand({ "serverStatus": 1, "repl": 1 })
```

The content of the `repl.replicationProgress` (page 503) section depends on the source of each member's replication. This section supports internal operation and is for internal and diagnostic use only.

`repl.replicationProgress[n].`**rid**

An ObjectId used as an ID for the members of the replica set. For internal use only.

`repl.replicationProgress[n].`**optime**

Information regarding the last operation from the *oplog* that the member applied, as reported from this member.

`repl.replicationProgress[n].`**host**

The name of the host in `[hostname]:[port]` format for the member of the replica set.

`repl.replicationProgress[n].`**`memberID`**

The integer identifier for this member of the replica set.

security

```
"security" : {
   "SSLServerSubjectName": <string>,
   "SSLServerHasCertificateAuthority": <boolean>,
   "SSLServerCertificateExpirationDate": <date>
},
```

`security`

New in version 3.0.

A document that reports on security configuration and details. Only appears for `mongod` (page 770) instances compiled with support for TLS/SSL.

`security.`**`SSLServerSubjectName`**

The subject name associated with the TLS/SSL certificate specified by `net.ssl.PEMKeyFile` (page 906).

`security.`**`SSLServerHasCertificateAuthority`**

A boolean that is `true` when the TLS/SSL certificate specified by `net.ssl.PEMKeyFile` (page 906) is associated with a certificate authority. `false` when the TLS/SSL certificate is self-signed.

`security.`**`SSLServerCertificateExpirationDate`**

A *date object* that represents the date when the TLS/SSL certificate specified by `net.ssl.PEMKeyFile` (page 906) expires.

storageEngine

New in version 3.0.

```
"storageEngine" : {
   "name" : <string>,
   "supportsCommittedReads" : <boolean>
},
```

`storageEngine`

A document with data about the current storage engine.

`storageEngine.`**`name`**

The name of the current storage engine.

`storageEngine.`**`supportsCommittedReads`**

New in version 3.2.

A boolean that indicates whether the storage engine supports `"majority"` `read concern`.

wiredTiger

`wiredTiger` information only appears if using the `WiredTiger` storage engine. Some of the statistics, such as `wiredTiger.LSM` (page 508), roll up for the server.

```
"wiredTiger" : {
   "uri" : "statistics:",
   "LSM" : {
         "sleep for LSM checkpoint throttle" : <num>,
         "sleep for LSM merge throttle" : <num>,
         "rows merged in an LSM tree" : <num>,
         "application work units currently queued" : <num>,
         "merge work units currently queued" : <num>,
         "tree queue hit maximum" : <num>,
         "switch work units currently queued" : <num>,
         "tree maintenance operations scheduled" : <num>,
         "tree maintenance operations discarded" : <num>,
         "tree maintenance operations executed" : <num>
   },
   "async" : {
         "number of allocation state races" : <num>,
         "number of operation slots viewed for allocation" : <num>,
         "current work queue length" : <num>,
         "number of flush calls" : <num>,
         "number of times operation allocation failed" : <num>,
         "maximum work queue length" : <num>,
         "number of times worker found no work" : <num>,
         "total allocations" : <num>,
         "total compact calls" : <num>,
         "total insert calls" : <num>,
         "total remove calls" : <num>,
         "total search calls" : <num>,
         "total update calls" : <num>
   },
   "block-manager" : {
         "mapped bytes read" : <num>,
         "bytes read" : <num>,
         "bytes written" : <num>,
         "mapped blocks read" : <num>,
         "blocks pre-loaded" : <num>,
         "blocks read" : <num>,
         "blocks written" : <num>
   },
   "cache" : {
         "tracked dirty bytes in the cache" : <num>,
         "tracked bytes belonging to internal pages in the cache" : <num>,
         "bytes currently in the cache" : <num>,
         "tracked bytes belonging to leaf pages in the cache" : <num>,
         "maximum bytes configured" : <num>,
         "tracked bytes belonging to overflow pages in the cache" : <num>,
         "bytes read into cache" : <num>,
         "bytes written from cache" : <num>,
         "pages evicted by application threads" : <num>,
         "checkpoint blocked page eviction" : <num>,
         "unmodified pages evicted" : <num>,
         "page split during eviction deepened the tree" : <num>,
         "modified pages evicted" : <num>,
         "pages selected for eviction unable to be evicted" : <num>,
         "pages evicted because they exceeded the in-memory maximum" : <num>,
         "pages evicted because they had chains of deleted items" : <num>,
         "failed eviction of pages that exceeded the in-memory maximum" : <num>,
         "hazard pointer blocked page eviction" : <num>,
         "internal pages evicted" : <num>,
         "maximum page size at eviction" : <num>,
         "eviction server candidate queue empty when topping up" : <num>,
         "eviction server candidate queue not empty when topping up" : <num>,
         "eviction server evicting pages" : <num>,
         "eviction server populating queue, but not evicting pages" : <num>,
         "eviction server unable to reach eviction goal" : <num>,
         "internal pages split during eviction" : <num>,
         "leaf pages split during eviction" : <num>,
         "pages walked for eviction" : <num>,
         "eviction worker thread evicting pages" : <num>,
         "in-memory page splits" : <num>,
         "in-memory page passed criteria to be split" : <num>,
         "lookaside table insert calls" : <num>,
         "lookaside table remove calls" : <num>,
         "percentage overhead" : <num>,
         "tracked dirty pages in the cache" : <num>,
         "pages currently held in the cache" : <num>,
         "pages read into cache" : <num>,
         "pages read into cache requiring lookaside entries" : <num>,
         "pages written from cache" : <num>,
         "page written requiring lookaside records" : <num>,
         "pages written requiring in-memory restoration" : <num>
   },
   "connection" : {
         "pthread mutex condition wait calls" : <num>,
         "files currently open" : <num>,
         "memory allocations" : <num>,
         "memory frees" : <num>,
         "memory re-allocations" : <num>,
         "total read I/Os" : <num>,
         "pthread mutex shared lock read-lock calls" : <num>,
         "pthread mutex shared lock write-lock calls" : <num>,
         "total write I/Os" : <num>
   },
   "cursor" : {
         "cursor create calls" : <num>,
         "cursor insert calls" : <num>,
         "cursor next calls" : <num>,
         "cursor prev calls" : <num>,
         "cursor remove calls" : <num>,
         "cursor reset calls" : <num>,
         "cursor restarted searches" : <num>,
         "cursor search calls" : <num>,
         "cursor search near calls" : <num>,
         "truncate calls" : <num>,
         "cursor update calls" : <num>
   },
   "data-handle" : {
         "connection data handles currently active" : <num>,
         "session dhandles swept" : <num>,
         "session sweep attempts" : <num>,
         "connection sweep dhandles closed" : <num>,
         "connection sweep candidate became referenced" : <num>,
         "connection sweep dhandles removed from hash list" : <num>,
         "connection sweep time-of-death sets" : <num>,
         "connection sweeps" : <num>
   },
   "log" : {
         "total log buffer size" : <num>,
         "log bytes of payload data" : <num>,
         "log bytes written" : <num>,
         "yields waiting for previous log file close" : <num>,
         "total size of compressed records" : <num>,
         "total in-memory size of compressed records" : <num>,
         "log records too small to compress" : <num>,
         "log records not compressed" : <num>,
         "log records compressed" : <num>,
         "log flush operations" : <num>,
         "maximum log file size" : <num>,
         "pre-allocated log files prepared" : <num>,
         "number of pre-allocated log files to create" : <num>,
         "pre-allocated log files not ready and missed" : <num>,
         "pre-allocated log files used" : <num>,
         "log release advances write LSN" : <num>,
         "records processed by log scan" : <num>,
         "log scan records requiring two reads" : <num>,
         "log scan operations" : <num>,
         "consolidated slot closures" : <num>,
         "written slots coalesced" : <num>,
         "logging bytes consolidated" : <num>,
         "consolidated slot joins" : <num>,
         "consolidated slot join races" : <num>,
         "busy returns attempting to switch slots" : <num>,
         "consolidated slot join transitions" : <num>,
         "consolidated slot unbuffered writes" : <num>,
         "log sync operations" : <num>,
         "log sync_dir operations" : <num>,
         "log server thread advances write LSN" : <num>,
         "log write operations" : <num>,
         "log files manually zero-filled" : <num>
   },
   "reconciliation" : {
         "pages deleted" : <num>,
         "fast-path pages deleted" : <num>,
         "page reconciliation calls" : <num>,
         "page reconciliation calls for eviction" : <num>,
         "split bytes currently awaiting free" : <num>,
         "split objects currently awaiting free" : <num>
   },
   "session" : {
         "open cursor count" : <num>,
         "open session count" : <num>
   },
   "thread-yield" : {
         "page acquire busy blocked" : <num>,
         "page acquire eviction blocked" : <num>,
         "page acquire locked blocked" : <num>,
         "page acquire read blocked" : <num>,
         "page acquire time sleeping (usecs)" : <num>
   },
   "transaction" : {
         "transaction begins" : <num>,
         "transaction checkpoints" : <num>,
         "transaction checkpoint generation" : <num>,
         "transaction checkpoint currently running" : <num>,
         "transaction checkpoint max time (msecs)" : <num>,
         "transaction checkpoint min time (msecs)" : <num>,
         "transaction checkpoint most recent time (msecs)" : <num>,
         "transaction checkpoint total time (msecs)" : <num>,
         "transactions committed" : <num>,
         "transaction failures due to cache overflow" : <num>,
         "transaction range of IDs currently pinned by a checkpoint" : <num>,
         "transaction range of IDs currently pinned" : <num>,
         "transaction range of IDs currently pinned by named snapshots" : <num>,
         "transactions rolled back" : <num>,
         "number of named snapshots created" : <num>,
         "number of named snapshots dropped" : <num>,
         "transaction sync calls" : <num>
   },
   "concurrentTransactions" : {
         "write" : {
            "out" : <num>,
            "available" : <num>,
            "totalTickets" : <num>
         },
         "read" : {
            "out" : <num>,
            "available" : <num>,
            "totalTickets" : <num>
         }
   }
},
```

`wiredTiger.`**`uri`**

New in version 3.0.

A string. For internal use by MongoDB.

`wiredTiger.`**`LSM`**

New in version 3.0.

A document that returns statistics on the LSM (Log-Structured Merge) tree. The values reflects the statistics for all LSM trees used in this server.

`wiredTiger.`**`async`**

New in version 3.0.

A document that returns statistics related to the asynchronous operations API. This is unused by MongoDB.

`wiredTiger.`**`block-manager`**

New in version 3.0.

A document that returns statistics on the block manager operations.

`wiredTiger.`**`cache`**

New in version 3.0: A document that returns statistics on the cache and page evictions from the cache.

The following describes some of the key `wiredTiger.cache` (page 508) statistics:

`wiredTiger.cache.`**`maximum bytes configured`**

Maximum cache size.

`wiredTiger.cache.`**`bytes currently in the cache`**

Size in byte of the data currently in cache. This value should not be greater than the `maximum bytes configured` value.

`wiredTiger.cache.`**`unmodified pages evicted`**

Main statistics for page eviction.

`wiredTiger.cache.`**`tracked dirty bytes in the cache`**

Size in bytes of the dirty data in the cache. This value should be less than the `bytes currently in the cache` value.

`wiredTiger.cache.`**`pages read into cache`**

Number of pages read into the cache. `wiredTiger.cache.pages read into cache` (page 508) with the `wiredTiger.cache.pages written from cache` can provide an overview of the I/O activity.

wiredTiger.cache.**pages written from cache**

Number of pages written from the cache. wiredTiger.cache.pages written from cache (page 508) with the wiredTiger.cache.pages read into cache can provide an overview of the I/O activity.

To adjust the size of the WiredTiger cache, see storage.wiredTiger.engineConfig.cacheSizeGB (page 919) and *--wiredTigerCacheSizeGB* (page 779). Avoid increasing the WiredTiger cache size above its default value.

wiredTiger.**connection**

New in version 3.0.

A document that returns statistics related to WiredTiger connections.

wiredTiger.**cursor**

New in version 3.0.

A document that returns statistics on WiredTiger cursor.

wiredTiger.**data-handle**

New in version 3.0.

A document that returns statistics on the data handles and sweeps.

wiredTiger.**log**

New in version 3.0.

A document that returns statistics on WiredTiger's write ahead log.

See also:

journaling-wiredTiger

wiredTiger.**reconciliation**

New in version 3.0.

A document that returns statistics on the reconciliation process.

wiredTiger.**session**

New in version 3.0.

A document that returns the open cursor count and open session count for the session.

wiredTiger.**thread-yield**

New in version 3.0.

A document that returns statistics on yields during page acquisitions.

wiredTiger.**transaction**

New in version 3.0.

A document that returns statistics on transaction checkpoints and operations.

wiredTiger.transaction.**transaction checkpoint most recent time**

Amount of time, in milliseconds, to create the most recent checkpoint. An increase in this value under stead write load may indicate saturation on the I/O subsystem.

wiredTiger.**concurrentTransactions**

New in version 3.0.

A document that returns information on the number of concurrent of read and write transactions allowed into the WiredTiger storage engine. These settings are MongoDB-specific.

To change the settings for concurrent reads and write transactions, see `wiredTigerConcurrentReadTransactions` (page 938) and `wiredTigerConcurrentWriteTransactions` (page 939).

writeBacksQueued

```
"writeBacksQueued" : <boolean>,
```

writeBacksQueued
A boolean that indicates whether there are operations from a `mongos` (page 792) instance queued for retrying. Typically, this value is false. See also *writeBacks*.

mem

```
"mem" : {
   "bits" : <int>,
   "resident" : <int>,
   "virtual" : <int>,
   "supported" : <boolean>,
   "mapped" : <int>,
   "mappedWithJournal" : <int>
},
```

mem
A document that reports on the system architecture of the `mongod` (page 770) and current memory use.

mem.**bits**
A number, either `64` or `32`, that indicates whether the MongoDB instance is compiled for 64-bit or 32-bit architecture.

mem.**resident**
The value of `mem.resident` (page 510) is roughly equivalent to the amount of RAM, in megabytes (MB), currently used by the database process. During normal use, this value tends to grow. In dedicated database servers, this number tends to approach the total amount of system memory.

mem.**virtual**
`mem.virtual` (page 510) displays the quantity, in megabytes (MB), of virtual memory used by the `mongod` (page 770) process.

With *journaling* enabled and if using MMAPv1 storage engine, the value of `mem.virtual` (page 510) is at least twice the value of `mem.mapped` (page 510). If `mem.virtual` (page 510) value is significantly larger than `mem.mapped` (page 510) (e.g. 3 or more times), this may indicate a memory leak.

mem.**supported**
A boolean that indicates whether 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 510) values may not be accessible to the database server.

mem.**mapped**
Only for the MMAPv1 storage engine.

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.

mem.**mappedWithJournal**
Only for the MMAPv1 storage engine.

The amount of mapped memory, in megabytes (MB), including the memory used for journaling. This value will always be twice the value of `mem.mapped` (page 510). This field is only included if journaling is enabled.

mem.**note**

The field mem.note (page 510) appears if mem.supported (page 510) is false.

The mem.note (page 510) field contains the text: "not all mem info support on this platform".

metrics

```
"metrics" : {
   "commands": {
         "<command>": {
            "failed": <num>,
            "total": <num>
         }
   },
   "cursor" : {
         "timedOut" : NumberLong(<num>),
         "open" : {
            "noTimeout" : NumberLong(<num>),
            "pinned" : NumberLong(<num>),
            "multiTarget" : NumberLong(<num>),
            "singleTarget" : NumberLong(<num>),
            "total" : NumberLong(<num>),
         }
   },
   "document" : {
         "deleted" : NumberLong(<num>),
         "inserted" : NumberLong(<num>),
         "returned" : NumberLong(<num>),
         "updated" : NumberLong(<num>)
   },
   "getLastError" : {
         "wtime" : {
            "num" : <num>,
            "totalMillis" : <num>
         },
         "wtimeouts" : NumberLong(<num>)
   },
   "operation" : {
         "fastmod" : NumberLong(<num>),
         "idhack" : NumberLong(<num>),
         "scanAndOrder" : NumberLong(<num>),
         "writeConflicts" : NumberLong(<num>)
   },
   "queryExecutor": {
         "scanned" : NumberLong(<num>),
         "scannedObjects" : NumberLong(<num>)
   },
   "record" : {
         "moves" : NumberLong(<num>)
   },
   "repl" : {
      "executor" : {
                  "counters" : {
                     "eventCreated" : <num>,
                     "eventWait" :  <num>,
                     "cancels" : <num>,
                     "waits" : <num>,
                     "scheduledNetCmd" : <num>,
                     "scheduledDBWork" : <num>,
                     "scheduledXclWork" : <num>,
                     "scheduledWorkAt" :  <num>,
                     "scheduledWork" :  <num>,
                     "schedulingFailures" :  <num>
                  },
                  "queues" : {
                     "networkInProgress" : <num>,
                     "dbWorkInProgress" : <num>,
                     "exclusiveInProgress" : <num>,
                     "sleepers" : <num>,
                     "ready" : <num>,
                     "free" : <num>
                  },
                  "unsignaledEvents" : <num>,
                  "eventWaiters" : <num>,
                  "shuttingDown" : <boolean>
                  "networkInterface" : "NetworkInterfaceASIO inShutdown: 0"
         },
         "apply" : {
            "batches" : {
               "num" : <num>,
               "totalMillis" : <num>
            },
            "ops" : NumberLong(<num>)
         },
         "buffer" : {
            "count" : NumberLong(<num>),
            "maxSizeBytes" : <num>,
            "sizeBytes" : NumberLong(<num>)
         },
         "network" : {
            "bytes" : NumberLong(<num>),
            "getmores" : {
               "num" : <num>,
               "totalMillis" : <num>
            },
            "ops" : NumberLong(<num>),
            "readersCreated" : NumberLong(<num>)
         },
         "oplog" : {
            "insert" : {
               "num" : <num>,
               "totalMillis" : <num>
            },
            "insertBytes" : NumberLong(<num>)
         },
         "preload" : {
            "docs" : {
               "num" : <num>,
               "totalMillis" : <num>
            },
            "indexes" : {
               "num" : <num>,
               "totalMillis" : <num>
            }
         }
   },
   "storage" : {
         "freelist" : {
            "search" : {
               "bucketExhausted" : <num>,
               "requests" : <num>,
               "scanned" : <num>
            }
         }
   },
   "ttl" : {
         "deletedDocuments" : NumberLong(<num>),
         "passes" : NumberLong(<num>)
   }
},
```

metrics
A document that returns various statistics that reflect the current use and state of a running `mongod` (page 770) instance.

metrics.**commands**
New in version 3.0.

A document that reports on the use of database commands. The fields in `metrics.commands` (page 513) are the names of *database commands* (page 303) and each value is a document that reports the total number of commands executed as well as the number of failed executions.

metrics.commands.<command>.**failed**
The number of times <command> failed on this `mongod` (page 770).

metrics.commands.<command>.**total**
The number of times <command> executed on this `mongod` (page 770).

metrics.**document**
A document that reflects document access and modification patterns. Compare these values to the data in the `opcounters` (page 500) document, which track total number of operations.

metrics.document.**deleted**
The total number of documents deleted.

metrics.document.**inserted**
The total number of documents inserted.

metrics.document.**returned**
The total number of documents returned by queries.

metrics.document.**updated**
The total number of documents updated.

metrics.**executor**
New in version 3.2.

A document that reports on various statistics for the replication executor.

metrics.**getLastError**
A document that reports on `getLastError` (page 355) use.

metrics.getLastError.**wtime**
A document that reports `getLastError` (page 355) operation counts with a `w` argument greater than `1`.

metrics.getLastError.wtime.**num**
The total number of `getLastError` (page 355) 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`.)

metrics.getLastError.wtime.**totalMillis**
The total amount of time in milliseconds that the mongod (page 770) has spent performing getLastError (page 355) 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.)

metrics.getLastError.**wtimeouts**
The number of times that *write concern* operations have timed out as a result of the wtimeout threshold to getLastError (page 355).

metrics.**operation**
A document that holds counters for several types of update and query operations that MongoDB handles using special operation types.

metrics.operation.**fastmod**
If using MMAPv1 storage engine, 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 595) operator to increment the value of a field that is not indexed.

metrics.operation.**idhack**
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.

metrics.operation.**scanAndOrder**
The total number of queries that return sorted numbers that cannot perform the sort operation using an index.

metrics.operation.**writeConflicts**
The total number of queries that encounted write conflicts.

metrics.**queryExecutor**
A document that reports data from the query execution system.

metrics.queryExecutor.**scanned**
The total number of index items scanned during queries and query-plan evaluation. This counter is the same as totalKeysExamined (page 949) in the output of explain() (page 140).

metrics.queryExecutor.**scannedObjects**
The total number of documents scanned during queries and query-plan evaluation. This counter is the same as totalDocsExamined (page 949) in the output of explain() (page 140).

metrics.**record**
A document that reports on data related to record allocation in the on-disk memory files.

metrics.record.**moves**
For https://docs.mongodb.org/manual/core/mmapv1, metrics.record.moves (page 514) 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.

metrics.**repl**
A document that reports metrics related to the replication process. metrics.repl (page 514) document appears on all mongod (page 770) instances, even those that aren't members of *replica sets*.

metrics.repl.**apply**
A document that reports on the application of operations from the replication *oplog*.

metrics.repl.apply.**batches**
metrics.repl.apply.batches (page 514) 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

`metrics.repl.apply.batches.`**`num`**
The total number of batches applied across all databases.

`metrics.repl.apply.batches.`**`totalMillis`**
The total amount of time in milliseconds the `mongod` (page 770) has spent applying operations from the oplog.

`metrics.repl.apply.`**`ops`**
The total number of *oplog* operations applied.

`metrics.repl.`**`buffer`**
MongoDB buffers oplog operations from the replication sync source buffer before applying oplog entries in a batch. `metrics.repl.buffer` (page 515) provides a way to track the oplog buffer. See *replica-set-internals-multi-threaded-replication* for more information on the oplog application process.

`metrics.repl.buffer.`**`count`**
The current number of operations in the oplog buffer.

`metrics.repl.buffer.`**`maxSizeBytes`**
The maximum size of the buffer. This value is a constant setting in the `mongod` (page 770), and is not configurable.

`metrics.repl.buffer.`**`sizeBytes`**
The current size of the contents of the oplog buffer.

`metrics.repl.`**`network`**
`metrics.repl.network` (page 515) reports network use by the replication process.

`metrics.repl.network.`**`bytes`**
`metrics.repl.network.bytes` (page 515) reports the total amount of data read from the replication sync source.

`metrics.repl.network.`**`getmores`**
`metrics.repl.network.getmores` (page 515) reports on the `getmore` operations, which are requests for additional results from the oplog *cursor* as part of the oplog replication process.

`metrics.repl.network.getmores.`**`num`**
`metrics.repl.network.getmores.num` (page 515) reports the total number of `getmore` operations, which are operations that request an additional set of operations from the replication sync source.

`metrics.repl.network.getmores.`**`totalMillis`**
`metrics.repl.network.getmores.totalMillis` (page 515) 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.

`metrics.repl.network.`**`ops`**
`metrics.repl.network.ops` (page 515) reports the total number of operations read from the replication source.

`metrics.repl.network.`**`readersCreated`**
`metrics.repl.network.readersCreated` (page 515) 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, `metrics.repl.network.readersCreated` (page 515) will increment every time MongoDB selects a new source for replication.

`metrics.repl.`**`oplog`**
A document that reports on the size and use of the *oplog* by this `mongod` (page 770) instance.

`metrics.repl.oplog.`**`insert`**
A document that reports insert operations into the *oplog*.

`metrics.repl.oplog.insert.`**`num`**
The total number of items inserted into the *oplog*.

`metrics.repl.oplog.insert.`**`totalMillis`**
The total amount of time spent for the `mongod` (page 770) to insert data into the *oplog*.

`metrics.repl.oplog.`**`insertBytes`**
The total size of documents inserted into the oplog.

`metrics.repl.`**`preload`**
`metrics.repl.preload` (page 516) 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.

`metrics.repl.preload.`**`docs`**
A document that reports on the documents loaded into memory during the *pre-fetch* stage.

`metrics.repl.preload.docs.`**`num`**
The total number of documents loaded during the *pre-fetch* stage of replication.

`metrics.repl.preload.docs.`**`totalMillis`**
The total amount of time spent loading documents as part of the *pre-fetch* stage of replication.

`metrics.repl.preload.`**`indexes`**
A 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.

`metrics.repl.preload.indexes.`**`num`**
The total number of index entries loaded by members before updating documents as part of the *pre-fetch* stage of replication.

`metrics.repl.preload.indexes.`**`totalMillis`**
The total amount of time, in milliseconds, spent loading index entries as part of the *pre-fetch* stage of replication.

`metrics.storage.freelist.search.`**`bucketExhausted`**
The number of times that `mongod` (page 770) has checked the free list without finding a suitably large record allocation.

`metrics.storage.freelist.search.`**`requests`**
The number of times `mongod` (page 770) has searched for available record allocations.

`metrics.storage.freelist.search.`**`scanned`**
The number of available record allocations `mongod` (page 770) has searched.

`metrics.`**`ttl`**
A document that reports on the operation of the resource use of the `ttl index` process.

`metrics.ttl.`**`deletedDocuments`**
The total number of documents deleted from collections with a `ttl index`.

`metrics.ttl.`**`passes`**
The number of times the background process removes documents from collections with a `ttl index`.

`metrics.`**`cursor`**
New in version 2.6.

A document that contains data regarding cursor state and use.

`metrics.cursor.`**`timedOut`**
New in version 2.6.

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.

`metrics.cursor.`**`open`**

New in version 2.6.

A document that contains data regarding open cursors.

`metrics.cursor.open.`**`noTimeout`**

New in version 2.6.

The number of open cursors with the option `DBQuery.Option.noTimeout` set to prevent timeout after a period of inactivity.

`metrics.cursor.open.`**`pinned`**

New in version 2.6.

The number of "pinned" open cursors.

`metrics.cursor.open.`**`total`**

New in version 2.6.

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.

`metrics.cursor.open.`**`singleTarget`**

New in version 3.0.

The total number of cursors that only target a *single* shard. Only `mongos` (page 792) instances report `metrics.cursor.open.singleTarget` (page 517) values.

`metrics.cursor.open.`**`multiTarget`**

New in version 3.0.

The total number of cursors that only target *more than one* shard. Only `mongos` (page 792) instances report `metrics.cursor.open.multiTarget` (page 517) values.

features

`features`

`features` (page 517) is an internal command that returns the build-level feature settings.

isSelf

`_isSelf`

`_isSelf` (page 517) is an internal command.

2.2.3 Internal Commands

Internal Commands

Name	Description
handshake (page 518)	Internal command.
_recvChunkAbort (page 518)	Internal command that supports chunk migrations in sharded clusters. Do not call directly.
_recvChunkCommit (page 518)	Internal command that supports chunk migrations in sharded clusters. Do not call directly.
_recvChunkStart (page 518)	Internal command that facilitates chunk migrations in sharded clusters.. Do not call directly.
_recvChunkStatus (page 519)	Internal command that returns data to support chunk migrations in sharded clusters. Do not call directly.
_replSetFresh	Internal command that supports replica set election operations.
mapreduce.shardedfinish (page 519)	Internal command that supports *map-reduce* in *sharded cluster* environments.
_transferMods (page 519)	Internal command that supports chunk migrations. Do not call directly.
replSetHeartbeat (page 519)	Internal command that supports replica set operations.
replSetGetRBID (page 519)	Internal command that supports replica set operations.
_migrateClone (page 519)	Internal command that supports chunk migration. Do not call directly.
replSetElect (page 519)	Internal command that supports replica set functionality.
writeBacksQueued (page 520)	Internal command that supports chunk migrations in sharded clusters.
writebacklisten (page 520)	Internal command that supports chunk migrations in sharded clusters.

handshake

handshake

handshake (page 518) is an internal command.

recvChunkAbort

_recvChunkAbort

_recvChunkAbort (page 518) is an internal command. Do not call directly.

recvChunkCommit

_recvChunkCommit

_recvChunkCommit (page 518) is an internal command. Do not call directly.

recvChunkStart

_recvChunkStart

_recvChunkStart (page 518) 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 519) is an internal command. Do not call directly.

replSetFresh

replSetFresh

replSetFresh (page 519) is an internal command that supports replica set functionality.

mapreduce.shardedfinish

mapreduce.**shardedfinish**

Provides internal functionality to support *map-reduce* in *sharded* environments.

See also:

"mapReduce (page 318)"

transferMods

_transferMods

_transferMods (page 519) is an internal command. Do not call directly.

replSetHeartbeat

replSetHeartbeat

replSetHeartbeat (page 519) is an internal command that supports replica set functionality.

replSetGetRBID

replSetGetRBID

replSetGetRBID (page 519) is an internal command that supports replica set functionality.

migrateClone

_migrateClone

_migrateClone (page 519) is an internal command. Do not call directly.

replSetElect

replSetElect

replSetElect (page 519) is an internal command that support replica set functionality.

writeBacksQueued

writeBacksQueued

writeBacksQueued (page 520) is an internal command that returns a document reporting there are operations in the write back queue for the given mongos (page 792) and information about the queues.

writeBacksQueued.**hasOpsQueued**
Boolean.

hasOpsQueued (page 520) is `true` if there are `write Back` operations queued.

writeBacksQueued.**totalOpsQueued**
Integer.

totalOpsQueued (page 520) reflects the number of operations queued.

writeBacksQueued.**queues**
Document.

queues (page 520) holds an embedded 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 520) reflects the size, by number of items, in the queues.

writeBacksQueued.queues.**minutesSinceLastCall**
The number of minutes since the last time the mongos (page 792) touched this queue.

The command document has the following prototype form:

```
{writeBacksQueued: 1}
```

To call writeBacksQueued (page 520) from the mongo (page 803) shell, use the following db.runCommand() (page 196) 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 520) is an internal command.

2.2.4 Testing Commands

Testing Commands

Name	Description
testDistLockWithSkew	Internal command. Do not call this directly.
testDistLockWithSyncCluster	Internal command. Do not call this directly.
captrunc (page 522)	Internal command. Truncates capped collections.
emptycapped (page 522)	Internal command. Removes all documents from a capped collection.
godinsert (page 522)	Internal command for testing.
_hashBSONElement (page 523)	Internal command. Computes the MD5 hash of a BSON element.
journalLatencyTest (page 524)	Tests the time required to write and perform a file system sync for a file in the journal directory.
sleep (page 524)	Internal command for testing. Forces MongoDB to block all operations.
replSetTest (page 525)	Internal command for testing replica set functionality.
forceerror (page 525)	Internal command for testing. Forces a user assertion exception.
skewClockCommand	Internal command. Do not call this command directly.
configureFailPoint (page 526)	Internal command for testing. Configures failure points.

testDistLockWithSkew

_testDistLockWithSkew

_testDistLockWithSkew (page 521) is an internal command. Do not call directly.

Note: _testDistLockWithSkew (page 521) is an internal command that is not enabled by default. _testDistLockWithSkew (page 521) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 770) command line. _testDistLockWithSkew (page 521) cannot be enabled during run-time.

testDistLockWithSyncCluster

_testDistLockWithSyncCluster

_testDistLockWithSyncCluster (page 521) is an internal command. Do not call directly.

Note: _testDistLockWithSyncCluster (page 521) is an internal command that is not enabled by default. _testDistLockWithSyncCluster (page 521) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 770) command line. _testDistLockWithSyncCluster (page 521) cannot be enabled during run-time.

captrunc

On this page

- Definition (page 522)
- Examples (page 522)

Definition

captrunc

captrunc (page 522) 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 522) 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 522) is an internal command that is not enabled by default. captrunc (page 522) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 770) command line. captrunc (page 522) cannot be enabled during run-time.

Examples The following command truncates 10 older documents from the collection `records`:

```
db.runCommand({captrunc: "records" , n: 10})
```

The following command truncates 100 documents and the 101st document:

```
db.runCommand({captrunc: "records", n: 100, inc: true})
```

emptycapped

emptycapped

The `emptycapped` command removes all documents from a capped collection. Use the following syntax:

```
{ emptycapped: "events" }
```

This command removes all records from the capped collection named `events`.

Warning: This command obtains a write lock on the affected database and will block other operations until it has completed.

Note: emptycapped (page 522) is an internal command that is not enabled by default. emptycapped (page 522) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 770) command line. emptycapped (page 522) cannot be enabled during run-time.

godinsert

godinsert

godinsert (page 522) 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 522) is an internal command that is not enabled by default. godinsert (page 522) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 770) command line. godinsert (page 522) cannot be enabled during run-time.

_hashBSONElement

On this page

- Description (page 523)
- Output (page 523)
- Example (page 523)

Description

_hashBSONElement

New in version 2.4.

An internal command that computes the MD5 hash of a BSON element. The _hashBSONElement (page 523) command returns 8 bytes from the 16 byte MD5 hash.

The _hashBSONElement (page 523) command has the following form:

```
db.runCommand({ _hashBSONElement: <key> , seed: <seed> })
```

The _hashBSONElement (page 523) 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 523) is an internal command that is not enabled by default. _hashBSONElement (page 523) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 770) command line. _hashBSONElement (page 523) cannot be enabled during run-time.

Output The _hashBSONElement (page 523) 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 770) 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 524) 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 524) 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 524) is an internal command that is not enabled by default. journalLatencyTest (page 524) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 770) command line. journalLatencyTest (page 524) cannot be enabled during run-time.

sleep

On this page

- Definition (page 524)
- Behavior (page 525)

Definition

sleep

Forces the database to block all operations. This is an internal command for testing purposes.

The sleep (page 524) command takes the following prototype form:

```
{ sleep: 1, w: <true|false>, secs: <seconds> }
```

The `sleep` (page 524) 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 770) instance in a *write lock* state for `100` seconds. Without arguments, `sleep` (page 524) causes a "read lock" for 100 seconds.

Warning: `sleep` (page 524) claims the lock specified in the `w` argument and blocks *all* operations on the `mongod` (page 770) instance for the specified amount of time.

Note: `sleep` (page 524) is an internal command that is not enabled by default. `sleep` (page 524) must be enabled by using *`--setParameter enableTestCommands=1`* on the `mongod` (page 770) command line. `sleep` (page 524) cannot be enabled during run-time.

replSetTest

`replSetTest`

`replSetTest` (page 525) is internal diagnostic command used for regression tests that supports replica set functionality.

Note: `replSetTest` (page 525) is an internal command that is not enabled by default. `replSetTest` (page 525) must be enabled by using *`--setParameter enableTestCommands=1`* on the `mongod` (page 770) command line. `replSetTest` (page 525) cannot be enabled during run-time.

forceerror

`forceerror`

The `forceerror` (page 525) command is for testing purposes only. Use `forceerror` (page 525) to force a user assertion exception. This command always returns an `ok` value of 0.

skewClockCommand

`_skewClockCommand`

`_skewClockCommand` (page 525) is an internal command. Do not call directly.

Note: `_skewClockCommand` (page 525) is an internal command that is not enabled by default. `_skewClockCommand` (page 525) must be enabled by using *`--setParameter enableTestCommands=1`* on the `mongod` (page 770) command line. `_skewClockCommand` (page 525) cannot be enabled during run-time.

configureFailPoint

On this page

- Definition (page 526)
- Example (page 526)

Definition

`configureFailPoint`

Configures a failure point that you can turn on and off while MongoDB runs. `configureFailPoint` (page 526) is an internal command for testing purposes that takes the following form:

```
{configureFailPoint: "<failure_point>", mode: <behavior> }
```

You must issue `configureFailPoint` (page 526) against the *admin database*. `configureFailPoint` (page 526) 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 Optional. 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 526) is an internal command that is not enabled by default. `configureFailPoint` (page 526) must be enabled by using `--setParameter enableTestCommands=1` on the `mongod` (page 770) command line. `configureFailPoint` (page 526) cannot be enabled during run-time.

Example

```
db.adminCommand( { configureFailPoint: "blocking_thread", mode: {times: 21} } )
```

2.2.5 Auditing Commands

System Events Auditing Commands

Name	Description
`logApplicationMessage` (page 526)	Posts a custom message to the audit log.

logApplicationMessage

`logApplicationMessage`

Note: Available only in MongoDB Enterprise[12].

The `logApplicationMessage` (page 526) 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.

[12]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

The `logApplicationMessage` (page 526) 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 527)** Query operators provide ways to locate data within the database and projection operators modify how data is presented.

***Update Operators* (page 594)** Update operators are operators that enable you to modify the data in your database or add additional data.

***Aggregation Pipeline Operators* (page 629)** Aggregation pipeline operations have a collection of operators available to define and manipulate documents in pipeline stages.

2.3.1 Query and Projection Operators

On this page

- Query Selectors (page 527)
- Projection Operators (page 588)
- Additional Resources (page 594)

Query Selectors

Comparison

Comparison Query Operators For comparison of different BSON type values, see the *specified BSON comparison order*.

Name	Description
`$eq` (page 527)	Matches values that are equal to a specified value.
`$gt` (page 529)	Matches values that are greater than a specified value.
`$gte` (page 530)	Matches values that are greater than or equal to a specified value.
`$lt` (page 530)	Matches values that are less than a specified value.
`$lte` (page 531)	Matches values that are less than or equal to a specified value.
`$ne` (page 531)	Matches all values that are not equal to a specified value.
`$in` (page 532)	Matches any of the values specified in an array.
`$nin` (page 533)	Matches none of the values specified in an array.

$eq

On this page

- Behavior (page 528)
- Examples (page 528)

$eq

New in version 3.0.

Specifies equality condition. The $eq (page 527) operator matches documents where the value of a field equals the specified value.

```
{ <field>: { $eq: <value> } }
```

The $eq (page 527) expression is equivalent to `{ field: <value> }`.

Behavior

Comparison Order For comparison of different BSON type values, see the *specified BSON comparison order*.

Match a Document Value If the specified `<value>` is a document, the order of the fields in the document matters.

Match an Array Value If the specified `<value>` is an array, MongoDB matches documents where the `<field>` matches the array exactly or the `<field>` contains an element that matches the array exactly. The order of the elements matters. For an example, see *Equals an Array Value* (page 529).

Examples The following examples query against the `inventory` collection with the following documents:

```
{ _id: 1, item: { name: "ab", code: "123" }, qty: 15, tags: [ "A", "B", "C" ] }
{ _id: 2, item: { name: "cd", code: "123" }, qty: 20, tags: [ "B" ] }
{ _id: 3, item: { name: "ij", code: "456" }, qty: 25, tags: [ "A", "B" ] }
{ _id: 4, item: { name: "xy", code: "456" }, qty: 30, tags: [ "B", "A" ] }
{ _id: 5, item: { name: "mn", code: "000" }, qty: 20, tags: [ [ "A", "B" ], "C" ] }
```

Equals a Specified Value The following example queries the `inventory` collection to select all documents where the value of the `qty` field equals `20`:

```
db.inventory.find( { qty: { $eq: 20 } } )
```

The query is equivalent to:

```
db.inventory.find( { qty: 20 } )
```

Both queries match the following documents:

```
{ _id: 2, item: { name: "cd", code: "123" }, qty: 20, tags: [ "B" ] }
{ _id: 5, item: { name: "mn", code: "000" }, qty: 20, tags: [ [ "A", "B" ], "C" ] }
```

Field in Embedded Document Equals a Value The following example queries the `inventory` collection to select all documents where the value of the `name` field in the `item` document equals `"ab"`. To specify a condition on a field in an embedded document, use the *dot notation*.

```
db.inventory.find( { "item.name": { $eq: "ab" } } )
```

The query is equivalent to:

```
db.inventory.find( { "item.name": "ab" } )
```

Both queries match the following document:

```
{ _id: 1, item: { name: "ab", code: "123" }, qty: 15, tags: [ "A", "B", "C" ] }
```

See also:

Query Embedded Documents

Array Element Equals a Value The following example queries the `inventory` collection to select all documents where the `tags` array contains an element with the value `"B"` [13]:

```
db.inventory.find( { tags: { $eq: "B" } } )
```

The query is equivalent to:

```
db.inventory.find( { tags: "B" } )
```

Both queries match the following documents:

```
{ _id: 1, item: { name: "ab", code: "123" }, qty: 15, tags: [ "A", "B", "C" ] }
{ _id: 2, item: { name: "cd", code: "123" }, qty: 20, tags: [ "B" ] }
{ _id: 3, item: { name: "ij", code: "456" }, qty: 25, tags: [ "A", "B" ] }
{ _id: 4, item: { name: "xy", code: "456" }, qty: 30, tags: [ "B", "A" ] }
```

See also:

`$elemMatch` (page 579), *Query Arrays*

Equals an Array Value The following example queries the `inventory` collection to select all documents where the `tags` array equals exactly the specified array or the `tags` array contains an element that equals the array `[ "A", "B" ]`.

```
db.inventory.find( { tags: { $eq: [ "A", "B" ] } } )
```

The query is equivalent to:

```
db.inventory.find( { tags: [ "A", "B" ] } )
```

Both queries match the following documents:

```
{ _id: 3, item: { name: "ij", code: "456" }, qty: 25, tags: [ "A", "B" ] }
{ _id: 5, item: { name: "mn", code: "000" }, qty: 20, tags: [ [ "A", "B" ], "C" ] }
```

$gt

$gt

Syntax: `{field: {$gt: value} }`

`$gt` (page 529) 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`.

[13] The query will also match documents where the value of the `tags` field is the string `"B"`.

Consider the following example that uses the `$gt` (page 529) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gt: 2 } }, { $set: { price: 9.99 } } )
```

This `update()` (page 117) 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 117) method:

```
db.inventory.update(
   { "carrier.fee": { $gt: 2 } },
   { $set: { price: 9.99 } },
   { multi: true }
)
```

See also:

`find()` (page 51), `update()` (page 117), `$set` (page 600).

$gte

`$gte`

Syntax: `{field: {$gte: value} }`

`$gte` (page 530) 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 530) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gte: 2 } }, { $set: { price: 9.99 } } )
```

This `update()` (page 117) 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 51), `update()` (page 117), `$set` (page 600).

$lt

`$lt`

Syntax: `{field: {$lt: value} }`

`$lt` (page 530) 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 530) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lt: 20 } }, { $set: { price: 9.99 } } )
```

This update() (page 117) 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 51), update() (page 117), $set (page 600).

$lte

$lte

Syntax: { field: { $lte: value} }

$lte (page 531) 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 530) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lte: 5 } }, { $set: { price: 9.99 } } )
```

This update() (page 117) 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 51), update() (page 117), $set (page 600).

$ne

$ne

Syntax: {field: {$ne: value} }

$ne (page 531) 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 531) operator with a field in an embedded document:

```
db.inventory.update( { "carrier.state": { $ne: "NY" } }, { $set: { qty: 20 } } )
```

This `update()` (page 117) 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.

The inequality operator `$ne` (page 531) is *not* very selective since it often matches a large portion of the index. As a result, in many cases, a `$ne` (page 531) query with an index may perform no better than a `$ne` (page 531) query that must scan all documents in a collection. See also *read-operations-query-selectivity*.

See also:

`find()` (page 51), `update()` (page 117), `$set` (page 600).

$in

On this page

- Examples (page 532)
- Use the `$in` Operator to Match Values (page 532)
- Use the `$in` Operator to Match Values in an Array (page 532)
- Use the `$in` Operator with a Regular Expression (page 533)

`$in`

The `$in` (page 532) operator selects the documents where the value of a field equals any value in the specified array. To specify an `$in` (page 532) 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 532) 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 532) operator that exists for earlier versions[14] 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 `$or` (page 534) operator, choose the `$in` (page 532) operator rather than the `$or` (page 534) 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 117) 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"`.

[14] https://docs.mongodb.org/v2.4/reference/operator/query/in

```
db.inventory.update(
                     { tags: { $in: ["appliances", "school"] } },
                     { $set: { sale:true } }
                   )
```

Use the `$in` Operator with a Regular Expression The `$in` (page 532) operator can specify matching values using regular expressions of the form `https://docs.mongodb.org/manual/pattern/`. You *cannot* use `$regex` (page 546) operator expressions inside an `$in` (page 532).

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 51), `update()` (page 117), `$or` (page 534), `$set` (page 600).

$nin

`$nin`

Syntax: `{ field: { $nin: [ <value1>, <value2> ... <valueN> ]} }`

`$nin` (page 533) 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 533) 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 117) 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.

The inequality operator `$nin` (page 533) is *not* very selective since it often matches a large portion of the index. As a result, in many cases, a `$nin` (page 533) query with an index may perform no better than a `$nin` (page 533) query that must scan all documents in a collection. See also *read-operations-query-selectivity*.

See also:

`find()` (page 51), `update()` (page 117), `$set` (page 600).

Logical

Logical Query Operators

Name	Description
$or (page 534)	Joins query clauses with a logical OR returns all documents that match the conditions clause.
$and (page 535)	Joins query clauses with a logical AND returns all documents that match the condition clauses.
$not (page 536)	Inverts the effect of a query expression and returns documents that do *not* match the q expression.
$nor (page 537)	Joins query clauses with a logical NOR returns all documents that fail to match both cl

$or

On this page

- Behaviors (page 534)

$or

The $or (page 534) 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 534) 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 534) 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 534) expression, all the clauses in the $or (page 534) expression must be supported by indexes. Otherwise, MongoDB will perform a collection scan.

When using indexes with $or (page 534) queries, each clause of an $or (page 534) 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.createIndex( { quantity: 1 } )
db.inventory.createIndex( { price: 1 } )
```

MongoDB can use all but the geoHaystack index to support $or (page 534) clauses.

$or and text Queries Changed in version 2.6.

If $or (page 534) includes a $text (page 549) query, all clauses in the $or (page 534) array must be supported by an index. This is because a $text (page 549) query *must* use an index, and $or (page 534) can only use indexes if all its clauses are supported by indexes. If the `$text` (page 549) query cannot use an index, the query will return an error.

`$or` and GeoSpatial Queries Changed in version 2.6.

`$or` supports *geospatial clauses* (page 559) with the following exception for the near clause (near clause includes `$nearSphere` (page 567) and `$near` (page 565)). `$or` cannot contain a near clause with any other clause.

`$or` and Sort Operations Changed in version 2.6.

When executing `$or` (page 534) queries with a `sort()` (page 156), MongoDB can now use indexes that support the `$or` (page 534) clauses. Previous versions did not use the indexes.

`$or` versus `$in` When using `$or` (page 534) with `<expressions>` that are equality checks for the value of the same field, use the `$in` (page 532) operator instead of the `$or` (page 534) 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 532) operator:

```
db.inventory.find ( { quantity: { $in: [20, 50] } } )
```

Nested `$or` Clauses You may nest `$or` (page 534) operations.

See also:

`$and` (page 535), `find()` (page 51), `sort()` (page 156), `$in` (page 532)

$and

On this page

- Examples (page 535)

`$and`

New in version 2.0.

Syntax: `{ $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }`

`$and` (page 535) performs a logical `AND` operation on an array of *two or more* expressions (e.g. `<expression1>`, `<expression2>`, etc.) and selects the documents that satisfy *all* the expressions in the array. The `$and` (page 535) 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 535) 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 534) operator more than once.

See also:

`find()` (page 51), `update()` (page 117), `$ne` (page 531), `$exists` (page 538), `$set` (page 600).

$not

`$not`

Syntax: `{ field: { $not: { <operator-expression> } } }`

`$not` (page 536) 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 531) 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 `$not` (page 536) operator only affects *other operators* and cannot check fields and documents independently. So, use the `$not` (page 536) operator for logical disjunctions and the `$ne` (page 531) operator to test the contents of fields directly.

Consider the following behaviors when using the `$not` (page 536) operator:

- The operation of the `$not` (page 536) operator is consistent with the behavior of other operators but may yield unexpected results with some data types like arrays.
- The `$not` (page 536) operator does **not** support operations with the `$regex` (page 546) operator. Instead use `https://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 `https://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 51), `update()` (page 117), `$set` (page 600), `$gt` (page 529), `$regex` (page 546), PyMongo[15], *driver*.

$nor

On this page

- Examples (page 537)

`$nor`

`$nor` (page 537) performs a logical `NOR` operation on an array of one or more query expression and selects the documents that **fail** all the query expressions in the array. The `$nor` (page 537) has the following syntax:

```
{ $nor: [ { <expression1> }, { <expression2> }, ...  { <expressionN> } ] }
```

See also:

`find()` (page 51), `update()` (page 117), `$or` (page 534), `$set` (page 600), and `$exists` (page 538).

Examples

`$nor` Query with Two Expressions Consider the following query which uses only the `$nor` (page 537) 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

[15] https://api.mongodb.org/python/current

`$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 `$nor` (page 537) expression is when the `$nor` (page 537) operator is used with the `$exists` (page 538) operator.

`$nor` and `$exists` Compare that with the following query which uses the `$nor` (page 537) operator with the `$exists` (page 538) 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`

Element

Element Query Operators

Name	Description
`$exists` (page 538)	Matches documents that have the specified field.
`$type` (page 540)	Selects documents if a field is of the specified type.

$exists

On this page

- Definition (page 538)
- Examples (page 538)

Definition

`$exists`

Syntax: `{ field: { $exists: <boolean> } }`

When `<boolean>` is true, `$exists` (page 538) 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 532) operator.

See also:

`$nin` (page 533), `$in` (page 532), 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 The following examples uses 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 }
```

`$exists: true` The following query specifies the query predicate `a: { $exists: true }`:

```
db.records.find( { a: { $exists: true } } )
```

The results consist of those documents that contain the field `a`, including the document whose field `a` contains a null value:

```
{ 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 }
```

`$exists: false` The following query specifies the query predicate `b: { $exists: false }`:

```
db.records.find( { b: { $exists: false } } )
```

The results consist of those documents that do not contain the field `b`:

```
{ a: 2, c: 5 }
{ a: 4 }
{ c: 6 }
```

$type

On this page

- Definition (page 540)
- Behavior (page 540)
- Examples (page 541)
- Querying by Array Type (page 543)
- Additional Information (page 544)

Definition

$type

$type (page 540) selects the documents where the *value* of the field is an instance of the specified *BSON* type. Querying by data type is useful when dealing with highly unstructured data where data types are not predictable.

A $type (page 540) expression has the following syntax:

Changed in version 3.2.

```
{ field: { $type: <BSON type number> | <String alias> } }
```

Available Types (page 540) describes the BSON types and their corresponding numeric and string aliases.

Behavior $type (page 540) returns documents where the BSON type of the field matches the BSON type passed to $type (page 540).

Available Types Changed in version 3.2: $type (page 540) operator accepts string aliases for the BSON types in addition to the numbers corresponding to the BSON types. Previous versions only accepted the numbers corresponding to the BSON type.

Type	Number	Alias	Notes
Double	1	"double"	
String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary data	5	"binData"	
Undefined	6	"undefined"	Deprecated.
Object id	7	"objectId"	
Boolean	8	"bool"	
Date	9	"date"	
Null	10	"null"	
Regular Expression	11	"regex"	
DBPointer	12	"dbPointer"	
JavaScript	13	"javascript"	
Symbol	14	"symbol"	
JavaScript (with scope)	15	"javascriptWithScope"	
32-bit integer	16	"int"	
Timestamp	17	"timestamp"	
64-bit integer	18	"long"	
Min key	-1	"minKey"	
Max key	127	"maxKey"	

$type (page 540) supports the number alias, which will match against the following *BSON* types:

- double
- 32-bit integer
- 64-bit integer

See *Querying by Data Type* (page 541)

Arrays When applied to arrays, $type (page 540) matches any **inner** element that is of the specified *BSON* type. For example, when matching for $type : 'array', the document will match if the field has a nested array. It will not return results where the field itself is an array.

See *Querying by Array Type* (page 543) for an example.

MinKey and MaxKey `MinKey` (page 963) and `MaxKey` (page 963) are used in comparison operations and exist primarily for internal use. For all possible *BSON* element values, `MinKey` will always be the smallest value while `MaxKey` will always be the greatest value.

Querying for `minKey` or `maxKey` with `$type` (page 540) will only return fields that match the special `MinKey` or `MaxKey` values.

Suppose that the `data` collection has two documents with `MinKey` and `MaxKey`:

```
{ "_id" : 1, x : { "$minKey" : 1 } }
{ "_id" : 2, y : { "$maxKey" : 1 } }
```

The following query will return the document with `_id:    1`:

```
db.data.find( { x: { $type: "minKey" } } )
```

The following query will return the document with `_id:    2`:

```
db.data.find( { y: { $type: "maxKey" } } )
```

Examples

Querying by Data Type The `addressBook` contains addresses and zipcodes, where `zipCode` has `string`, `int`, `double`, and `long` values:

```
db.addressBook.insertMany(
   [
      { "_id" : 1, address : "2030 Martian Way", zipCode : "90698345" },
      { "_id" : 2, address: "156 Lunar Place", zipCode : 43339374 },
      { "_id" : 3, address : "2324 Pluto Place", zipCode: NumberLong(3921412) },
      { "_id" : 4, address : "55 Saturn Ring" , zipCode : NumberInt(88602117) }
   ]
)
```

The following queries return all documents where `zipCode` is the *BSON* type `string`:

```
db.addressBook.find( { "zipCode" : { $type : 2 } } );
db.addressBook.find( { "zipCode" : { $type : "string" } } );
```

These queries return:

```
{ "_id" : 1, "address" : "2030 Martian Way", "zipCode" : "90698345" }
```

The following queries return all documents where `zipCode` is the *BSON* type `double`:

```
db.addressBook.find( { "zipCode" : { $type : 1 } } )
db.addressBook.find( { "zipCode" : { $type : "double" } } )
```

These queries return:

```
{ "_id" : 2, "address" : "156 Lunar Place", "zip" : 43339374 }
```

The following query uses the `number` alias to return documents where `zipCode` is the *BSON* type `double`, `int`, or `long`:

```
db.addressBook.find( { "zipCode" : { $type : "number" } } )
```

These queries return:

```
{ "_id" : 2, address : "156 Lunar Place", zipCode : 43339374 }
{ "_id" : 3, address : "2324 Pluto Place", zipCode: NumberLong(3921412) }
{ "_id" : 4, address : "55 Saturn Ring" , zipCode : 88602117 }
```

Querying by MinKey and MaxKey The `restaurants` collection uses `minKey` for any grade that is a failing grade:

```
{
   "_id": 1,
   "address": {
      "building": "230",
      "coord": [ -73.996089, 40.675018 ],
      "street": "Huntington St",
      "zipcode": "11231"
   },
   "borough": "Brooklyn",
   "cuisine": "Bakery",
   "grades": [
      { "date": { "$date": 1393804800000 }, "grade": "C", "score": 15 },
      { "date": { "$date": 1378857600000 }, "grade": "C", "score": 16 },
      { "date": { "$date": 1358985600000 }, "grade": { "$minKey" : 1 }, "score": 30 },
      { "date": { "$date": 1322006400000 }, "grade": "C", "score": 15 }
   ],
   "name": "Dirty Dan's Donuts",
   "restaurant_id": "30075445"
}
```

And `maxKey` for any grade that is the highest passing grade:

```
{
   "_id": 2,
   "address": {
      "building": "1166",
      "coord": [ -73.955184, 40.738589 ],
      "street": "Manhattan Ave",
      "zipcode": "11222"
   },
   "borough": "Brooklyn",
   "cuisine": "Bakery",
   "grades": [
      { "date": { "$date": 1393804800000 }, "grade": { "$maxKey" : 1 }, "score": 2 },
      { "date": { "$date": 1378857600000 }, "grade": "B", "score": 6 },
      { "date": { "$date": 1358985600000 }, "grade": { "$maxKey" : 1 }, "score": 3 },
      { "date": { "$date": 1322006400000 }, "grade": "B", "score": 5 }
   ],
   "name": "Dainty Daisey's Donuts",
   "restaurant_id": "30075449"
}
```

The following query returns any restaurant whose `grades.grade` field contains `minKey`:

```
db.restaurant.find(
   { "grades.grade" : { $type : "minKey" } }
)
```

This returns

```
{
   "_id": 1,
   "address": {
      "building": "230",
      "coord": [ -73.996089, 40.675018 ],
      "street": "Huntington St",
      "zipcode": "11231"
   },
   "borough": "Brooklyn",
   "cuisine": "Bakery",
   "grades": [
      { "date": { "$date": 1393804800000 }, "grade": "C", "score": 15 },
      { "date": { "$date": 1378857600000 }, "grade": "C", "score": 16 },
      { "date": { "$date": 1358985600000 }, "grade": { "$minKey" : 1 }, "score": 30 },
      { "date": { "$date": 1322006400000 }, "grade": "C", "score": 15 }
   ],
   "name": "Dirty Dan's Donuts",
   "restaurant_id": "30075445"
}
```

The following query returns any restaurant whose `grades.grade` field contains `maxKey`:

```
db.restaurant.find(
   { "grades.grade" : { $type : "maxKey" } }
)
```

This returns

```
{
   "_id": 2,
   "address": {
      "building": "1166",
      "coord": [ -73.955184, 40.738589 ],
      "street": "Manhattan Ave",
      "zipcode": "11222"
   },
   "borough": "Brooklyn",
   "cuisine": "Bakery",
   "grades": [
      { "date": { "$date": 1393804800000 }, "grade": { "$maxKey" : 1 }, "score": 2 },
      { "date": { "$date": 1378857600000 }, "grade": "B", "score": 6 },
      { "date": { "$date": 1358985600000 }, "grade": { "$maxKey" : 1 }, "score": 3 },
      { "date": { "$date": 1322006400000 }, "grade": "B", "score": 5 }
   ],
   "name": "Dainty Daisey's Donuts",
   "restaurant_id": "30075449"
}
```

Querying by Array Type The `SensorReading` collection contains the following documents:

```
{
   "_id": 1,
   "readings": [
      25,
      23,
      [ "Warn: High Temp!", 55 ],
      [ "ERROR: SYSTEM SHUTDOWN!", 66 ]
   ]
},
{
   "_id": 2,
   "readings": [
      25,
      25,
      24,
      23
   ]
}
```

The following query returns any document where `readings` has an element of *BSON* type `array`:

```
db.SensorReading.find( "readings" : { $type: "array" } )
```

This returns

```
{
   "_id": 1,
   "readings": [
      25,
      23,
      [ "Warn: High Temp!", 55 ],
      [ "ERROR: SYSTEM SHUTDOWN!", 66 ]
   ]
}
```

Since the `readings` field has at least one array as an element, the `$type` (page 540) will return the first document.

Additional Information `find()` (page 51), `BSON Types`.

Evaluation

Evaluation Query Operators

Name	Description
`$mod` (page 544)	Performs a modulo operation on the value of a field and selects documents wit result.
`$regex` (page 546)	Selects documents where values match a specified regular expression.
`$text` (page 549)	Performs text search.
`$where` (page 558)	Matches documents that satisfy a JavaScript expression.

$mod

On this page

- Examples (page 545)

`$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 544) expression, use the following syntax:

```
{ field: { $mod: [ divisor, remainder ] } }
```

Changed in version 2.6: The `$mod` (page 544) operator errors when passed an array with fewer or more elements. In previous versions, if passed an array with one element, the `$mod` (page 544) operator uses `0` as the remainder value, and if passed an array with more than two elements, the `$mod` (page 544) ignores all but the first two elements. Previous versions do return an error when passed an empty array. See *Not Enough Elements Error* (page 545) and *Too Many Elements Error* (page 546) 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 544) operator errors when passed an array with fewer than two elements.

Array with Single Element The following operation incorrectly passes the `$mod` (page 544) 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 544) operator uses the specified element as the divisor and `0` as the remainder value.

Empty Array The following operation incorrectly passes the `$mod` (page 544) 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 544) operator errors when passed an array with more than two elements.

For example, the following operation attempts to use the $mod (page 544) 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 544) ignores all but the first two elements.

$regex

On this page

- Definition (page 546)
- Behavior (page 547)
- Examples (page 548)

Definition

$regex

Provides regular expression capabilities for pattern matching *strings* in queries. MongoDB uses Perl compatible regular expressions (i.e. "PCRE") version 8.36 with UTF-8 support.

To use $regex (page 546), 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. `https://docs.mongodb.org/manual/pattern/`) to specify regular expressions:

```
{ <field>: /pattern/<options> }
```

For restrictions on particular syntax use, see *$regex vs. /pattern/ Syntax* (page 547).

$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 548).	
`m`	For patterns that include anchors (i.e. `^` for the start, `$` for the end), match at the beginning or end of each line for strings with multiline values. Without this option, these anchors match at beginning or end of the string. For an example, see *Multiline Match for Lines Starting with Specified Pattern* (page 548). If the pattern contains no anchors or if the string value has no newline characters (e.g. `\n`), the `m` option has no effect.	
`x`	"Extended" capability to ignore all white space characters in the `$regex` (page 546) 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 549).	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. `https://docs.mongodb.org/manual/pattern/`). For example:

```
{ name: { $in: [ /^acme/i, /^ack/ ] } }
```

You *cannot* use `$regex` (page 546) operator expressions inside an `$in` (page 532).

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 546) 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 `$regex` (page 546) operator expression *with* the `$options` (page 546) operator. For example, to specify the `i` and the `s` options, you must use `$options` (page 546) 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 `$regex` (page 546) operator expression with the pattern as a string. For example, to use `(?i)` in the pattern to turn case-insensitivity on for the remaining pattern and `(?-i)` to turn case-sensitivity on for the remaining pattern, you must use the `$regex` (page 546) 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 `https://docs.mongodb.org/manual/^abc.*/` will be optimized by matching only against the values from the index that start with `abc`.

Additionally, while `https://docs.mongodb.org/manual/^a/`, `https://docs.mongodb.org/manual/^a.*/`, and `https://docs.mongodb.org/manual/^a.*$/` match equivalent strings, they have different performance characteristics. All of these expressions use an index if an appropriate index exists; however, `https://docs.mongodb.org/manual/^a.*/`, and `https://docs.mongodb.org/manual/^a.*$/` are slower. `https://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 546) 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 546) 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

On this page

- Definition (page 549)
- Behavior (page 550)
- Examples (page 553)

Definition

$text

$text (page 549) performs a text search on the content of the fields indexed with a text index. A $text (page 549) expression has the following syntax:

Changed in version 3.2.

```
{
  $text:
    {
      $search: <string>,
      $language: <string>,
      $caseSensitive: <boolean>,
      $diacriticSensitive: <boolean>
    }
}
```

The `$text` (page 549) 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 550) for more information on the field.

field string $language Optional. 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.

field boolean $caseSensitive Optional. A boolean flag to enable or disable case sensitive search. Defaults to `false`; i.e. the search defers to the case insensitivity of the `text` index.

For more information, see *Case Insensitivity* (page 552).

New in version 3.2.

field boolean $diacriticSensitive Optional. A boolean flag to enable or disable diacritic sensitive search against `version 3 text indexes`. Defaults to `false`; i.e. the search defers to the diacritic insensitivity of the `text` index.

Text searches against earlier versions of the text index are inherently diacritic sensitive and cannot be diacritic insensitive. As such, the `$diacriticSensitive` option has no effect with earlier versions of the text index.

For more information, see *Diacritic Insensitivity* (page 552).

New in version 3.2.

The `$text` (page 549) operator, by default, does *not* return results sorted in terms of the results' scores. For more information on sorting by the text search scores, see the *Text Score* (page 553) documentation.

Behavior

Restrictions

- A query can specify, at most, one `$text` (page 549) expression.
- The `$text` (page 549) query can not appear in `$nor` (page 537) expressions.
- To use a `$text` (page 549) query in an `$or` (page 534) expression, all clauses in the `$or` (page 534) array must be indexed.
- You cannot use `hint()` (page 142) if the query includes a `$text` (page 549) query expression.
- You cannot specify `$natural` sort order if the query includes a `$text` (page 549) expression.
- You cannot combine the `$text` (page 549) expression, which requires a special *text index*, with a query operator that requires a different type of special index. For example you cannot combine `$text` (page 549) expression with the `$near` (page 565) operator.

If using the `$text` (page 549) operator in aggregation, the following restrictions also apply.

- The `$match` (page 635) stage that includes a `$text` (page 549) must be the **first** stage in the pipeline.
- A `text` operator can only occur once in the stage.
- The `text` operator expression cannot appear in `$or` (page 661) or `$not` (page 662) expressions.
- The text search, by default, does not return the matching documents in order of matching scores. Use the `$meta` (page 701) aggregation expression in the `$sort` (page 649) stage.

`$search` Field In the `$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-minus (`-`) 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 549) operator searches for the phrase `"ssl certificate"` **and** (`"authority"` **or** `"key"` **or** `"ssl"` **or** `"certificate"`).

Negations Prefixing a word with a hyphen-minus (`-`) 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 549) operator treats the hyphen-minus (`-`) as a delimiter.

The `$text` (page 549) operator adds all negations to the query with the logical `AND` operator.

Match Operation

Stop Words The `$text` (page 549) operator ignores language-specific stop words, such as `the` and `and` in English.

Stemmed Words For case insensitive and diacritic insensitive text searches, the `$text` (page 549) 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.

Case Sensitive Search and Stemmed Words For *case sensitive* (page 552) search (i.e. `$caseSensitive: true`), if the suffix stem contains uppercase letters, the `$text` (page 549) operator matches on the exact word.

Diacritic Sensitive Search and Stemmed Words For *diacritic sensitive* (page 552) search (i.e. `$diacriticSensitive: true`), if the suffix stem contains the diacritic mark or marks, the `$text` (page 549) operator matches on the exact word.

Case Insensitivity Changed in version 3.2.

The `$text` (page 549) operator defaults to the case insensitivity of the `text` index:

- The *version 3 text index* is case insensitive for Latin characters with or without diacritics and characters from non-Latin alphabets, such as the Cyrillic alphabet. See *text* index for details.
- Earlier versions of the `text` index are case insensitive for Latin characters without diacritic marks; i.e. for `[A-z]`.

`$caseSensitive` Option To support case sensitive search where the `text` index is case insensitive, specify `$caseSensitive: true`.

Case Sensitive Search Process When performing a case sensitive search (`$caseSensitive: true`) where the `text` index is case insensitive, the `$text` (page 549) operator:

- First searches the `text` index for case insensitive and diacritic matches.
- Then, to return just the documents that match the case of the search terms, the `$text` (page 549) query operation includes an additional stage to filter out the documents that do not match the specified case.

For case sensitive search (i.e. `$caseSensitive: true`), if the suffix stem contains uppercase letters, the `$text` (page 549) operator matches on the exact word.

Specifying `$caseSensitive: true` may impact performance.

See also:

Stemmed Words (page 551)

Diacritic Insensitivity Changed in version 3.2.

The `$text` (page 549) operator defaults to the diacritic insensitivity of the `text` index:

- The *version 3 text index* is diacritic insensitive. That is, the index does not distinguish between characters that contain diacritical marks and their non-marked counterpart, such as `é`, `ê`, and `e`.
- Earlier versions of the `text` index are diacritic sensitive.

`$diacriticSensitive` Option To support diacritic sensitive text search against the version 3 `text` index, specify `$diacriticSensitive: true`.

Text searches against earlier versions of the `text` index are inherently diacritic sensitive and cannot be diacritic insensitive. As such, the `$diacriticSensitive` option for the `$text` (page 549) operator has no effect with earlier versions of the `text` index.

Diacritic Sensitive Search Process To perform a diacritic sensitive text search (`$diacriticSensitive: true`) against a version 3 `text` index, the `$text` (page 549) operator:

- First searches the `text` index, which is diacritic insensitive.
- Then, to return just the documents that match the diacritic marked characters of the search terms, the `$text` (page 549) query operation includes an additional stage to filter out the documents that do not match.

Specifying `$diacriticSensitive: true` may impact performance.

To perform a diacritic sensitive search against an earlier version of the `text` index, the `$text` (page 549) operator searches the `text` index which is diacritic sensitive.

For diacritic sensitive search, if the suffix stem contains the diacritic mark or marks, the `$text` (page 549) operator matches on the exact word.

See also:

Stemmed Words (page 551)

Text Score The `$text` (page 549) 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 156) method specification as well as part of the projection expression. The `{ $meta: "textScore" }` expression provides information on the processing of the `$text` (page 549) operation. See `$meta` (page 592) projection operator for details on accessing the score for projection or sort.

Examples The following examples assume a collection `articles` that has a `version 3 text` index on the field `subject`:

```
db.articles.createIndex( { subject: "text" } )
```

Populate the collection with the following documents:

```
db.articles.insert(
   [
     { _id: 1, subject: "coffee", author: "xyz", views: 50 },
     { _id: 2, subject: "Coffee Shopping", author: "efg", views: 5 },
     { _id: 3, subject: "Baking a cake", author: "abc", views: 90  },
     { _id: 4, subject: "baking", author: "xyz", views: 100 },
     { _id: 5, subject: "Café Con Leche", author: "abc", views: 200 },
     { _id: 6, subject: "", author: "jkl", views: 80 },
     { _id: 7, subject: "coffee and cream", author: "efg", views: 10 },
     { _id: 8, subject: "Cafe con Leche", author: "xyz", views: 10 }
   ]
)
```

Search for a Single Word The following query specifies a `$search` string of `coffee`:

```
db.articles.find( { $text: { $search: "coffee" } } )
```

This query returns the documents that contain the term `coffee` in the indexed `subject` field, or more precisely, the stemmed version of the word:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
```

See also:

Case Insensitivity (page 552), *Stemmed Words* (page 551)

Match Any of the Search Terms If the search string is a space-delimited string, `$text` (page 549) operator performs a logical `OR` search on each term and returns documents that contains any of the terms.

The following query 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, or more precisely, the stemmed version of these words:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
{ "_id" : 3, "subject" : "Baking a cake", "author" : "abc", "views" : 90 }
{ "_id" : 4, "subject" : "baking", "author" : "xyz", "views" : 100 }
```

See also:

Case Insensitivity (page 552), *Stemmed Words* (page 551)

Search for a Phrase To match the exact phrase as a single term, escape the quotes.

The following query searches for the phrase `coffee shop`:

```
db.articles.find( { $text: { $search: "\"coffee shop\"" } } )
```

This query returns documents that contain the phrase `coffee shop`:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
```

See also:

Phrases (page 551)

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 549) operator will exclude the documents that contain those terms from the results.

The following example searches for documents that contain the words `coffee` but do **not** contain the term `shop`, or more precisely the stemmed version of the words:

```
db.articles.find( { $text: { $search: "coffee -shop" } } )
```

The query returns the following documents:

```
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
```

See also:

Negations (page 551), *Stemmed Words* (page 551)

Search a Different Language Use the optional `$language` field in the `$text` (page 549) expression to specify a language that determines the list of stop words and the rules for the stemmer and tokenizer for the search string.

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`, i.e. Spanish, as the language that determines the tokenization, stemming, and stop words:

```
db.articles.find(
   { $text: { $search: "leche", $language: "es" } }
)
```

The query returns the following documents:

```
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc", "views" : 200 }
{ "_id" : 8, "subject" : "Cafe con Leche", "author" : "xyz", "views" : 10 }
```

The `$text` (page 549) expression can also accept the language by name, `spanish`. See *text-search-languages* for the supported languages.

See also:

Case Insensitivity (page 552)

Case and Diacritic Insensitive Search Changed in version 3.2.

The `$text` (page 549) operator defers to the case and diacritic insensitivity of the `text` index. The version 3 `text` index is diacritic insensitive and expands its case insensitivity to include the Cyrillic alphabet as well as characters with diacritics. For details, see *text Index Case Insensitivity* and *text Index Diacritic Insensitivity*.

The following query performs a case and diacritic insensitive text search for the term `CAFÉS`:

```
db.articles.find( { $text: { $search: "CAFÉS" } } )
```

Using the version 3 `text` index, the query matches the following documents.

```
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc", "views" : 200 }
{ "_id" : 8, "subject" : "Cafe con Leche", "author" : "xyz", "views" : 10 }
```

With the previous versions of the `text` index, the query would not match any document.

See also:

Case Insensitivity (page 552), *Diacritic Insensitivity* (page 552), *Stemmed Words* (page 551), `https://docs.mongodb.org/manual/core/index-text`

Perform Case Sensitive Search Changed in version 3.2.

To enable case sensitive search, specify `$caseSensitive: true`. Specifying `$caseSensitive: true` may impact performance.

Case Sensitive Search for a Term The following query performs a case sensitive search for the term `Coffee`:

```
db.articles.find( { $text: { $search: "Coffee", $caseSensitive: true } } )
```

The search matches just the document:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
```

See also:

Case Insensitivity (page 552), *Case Sensitive Search and Stemmed Words* (page 551)

Case Sensitive Search for a Phrase The following query performs a case sensitive search for the phrase `Café Con Leche`:

```
db.articles.find( {
   $text: { $search: "\"Café Con Leche\"", $caseSensitive: true }
} )
```

The search matches just the document:

```
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc", "views" : 200 }
```

See also:

Case Sensitive Search and Stemmed Words (page 551), *Case Insensitivity* (page 552)

Case Sensitivity with Negated Term A *negated* term is a term that is prefixed by a minus sign `-`. If you negate a term, the `$text` (page 549) operator will exclude the documents that contain those terms from the results. You can also specify case sensitivity for negated terms.

The following example performs a case sensitive search for documents that contain the word `Coffee` but do **not** contain the lower-case term `shop`, or more precisely the stemmed version of the words:

```
db.articles.find( { $text: { $search: "Coffee -shop", $caseSensitive: true } } )
```

The query matches the following document:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg" }
```

See also:

Case Sensitive Search and Stemmed Words (page 551), *Negations* (page 551)

Diacritic Sensitive Search Changed in version 3.2.

To enable diacritic sensitive search against a version 3 `text` index, specify `$diacriticSensitive:  true`. Specifying `$diacriticSensitive:  true` may impact performance.

Diacritic Sensitive Search for a Term The following query performs a diacritic sensitive text search on the term `CAFÉ`, or more precisely the stemmed version of the word:

```
db.articles.find( { $text: { $search: "CAFÉ", $diacriticSensitive: true } } )
```

The query only matches the following document:

```
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc" }
```

See also:

Diacritic Sensitive Search and Stemmed Words (page 551), *Diacritic Insensitivity* (page 552), *Case Insensitivity* (page 552)

Diacritic Sensitivity with Negated Term The `$diacriticSensitive` option applies also to negated terms. A negated term is a term that is prefixed by a minus sign `-`. If you negate a term, the `$text` (page 549) operator will exclude the documents that contain those terms from the results.

The following query performs a diacritic sensitive text search for document that contains the term `leches` but not the term `cafés`, or more precisely the stemmed version of the words:

```
db.articles.find(
  { $text: { $search: "leches -cafés", $diacriticSensitive: true } }
)
```

The query matches the following document:

```
{ "_id" : 8, "subject" : "Cafe con Leche", "author" : "xyz" }
```

See also:

Diacritic Sensitive Search and Stemmed Words (page 551), *Diacritic Insensitivity* (page 552), *Case Insensitivity* (page 552)

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" } }
)
```

The returned document includes an *additional* field `score` that contains the document's score associated with the text search. [16]

See also:

Text Score (page 553)

Sort by Text Search Score To sort by the text score, include the **same** `$meta` (page 592) expression in **both** the projection document and the sort expression. [1] The following query searches for the term `coffee` and sorts the results by the descending score:

```
db.articles.find(
   { $text: { $search: "coffee" } },
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } )
```

The query returns the matching documents sorted by descending score.

See also:

Text Score (page 553)

Return Top 2 Matching Documents Use the `limit()` (page 144) method in conjunction with a `sort()` (page 156) to return the top `n` matching documents.

The following query searches for the term `coffee` and sorts the results by the descending score, limiting the results to the top two matching documents:

```
db.articles.find(
   { $text: { $search: "coffee" } },
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } ).limit(2)
```

See also:

Text Score (page 553)

Text Search with Additional Query and Sort Expressions The following query searches for documents where the `author` equals `"xyz"` and the indexed field `subject` contains the terms `coffee` or `bake`. The operation also specifies a sort order of ascending `_id`, then descending text search score:

```
db.articles.find(
   { author: "xyz", $text: { $search: "coffee bake" } },
   { score: { $meta: "textScore" } }
).sort( { date: 1, score: { $meta: "textScore" } } )
```

See also:

`https://docs.mongodb.org/manual/tutorial/text-search-in-aggregation`

[16] The behavior and requirements of the `$meta` (page 592) operator differs from that of the `$meta` (page 701) aggregation operator. See the `$meta` (page 701) aggregation operator for details.

$where

On this page

• Behavior (page 558)

$where

Use the $where (page 558) operator to pass either a string containing a JavaScript expression or a full JavaScript function to the query system. The $where (page 558) 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

Restrictions Changed in version 2.4.

In MongoDB 2.4, map-reduce operations (page 318), the group (page 313) command, and $where (page 558) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the mongo (page 803) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your map-reduce operations (page 318), group (page 313) commands, or $where (page 558) 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 318), the group (page 313) command, and $where (page 558) operator expressions in MongoDB 2.4:

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

elemMatch Changed in version 2.6.

Only apply the $where (page 558) query operator to top-level documents. The $where (page 558) query operator will not work inside a nested document, for instance, in an $elemMatch (page 579) query.

Considerations

- Do not use global variables.
- `$where` (page 558) evaluates JavaScript and cannot take advantage of indexes. Therefore, query performance improves when you express your query using the standard MongoDB operators (e.g., `$gt` (page 529), `$in` (page 532)).
- In general, you should use `$where` (page 558) only when you can't express your query using another operator. If you must use `$where` (page 558), try to include at least one other standard query operator to filter the result set. Using `$where` (page 558) alone requires a table scan.

Using normal non-`$where` (page 558) query statements provides the following performance advantages:

- MongoDB will evaluate non-`$where` (page 558) components of query before `$where` (page 558) statements. If the non-`$where` (page 558) statements match no documents, MongoDB will not perform any query evaluation using `$where` (page 558).
- The non-`$where` (page 558) 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 558) 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 558) 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

On this page

- Operators (page 559)

Operators

Query Selectors

Name	Description
`$geoWithin` (page 560)	Selects geometries within a bounding *GeoJSON geometry*. The `2dsphere` and `2d` indexes support `$geoWithin` (page 560).
`$geoIntersects` (page 562)	Selects geometries that intersect with a *GeoJSON* geometry. The `2dsphere` index supports `$geoIntersects` (page 562).
`$near` (page 565)	Returns geospatial objects in proximity to a point. Requires a geospatial index. The `2dsphere` and `2d` indexes support `$near` (page 565).
`$nearSphere` (page 567)	Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The `2dsphere` and `2d` indexes support `$nearSphere` (page 567).

$geoWithin

On this page

- Definition (page 560)
- Behavior (page 561)
- Examples (page 561)

Definition

`$geoWithin`

New in version 2.4: `$geoWithin` (page 560) replaces `$within` (page 562) 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 `$geoWithin` (page 560) operator uses the `$geometry` (page 569) 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 560) 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 3.0: 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 569) 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 `$geoWithin` (page 560) to support queries with a single-ringed GeoJSON *polygon* whose area is greater than or equal to a single hemisphere. If the specified polygon is smaller than a single hemisphere, the behavior of `$geoWithin` (page 560) with the MongoDB CRS is the same as with the default CRS. See also *"Big" Polygons* (page 561).

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 573),

• `$polygon` (page 574),

• `$center` (page 572) (defines a circle), and

• `$centerSphere` (page 572) (defines a circle on a sphere).

Important: If you use longitude and latitude, specify coordinates in order of `longitude, latitude`.

Behavior

Geospatial Indexes `$geoWithin` (page 560) does not require a geospatial index. However, a geospatial index will improve query performance. Both `2dsphere` and `2d` geospatial indexes support `$geoWithin` (page 560).

Unsorted Results The `$geoWithin` (page 560) operator does not return sorted results. As such, MongoDB can return `$geoWithin` (page 560) queries more quickly than geospatial `$near` (page 565) or `$nearSphere` (page 567) queries, which sort results.

"Big" Polygons For `$geoWithin` (page 560), 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 $geometry` (page 569) expression; otherwise, `$geoWithin` (page 560) queries for the complementary geometry. For all other GeoJSON polygons with areas greater than a hemisphere, `$geoWithin` (page 560) 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 562).

Within a "Big" Polygon To query with a single-ringed GeoJSON polygon whose area is greater than a single hemisphere, the `$geometry` (page 569) 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: `$geoWithin` (page 560) replaces `$within` (page 562) in MongoDB 2.4.

$geoIntersects

On this page

- Definition (page 562)
- Behavior (page 563)
- Examples (page 564)

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 `$geoIntersects` (page 562) operator uses the `$geometry` (page 569) 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 562) 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 3.0: 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 569) 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 `$geoIntersects` (page 562) to support queries with a single-ringed GeoJSON *polygon* whose area is greater than or equal to a single hemisphere. If the specified polygon is smaller than a single hemisphere, the behavior of `$geoIntersects` (page 562) with the MongoDB CRS is the same as with the default CRS. See also *"Big" Polygons* (page 563).

Important: If you use longitude and latitude, specify coordinates in order of: **longitude, latitude.**

Behavior

Geospatial Indexes `$geoIntersects` (page 562) uses spherical geometry. `$geoIntersects` (page 562) does not require a geospatial index. However, a geospatial index will improve query performance. Only the `2dsphere` geospatial index supports `$geoIntersects` (page 562).

"Big" Polygons For `$geoIntersects` (page 562), if you specify a single-ringed polygon that has an area greater than a single hemisphere, include `the custom MongoDB coordinate reference system in` the `$geometry` (page 569) expression; otherwise, `$geoIntersects` (page 562) queries for the complementary geometry. For all other GeoJSON polygons with areas greater than a hemisphere, `$geoIntersects` (page 562) queries for the complementary geometry.

Examples

Intersects a Polygon The following example uses `$geoIntersects` (page 562) 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 564).

Intersects a "Big" Polygon To query with a single-ringed GeoJSON polygon whose area is greater than a single hemisphere, the `$geometry` (page 569) 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" }
             }
          }
       }
     }
   }
)
```

$near

On this page

- Definition (page 565)
- Behavior (page 565)
- Examples (page 566)

Definition

`$near`

Specifies a point for which a *geospatial* query returns the documents from nearest to farthest. The `$near` (page 565) operator can specify either a *GeoJSON* point or legacy coordinate point.

`$near` (page 565) 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 565) 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* `$minDistance` (page 570) and `$maxDistance` (page 571) specifications to limit the `$near` (page 565) results by distance in *meters*:

- `$minDistance` (page 570) limits the results to those documents that are *at least* the specified distance from the center point. `$minDistance` (page 570) is only available for use with `2dsphere` index.

  New in version 2.6.

- `$maxDistance` (page 571) 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 565) 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* `$maxDistance` (page 571) specification to limit the `$near` (page 565) results by distance in *radians*. `$maxDistance` (page 571) limits the results to those documents that are *at most* the specified distance from the center point.

Behavior

Special Indexes Restriction You cannot combine the `$near` (page 565) operator, which requires a special *geospatial index*, with a query operator or command that requires another special index. For example you cannot combine `$near` (page 565) with the `$text` (page 549) query.

Sharded Collections Restrictions For sharded collections, queries using `$near` (page 565) are not supported. You can instead use either the `geoNear` (page 327) command or the `$geoNear` (page 651) aggregation stage.

Sort Operation `$near` (page 565) sorts documents by distance. If you also include a `sort()` (page 156) for the query, `sort()` (page 156) re-orders the matching documents, effectively overriding the sort operation already performed by `$near` (page 565). When using `sort()` (page 156) with geospatial queries, consider using `$geoWithin` (page 560) operator, which does not sort documents, instead of `$near` (page 565).

See also:

2d Indexes and Geospatial Near Queries (page 1051)

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 } }
)
```

$nearSphere

On this page

- Definition (page 567)
- Behavior (page 568)
- Examples (page 568)

Definition

`$nearSphere`

Specifies a point for which a *geospatial* query returns the documents from nearest to farthest. MongoDB calculates distances for `$nearSphere` (page 567) using spherical geometry.

`$nearSphere` (page 567) *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 567) 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* `$minDistance` (page 570) is available only if the query uses the `2dsphere` index. `$minDistance` (page 570) 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 571) 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* `$minDistance` (page 570) is available only if the query uses the `2dsphere` index. `$minDistance` (page 570) 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 571) 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 1051)

Behavior

Special Indexes Restriction You cannot combine the `$nearSphere` (page 567) operator, which requires a special *geospatial index*, with a query operator or command that requires another special index. For example you cannot combine `$nearSphere` (page 567) with the `$text` (page 549) query.

Sharded Collections Restrictions For sharded collections, queries using `$nearSphere` (page 567) are not supported. You can instead use either the `geoNear` (page 327) command or the `$geoNear` (page 651) aggregation stage.

Sort Operation `$nearSphere` (page 567) sorts documents by distance. If you also include a `sort()` (page 156) for the query, `sort()` (page 156) re-orders the matching documents, effectively overriding the sort operation already performed by `$nearSphere` (page 567). When using `sort()` (page 156) with geospatial queries, consider using `$geoWithin` (page 560) operator, which does not sort documents, instead of `$nearSphere` (page 567).

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 570) 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 } }
)
```

Geometry Specifiers

Name	Description
`$geometry` (page 569)	Specifies a geometry in *GeoJSON* format to geospatial query operators.
`$minDistance` (page 570)	Specifies a minimum distance to limit the results of `$near` (page 565) and `$nearSphe` (page 567) queries. For use with `2dsphere` index only.
`$maxDistance` (page 571)	Specifies a maximum distance to limit the results of `$near` (page 565) and `$nearSphe` (page 567) queries. The `2dsphere` and `2d` indexes support `$centerSphere` (page 57
`$center` (page 572)	Specifies a circle using legacy coordinate pairs to `$geoWithin` (page 560) queries wher using planar geometry. The `2d` index supports `$center` (page 572).
`$centerSphere` (page 572)	Specifies a circle using either legacy coordinate pairs or *GeoJSON* format for `$geoWith` (page 560) queries when using spherical geometry. The `2dsphere` and `2d` indexes supp `$centerSphere` (page 572).
`$box` (page 573)	Specifies a rectangular box using legacy coordinate pairs for `$geoWithin` (page 560) queries. The `2d` index supports `$box` (page 573).
`$polygon` (page 574)	Specifies a polygon to using legacy coordinate pairs for `$geoWithin` (page 560) querie The `2d` index supports `$center` (page 572).
`$uniqueDocs` (page 575)	Deprecated. Modifies a `$geoWithin` (page 560) and `$near` (page 565) queries to ensu that even if a document matches the query multiple times, the query returns the document once.

$geometry

`$geometry`

New in version 2.4.

Changed in version 3.0: Add support to specify single-ringed GeoJSON *polygons* with areas greater than a single hemisphere.

The `$geometry` (page 569) operator specifies a *GeoJSON* geometry for use with the following geospatial query operators: `$geoWithin` (page 560), `$geoIntersects` (page 562), `$near` (page 565), and `$nearSphere` (page 567). `$geometry` (page 569) 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 569):

```
$geometry: {
   type: "<GeoJSON object type>",
   coordinates: [ <coordinates> ]
}
```

New in version 3.0: To specify a single-ringed GeoJSON *polygon* with a custom MongoDB CRS, use the following prototype (available only for `$geoWithin` (page 560) and `$geoIntersects` (page 562)):

```
$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.**

$minDistance

On this page

- Definition (page 570)
- Examples (page 570)

Definition

`$minDistance`

New in version 2.6.

Filters the results of a geospatial `$near` (page 565) or `$nearSphere` (page 567) query to those documents that are *at least* the specified distance from the center point.

`$minDistance` (page 570) is available for use with `2dsphere` index only.

If `$near` (page 565) or `$nearSphere` (page 567) query specifies the center point as a *GeoJSON point*, specify the distance as a non-negative number in *meters*.

If `$nearSphere` (page 567) query specifies the center point as *legacy coordinate pair*, specify the distance as a non-negative number in *radians*. `$near` (page 565) 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 567)

$maxDistance

On this page

- Definition (page 571)
- Example (page 571)

Definition

`$maxDistance`

The `$maxDistance` (page 571) operator constrains the results of a geospatial `$near` (page 565) or `$nearSphere` (page 567) 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 571).

The `2dsphere` and `2d` geospatial indexes both support `$maxDistance` (page 571): .

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 144) method.

$center

On this page

- Definition (page 572)
- Behavior (page 572)
- Example (page 572)

Definition

$center

New in version 1.4.

The $center (page 572) operator specifies a circle for a $geoWithin (page 560) 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 572) 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 572) *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 572).

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 ] } } }
)
```

$centerSphere

On this page

- Definition (page 572)
- Behavior (page 573)
- Example (page 573)

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 572) operator on both *GeoJSON* objects and legacy coordinate pairs.

To use `$centerSphere` (page 572), 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 `https://docs.mongodb.org/manual/tutorial/calculate-distances-using-spherical-geomet`

```
{
   <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 572) *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 572).

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 equatorial radius of the earth, 3963.2 miles:

```
db.places.find( {
  loc: { $geoWithin: { $centerSphere: [ [ -88, 30 ], 10/3963.2 ] } }
} )
```

$box

On this page

- Definition (page 573)
- Behavior (page 574)
- Example (page 574)

Definition

$box

Specifies a rectangle for a *geospatial* `$geoWithin` (page 560) query to return documents that are within the bounds of the rectangle, according to their point-based location data. When used with the `$box` (page 573) operator, `$geoWithin` (page 560) returns documents based on *grid coordinates* and does *not* query for GeoJSON shapes.

To use the `$box` (page 573) 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 573) *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 573).

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 ] ] } }
} )
```

$polygon

On this page

- Definition (page 574)
- Behavior (page 575)
- Example (page 575)

Definition

`$polygon`

New in version 1.9.

Specifies a polygon for a *geospatial* `$geoWithin` (page 560) 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 574) operator calculates distances using flat (planar) geometry.

Changed in version 2.2.3: Applications can use `$polygon` (page 574) *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 574) 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

On this page

- Definition (page 575)

Definition

`$uniqueDocs`

Deprecated since version 2.6: Geospatial queries no longer return duplicate results. The `$uniqueDocs` (page 575) operator has no impact on results.

Returns a document only once for a geospatial query even if the document matches the query multiple times.

Array

Query Operator Array

Name	Description
`$all` (page 575)	Matches arrays that contain all elements specified in the query.
`$elemMatch` (page 579)	Selects documents if element in the array field matches all the specified `$elemMa` (page 579) conditions.
`$size` (page 580)	Selects documents if the array field is a specified size.

$all

On this page

- Behavior (page 576)
- Examples (page 576)

`$all`

The `$all` (page 575) operator selects the documents where the value of a field is an array that contains all the specified elements. To specify an `$all` (page 575) expression, use the following prototype:

```
{ <field>: { $all: [ <value1> , <value2> ... ] } }
```

Behavior

Equivalent to `$and` Operation Changed in version 2.6.

The `$all` (page 575) is equivalent to an `$and` (page 535) 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 575) 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 [17]:

```
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 575) 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 575) allows for more matches than previous versions of MongoDB. Earlier versions could only match documents where the field contains the nested array.

Performance Queries that use the `$all` (page 575) operator must scan all the documents that match the first element in the `$all` (page 575) 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 575) 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" ],
```

[17] The `$all` (page 575) expression with a *single* element is for illustrative purposes since the `$all` (page 575) 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.

```
   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 575) 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 `$all` (page 575) with the `$elemMatch` (page 579) 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 579) 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 `$all` (page 575) operator exists to support queries on arrays. But you may use the `$all` (page 575) 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 51), `update()` (page 117), and `$set` (page 600).

$elemMatch (query)

On this page

- Definition (page 579)
- Behavior (page 579)
- Examples (page 579)

See also:

$elemMatch (projection) (page 590)

Definition

`$elemMatch`

The `$elemMatch` (page 579) operator matches documents that contain an array field with at least one element that matches all the specified query criteria.

```
{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }
```

If you specify only a single `<query>` condition in the `$elemMatch` (page 579) expression, you do not need to use `$elemMatch` (page 579).

Behavior You cannot specify a `$where` (page 558) expression as a query criterion for `$elemMatch` (page 579).

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 } ]
```

Single Query Condition If you specify a single query predicate in the `$elemMatch` (page 579) expression, `$elemMatch` (page 579) is not necessary.

For example, consider the following example where `$elemMatch` (page 579) specifies only a single query predicate `{ product: "xyz" }`:

```
db.survey.find(
   { results: { $elemMatch: { product: "xyz" } } }
)
```

Since the `$elemMatch` (page 579) only specifies a single condition, the `$elemMatch` (page 579) expression is not necessary, and instead you can use the following query:

```
db.survey.find(
   { "results.product": "xyz" }
)
```

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 580) 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 580) with a value of 1, as follows:

```
db.collection.find( { field: { $size: 1 } } );
```

`$size` (page 580) 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 580) portion of a query, although the other portions of a query can use indexes if applicable.

Bitwise

Bitwise Query Operators

Name	Description
$bitsAllSet (page 581)	Matches numeric or binary values in which a set of bit positions *all* have a 1.
$bitsAnySet (page 582)	Matches numeric or binary values in which *any* bit from a set of bit positic value of 1.
$bitsAllClear (page 584)	Matches numeric or binary values in which a set of bit positions *all* have a 0.
$bitsAnyClear (page 585)	Matches numeric or binary values in which *any* bit from a set of bit positic value of 0.

$bitsAllSet

On this page

- Behavior (page 581)
- Examples (page 582)

$bitsAllSet

New in version 3.2.

$bitsAllSet (page 581) matches documents where *all* of the bit positions given by the query are set (i.e. 1) in field.

```
{ <field>:  { $bitsAllSet:  <numeric bitmask> } }
{ <field>:  { $bitsAllSet:  < BinData (page 961) bitmask> } }
{ <field>:  { $bitsAllSet:  [ <position1>, <position2>, ...  ]  } }
```

The field value must be either numerical or a BinData (page 961) instance. Otherwise, $bitsAllSet (page 581) will not match the current document.

Numeric Bitmask You can provide a numeric bitmask to be matched against the operand field. It must be representable as a non-negative 32-bit signed integer. Otherwise, $bitsAllSet (page 581) will return an error.

BinData Bitmask You can also use an arbitrarily large BinData (page 961) instance as a bitmask.

Position List If querying a list of bit positions, each <position> must be a non-negative integer. Bit positions start at 0 from the least significant bit. For example, the decimal number 254 would have the following bit positions:

Bit Value	1	1	1	1	1	1	1	0
Position	7	6	5	4	3	2	1	0

Behavior

Floating Point Values $bitsAllSet (page 581) will not match numerical values that cannot be represented as a signed 64-bit integer. This can be the case if a value is either too large or small to fit in a signed 64-bit integer, or if it has a fractional component.

Sign Extension Numbers are sign extended. For example, $bitsAllSet considers bit position 200 to be set for the negative number -5, but bit position 200 to be clear for the positive number +5.

In contrast, BinData (page 961) instances are zero-extended. For example, given the following document:

```
db.collection.save({ x: BinData(0, "ww=="), binaryValueofA: "11000011" })
```

$bitsAllSet will consider all bits outside of `x` to be clear.

Examples The following examples will use a collection with the following documents:

```
db.collection.save({ _id: 1, a: 54, binaryValueofA: "00110110" })
db.collection.save({ _id: 2, a: 20, binaryValueofA: "00010100" })
db.collection.save({ _id: 3, a: 20.0, binaryValueofA: "00010100" })
db.collection.save({ _id: 4, a: BinData(0, "Zg=="), binaryValueofA: "01100110" })
```

Bit Position Array The following query uses the `$bitsAllSet` (page 581) operator to test whether field `a` has bits set at position `1` and position `5`, where the least significant bit is position `0`.

```
db.collection.find( { a: { $bitsAllSet: [ 1, 5 ] } } )
```

The query matches the following documents:

```
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
```

Integer Bitmask The following query uses the `$bitsAllSet` (page 581) operator to test whether field `a` has bits set at positions `1`, `4`, and `5` (the binary representation of the bitmask `50` is `00110010`).

```
db.collection.find( { a: { $bitsAllSet: 35 } } )
```

The query matches the following document:

```
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
```

BinData Bitmask The following query uses the `$bitsAllSet` (page 581) operator to test whether field `a` has bits set at positions *4* and *5* (the binary representation of `BinData(0, "MC==")` is `00110000`).

```
db.collection.find( { a: { $bitsAllSet: BinData(0, "MC==") } } )
```

The query matches the following document:

```
{ _id: 1, a: 54, binaryValueofA: "00110110" }
```

$bitsAnySet

On this page

- Behavior (page 583)
- Examples (page 583)

`$bitsAnySet`

New in version 3.2.

`$bitsAnySet` (page 582) matches documents where *any* of the bit positions given by the query are set (i.e. `1`) in `field`.

```
{ <field>: { $bitsAnySet: <numeric bitmask> } }
{ <field>: { $bitsAnySet: < BinData (page 961) bitmask> } }
{ <field>: { $bitsAnySet: [ <position1>, <position2>, ... ] } }
```

The `field` value must be either numerical or a `BinData` (page 961) instance. Otherwise, `$bitsAnySet` (page 582) will not match the current document.

Numeric Bitmask You can provide a numeric bitmask to be matched against the operand field. It must be representable as a non-negative 32-bit signed integer. Otherwise, `$bitsAnySet` (page 582) will return an error.

BinData Bitmask You can also use an arbitrarily large `BinData` (page 961) instance as a bitmask.

Position List If querying a list of bit positions, each `<position>` must be a non-negative integer. Bit positions start at `0` from the least significant bit. For example, the decimal number `254` would have the following bit positions:

Bit Value	1	1	1	1	1	1	1	0
Position	7	6	5	4	3	2	1	0

Behavior

Floating Point Values `$bitsAnySet` (page 582) will not match numerical values that cannot be represented as a signed 64-bit integer. This can be the case if a value is either too large or small to fit in a signed 64-bit integer, or if it has a fractional component.

Sign Extension Numbers are sign extended. For example, $bitsAllSet considers bit position `200` to be set for the negative number `-5`, but bit position `200` to be clear for the positive number `+5`.

In contrast, `BinData` (page 961) instances are zero-extended. For example, given the following document:

```
db.collection.save({ x: BinData(0, "ww=="), binaryValueofA: "11000011" })
```

$bitsAllSet will consider all bits outside of `x` to be clear.

Examples The following examples will use a collection with the following documents:

```
db.collection.save({ _id: 1, a: 54, binaryValueofA: "00110110" })
db.collection.save({ _id: 2, a: 20, binaryValueofA: "00010100" })
db.collection.save({ _id: 3, a: 20.0, binaryValueofA: "00010100" })
db.collection.save({ _id: 4, a: BinData(0, "Zg=="), binaryValueofA: "01100110" })
```

Bit Position Array The following query uses the `$bitsAnySet` (page 582) operator to test whether field `a` has either bit position `1` or bit position `5` set, where the least significant bit is position `0`.

```
db.collection.find( { a: { $bitsAnySet: [ 1, 5 ] } } )
```

The query matches the following documents:

```
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
```

Integer Bitmask The following query uses the `$bitsAnySet` (page 582) operator to test whether field `a` has any bits set at positions `0`, `1`, and `5` (the binary representation of the bitmask `35` is `00100011`).

```
db.collection.find( { a: { $bitsAnySet: 35 } } )
```

The query matches the following documents:

```
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
```

BinData Bitmask The following query uses the $bitsAnySet (page 582) operator to test whether field a has any bits set at positions *4*, and *5* (the binary representation of BinData(0, "MC==") is 00110000).

```
db.collection.find( { a: { $bitsAnySet: BinData(0, "MC==") } } )
```

The query matches the following documents:

```
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20.0, "binaryValueofA" : "00010100" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
```

$bitsAllClear

On this page

- Behavior (page 584)
- Examples (page 585)

$bitsAllClear

New in version 3.2.

$bitsAllClear (page 584) matches documents where *all* of the bit positions given by the query are clear (i.e. 0) in field.

```
{ <field>: { $bitsAllClear: <numeric bitmask> } }
{ <field>: { $bitsAllClear: < BinData (page 961) bitmask> } }
{ <field>: { $bitsAllClear: [ <position1>, <position2>, ... ] } }
```

The field value must be either numerical or a BinData (page 961) instance. Otherwise, $bitsAllClear (page 584) will not match the current document.

Numeric Bitmask You can provide a numeric bitmask to be matched against the operand field. It must be representable as a non-negative 32-bit signed integer. Otherwise, $bitsAllClear (page 584) will return an error.

BinData Bitmask You can also use an arbitrarily large BinData (page 961) instance as a bitmask.

Position List If querying a list of bit positions, each <position> must be a non-negative integer. Bit positions start at 0 from the least significant bit. For example, the decimal number 254 would have the following bit positions:

Bit Value	1	1	1	1	1	1	1	0
Position	7	6	5	4	3	2	1	0

Behavior

Floating Point Values $bitsAllClear (page 584) will not match numerical values that cannot be represented as a signed 64-bit integer. This can be the case if a value is either too large or small to fit in a signed 64-bit integer, or if it has a fractional component.

Sign Extension Numbers are sign extended. For example, $bitsAllSet considers bit position `200` to be set for the negative number `-5`, but bit position `200` to be clear for the positive number `+5`.

In contrast, `BinData` (page 961) instances are zero-extended. For example, given the following document:

```
db.collection.save({ x: BinData(0, "ww=="), binaryValueofA: "11000011" })
```

$bitsAllSet will consider all bits outside of `x` to be clear.

Examples The following examples will use a collection with the following documents:

```
db.collection.save({ _id: 1, a: 54, binaryValueofA: "00110110" })
db.collection.save({ _id: 2, a: 20, binaryValueofA: "00010100" })
db.collection.save({ _id: 3, a: 20.0, binaryValueofA: "00010100" })
db.collection.save({ _id: 4, a: BinData(0, "Zg=="), binaryValueofA: "01100110" })
```

Bit Position Array The following query uses the `$bitsAllClear` (page 584) operator to test whether field `a` has bits clear at position `1` and position `5`, where the least significant bit is position `0`.

```
db.collection.find( { a: { $bitsAllClear: [ 1, 5 ] } } )
```

The query matches the following documents:

```
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20, "binaryValueofA" : "00010100" }
```

Integer Bitmask The following query uses the `$bitsAllClear` (page 584) operator to test whether field `a` has bits clear at positions `0`, `1`, and `5` (the binary representation of the bitmask `35` is `00100011`).

```
db.collection.find( { a: { $bitsAllClear: 35 } } )
```

The query matches the following documents:

```
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20, "binaryValueofA" : "00010100" }
```

BinData Bitmask The following query uses the `$bitsAllClear` (page 584) operator to test whether field `a` has bits clear at positions *2* and *4* (the binary representation of `BinData(0, "ID==")` is `00010100`.

```
db.collection.find( { a: { $bitsAllClear: BinData(0, "ID==") } } )
```

The query matches the following documents:

```
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20, "binaryValueofA" : "00010100" }
```

$bitsAnyClear

On this page

- Behavior (page 586)
- Examples (page 586)

$bitsAnyClear

New in version 3.2.

$bitsAnyClear (page 585) matches documents where *any* of the bit positions given by the query are clear (i.e. 0) in field.

```
{ <field>:  { $bitsAnyClear:  <numeric bitmask> } }
{ <field>:  { $bitsAnyClear:  < BinData (page 961) bitmask> } }
{ <field>:  { $bitsAnyClear:  [ <position1>, <position2>, ...  ]  } }
```

The field value must be either numerical or a BinData (page 961) instance. Otherwise, $bitsAnyClear (page 585) will not match the current document.

Numeric Bitmask You can provide a numeric bitmask to be matched against the operand field. It must be representable as a non-negative 32-bit signed integer. Otherwise, $bitsAnyClear (page 585) will return an error.

BinData Bitmask You can also use an arbitrarily large BinData (page 961) instance as a bitmask.

Position List If querying a list of bit positions, each <position> must be a non-negative integer. Bit positions start at 0 from the least significant bit. For example, the decimal number 254 would have the following bit positions:

Bit Value	1	1	1	1	1	1	1	0
Position	7	6	5	4	3	2	1	0

Behavior

Floating Point Values $bitsAnyClear (page 585) will not match numerical values that cannot be represented as a signed 64-bit integer. This can be the case if a value is either too large or small to fit in a signed 64-bit integer, or if it has a fractional component.

Sign Extension Numbers are sign extended. For example, $bitsAllSet considers bit position 200 to be set for the negative number -5, but bit position 200 to be clear for the positive number +5.

In contrast, BinData (page 961) instances are zero-extended. For example, given the following document:

```
db.collection.save({ x: BinData(0, "ww=="), binaryValueofA: "11000011" })
```

$bitsAllSet will consider all bits outside of x to be clear.

Examples The following examples will use a collection with the following documents:

```
db.collection.save({ _id: 1, a: 54, binaryValueofA: "00110110" })
db.collection.save({ _id: 2, a: 20, binaryValueofA: "00010100" })
db.collection.save({ _id: 3, a: 20.0, binaryValueofA: "00010100" })
db.collection.save({ _id: 4, a: BinData(0, "Zg=="), binaryValueofA: "01100110" })
```

Bit Position Array The following query uses the $bitsAnyClear (page 585) operator to test whether field a has either bit position 1 or bit position 5 clear, where the least significant bit is position 0.

```
db.collection.find( { a: { $bitsAnyClear: [ 1, 5 ] } } )
```

The query matches the following documents:

```
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20.0, "binaryValueofA" : "00010100" }
```

Integer Bitmask The following query uses the `$bitsAnyClear` (page 585) operator to test whether field `a` has any bits clear at positions `0`, `1`, and `5` (the binary representation of the bitmask `35` is `00100011`).

```
db.collection.find( { a: { $bitsAnyClear: 35 } } )
```

The query matches the following documents:

```
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20.0, "binaryValueofA" : "00010100" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
```

BinData Bitmask The following query uses the `$bitsAnyClear` (page 585) operator to test whether field `a` has any bits clear at positions *4* and *5* (the binary representation of `BinData(0, "MC==")` is `00110000`).

```
db.collection.find( { a: { $bitsAnyClear: BinData(0, "MC==") } } )
```

The query matches the following documents:

```
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20.0, "binaryValueofA" : "00010100" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
```

Comments

$comment	**On this page** • Definition (page 587) • Behavior (page 587) • Examples (page 587)

Definition

`$comment`

The `$comment` (page 587) query operator associates a comment to any expression taking a query predicate.

Because comments propagate to the `profile` (page 484) log, adding a comment can make your profile data easier to interpret and trace.

The `$comment` (page 587) operator has the form:

```
db.collection.find( { <query>, $comment: <comment> } )
```

Behavior You can use the `$comment` (page 587) with any expression taking a query predicate, such as the query predicate in `db.collection.update()` (page 117) or in the `$match` (page 635) stage of the *aggregation pipeline* (page 746). For an example, see *Attach a Comment to an Aggregation Expression* (page 588).

Examples

Attach a Comment to `find` The following example adds a `$comment` (page 587) to a `find()` (page 51) 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 587) with any expression taking a query predicate.

The following examples uses the `$comment` (page 587) operator in the `$match` (page 635) 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`

Projection Operators

Projection Operators

Name	Description
`$` (page 588)	Projects the first element in an array that matches the query condition.
`$elemMatch` (page 591)	Projects the first element in an array that matches the specified `$elemMatch` (page 591) condition.
`$meta` (page 592)	Projects the document's score assigned during `$text` (page 549) operation.
`$slice` (page 594)	Limits the number of elements projected from an array. Supports skip and limit slices.

$ (projection)

On this page

- Definition (page 588)
- Usage Considerations (page 589)
- Behavior (page 589)
- Examples (page 589)
- Further Reading (page 590)

Definition

$

The positional `$` (page 588) 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 607).

Use `$` (page 588) in the *projection* document of the `find()` (page 51) method or the `findOne()` (page 62) method when you only need one particular array element in selected documents.

Usage Considerations Both the `$` (page 588) operator and the `$elemMatch` (page 591) operator project a subset of elements from an array based on a condition.

The `$` (page 588) operator projects the array elements based on some condition from the query statement.

The `$elemMatch` (page 591) 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 embedded documents. See *Array Field Limitations* (page 589) 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 527).

Array Field Limitations MongoDB requires the following when dealing with projection over arrays:

- Only one positional `$` (page 588) 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 579) query operator. The following query will return any embedded documents 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 591) 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 51) method includes a `sort()` (page 156), the `find()` (page 51) method applies the `sort()` (page 156) to order the matching documents **before** it applies the positional `$` (page 588) projection operator.

If an array field contains multiple documents with the same field name and the `find()` (page 51) method includes a `sort()` (page 156) 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 588) 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 588) 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 591)

$elemMatch (projection) **See also:**

$elemMatch (query) (page 579)

On this page

- Definition (page 591)
- Usage Considerations (page 591)
- Examples (page 591)

Definition

`$elemMatch`

New in version 2.2.

The `$elemMatch` (page 591) operator limits the contents of an `<array>` field from the query results to contain only the **first** element matching the `$elemMatch` (page 591) condition.

Usage Considerations Both the `$` (page 588) operator and the `$elemMatch` (page 591) operator project a subset of elements from an array based on a condition.

The `$` (page 588) operator projects the array elements based on some condition from the query statement.

The `$elemMatch` (page 591) 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 embedded documents. See *Array Field Limitations* (page 589) for an example.

Examples The examples on the `$elemMatch` (page 591) 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 51) operation queries for all documents where the value of the zipcode field is 63109. The $elemMatch (page 591) 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 591) 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 591) condition.

$elemMatch with Multiple Fields The $elemMatch (page 591) projection can specify criteria on multiple fields:

The following find() (page 51) 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 591) criteria.

See also:

$ (projection) (page 588) operator

$meta

On this page

- Behaviors (page 593)
- Examples (page 594)

$meta

New in version 2.6.

The $meta (page 592) projection operator returns for each matching document the metadata (e.g. "textScore") associated with the query.

A $meta (page 592) expression has the following syntax:

```
{ $meta: <metaDataKeyword> }
```

The $meta (page 592) expression can specify the following keyword as the <metaDataKeyword>:

Keyword	Description	Sort Order
"textScore"	Returns the score associated with the corresponding $text (page 549) query for each matching document. The text score signifies how well the document matched the *search term or terms* (page 551). If not used in conjunction with a $text (page 549) query, returns a score of 0.	Descending

Behaviors The $meta (page 592) expression can be a part of the *projection* document as well as a sort() (page 156) 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 592) values instead of with the stored values.

Projection The $meta (page 592) expression can be used in the *projection* document, as in:

```
db.collection.find(
   <query>,
   { score: { $meta: "textScore" } }
)
```

The $meta (page 592) expression specifies the inclusion of the field to the result set and does *not* specify the exclusion of the other fields.

The $meta (page 592) 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 592) expression can be part of a sort() (page 156) expression, as in:

```
db.collection.find(
   <query>,
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } )
```

To include a $meta (page 592) expression in a sort() (page 156) expression, the *same* $meta (page 592) 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 557).

Examples For examples of `"textScore"` projections and sorts, see `$text` (page 549).

$slice (projection)

`$slice`

The `$slice` (page 594) operator controls the number of items of an array that a query returns. For information on limiting the size of an array during an update with `$push` (page 616), see the `$slice` (page 619) 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 594) 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 594) 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 594). 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.

Additional Resources

- Quick Reference Cards[18]

2.3.2 Update Operators

On this page

- Update Operators (page 595)

The following modifiers are available for use in update operations; e.g. in `db.collection.update()` (page 117) and `db.collection.findAndModify()` (page 57).

[18] https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs

Update Operators

Fields

	Name	Description
Field Update Operators	`$inc` (page 595)	Increments the value of the field by the specified amount.
	`$mul` (page 596)	Multiplies the value of the field by the specified amount.
	`$rename` (page 598)	Renames a field.
	`$setOnInsert` (page 599)	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.
	`$set` (page 600)	Sets the value of a field in a document.
	`$unset` (page 602)	Removes the specified field from a document.
	`$min` (page 602)	Only updates the field if the specified value is less than the existing field value.
	`$max` (page 604)	Only updates the field if the specified value is greater than the existing field value.
	`$currentDate` (page 605)	Sets the value of a field to current date, either as a Date or a Timestamp.

$inc

On this page

- Definition (page 595)
- Behavior (page 595)
- Example (page 595)

Definition

`$inc`

The `$inc` (page 595) 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 595) operator accepts positive and negative values.

If the field does not exist, `$inc` (page 595) creates the field and sets the field to the specified value.

Use of the `$inc` (page 595) operator on a field with a null value will generate an error.

`$inc` (page 595) 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 117) operation uses the `$inc` (page 595) 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 117), `db.collection.findAndModify()` (page 57)

$mul

On this page

- Definition (page 596)
- Behavior (page 596)
- Examples (page 596)

Definition

`$mul`

New in version 2.6.

Multiply the value of a field by a number. To specify a `$mul` (page 596) 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 596) 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 596) 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 117) operation updates the document, using the `$mul` (page 596) 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 117) operation updates the document, applying the `$mul` (page 596) 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 117) operation uses the `$mul` (page 596) 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 117), `db.collection.findAndModify()` (page 57)

$rename

On this page

- Definition (page 598)
- Behavior (page 598)
- Examples (page 598)

Definition

`$rename`

The `$rename` (page 598) 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 `$rename` (page 598) operator logically performs an `$unset` (page 602) of both the old name and the new name, and then performs a `$set` (page 600) 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 598) operator removes that field and renames the specified `<field>` to `<newName>`.

If the field to rename does not exist in a document, `$rename` (page 598) does nothing (i.e. no operation).

For fields in embedded documents, the `$rename` (page 598) operator can rename these fields as well as move the fields in and out of embedded documents. `$rename` (page 598) 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 598) 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 598) 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 598) 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 117), `db.collection.findAndModify()` (page 57)

$setOnInsert

On this page

- Definition (page 599)
- Example (page 600)

Definition

`$setOnInsert`

New in version 2.4.

If an update operation with *upsert: true* (page 118) results in an insert of a document, then `$setOnInsert` (page 599) assigns the specified values to the fields in the document. If the update operation does not result in an insert, `$setOnInsert` (page 599) does nothing.

You can specify the `upsert` option for either the `db.collection.update()` (page 117) or `db.collection.findAndModify()` (page 57) 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 117) with *upsert: true* (page 118) 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 `$set` (page 600) and `$setOnInsert` (page 599) operations to this document.

The `products` collection contains the newly-inserted document:

```
{ "_id" : 1, "item" : "apple", "defaultQty" : 100 }
```

If the `db.collection.update()` (page 117) with *upsert: true* (page 118) had found a matching document, then MongoDB performs an update, applying the `$set` (page 600) operation but ignoring the `$setOnInsert` (page 599) operation.

See also:

`db.collection.update()` (page 117), `db.collection.findAndModify()` (page 57)

$set

On this page

- Definition (page 600)
- Behavior (page 600)
- Examples (page 600)

Definition

`$set`

The `$set` (page 600) operator replaces the value of a field with the specified value.

The `$set` (page 600) 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, `$set` (page 600) will add a new field with the specified value, provided that the new field does not violate a type constraint. If you specify a dotted path for a non-existent field, `$set` (page 600) will create the embedded documents *as needed* to fulfill the dotted path to the field.

If you specify multiple field-value pairs, `$set` (page 600) 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 600) 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 606).

See also:

`db.collection.update()` (page 117), `db.collection.findAndModify()` (page 57)

$unset

On this page

- Behavior (page 602)
- Example (page 602)

$unset

The $unset (page 602) operator deletes a particular field. Consider the following syntax:

```
{ $unset: { <field1>: "", ... } }
```

The specified value in the $unset (page 602) 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 602) does nothing (i.e. no operation).

When used with $ (page 607) to match an array element, $unset (page 602) 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 117) operation uses the $unset (page 602) 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 117), `db.collection.findAndModify()` (page 57)

$min

On this page

- Definition (page 602)
- Behavior (page 602)
- Examples (page 603)

Definition

$min

The $min (page 602) 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 602) 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 602) operator sets the field to the specified value.

For comparisons between values of different types, such as a number and a null, $min (page 602) uses the *BSON comparison order*.

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 602) 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 `250`:

```
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 117), `db.collection.findAndModify()` (page 57)

$max

On this page

- Definition (page 604)
- Behavior (page 604)
- Examples (page 604)

Definition

$max

The $max (page 604) 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 604) operator can compare values of different types, using the *BSON comparison order*.

The $max (page 604) 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 604) 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 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 117), `db.collection.findAndModify()` (page 57)

$currentDate

On this page

- Definition (page 605)
- Behavior (page 605)
- Example (page 605)

Definition

`$currentDate`

The `$currentDate` (page 605) operator sets the value of a field to the current date, either as a *Date* or a *timestamp*. The default type is *Date*.

Changed in version 3.0: MongoDB no longer treats the *timestamp* and the *Date* data types as equivalent for comparison/sorting purposes. For details, see *Date and Timestamp Comparison Order* (page 1053).

The `$currentDate` (page 605) 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 605) 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 117), `db.collection.findAndModify()` (page 57)

Array

Array Update Operators

On this page

- Update Operators (page 606)
- Update Operator Modifiers (page 618)

Update Operators

Name	Description
`$` (page 607)	Acts as a placeholder to update the first element that matches the query condition in a update.
`$addToSet` (page 609)	Adds elements to an array only if they do not already exist in the set.
`$pop` (page 611)	Removes the first or last item of an array.
`$pullAll` (page 612)	Removes all matching values from an array.
`$pull` (page 613)	Removes all array elements that match a specified query.
`$pushAll` (page 616)	*Deprecated.* Adds several items to an array.
`$push` (page 616)	Adds an item to an array.

$ (update)

On this page

- Definition (page 607)
- Behavior (page 607)
- Examples (page 607)

Definition

$

The positional $ (page 607) 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 588) projection operator.

The positional $ (page 607) operator has the form:

```
{ "<array>.$" : value }
```

When used with update operations, e.g. `db.collection.update()` (page 117) and `db.collection.findAndModify()` (page 57),

- the positional $ (page 607) 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 607) with *upsert* operations because inserts will use the `$` as a field name in the inserted document.

Nested Arrays The positional $ (page 607) 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 607) placeholder is a single value

Unsets When used with the `$unset` (page 602) operator, the positional $ (page 607) 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 `$ne` (page 531), `$not` (page 536), or `$nin` (page 533), 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 579) 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 607) 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 607) operator acts as a placeholder for the **first match** of the update *query document*.

Update Documents in an Array The positional `$` (page 607) operator facilitates updates to arrays that contain embedded documents. Use the positional `$` (page 607) operator to access the fields in the embedded documents with the *dot notation* on the `$` (page 607) 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 607) 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 `$` (page 607) operator can update the first array element that matches multiple query criteria specified with the `$elemMatch()` (page 579) 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 607) 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 117), db.collection.findAndModify() (page 57), $elemMatch() (page 579)

$addToSet

On this page

- Definition (page 609)
- Behavior (page 609)
- Examples (page 610)

Definition

$addToSet

The $addToSet (page 609) operator adds a value to an array unless the value is already present, in which case $addToSet (page 609) does nothing to that array.

The $addToSet (page 609) 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 609) only ensures that there are no duplicate items *added* to the set and does not affect existing duplicate elements. $addToSet (page 609) does not guarantee a particular ordering of elements in the modified set.

If the field is absent in the document to update, $addToSet (page 609) 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 609) appends the whole array as a *single* element.

Consider a document in a collection `test` containing an array field `letters`:

```
{ _id: 1, letters: ["a", "b"] }
```

The following operation appends the array `[ "c", "d" ]` to the `letters` field:

```
db.test.update(
   { _id: 1 },
   { $addToSet: {letters: [ "c", "d" ] } }
)
```

The `letters` array now includes the `[ "c", "d" ]` array as an element:

```
{ _id: 1, letters: [ "a", "b", [ "c", "d" ] ] }
```

To add each element of the value **separately**, use the `$each` (page 619) modifier with `$addToSet` (page 609). See *$each Modifier* (page 610) 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: "polarizing_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 609) 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 609) operator with the `$each` (page 619) modifier. The `$each` (page 619) modifier allows the `$addToSet` (page 609) 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 `$addToSet` (page 609) operator with the `$each` (page 619) 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 117), db.collection.findAndModify() (page 57), $push (page 616)

$pop

On this page

- Definition (page 611)
- Behavior (page 611)
- Examples (page 611)

Definition

$pop

The $pop (page 611) operator removes the first or last element of an array. Pass $pop (page 611) 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 611) 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 611) operation fails if the <field> is not an array.

If the $pop (page 611) 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 611) 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 117), `db.collection.findAndModify()` (page 57)

$pullAll

On this page

- Definition (page 612)
- Behavior (page 612)
- Examples (page 612)

Definition

`$pullAll`

The `$pullAll` (page 612) operator removes all instances of the specified values from an existing array. Unlike the `$pull` (page 613) operator that removes elements by specifying a query, `$pullAll` (page 612) removes elements that match the listed values.

The `$pullAll` (page 612) 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 612) 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 117), `db.collection.findAndModify()` (page 57)

$pull

On this page

- Behavior (page 613)
- Examples (page 613)

`$pull`

The `$pull` (page 613) operator removes from an existing array all instances of a value or values that match a specified condition.

The `$pull` (page 613) operator has the form:

```
{ $pull: { <field1>: <value|condition>, <field2>: <value|condition>, ... } }
```

To specify a `<field>` in an embedded document or in an array, use *dot notation*.

Behavior If you specify a `<condition>` and the array elements are embedded documents, `$pull` (page 613) operator applies the `<condition>` as if each array element were a document in a collection. See *Remove Items from an Array of Documents* (page 614) for an example.

If the specified `<value>` to remove is an array, `$pull` (page 613) 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 613) 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 document in the `stores` collection:

```
{
   _id: 1,
   fruits: [ "apples", "pears", "oranges", "grapes", "bananas" ],
   vegetables: [ "carrots", "celery", "squash", "carrots" ]
}
{
   _id: 2,
   fruits: [ "plums", "kiwis", "oranges", "bananas", "apples" ],
   vegetables: [ "broccoli", "zucchini", "carrots", "onions" ]
}
```

The following operation updates all documents in the collection to remove `"apples"` and `"oranges"` from the array `fruits` and remove `"carrots"` from the array `vegetables`:

```
db.stores.update(
    { },
    { $pull: { fruits: { $in: [ "apples", "oranges" ] }, vegetables: "carrots" } },
    { multi: true }
)
```

After the operation, the `fruits` array no longer contains any `"apples"` or `"oranges"` values, and the `vegetables` array no longer contains any `"carrots"` values:

```
{
  "_id" : 1,
  "fruits" : [ "pears", "grapes", "bananas" ],
  "vegetables" : [ "celery", "squash" ]
}
{
  "_id" : 2,
  "fruits" : [ "plums", "kiwis", "bananas" ],
  "vegetables" : [ "broccoli", "zucchini", "onions" ]
}
```

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 530)) `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 both a `score` field equal to `8` and an `item` field equal to `"B"`:

```
db.survey.update(
  { },
  { $pull: { results: { score: 8 , item: "B" } } },
  { multi: true }
)
```

The `$pull` (page 613) 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 contain both a `score` field equal to `8` and an `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 `$pull` (page 613) operator applies its query to each element as though it were a top-level object, the expression did not require the use of `$elemMatch` (page 579) 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 579):

```
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 117), `db.collection.findAndModify()` (page 57)

$pushAll

`$pushAll`

Deprecated since version 2.4: Use the `$push` (page 616) operator with `$each` (page 619) instead.

The `$pushAll` (page 616) operator appends the specified values to an array.

The `$pushAll` (page 616) operator has the form:

```
{ $pushAll: { <field>: [ <value1>, <value2>, ... ] } }
```

If you specify a single value, `$pushAll` (page 616) will behave as `$push` (page 616).

$push

On this page

- Definition (page 616)
- Behavior (page 616)
- Modifiers (page 616)
- Examples (page 617)

Definition

`$push`

The `$push` (page 616) operator appends a specified value to an array.

The `$push` (page 616) 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 616) 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, `$push` (page 616) appends the whole array as a *single* element. To add each element of the value separately, use the `$each` (page 619) modifier with `$push` (page 616). For an example, see *Append Multiple Values to an Array* (page 617). For a list of modifiers available for `$push` (page 616), see *Modifiers* (page 616).

Changed in version 2.4: MongoDB adds support for the `$each` (page 619) modifier to the `$push` (page 616) operator. Before 2.4, use `$pushAll` (page 616) for similar functionality.

Modifiers New in version 2.4.

You can use the `$push` (page 616) operator with the following modifiers:

Modifier	Description
`$each` (page 619)	Appends multiple values to the array field. Changed in version 2.6: When used in conjunction with the other modifiers, the `$each` (page 619) modifier no longer needs to be first.
`$slice` (page 619)	Limits the number of array elements. Requires the use of the `$each` (page 619) modifier.
`$sort` (page 622)	Orders elements of the array. Requires the use of the `$each` (page 619) modifier. Changed in version 2.6: In previous versions, `$sort` (page 622) is only available when used with both `$each` (page 619) and `$slice` (page 619).
`$position` (page 625)	Specifies the location in the array at which to insert the new elements. Requires the use of the `$each` (page 619) modifier. Without the `$position` (page 625) modifier, the `$push` (page 616) appends the elements to the end of the array. New in version 2.6.

When used with modifiers, the `$push` (page 616) 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 `$push` (page 616) with the `$each` (page 619) 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 616) operation uses:

- the $each (page 619) modifier to add multiple documents to the quizzes array,
- the $sort (page 622) modifier to sort all the elements of the modified quizzes array by the score field in descending order, and
- the $slice (page 619) 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 117), db.collection.findAndModify() (page 57)

Update Operator Modifiers

Name	Description
$each (page 619)	Modifies the $push (page 616) and $addToSet (page 609) operators to appe items for array updates.
$slice (page 619)	Modifies the $push (page 616) operator to limit the size of updated arrays.
$sort (page 622)	Modifies the $push (page 616) operator to reorder documents stored in an arra
$position (page 625)	Modifies the $push (page 616) operator to specify the position in the array to a elements.

$each

On this page

- Definition (page 619)
- Examples (page 619)

Definition

$each

The $each (page 619) modifier is available for use with the $addToSet (page 609) operator and the $push (page 616) operator.

Use with the $addToSet (page 609) 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 616) 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 619) modifier to the $push (page 616) operator. The $push (page 616) operator can use $each (page 619) modifier with other modifiers. For a list of modifiers available for $push (page 616), see *Modifiers* (page 616).

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 609) operator with the $each (page 619) 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" ]
}
```

$slice

On this page

- Behavior (page 620)
- Examples (page 620)

$slice

New in version 2.4.

The $slice (page 619) modifier limits the number of array elements during a $push (page 616) operation. To project, or return, a specified number of array elements from a read operation, see the $slice (page 594) projection operator instead.

To use the $slice (page 619) modifier, it **must** appear with the $each (page 619) modifier. You can pass an empty array [] to the $each (page 619) modifier such that only the $slice (page 619) 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 619) modifier to appear as the first modifier if used in conjunction with $slice (page 619). For a list of modifiers available for $push (page 616), see *Modifiers* (page 616).

Trying to use the $slice (page 619) modifier without the $each (page 619) 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 619) 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 619) 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 `$slice` (page 619) modifier, specify the number of elements to slice (e.g. `-3`) for the `$slice` (page 619) modifier and an empty array `[]` for the `$each` (page 619) 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 616) operation uses:

- the `$each` (page 619) modifier to add multiple documents to the `quizzes` array,
- the `$sort` (page 622) modifier to sort all the elements of the modified `quizzes` array by the `score` field in descending order, and
- the `$slice` (page 619) 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 616) for details.

$sort

On this page

- Behavior (page 623)
- Examples (page 623)

`$sort`

New in version 2.4.

The `$sort` (page 622) modifier orders the elements of an array during a `$push` (page 616) operation.

To use the `$sort` (page 622) modifier, it **must** appear with the `$each` (page 619) modifier. You can pass an empty array `[]` to the `$each` (page 619) modifier such that only the `$sort` (page 622) 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 `$sort` (page 622) modifier can sort array elements that are not documents. In previous versions, the `$sort` (page 622) 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 622) modifier can only sort by a specific field in the documents.

Trying to use the `$sort` (page 622) modifier without the `$each` (page 619) modifier results in an error. The `$sort` (page 622) no longer requires the `$slice` (page 619) modifier. For a list of modifiers available for `$push` (page 616), see *Modifiers* (page 616).

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 619) 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 616) operation uses:

- the `$each` (page 619) modifier to add multiple documents to the `quizzes` array,

- the `$sort` (page 622) modifier to sort all the elements of the modified `quizzes` array by the `score` field in descending order, and
- the `$slice` (page 619) 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 616) for details.

$position

On this page

- Definition (page 625)
- Examples (page 626)

Definition

`$position`

New in version 2.6.

The `$position` (page 625) modifier specifies the location in the array at which the `$push` (page 616) operator insert elements. Without the `$position` (page 625) modifier, the `$push` (page 616) operator inserts elements to the end of the array. See *$push modifiers* (page 616) for more information.

To use the `$position` (page 625) modifier, it **must** appear with the `$each` (page 619) 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 625) modifier has no effect and $push (page 616) 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 ] }
```

Bitwise

Bitwise Update Operator

Name	Description
$bit (page 627)	Performs bitwise AND, OR, and XOR updates of integer values.

$bit

On this page

- Definition (page 627)
- Examples (page 627)

Definition

`$bit`

Changed in version 2.6: Added support for bitwise `xor` operation.

The `$bit` (page 627) operator performs a bitwise update of a field. The operator supports bitwise `and`, bitwise `or`, and bitwise `xor` (i.e. exclusive or) operations. To specify a `$bit` (page 627) 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 803) 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 117) 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 803) 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 117) 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 117) 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 117), db.collection.findAndModify() (page 57)

Isolation

Isolation Update Operator

Name	Description
$isolated (page 629)	Modifies the behavior of a write operation to increase the isolation of the op

$isolated

On this page

- Definition (page 629)
- Behavior (page 629)
- Example (page 629)

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 629) 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 for storage engies that take a write lock (e.g. MMAPv1), or for document-level locking storage engine that normally do not take a write lock (e.g. WiredTiger), `$isolated` (page 629) operator will make WiredTiger single-threaded for the duration of the operation.

Behavior The `$isolated` (page 629) isolation operator does **not** provide "all-or-nothing" atomicity for write operations.

`$isolated` (page 629) 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 629) operator, the `multi`-update operation will allow other operations to interleave with its update of the matched documents.

See also:

`db.collection.update()` (page 117) and `db.collection.remove()` (page 101)

$atomic

Deprecated since version 2.2: The `$isolated` (page 629) operator replaces `$atomic`.

2.3.3 Aggregation Pipeline Operators

On this page

- Stage Operators (page 629)
- Expression Operators (page 659)
- Accumulators (page 728)

Stage Operators

In the `db.collection.aggregate` (page 20) method, pipeline stages appear in an array. Documents pass through the stages in sequence.

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

Name	Description
`$project` (page 631)	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
`$match` (page 635)	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. `$match` (page 635) uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
`$redact` (page 637)	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 `$project` (page 631) and `$match` (page 635). Can be used to implement field level redaction. For each input document, outputs either one or zero document.
`$limit` (page 640)	Passes the first *n* documents unmodified to the pipeline where *n* is the specified limit. For each input document, outputs either one document (for the first *n* documents) or zero documents (after the first *n* documents).
`$skip` (page 641)	Skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents).
`$unwind` (page 641)	Deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array.
`$group` (page 644)	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.
`$sample` (page 648)	Randomly selects the specified number of documents from its input.
`$sort` (page 649)	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
`$geoNear` (page 651)	Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of `$match` (page 635), `$sort` (page 649), and `$limit` (page 640) for geospatial data. The output documents include an additional distance field and can include a location identifier field.
`$lookup` (page 654)	Performs a left outer join to another collection in the *same* database to filter in documents from the “joined” collection for processing.
`$out` (page 656)	Writes the resulting documents of the aggregation pipeline to a collection. To use the `$out` (page 656) stage, it must be the last stage in the pipeline.
`$indexStats` (page 657)	Returns statistics regarding the use of each index for the collection.

Pipeline Aggregation Stages

On this page

• Additional Resources (page 659)

Name	Description
`$project` (page 631)	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
`$match` (page 635)	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. `$match` (page 635) uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
`$redact` (page 637)	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 `$project` (page 631) and `$match` (page 635). Can be used to implement field level redaction. For each input document, outputs either one or zero document.
`$limit` (page 640)	Passes the first *n* documents unmodified to the pipeline where *n* is the specified limit. For each input document, outputs either one document (for the first *n* documents) or zero documents (after the first *n* documents).
`$skip` (page 641)	Skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents).
`$unwind` (page 641)	Deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array.
`$group` (page 644)	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.
`$sample` (page 648)	Randomly selects the specified number of documents from its input.
`$sort` (page 649)	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
`$geoNear` (page 651)	Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of `$match` (page 635), `$sort` (page 649), and `$limit` (page 640) for geospatial data. The output documents include an additional distance field and can include a location identifier field.
`$lookup` (page 654)	Performs a left outer join to another collection in the *same* database to filter in documents from the "joined" collection for processing.
`$out` (page 656)	Writes the resulting documents of the aggregation pipeline to a collection. To use the `$out` (page 656) stage, it must be the last stage in the pipeline.
`$indexStats` (page 657)	Returns statistics regarding the use of each index for the collection.

$project (aggregation)

On this page

- Definition (page 631)
- Considerations (page 632)
- Examples (page 632)

Definition

`$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 631) stage has the following prototype form:

```
{ $project: { <specifications> } }
```

The `$project` (page 631) 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.

Considerations

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 `$project` (page 631).
- If you specify an inclusion of a field that does not exist in the document, `$project` (page 631) ignores that field inclusion; i.e. `$project` (page 631) 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 631).

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 747).

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 `$literal` (page 712) operator. Otherwise, `$project` (page 631) 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.

Changed in version 3.2: Starting in MongoDB 3.2, `$project` (page 631) stage supports using the square brackets `[]` to directly create new array fields. If array specification includes fields that are non-existent in a document, the operation substitutes `null` as the value for that field. For an example, see *Project New Array Fields* (page 634).

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 631) 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 631) 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 631) 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 631) 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
}
```

Project New Array Fields For example, if a collection includes the following document:

```
{ "_id" : ObjectId("55ad167f320c6be244eb3b95"), "x" : 1, "y" : 1 }
```

The following operation projects the fields `x` and `y` as elements in a new field `myArray`:

```
db.collection.aggregate( [ { $project: { myArray: [ "$x", "$y" ] } } ] )
```

The operation returns the following document:

```
{ "_id" : ObjectId("55ad167f320c6be244eb3b95"), "myArray" : [ 1, 1 ] }
```

If array specification includes fields that are non-existent in a document, the operation substitutes `null` as the value for that field.

For example, given the same document as above, the following operation projects the fields `x`, `y`, and a non-existing field `$someField` as elements in a new field `myArray`:

```
db.collection.aggregate( [ { $project: { myArray: [ "$x", "$y", "$someField" ] } } ] )
```

The operation returns the following document:

```
{ "_id" : ObjectId("55ad167f320c6be244eb3b95"), "myArray" : [ 1, 1, null ] }
```

See also:

`https://docs.mongodb.org/manual/tutorial/aggregation-zip-code-data-set`, `https://docs.mongodb.org/manual/tutorial/aggregation-with-user-preference-data`

$match (aggregation)

On this page

- Definition (page 635)
- Behavior (page 635)
- Examples (page 636)

Definition

`$match`

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

The `$match` (page 635) stage has the following prototype form:

```
{ $match: { <query> } }
```

`$match` (page 635) takes a document that specifies the query conditions. The query syntax is identical to the *read operation query* syntax.

Behavior

Pipeline Optimization

- Place the `$match` (page 635) as early in the aggregation *pipeline* as possible. Because `$match` (page 635) limits the total number of documents in the aggregation pipeline, earlier `$match` (page 635) operations minimize the amount of processing down the pipe.
- If you place a `$match` (page 635) at the very beginning of a pipeline, the query can take advantage of *indexes* like any other `db.collection.find()` (page 51) or `db.collection.findOne()` (page 62).

Restrictions

- You cannot use `$where` (page 558) in `$match` (page 635) queries as part of the aggregation pipeline.
- To use `$text` (page 549) in the `$match` (page 635) stage, the `$match` (page 635) stage has to be the first stage of the pipeline.

Examples The examples use a collection named `articles` with the following documents:

```
{ "_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }
{ "_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }
{ "_id" : ObjectId("55f5a192d4bede9ac365b257"), "author" : "ahn", "score" : 60, "views" : 1000 }
{ "_id" : ObjectId("55f5a192d4bede9ac365b258"), "author" : "li", "score" : 55, "views" : 5000 }
{ "_id" : ObjectId("55f5a1d3d4bede9ac365b259"), "author" : "annT", "score" : 60, "views" : 50 }
{ "_id" : ObjectId("55f5a1d3d4bede9ac365b25a"), "author" : "li", "score" : 94, "views" : 999 }
{ "_id" : ObjectId("55f5a1d3d4bede9ac365b25b"), "author" : "ty", "score" : 95, "views" : 1000 }
```

Equality Match The following operation uses `$match` (page 635) to perform a simple equality match:

```
db.articles.aggregate(
    [ { $match : { author : "dave" } } ]
);
```

The `$match` (page 635) selects the documents where the `author` field equals `dave`, and the aggregation returns the following:

```
{ "_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }
{ "_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }
```

Perform a Count The following example selects documents to process using the `$match` (page 635) pipeline operator and then pipes the results to the `$group` (page 644) pipeline operator to compute a count of the documents:

```
db.articles.aggregate( [
  { $match: { $or: [ { score: { $gt: 70, $lt: 90 } }, { views: { $gte: 1000 } } ] } },
  { $group: { _id: null, count: { $sum: 1 } } }
] );
```

In the aggregation pipeline, `$match` (page 635) selects the documents where either the `score` is greater than `70` and less than `90` or the `views` is greater than or equal to `1000`. These documents are then piped to the `$group` (page 644) to perform a count. The aggregation returns the following:

```
{ "_id" : null, "count" : 5 }
```

See also:

`https://docs.mongodb.org/manual/tutorial/aggregation-zip-code-data-set`, `https://docs.mongodb.org/manual/tutorial/aggregation-with-user-preference-data`

$redact (aggregation)

On this page

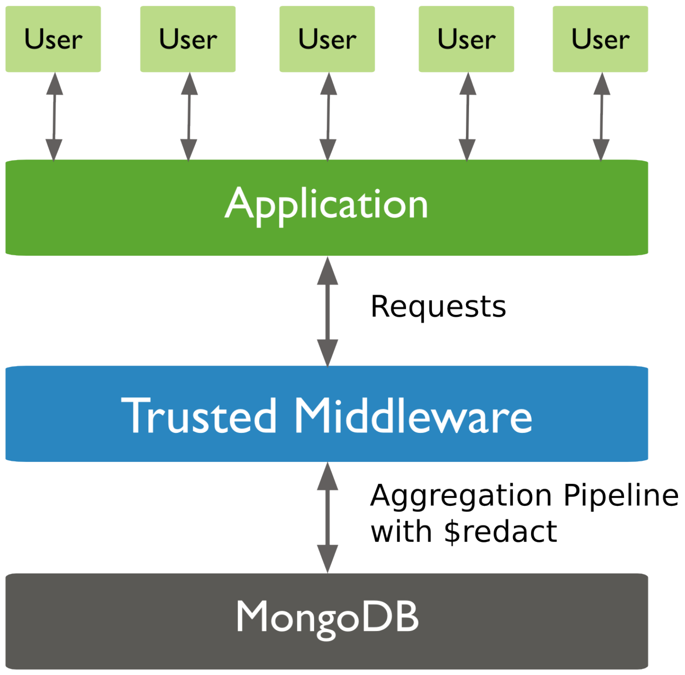
- Definition (page 637)
- Examples (page 637)

Definition

`$redact`

New in version 2.6.

Restricts the contents of the documents based on information stored in the documents themselves.



The `$redact` (page 637) stage has the following prototype form:

```
{ $redact: <expression> }
```

The argument can be any valid *expression* (page 747) as long as it resolves to $$DESCEND (page 637), $$PRUNE (page 637), or $$KEEP (page 637) system variables. For more information on expressions, see *Expressions* (page 747).

System Variable	Description
$$DESCEND	`$redact` (page 637) returns the fields at the current document level, excluding embedded documents. To include embedded documents and embedded documents within arrays, apply the `$cond` (page 726) expression to the embedded documents to determine access for these embedded documents.
$$PRUNE	`$redact` (page 637) excludes all fields at this current document/embedded document level, **without** further inspection of any of the excluded fields. This applies even if the excluded field contains embedded documents that may have different access levels.
$$KEEP	`$redact` (page 637) returns or keeps all fields at this current document/embedded document level, **without** further inspection of the fields at this level. This applies even if the included field contains embedded documents that may have different access levels.

Examples The examples in this section use the `db.collection.aggregate()` (page 20) helper provided in the 2.6 version of the `mongo` (page 803) shell.

Evaluate Access at Every Document Level A `forecasts` collection contains documents of the following form where the `tags` field lists the different access values for that document/embedded document 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 637) 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 707), $setIntersection (page 665)

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/embedded document at level `5`, include a `$redact` (page 637) 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 637) 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 embedded documents 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 637) stage excluded the field `cc` as a whole, including the `shipping_addr` field which contained embedded documents that had `level` field values equal to `3` and not `5`.

See also:

`https://docs.mongodb.org/manual/tutorial/implement-field-level-redaction` for steps to set up multiple combinations of access for the same data.

$limit (aggregation)

On this page

- Definition (page 640)
- Example (page 640)

Definition

`$limit`

Limits the number of documents passed to the next stage in the *pipeline*.

The `$limit` (page 640) stage has the following prototype form:

```
{ $limit: <positive integer> }
```

`$limit` (page 640) 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 640) has no effect on the content of the documents it passes.

Note: When a `$sort` (page 649) immediately precedes a `$limit` (page 640) in the pipeline, the `$sort` (page 649) 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 649) would sort all the results in memory, and then limit the results to n results.

See also:

`https://docs.mongodb.org/manual/tutorial/aggregation-zip-code-data-set`, `https://docs.mongodb.org/manual/tutorial/aggregation-with-user-preference-data`

$skip (aggregation)

On this page

- Definition (page 641)
- Example (page 641)

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 641) stage has the following prototype form:

```
{ $skip: <positive integer> }
```

`$skip` (page 641) 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 641) has no effect on the content of the documents it passes along the pipeline.

See also:

`https://docs.mongodb.org/manual/tutorial/aggregation-zip-code-data-set`, `https://docs.mongodb.org/manual/tutorial/aggregation-with-user-preference-data`

$unwind (aggregation)

On this page

- Definition (page 641)
- Behaviors (page 642)
- Examples (page 642)

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 641) stage has one of two syntaxes:

- The operand is a field path:

  ```
  { $unwind: <field path> }
  ```

  To specify a field path, prefix the field name with a dollar sign $ and enclose in quotes.

- The operand is a document:

  New in version 3.2.

  ```
  {
    $unwind:
      {
        path: <field path>,
        includeArrayIndex: <string>,
        preserveNullAndEmptyArrays: <boolean>
      }
  }
  ```

  field string path Field path to an array field. To specify a field path, prefix the field name with a dollar sign $ and enclose in quotes.

  field string includeArrayIndex Optional. The name of a new field to hold the array index of the element. The name cannot start with a dollar sign $.

  field boolean preserveNullAndEmptyArrays Optional. If true, if the path is null, missing, or an empty array, $unwind (page 641) outputs the document. If false, $unwind (page 641) does not output a document if the path is null, missing, or an empty array.

  The default value is false.

Behaviors

Non-Array Field Path Changed in version 3.2: $unwind (page 641) stage no longer errors on non-array operands. If the operand does not resolve to an array but is not missing, null, or an empty array, $unwind (page 641) treats the operand as a single element array.

Previously, if a value in the field specified by the field path is *not* an array, db.collection.aggregate() (page 20) generates an error.

Missing Field If you specify a path for a field that does not exist in an input document or the field is an empty array, $unwind (page 641), by default, ignores the input document and will not output documents for that input document.

New in version 3.2: To output documents where the array field is missing, null or an empty array, use the option preserveNullAndEmptyArrays.

Examples

Unwind Array Consider an inventory with the following document:

```
{ "_id" : 1, "item" : "ABC1", sizes: [ "S", "M", "L"] }
```

The following aggregation uses the $unwind (page 641) 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.

See also:

https://docs.mongodb.org/manual/tutorial/aggregation-zip-code-data-set,
https://docs.mongodb.org/manual/tutorial/aggregation-with-user-preference-data

`includeArrayIndex` and `preserveNullAndEmptyArrays` New in version 3.2.

A collection `inventory` has the following documents:

```
{ "_id" : 1, "item" : "ABC", "sizes": [ "S", "M", "L"] }
{ "_id" : 2, "item" : "EFG", "sizes" : [ ] }
{ "_id" : 3, "item" : "IJK", "sizes": "M" }
{ "_id" : 4, "item" : "LMN" }
{ "_id" : 5, "item" : "XYZ", "sizes" : null }
```

The following `$unwind` (page 641) operations are equivalent and return a document for each element in the `sizes` field. If the `sizes` field does not resolve to an array but is not missing, null, or an empty array, `$unwind` (page 641) treats the non-array operand as a single element array.

```
db.inventory.aggregate( [ { $unwind: "$sizes" } ] )
db.inventory.aggregate( [ { $unwind: { path: "$sizes" } } ] )
```

The operation returns the following documents:

```
{ "_id" : 1, "item" : "ABC", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC", "sizes" : "L" }
{ "_id" : 3, "item" : "IJK", "sizes" : "M" }
```

The following `$unwind` (page 641) operation uses the `includeArrayIndex` option to output also the array index of the array element.

```
db.inventory.aggregate( [ { $unwind: { path: "$sizes", includeArrayIndex: "arrayIndex" } } ] )
```

The operation unwinds the `sizes` array and includes the array index of the array index in the new `arrayIndex` field. If the `sizes` field does not resolve to an array but is not missing, null, or an empty array, the `arrayIndex` field is `null`.

```
{ "_id" : 1, "item" : "ABC", "sizes" : "S", "arrayIndex" : NumberLong(0) }
{ "_id" : 1, "item" : "ABC", "sizes" : "M", "arrayIndex" : NumberLong(1) }
{ "_id" : 1, "item" : "ABC", "sizes" : "L", "arrayIndex" : NumberLong(2) }
{ "_id" : 3, "item" : "IJK", "sizes" : "M", "arrayIndex" : null }
```

The following `$unwind` (page 641) operation uses the `preserveNullAndEmptyArrays` option to include in the output those documents where `sizes` field is missing, null or an empty array.

```
db.inventory.aggregate( [
   { $unwind: { path: "$sizes", preserveNullAndEmptyArrays: true } }
] )
```

In addition to unwinding the documents where the `sizes` is an array of elements or a non-null, non-array field, the operation outputs, without modification, those documents where the `sizes` field is missing, null or an empty array:

```
{ "_id" : 1, "item" : "ABC", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC", "sizes" : "L" }
{ "_id" : 2, "item" : "EFG" }
{ "_id" : 3, "item" : "IJK", "sizes" : "M" }
{ "_id" : 4, "item" : "LMN" }
{ "_id" : 5, "item" : "XYZ", "sizes" : null }
```

$group (aggregation)

On this page

- Definition (page 644)
- Considerations (page 644)
- Examples (page 645)
- Additional Resources (page 648)

Definition

`$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 `$group` (page 644)'s `_id` field. `$group` (page 644) does *not* order its output documents.

The `$group` (page 644) 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 747). For more information on expressions, see *Expressions* (page 747).

Considerations

Accumulator Operator The `<accumulator>` operator must be one of the following accumulator operators:

Name	Description
`$sum` (page 729)	Returns a sum of numerical values. Ignores non-numeric values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$avg` (page 732)	Returns an average of numerical values. Ignores non-numeric values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$first` (page 734)	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order. Available in `$group` (page 644) stage only.
`$last` (page 735)	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order. Available in `$group` (page 644) stage only.
`$max` (page 736)	Returns the highest expression value for each group. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$min` (page 738)	Returns the lowest expression value for each group. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$push` (page 740)	Returns an array of expression values for each group. Available in `$group` (page 644) stage only.
`$addToSet` (page 741)	Returns an array of *unique* expression values for each group. Order of the array elements is undefined. Available in `$group` (page 644) stage only.
`$stdDevPop` (page 742)	Returns the population standard deviation of the input values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$stdDevSamp` (page 744)	Returns the sample standard deviation of the input values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.

`$group` Operator and Memory The `$group` (page 644) stage has a limit of 100 megabytes of RAM. By default, if the stage exceeds this limit, `$group` (page 644) will produce an error. However, to allow for the handling of large datasets, set the `allowDiskUse` (page 20) option to `true` to enable `$group` (page 644) operations to write to temporary files. See `db.collection.aggregate()` (page 20) method and the `aggregate` (page 303) command for details.

Changed in version 2.6: MongoDB introduces a limit of 100 megabytes of RAM for the `$group` (page 644) stage as well as the `allowDiskUse` (page 20) 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("2014-04-04T11:21:39.736
{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.33
```

Group by Month, Day, and Year The following aggregation operation uses the `$group` (page 644) 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, "co
{ "_id" : { "month" : 4, "day" : 4, "year" : 2014 }, "totalPrice" : 200, "averageQuantity" : 15, "co
{ "_id" : { "month" : 3, "day" : 1, "year" : 2014 }, "totalPrice" : 40, "averageQuantity" : 1.5, "co
```

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("2014-04-04T11:21:39.736
{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.33
```

The following aggregation operation uses the `$group` (page 644) 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 755) system variable to group the documents by authors. The resulting documents must not exceed the `BSON Document Size` (page 940) 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 `https://docs.mongodb.org/manual/tutorial/aggregation-zip-code-data-set` tutorial provides an extensive example of the `$group` (page 644) operator in a common use case.

Additional Resources

- MongoDB Analytics: Learn Aggregation by Example: Exploratory Analytics and Visualization Using Flight Data[19]
- MongoDB for Time Series Data: Analyzing Time Series Data Using the Aggregation Framework and Hadoop[20]
- The Aggregation Framework[21]
- Webinar: Exploring the Aggregation Framework[22]
- Quick Reference Cards[23]

$sample (aggregation)

On this page

- Definition (page 648)
- Behavior (page 648)
- Example (page 648)

Definition

`$sample`

New in version 3.2.

Randomly selects the specified number of documents from its input.

The `$sample` (page 648) stage has the following syntax:

```
{ $sample: { size: <positive integer> } }
```

Behavior In order to get N random documents:

- If N is greater than or equal to 5% of the total documents in the collection, `$sample` (page 648) performs a collection scan, performs a sort, and then select the top N documents. As such, the `$sample` (page 648) stage is subject to the *sort memory restrictions* (page 651).
- If N is less than 5% of the total documents in the collection,
  - If using `https://docs.mongodb.org/manual/core/wiredtiger`, `$sample` (page 648) uses a pseudo-random cursor over the collection to sample N documents.
  - If using `https://docs.mongodb.org/manual/core/mmapv1`, `$sample` (page 648) uses the `_id` index to randomly select N documents.

Warning: `$sample` (page 648) may output the same document more than once in its result set. For more information, see *cursor-isolation*.

Example Given a collection named `users` with the following documents:

[19] http://www.mongodb.com/presentations/mongodb-analytics-learn-aggregation-example-exploratory-analytics-and-visualization?jmp=docs
[20] http://www.mongodb.com/presentations/mongodb-time-series-data-part-2-analyzing-time-series-data-using-aggregation-framework?jmp=docs
[21] https://www.mongodb.com/presentations/aggregation-framework-0?jmp=docs
[22] https://www.mongodb.com/webinar/exploring-the-aggregation-framework?jmp=docs
[23] https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs

```
{ "_id" : 1, "name" : "dave123", "q1" : true, "q2" : true }
{ "_id" : 2, "name" : "dave2", "q1" : false, "q2" : false  }
{ "_id" : 3, "name" : "ahn", "q1" : true, "q2" : true  }
{ "_id" : 4, "name" : "li", "q1" : true, "q2" : false  }
{ "_id" : 5, "name" : "annT", "q1" : false, "q2" : true  }
{ "_id" : 6, "name" : "li", "q1" : true, "q2" : true  }
{ "_id" : 7, "name" : "ty", "q1" : false, "q2" : true  }
```

The following aggregation operation randomly selects `3` documents from the collection:

```
db.users.aggregate(
   [ { $sample: { size: 3 } } ]
)
```

The operation returns three random documents.

$sort (aggregation)

On this page

- Definition (page 649)
- Examples (page 649)
- `$sort` Operator and Memory (page 651)
- `$sort` Operator and Performance (page 651)

Definition

`$sort`

Sorts all input documents and returns them to the pipeline in sorted order.

The `$sort` (page 649) stage has the following prototype form:

```
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
```

`$sort` (page 649) 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 650) for an example.

Examples

Ascending/Descending Sort For the field or fields to sort by, set the sort order to `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
11. Timestamp
12. Regular Expression
13. MaxKey (internal type)

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

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

The comparison treats a non-existent field as it would an empty BSON Object. As such, a sort on the `a` field in documents `{ }` and `{ a: null }` would treat the documents as equivalent in sort order.

With arrays, a less-than comparison or an ascending sort compares the smallest element of arrays, and a greater-than comparison or a descending sort compares the largest element of the arrays. As such, when comparing a field whose value is a single-element array (e.g. `[ 1 ]`) with non-array fields (e.g. `2`), the comparison is between `1` and `2`. A comparison of an empty array (e.g. `[ ]`) treats the empty array as less than `null` or a missing field.

MongoDB sorts `BinData` in the following order:

1. First, the length or size of the data.
2. Then, by the BSON one-byte subtype.
3. Finally, by the data, performing a byte-by-byte comparison.

Metadata Sort Specify in the `{ <sort-key> }` document, a new field name for the computed metadata and specify the `$meta` (page 701) 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 `$text` (page 549) operator to match the documents, and then sorts first by the `"textScore"` metadata and then by descending order of the `posts` field. The specified metadata determines the sort order. For example, the `"textScore"` metadata sorts in descending order. See `$meta` (page 701) for more information on metadata.

`$sort` Operator and Memory

`$sort` + `$limit` Memory Optimization When a `$sort` (page 649) immediately precedes a `$limit` (page 640) in the pipeline, the `$sort` (page 649) 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 649) 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 649) stage has a limit of 100 megabytes of RAM. By default, if the stage exceeds this limit, `$sort` (page 649) will produce an error. To allow for the handling of large datasets, set the `allowDiskUse` option to `true` to enable `$sort` (page 649) operations to write to temporary files. See the `allowDiskUse` option in `db.collection.aggregate()` (page 20) method and the `aggregate` (page 303) command for details.

Changed in version 2.6: The memory limit for `$sort` (page 649) changed from 10 percent of RAM to 100 megabytes of RAM.

`$sort` Operator and Performance `$sort` (page 649) operator can take advantage of an index when placed at the **beginning** of the pipeline or placed **before** the `$project` (page 631), `$unwind` (page 641), and `$group` (page 644) aggregation operators. If `$project` (page 631), `$unwind` (page 641), or `$group` (page 644) occur prior to the `$sort` (page 649) operation, `$sort` (page 649) cannot use any indexes.

See also:

`https://docs.mongodb.org/manual/tutorial/aggregation-zip-code-data-set`, `https://docs.mongodb.org/manual/tutorial/aggregation-with-user-preference-data`

$geoNear (aggregation)

On this page

- Definition (page 651)
- Behavior (page 653)
- Example (page 653)

Definition

`$geoNear`

New in version 2.4.

Outputs documents in order of nearest to farthest from a specified point.

The `$geoNear` (page 651) stage has the following prototype form:

```
{ $geoNear: { <geoNear options> } }
```

The `$geoNear` (page 651) operator accepts a *document* that contains the following `$geoNear` (page 651) options. Specify all distances in the same units as those of the processed documents' coordinate system:

field boolean spherical Required if using a `2dsphere` index. Determines how MongoDB calculates the distance. The default value is `false`.

If `true`, then MongoDB uses spherical geometry to calculate distances in meters if the specified (`near`) point is a GeoJSON point and in radians if the specified (`near`) point is a legacy coordinate pair.

If `false`, then MongoDB uses 2d planar geometry to calculate distance between points.

If using a `2dsphere` index, `spherical` must be `true`.

field number limit Optional. The maximum number of documents to return. The default value is `100`. See also the `num` option.

field number num Optional. 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 Optional. 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 Optional. 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 `$near` (page 565) predicate in the `query` field of the `$geoNear` (page 651) stage.

field number distanceMultiplier Optional. 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 uniqueDocs Optional. 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 575) operator has no impact on results.

:field GeoJSON point, *legacy coordinate pair* near:

The point for which to find the closest documents.

If using a `2dsphere` index, you can specify the point as either a GeoJSON point or legacy coordinate pair.

If using a `2d` index, specify the point as a legacy coordinate pair.

field string distanceField The output field that contains the calculated distance. To specify a field within an embedded document, use *dot notation*.

field string includeLocs Optional. 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 an embedded document, use *dot notation*.

field number minDistance Optional. The minimum distance from the center point that the documents can be. MongoDB limits the results to those documents that fall outside the specified distance from the center point.

Specify the distance in meters for GeoJSON data and in radians for legacy coordinate pairs.

New in version 3.2.

Behavior When using `$geoNear` (page 651), consider that:

- You can only use `$geoNear` (page 651) as the first stage of a pipeline.
- You must include the `distanceField` option. The `distanceField` option specifies the field that will contain the calculated distance.
- The collection must have a `geospatial index`.
- The `$geoNear` (page 651) 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 `$geoNear` (page 651) requires that the collection have a single geospatial index, `$geoNear` (page 651) implicitly uses the indexed field.
- If using a `2dsphere index`, you must specify `spherical: true`.
- You cannot specify a `$near` (page 565) predicate in the `query` field of the `$geoNear` (page 651) stage.

Generally, the options for `$geoNear` (page 651) are similar to the `geoNear` (page 327) command with the following exceptions:

- `distanceField` is a mandatory field for the `$geoNear` (page 651) pipeline operator; the option does not exist in the `geoNear` (page 327) command.
- `includeLocs` accepts a `string` in the `$geoNear` (page 651) pipeline operator and a `boolean` in the `geoNear` (page 327) 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.

Minimum Distance New in version 3.2.

The following example uses the option `minDistance` to specify the minimum distance from the center point that the documents can be. MongoDB limits the results to those documents that fall outside the specified distance from the center point.

```
db.places.aggregate([
    {
      $geoNear: {
         near: { type: "Point", coordinates: [ -73.99279 , 40.719296 ] },
         distanceField: "dist.calculated",
         minDistance: 2,
         query: { type: "public" },
         includeLocs: "dist.location",
         num: 5,
         spherical: true
      }
    }
])
```

$lookup (aggregation)

On this page

- Definition (page 654)
- Example (page 655)

Definition

`$lookup`

New in version 3.2.

Performs a left outer join to an unsharded collection in the *same* database to filter in documents from the "joined" collection for processing. The `$lookup` (page 654) stage does an equality match between a field from the input documents with a field from the documents of the "joined" collection.

To each input document, the `$lookup` (page 654) stage adds a new array field whose elements are the matching documents from the "joined" collection. The `$lookup` (page 654) stage passes these reshaped documents to the next stage.

The `$lookup` (page 654) stage has the following syntax:

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

The $lookup (page 654) takes a document with the following fields:

Field	Description
from	Specifies the collection in the *same* database to perform the join with. The from collection cannot be sharded.
localField	Specifies the field from the documents input to the $lookup (page 654) stage. $lookup (page 654) performs an equality match on the localField to the foreignField from the documents of the from collection. If an input document does not contain the localField, the $lookup (page 654) treats the field as having a value of null for matching purposes.
foreignField	Specifies the field from the documents in the from collection. $lookup (page 654) performs an equality match on the foreignField to the localField from the input documents. If a document in the from collection does not contain the foreignField, the $lookup (page 654) treats the value as null for matching purposes.
as	Specifies the name of the new array field to add to the input documents. The new array field contains the matching documents from the from collection. If the specified name already exists in the input document, the existing field is *overwritten*.

Example A collection orders contains the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 12, "quantity" : 2 }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1 }
{ "_id" : 3  }
```

Another collection inventory contains the following documents:

```
{ "_id" : 1, "sku" : "abc", description: "product 1", "instock" : 120 }
{ "_id" : 2, "sku" : "def", description: "product 2", "instock" : 80 }
{ "_id" : 3, "sku" : "ijk", description: "product 3", "instock" : 60 }
{ "_id" : 4, "sku" : "jkl", description: "product 4", "instock" : 70 }
{ "_id" : 5, "sku": null, description: "Incomplete" }
{ "_id" : 6 }
```

The following aggregation operation on the orders collection joins the documents from orders with the documents from the inventory collection using the fields item from the orders collection and the sku field from the inventory collection:

```
db.orders.aggregate([
    {
      $lookup:
        {
          from: "inventory",
          localField: "item",
          foreignField: "sku",
          as: "inventory_docs"
        }
   }
])
```

The operation returns the following documents:

```
{
  "_id" : 1,
   "item" : "abc",
  "price" : 12,
  "quantity" : 2,
  "inventory_docs" : [
    { "_id" : 1, "sku" : "abc", description: "product 1", "instock" : 120 }
  ]
}
{
  "_id" : 2,
  "item" : "jkl",
  "price" : 20,
  "quantity" : 1,
  "inventory_docs" : [
    { "_id" : 4, "sku" : "jkl", "description" : "product 4", "instock" : 70 }
  ]
}
{
  "_id" : 3,
  "inventory_docs" : [
    { "_id" : 5, "sku" : null, "description" : "Incomplete" },
    { "_id" : 6 }
  ]
}
```

$out (aggregation)

On this page

- Definition (page 656)
- Behaviors (page 657)
- Example (page 657)

New in version 2.6.

Definition

`$out`

Takes the documents returned by the aggregation pipeline and writes them to a specified collection. The `$out` (page 656) operator must be *the last stage* in the pipeline. The `$out` (page 656) operator lets the aggregation framework return result sets of any size.

Changed in version 3.2.0: MongoDB 3.2 added support for *document validation* (page 991). The `bypassDocumentValidation` field enables you to bypass document validation during the `$out` (page 656) phase of the aggregation operation. This lets you insert documents that do not meet the validation requirements. Specify `bypassDocumentValidation` as an option on the aggregation method or command.

The `$out` (page 656) stage has the following prototype form:

```
{ $out: "<output-collection>" }
```

`$out` (page 656) 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 656) operator cannot write results to a `capped collection`.

Behaviors

Create New Collection The `$out` (page 656) 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 `$out` (page 656) operation already exists, then upon completion of the aggregation, the `$out` (page 656) stage atomically replaces the existing collection with the new results collection. The `$out` (page 656) operation does not change any indexes that existed on the previous collection. If the aggregation fails, the `$out` (page 656) 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" ] }
```

$indexStats (aggregation)

On this page

- Definition (page 657)
- Behavior (page 658)
- Example (page 658)

Definition

`$indexStats`

New in version 3.2.

Returns statistics regarding the use of each index for the collection. If running with `access control`, the user must have privileges that include `indexStats` action.

The `$indexStats` (page 657) stage takes an empty document and has the following syntax:

```
{ $indexStats: { } }
```

The return document include the following fields:

Output Field	Description
`name`	Index name.
`key`	Index key specification.
`host`	The hostname and port of process.
`accesses`	Statistics on the index use: •`ops` is the number of operations that used the index. •`since` is the time from which MongoDB gathered the statistics.

Statistics for an index will be reset on `mongod` (page 770) restart or index drop and recreation.

Behavior The statistics reported by the `accesses` field only includes index access driven by user requests. It does not include internal operations like deletion via `https://docs.mongodb.org/manual/core/index-ttl` or chunk split and migration operations.

Example For example, a collection `orders` contains the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 12, "quantity" : 2, "type": "apparel" }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "type": "electronics" }
{ "_id" : 3, "item" : "abc", "price" : 10, "quantity" : 5, "type": "apparel" }
```

Create the following two indexes on the collection:

```
db.orders.createIndex( { item: 1, quantity: 1 } )
db.orders.createIndex( { type: 1, item: 1 } )
```

Run some queries against the collection:

```
db.orders.find( { type: "apparel"} )
db.orders.find( { item: "abc" } ).sort( { quantity: 1 } )
```

To view statistics on the index use on the `orders` collection, run the following aggregation operation:

```
db.orders.aggregate( [ { $indexStats: { } } ] )
```

The operation returns a document that contains usage statistics for each index:

```
{
   "name" : "item_1_quantity_1",
   "key" : {
      "item" : 1,
      "quantity" : 1
   },
   "host" : "examplehost.local:27017",
   "accesses" : {
      "ops" : NumberLong(1),
      "since" : ISODate("2015-10-02T14:31:53.685Z")
   }
}
{
   "name" : "_id_",
   "key" : {
      "_id" : 1
   },
   "host" : "examplehost.local:27017",
   "accesses" : {
      "ops" : NumberLong(0),
      "since" : ISODate("2015-10-02T14:31:32.479Z")
   }
}
{
   "name" : "type_1_item_1",
   "key" : {
      "type" : 1,
      "item" : 1
   },
   "host" : "examplehost.local:27017",
   "accesses" : {
      "ops" : NumberLong(1),
      "since" : ISODate("2015-10-02T14:31:58.321Z")
   }
}
```

Additional Resources

- MongoDB Analytics: Learn Aggregation by Example: Exploratory Analytics and Visualization Using Flight Data[24]
- MongoDB for Time Series Data: Analyzing Time Series Data Using the Aggregation Framework and Hadoop[25]
- The Aggregation Framework[26]
- Webinar: Exploring the Aggregation Framework[27]
- Quick Reference Cards[28]

Expression Operators

These expression operators are available to construct *expressions* (page 747) 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 712) expression or keep the outer array that designates the argument list.

Boolean Operators

Boolean expressions evaluate their argument expressions as booleans and return a boolean as the result.

[24] http://www.mongodb.com/presentations/mongodb-analytics-learn-aggregation-example-exploratory-analytics-and-visualization?jmp=docs
[25] http://www.mongodb.com/presentations/mongodb-time-series-data-part-2-analyzing-time-series-data-using-aggregation-framework?jmp=docs
[26] https://www.mongodb.com/presentations/aggregation-framework-0?jmp=docs
[27] https://www.mongodb.com/webinar/exploring-the-aggregation-framework?jmp=docs
[28] https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs

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 evaluate their argument expressions as booleans and return a boolean as the result.

In addition to the false boolean value, Boolean expression evaluates as false the following: null, 0, and undefined values. The Boolean expression evaluates all other values as true, including non-zero numeric values and arrays.

Name	Description
$and (page 660)	Returns true only when *all* its expressions evaluate to true. Accepts any number of argument expressions.
$or (page 661)	Returns true when *any* of its expressions evaluates to true. Accepts any number of argument expressions.
$not (page 662)	Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression.

$and (aggregation)

On this page

- Definition (page 660)
- Behavior (page 660)
- Example (page 660)

Definition

$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 660) returns false.

$and (page 660) has the following syntax:

```
{ $and: [ <expression1>, <expression2>, ... ] }
```

For more information on expressions, see *Expressions* (page 747).

Behavior $and (page 660) uses short-circuit logic: the operation stops evaluation after encountering the first false expression.

In addition to the false boolean value, $and (page 660) evaluates as false the following: null, 0, and undefined values. The $and (page 660) 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 660) 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 }
```

$or (aggregation)

On this page

- Definition (page 661)
- Behavior (page 661)
- Example (page 662)

Definition

`$or`

Evaluates one or more expressions and returns `true` if *any* of the expressions are `true`. Otherwise, `$or` (page 661) returns `false`.

`$or` (page 661) has the following syntax:

```
{ $or: [ <expression1>, <expression2>, ... ] }
```

For more information on expressions, see *Expressions* (page 747).

Behavior `$or` (page 661) uses short-circuit logic: the operation stops evaluation after encountering the first `true` expression.

In addition to the `false` boolean value, `$or` (page 661) evaluates as `false` the following: `null`, `0`, and `undefined` values. The `$or` (page 661) 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 661) 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 }
```

$not (aggregation)

On this page

- Definition (page 662)
- Behavior (page 663)
- Example (page 663)

Definition

`$not`

Evaluates a boolean and returns the opposite boolean value; i.e. when passed an expression that evaluates to `true`, `$not` (page 662) returns `false`; when passed an expression that evaluates to `false`, `$not` (page 662) returns `true`.

`$not` (page 662) has the following syntax:

```
{ $not: [ <expression> ] }
```

For more information on expressions, see *Expressions* (page 747).

Behavior In addition to the `false` boolean value, `$not` (page 662) evaluates as `false` the following: `null`, `0`, and `undefined` values. The `$not` (page 662) 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 661) operator to determine if `qty` is not greater than `250`:

```
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 }
```

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
`$setEquals` (page 664)	Returns `true` if the input sets have the same distinct elements. Accepts two or more argument expressions.
`$setIntersection` (page 665)	Returns a set with elements that appear in *all* of the input sets. Accepts any number of argument expressions.
`$setUnion` (page 666)	Returns a set with elements that appear in *any* of the input sets. Accepts any number of argument expressions.
`$setDifference` (page 668)	Returns a set with elements that appear in the first set but not in the second set; i.e. performs a relative complement[29] of the second set relative to the first. Accepts exactly two argument expressions.
`$setIsSubset` (page 669)	Returns `true` if all elements of the first set appear in the second set, including when the first set equals the second set; i.e. not a strict subset[30]. Accepts exactly two argument expressions.
`$anyElementTrue` (page 670)	Returns `true` if *any* elements of a set evaluate to `true`; otherwise, returns `false`. Accepts a single argument expression.
`$allElementsTrue` (page 671)	Returns `true` if *no* element of a set evaluates to `false`, otherwise, returns `false`. Accepts a single argument expression.

$setEquals (aggregation)

On this page

- Definition (page 664)
- Behavior (page 664)
- Example (page 665)

Definition

`$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 664) has the following syntax:

```
{ $setEquals: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 747) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 747).

Behavior `$setEquals` (page 664) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setEquals` (page 664) ignores the duplicate entries. `$setEquals` (page 664) ignores the order of the elements.

If a set contains a nested array element, `$setEquals` (page 664) 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`

[29] http://en.wikipedia.org/wiki/Complement_(set_theory)

[30] http://en.wikipedia.org/wiki/Subset

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 664) 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)

On this page

- Definition (page 665)
- Behavior (page 665)
- Example (page 666)

Definition

`$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 665) has the following syntax:

```
{ $setIntersection: [ <array1>, <array2>, ... ] }
```

The arguments can be any valid *expression* (page 747) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 747).

Behavior `$setIntersection` (page 665) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setIntersection` (page 665) ignores the duplicate entries. `$setIntersection` (page 665) ignores the order of the elements.

`$setIntersection` (page 665) 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 665) 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 665) 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" : [ ] }
```

$setUnion (aggregation)

On this page

- Definition (page 666)
- Behavior (page 667)
- Example (page 667)

Definition

`$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 666) has the following syntax:

```
{ $setUnion: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 747) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 747).

Behavior `$setUnion` (page 666) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setUnion` (page 666) ignores the duplicate entries. `$setUnion` (page 666) ignores the order of the elements.

`$setUnion` (page 666) 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 666) 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 666) 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" ] }
```

$setDifference (aggregation)

On this page

- Definition (page 668)
- Behavior (page 668)
- Example (page 668)

Definition

`$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[31] of the second set relative to the first.

`$setDifference` (page 668) has the following syntax:

```
{ $setDifference: [ <expression1>, <expression2> ] }
```

The arguments can be any valid *expression* (page 747) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 747).

Behavior

`$setDifference` (page 668) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setDifference` (page 668) ignores the duplicate entries. `$setDifference` (page 668) ignores the order of the elements.

`$setDifference` (page 668) 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 668) 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 668) 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 } }
   ]
)
```

[31] http://en.wikipedia.org/wiki/Complement_(set_theory)

The operation returns the following results:

```
{ "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" ] }
```

$setIsSubset (aggregation)

On this page

- Definition (page 669)
- Behavior (page 669)
- Example (page 669)

Definition

`$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 669) has the following syntax:

```
{ $setIsSubset: [ <expression1>, <expression2> ] }
```

The arguments can be any valid *expression* (page 747) as long as they each resolve to an array. For more information on expressions, see *Expressions* (page 747).

Behavior `$setIsSubset` (page 669) performs set operation on arrays, treating arrays as sets. If an array contains duplicate entries, `$setIsSubset` (page 669) ignores the duplicate entries. `$setIsSubset` (page 669) ignores the order of the elements.

If a set contains a nested array element, `$setIsSubset` (page 669) 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 669) 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 }
```

$anyElementTrue (aggregation)

On this page

- Definition (page 670)
- Behavior (page 670)
- Example (page 671)

Definition

`$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 670) 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 747).

Behavior If a set contains a nested array element, `$anyElementTrue` (page 670) does *not* descend into the nested array but evaluates the array at top-level.

In addition to the `false` boolean value, `$anyElementTrue` (page 670) evaluates as `false` the following: `null`, `0`, and `undefined` values. The `$anyElementTrue` (page 670) 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 670) 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 }
```

$allElementsTrue (aggregation)

On this page

- Definition (page 671)
- Behavior (page 672)
- Example (page 672)

Definition

`$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`.

`$allElementsTrue` (page 671) 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 747).

Behavior If a set contains a nested array element, `$allElementsTrue` (page 671) does *not* descend into the nested array but evaluates the array at top-level.

In addition to the `false` boolean value, `$allElementsTrue` (page 671) evaluates as `false` the following: `null`, `0`, and `undefined` values. The `$allElementsTrue` (page 671) 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 671) 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 }
```

Comparison Operators

Comparison expressions return a boolean except for `$cmp` (page 673) 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 673) 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
`$cmp` (page 673)	Returns: `0` if the two values are equivalent, `1` if the first value is greater than the second, and `-1` if the first value is less than the second.
`$eq` (page 674)	Returns `true` if the values are equivalent.
`$gt` (page 675)	Returns `true` if the first value is greater than the second.
`$gte` (page 676)	Returns `true` if the first value is greater than or equal to the second.
`$lt` (page 677)	Returns `true` if the first value is less than the second.
`$lte` (page 678)	Returns `true` if the first value is less than or equal to the second.
`$ne` (page 679)	Returns `true` if the values are *not* equivalent.

$cmp (aggregation)

On this page

- Definition (page 673)
- Example (page 673)

Definition

`$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 673) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$cmp` (page 673) has the following syntax:

```
{ $cmp: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 747).

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 673) 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)

On this page

- Definition (page 674)
- Example (page 674)

Definition

`$eq`

Compares two values and returns:

- `true` when the values are equivalent.
- `false` when the values are **not** equivalent.

The `$eq` (page 674) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$eq` (page 674) has the following syntax:

```
{ $eq: [ <expression1>, <expression2> ] }
```

The arguments can be any valid *expression* (page 747). For more information on expressions, see *Expressions* (page 747).

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 674) 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)

On this page

- Definition (page 675)
- Example (page 675)

Definition

`$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 675) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$gt` (page 675) has the following syntax:

```
{ $gt: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 747).

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 675) 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)

On this page

- Definition (page 676)
- Example (page 676)

Definition

$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 676) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$gte` (page 676) has the following syntax:

```
{ $gte: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 747).

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 676) 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)

On this page

- Definition (page 677)
- Example (page 677)

Definition

$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 677) compares both value and type, using the *specified BSON comparison order* for values of different types.

$lt (page 677) has the following syntax:

```
{ $lt: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 747).

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 677) 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)

On this page

- Definition (page 678)
- Example (page 678)

Definition

`$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 678) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$lte` (page 678) has the following syntax:

```
{ $lte: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 747).

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 678) 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)

On this page

- Definition (page 679)
- Example (page 679)

Definition

$ne

Compares two values and returns:

- `true` when the values are not equivalent.
- `false` when the values are equivalent.

The `$lte` (page 678) compares both value and type, using the *specified BSON comparison order* for values of different types.

`$ne` (page 679) has the following syntax:

```
{ $ne: [ <expression1>, <expression2> ] }
```

For more information on expressions, see *Expressions* (page 747).

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 679) 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
`$abs` (page 681)	Returns the absolute value of a number.
`$add` (page 682)	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.
`$ceil` (page 683)	Returns the smallest integer greater than or equal to the specified number.
`$divide` (page 684)	Returns the result of dividing the first number by the second. Accepts two argument expressions.
`$exp` (page 685)	Raises *e* to the specified exponent.
`$floor` (page 686)	Returns the largest integer less than or equal to the specified number.
`$ln` (page 687)	Calculates the natural log of a number.
`$log` (page 687)	Calculates the log of a number in the specified base.
`$log10` (page 689)	Calculates the log base 10 of a number.
`$mod` (page 689)	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
`$multiply` (page 690)	Multiplies numbers to return the product. Accepts any number of argument expressions.
`$pow` (page 691)	Raises a number to the specified exponent.
`$sqrt` (page 692)	Calculates the square root.
`$subtract` (page 693)	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.
`$trunc` (page 695)	Truncates a number to its integer.

$abs (aggregation)

On this page

- Definition (page 681)
- Behavior (page 682)
- Example (page 682)

Definition

`$abs`

New in version 3.2.

Returns the absolute value of a number.

`$abs` (page 681) has the following syntax:

```
{ $abs: <number> }
```

The `<number>` expression can be any valid *expression* (page 747) as long as it resolves to a number. For more information on expressions, see *Expressions* (page 747).

Behavior If the argument resolves to a value of `null` or refers to a field that is missing, `$abs` returns `null`. If the argument resolves to `NaN`, `$abs` returns `NaN`.

If the argument resolves to a value of `null` or refers to a field that is missing, `$abs` (page 681) returns `null`. If the argument resolves to `NaN`, `$abs` (page 681) returns `NaN`.

Example	Results
`{ $abs: -1 }`	`1`
`{ $abs: 1 }`	`1`
`{ $abs: null }`	`null`

Example A collection `ratings` contains the following documents:

```
{ _id: 1, start: 5, end: 8 }
{ _id: 2, start: 4, end: 4 }
{ _id: 3, start: 9, end: 7 }
{ _id: 4, start: 6, end: 7 }
```

The following example calculates the magnitude of difference between the `start` and `end` ratings:

```
db.ratings.aggregate([
   {
     $project: { delta: { $abs: { $subtract: [ "$start", "$end" ] } } }
   }
])
```

The operation returns the following results:

```
{ "_id" : 1, "delta" : 3 }
{ "_id" : 2, "delta" : 0 }
{ "_id" : 3, "delta" : 2 }
{ "_id" : 4, "delta" : 1 }
```

$add (aggregation)

On this page

- Definition (page 682)
- Examples (page 683)

Definition

`$add`

Adds numbers together or adds numbers and a date. If one of the arguments is a date, `$add` (page 682) treats the other arguments as milliseconds to add to the date.

The `$add` (page 682) expression has the following syntax:

```
{ $add: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 747) as long as they resolve to either all numbers or to numbers and a date. For more information on expressions, see *Expressions* (page 747).

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 `$add` (page 682) expression in the `$project` (page 631) 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 682) 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") }
```

$ceil (aggregation)

On this page

- Definition (page 683)
- Behavior (page 684)
- Example (page 684)

Definition

`$ceil`

New in version 3.2.

Returns the smallest integer greater than or equal to the specified number.

`$ceil` (page 683) has the following syntax:

```
{ $ceil: <number> }
```

The `<number>` expression can be any valid *expression* (page 747) as long as it resolves to a number. For more information on expressions, see *Expressions* (page 747).

Behavior If the argument resolves to a value of `null` or refers to a field that is missing, `$ceil` returns `null`. If the argument resolves to `NaN`, `$ceil` returns `NaN`.

Example	Results
`{ $ceil:  1 }`	`1`
`{ $ceil:  7.80 }`	`8`
`{ $ceil:  -2.8 }`	`-2`

Example A collection named `samples` contains the following documents:

```
{ _id: 1, value: 9.25 }
{ _id: 2, value: 8.73 }
{ _id: 3, value: 4.32 }
{ _id: 4, value: -5.34 }
```

The following example returns both the original value and the ceiling value:

```
db.samples.aggregate([
   { $project: { value: 1, ceilingValue: { $ceil: "$value" } } }
])
```

The operation returns the following results:

```
{ "_id" : 1, "value" : 9.25, "ceilingValue" : 10 }
{ "_id" : 2, "value" : 8.73, "ceilingValue" : 9 }
{ "_id" : 3, "value" : 4.32, "ceilingValue" : 5 }
{ "_id" : 4, "value" : -5.34, "ceilingValue" : -5 }
```

$divide (aggregation)

On this page

- Definition (page 684)
- Examples (page 684)

Definition

`$divide`

Divides one number by another and returns the result. Pass the arguments to `$divide` (page 684) in an array.

The `$divide` (page 684) 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 747) as long as the resolve to numbers. For more information on expressions, see *Expressions* (page 747).

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 684) 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 }
```

$exp (aggregation)

On this page

- Definition (page 685)
- Behavior (page 685)
- Example (page 685)

Definition

`$exp`

New in version 3.2.

Raises Euler's number (i.e. *e*) to the specified exponent and returns the result.

`$exp` (page 685) has the following syntax:

```
{ $exp: <exponent> }
```

The `<exponent>` expression can be any valid *expression* (page 747) as long as it resolves to a number. For more information on expressions, see *Expressions* (page 747).

Behavior If the argument resolves to a value of `null` or refers to a field that is missing, `$exp` returns `null`. If the argument resolves to `NaN`, `$exp` returns `NaN`.

Example	Results
`{ $exp:  0 }`	1
`{ $exp:  2 }`	7.38905609893065
`{ $exp:  -2 }`	0.1353352832366127

Example A collection named `accounts` contains the following documents:

```
{ _id: 1, rate: .08, pv: 10000 }
{ _id: 2, rate: .0825, pv: 250000 }
{ _id: 3, rate: .0425, pv: 1000 }
```

The following example calculates the effective interest rate for continuous compounding:

```
db.accounts.aggregate( [ { $project: { effectiveRate: { $subtract: [ { $exp: "$rate"}, 1 ] } } } ] )
```

The operation returns the following results:

```
{ "_id" : 1, "effectiveRate" : 0.08328706767495864 }
{ "_id" : 2, "effectiveRate" : 0.08599867343905654 }
{ "_id" : 3, "effectiveRate" : 0.04341605637367807 }
```

$floor (aggregation)

On this page

- Definition (page 686)
- Behavior (page 686)
- Example (page 686)

Definition

$floor

New in version 3.2.

Returns the largest integer less than or equal to the specified number.

$floor (page 686) has the following syntax:

```
{ $floor: <number> }
```

The <number> expression can be any valid *expression* (page 747) as long as it resolves to a number. For more information on expressions, see *Expressions* (page 747).

Behavior If the argument resolves to a value of null or refers to a field that is missing, $floor returns null. If the argument resolves to NaN, $floor returns NaN.

Example	Results
`{ $floor: 1 }`	1
`{ $floor: 7.80 }`	7
`{ $floor: -2.8 }`	-3

Example A collection named samples contains the following documents:

```
{ _id: 1, value: 9.25 }
{ _id: 2, value: 8.73 }
{ _id: 3, value: 4.32 }
{ _id: 4, value: -5.34 }
```

The following example returns both the original value and the floor value:

```
db.samples.aggregate([
   { $project: { value: 1, floorValue: { $floor: "$value" } } }
])
```

The operation returns the following results:

```
{ "_id" : 1, "value" : 9.25, "floorValue" : 9 }
{ "_id" : 2, "value" : 8.73, "floorValue" : 8 }
{ "_id" : 3, "value" : 4.32, "floorValue" : 4 }
{ "_id" : 4, "value" : -5.34, "floorValue" : -6 }
```

$ln (aggregation)

On this page

- Definition (page 687)
- Behavior (page 687)
- Example (page 687)

Definition

$ln

New in version 3.2.

Calculates the natural logarithm *ln* (i.e $\log_e$) of a number and returns the result as a double.

`$ln` (page 687) has the following syntax:

```
{ $ln: <number> }
```

The `<number>` expression can be any valid *expression* (page 747) as long as it resolves to a non-negative number. For more information on expressions, see *Expressions* (page 747).

`$ln` (page 687) is equivalent to `$log: [ <number>, Math.E ]` expression, where `Math.E` is a JavaScript representation for Euler's number *e*.

Behavior If the argument resolves to a value of `null` or refers to a field that is missing, `$ln` returns `null`. If the argument resolves to `NaN`, `$ln` returns `NaN`.

Example	Results
`{ $ln: 1 }`	`0`
`{ $ln: Math.E }` where `Math.E` is a JavaScript representation for *e*.	`1`
`{ $ln: 10 }`	`2.302585092994046`

Example A collection `sales` contains the following documents:

```
{ _id: 1, year: "2000", sales: 8700000 }
{ _id: 2, year: "2005", sales: 5000000 }
{ _id: 3, year: "2010", sales: 6250000 }
```

The following example transforms the `sales` data:

```
db.sales.aggregate( [ { $project: { x: "$year", y: { $ln: "$sales"  } } } ] )
```

The operation returns the following results:

```
{ "_id" : 1, "x" : "2000", "y" : 15.978833583624812 }
{ "_id" : 2, "x" : "2005", "y" : 15.424948470398375 }
{ "_id" : 3, "x" : "2010", "y" : 15.648092021712584 }
```

See also:

`$log` (page 687)

$log (aggregation)

On this page

- Definition (page 687)
- Behavior (page 688)
- Example (page 688)

Definition

$log

New in version 3.2.

Calculates the log of a number in the specified base and returns the result as a double.

`$log` (page 687) has the following syntax:

```
{ $log: [ <number>, <base> ] }
```

The `<number>` expression can be any valid *expression* (page 747) as long as it resolves to a non-negative number.

The `<base>` expression can be any valid *expression* (page 747) as long as it resolves to a positive number greater than `1`.

For more information on expressions, see *Expressions* (page 747).

Behavior If either argument resolves to a value of `null` or refers to a field that is missing, `$log` returns `null`. If either argument resolves to `NaN`, `$log` returns `NaN`.

Example	Results
`{ $log: [ 100, 10 ] }`	`2`
`{ $log: [ 100, Math.E ] }` where `Math.E` is a JavaScript representation for *e*.	`4.605170185988092`

Example A collection `examples` contains the following documents:

```
{ _id: 1, positiveInt: 5 }
{ _id: 2, positiveInt: 2 }
{ _id: 3, positiveInt: 23 }
{ _id: 4, positiveInt: 10 }
```

The following example uses $\log_2$ in its calculation to determine the number of bits required to represent the value of `positiveInt`.

```
db.examples.aggregate([
   { $project: { bitsNeeded:
      {
         $floor: { $add: [ 1, { $log: [ "$positiveInt", 2 ] } ] } } }
      }
])
```

The operation returns the following results:

```
{ "_id" : 1, "bitsNeeded" : 3 }
{ "_id" : 2, "bitsNeeded" : 2 }
{ "_id" : 3, "bitsNeeded" : 5 }
{ "_id" : 4, "bitsNeeded" : 4 }
```

See also:

`$log10` (page 689) and `$ln` (page 687)

$log10 (aggregation)

On this page

- Definition (page 689)
- Behavior (page 689)
- Example (page 689)

Definition

`$log10`

New in version 3.2.

Calculates the log base 10 of a number and returns the result as a double.

`$log10` (page 689) has the following syntax:

```
{ $log10: <number> }
```

The `<number>` expression can be any valid *expression* (page 747) as long as it resolves to a non-negative number. For more information on expressions, see *Expressions* (page 747).

`$log10` (page 689) is equivalent to `$log: [ <number>, 10 ]` expression.

Behavior If the argument resolves to a value of `null` or refers to a field that is missing, `$log10` returns `null`. If the argument resolves to `NaN`, `$log10` returns `NaN`.

Example	Results
`{ $log10: 1 }`	0
`{ $log10: 10 }`	1
`{ $log10: 100 }`	2
`{ $log10: 1000 }`	3

Example A collection `samples` contains the following documents:

```
{ _id: 1, H3O: 0.0025 }
{ _id: 2, H3O: 0.001 }
{ _id: 3, H3O: 0.02 }
```

The following example calculates the pH value of the samples:

```
db.samples.aggregate( [
   { $project: { pH: { $multiply: [ -1, { $log10: "$H3O" } ] } } }
] )
```

The operation returns the following results:

```
{ "_id" : 1, "pH" : 2.6020599913279625 }
{ "_id" : 2, "pH" : 3 }
{ "_id" : 3, "pH" : 1.6989700043360187 }
```

See also:

`$log` (page 687)

$mod (aggregation)

On this page

- Definition (page 689)
- Example (page 690)

Definition

`$mod`

Divides one number by another and returns the *remainder*.

The `$mod` (page 689) 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 747) as long as they resolve to numbers. For more information on expressions, see *Expressions* (page 747).

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 689) 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)

On this page

- Definition (page 690)
- Example (page 690)

Definition

`$multiply`

Multiplies numbers together and returns the result. Pass the arguments to `$multiply` (page 690) in an array.

The `$multiply` (page 690) expression has the following syntax:

```
{ $multiply: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 747) as long as they resolve to numbers. For more information on expressions, see *Expressions* (page 747).

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 `$multiply` (page 690) expression in the `$project` (page 631) 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 }
```

$pow (aggregation)

On this page

- Definition (page 691)
- Behavior (page 691)
- Example (page 691)

Definition

$pow

New in version 3.2.

Raises a number to the specified exponent and returns the result. `$pow` (page 691) has the following syntax:

```
{ $pow: [ <number>, <exponent> ] }
```

The `<number>` expression can be any valid *expression* (page 747) as long as it resolves to a non-negative number.

The `<exponent>` expression can be any valid *expression* (page 747) as long as it resolves to a number.

You cannot raise `0` to a negative exponent.

Behavior The result will have the same type as the input except when it cannot be represented accurately in that type. In these cases:

- A 32-bit integer will be converted to a 64-bit integer if the result is representable as a 64-bit integer.
- A 32-bit integer will be converted to a double if the result is not representable as a 64-bit integer.
- A 64-bit integer will be converted to double if the result is not representable as a 64-bit integer.

If either argument resolves to a value of `null` or refers to a field that is missing, `$pow` returns `null`. If either argument resolves to `NaN`, `$pow` returns `NaN`.

Example	Results
`{ $pow: [ 5, 0 ] }`	1
`{ $pow: [ 5, 2 ] }`	25
`{ $pow: [ 5, -2 ] }`	0.04

Example A collection named `quizzes` contains the following documents:

```
{
   "_id" : 1,
   "scores" : [
      {
         "name" : "dave123",
         "score" : 85
      },
      {
         "name" : "dave2",
         "score" : 90
      },
      {
         "name" : "ahn",
         "score" : 71
      }
   ]
}
{
   "_id" : 2,
   "scores" : [
      {
         "name" : "li",
         "quiz" : 2,
         "score" : 96
      },
      {
         "name" : "annT",
         "score" : 77
      },
      {
         "name" : "ty",
         "score" : 82
      }
   ]
}
```

The following example calculates the variance for each quiz:

```
db.quizzes.aggregate([
   { $project: { variance: { $pow: [ { $stdDevPop: "$scores.score" }, 2 ] } } }
])
```

The operation returns the following results:

```
{ "_id" : 1, "variance" : 64.66666666666667 }
{ "_id" : 2, "variance" : 64.66666666666667 }
```

$sqrt (aggregation)

On this page

- Definition (page 692)
- Behavior (page 693)
- Example (page 693)

Definition

`$sqrt`

New in version 3.2.

Calculates the square root of a positive number and returns the result as a double.

$sqrt (page 692) has the following syntax:

```
{ $sqrt: <number> }
```

The argument can be any valid *expression* (page 747) as long as it resolves to a *non-negative* number. For more information on expressions, see *Expressions* (page 747).

Behavior If the argument resolves to a value of `null` or refers to a field that is missing, `$sqrt` returns `null`. If the argument resolves to `NaN`, `$sqrt` returns `NaN`.

`$sqrt` (page 692) errors on negative numbers.

Example	Results
`{ $sqrt: 25 }`	`5`
`{ $sqrt: 30 }`	`5.477225575051661`
`{ $sqrt: null }`	`null`

Example A collection `points` contains the following documents:

```
{ _id: 1, p1: { x: 5, y: 8 }, p2: { x: 0, y: 5} }
{ _id: 2, p1: { x: -2, y: 1 }, p2: { x: 1, y: 5} }
{ _id: 3, p1: { x: 4, y: 4 }, p2: { x: 4, y: 0} }
```

The following example uses `$sqrt` (page 692) to calculate the distance between `p1` and `p2`:

```
db.points.aggregate([
   {
     $project: {
        distance: {
           $sqrt: {
               $add: [
                  { $pow: [ { $subtract: [ "$p2.y", "$p1.y" ] }, 2 ] },
                  { $pow: [ { $subtract: [ "$p2.x", "$p1.x" ] }, 2 ] }
               ]
           }
        }
     }
   }
])
```

The operation returns the following results:

```
{ "_id" : 1, "distance" : 5.830951894845301 }
{ "_id" : 2, "distance" : 5 }
{ "_id" : 3, "distance" : 4 }
```

$subtract (aggregation)

On this page

- Definition (page 693)
- Examples (page 694)

Definition

$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 693) 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 747) 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 747).

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 693) 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 `$subtract` (page 693) expression to subtract `$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 `$subtract` (page 693) expression to subtract 5 * 60 * 1000 milliseconds (5 minutes) from the "$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") }
```

$trunc (aggregation)

On this page

- Definition (page 695)
- Behavior (page 695)
- Example (page 695)

Definition

$trunc

New in version 3.2.

Truncates a number to its integer.

$trunc (page 695) has the following syntax:

```
{ $trunc: <number> }
```

The <number> expression can be any valid *expression* (page 747) as long as it resolves to a number. For more information on expressions, see *Expressions* (page 747).

Behavior If the argument resolves to a value of null or refers to a field that is missing, $trunc returns null. If the argument resolves to NaN, $trunc returns NaN.

Example	Results
{ $trunc: 0 }	0
{ $trunc: 7.80 }	7
{ $trunc: -2.3 }	-2

Example A collection named samples contains the following documents:

```
{ _id: 1, value: 9.25 }
{ _id: 2, value: 8.73 }
{ _id: 3, value: 4.32 }
{ _id: 4, value: -5.34 }
```

The following example returns both the original value and the truncated value:

```
db.samples.aggregate([
   { $project: { value: 1, truncatedValue: { $trunc: "$value" } } }
])
```

The operation returns the following results:

```
{ "_id" : 1, "value" : 9.25, "truncatedValue" : 9 }
{ "_id" : 2, "value" : 8.73, "truncatedValue" : 8 }
{ "_id" : 3, "value" : 4.32, "truncatedValue" : 4 }
{ "_id" : 4, "value" : -5.34, "truncatedValue" : -5 }
```

String Operators

String expressions, with the exception of $concat (page 696), only have a well-defined behavior for strings of ASCII characters.

$concat (page 696) behavior is well-defined regardless of the characters used.

String Aggregation Operators String expressions, with the exception of $concat (page 696), only have a well-defined behavior for strings of ASCII characters.

$concat (page 696) behavior is well-defined regardless of the characters used.

Name	Description
`$concat` (page 696)	Concatenates any number of strings.
`$substr` (page 697)	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.
`$toLower` (page 698)	Converts a string to lowercase. Accepts a single argument expression.
`$toUpper` (page 699)	Converts a string to uppercase. Accepts a single argument expression.
`$strcasecmp` (page 699)	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.

$concat (aggregation)

On this page

- Definition (page 696)
- Examples (page 696)

Definition

`$concat`

New in version 2.4.

Concatenates strings and returns the concatenated string.

`$concat` (page 696) has the following syntax:

```
{ $concat: [ <expression1>, <expression2>, ... ] }
```

The arguments can be any valid *expression* (page 747) as long as they resolve to strings. For more information on expressions, see *Expressions* (page 747).

If the argument resolves to a value of `null` or refers to a field that is missing, `$concat` (page 696) 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 696) 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 }
```

$substr (aggregation)

On this page

- Definition (page 697)
- Behavior (page 697)
- Example (page 697)

Definition

`$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 697) has the following syntax:

```
{ $substr: [ <string>, <start>, <length> ] }
```

The arguments can be any valid *expression* (page 747) 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 747).

Behavior

If `<start>` is a negative number, `$substr` (page 697) returns an empty string `""`.

If `<length>` is a negative number, `$substr` (page 697) returns a substring that starts at the specified index and includes the rest of the string.

`$substr` (page 697) 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 697) 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)

On this page

- Definition (page 698)
- Behavior (page 698)
- Example (page 698)

Definition

`$toLower`

Converts a string to lowercase, returning the result.

`$toLower` (page 698) has the following syntax:

```
{ $toLower: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a string. For more information on expressions, see *Expressions* (page 747).

If the argument resolves to null, `$toLower` (page 698) returns an empty string `""`.

Behavior

`$toLower` (page 698) 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 698) 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)

On this page

- Definition (page 699)
- Behavior (page 699)
- Example (page 699)

Definition

`$toUpper`

Converts a string to uppercase, returning the result.

`$toUpper` (page 699) has the following syntax:

```
{ $toUpper: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a string. For more information on expressions, see *Expressions* (page 747).

If the argument resolves to null, `$toLower` (page 698) returns an empty string `""`.

Behavior `$toUpper` (page 699) 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 699) 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" : "" }
```

$strcasecmp (aggregation)	**On this page**
	• Definition (page 699) • Behavior (page 700) • Example (page 700)

Definition

`$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 699) has the following syntax:

```
{ $strcasecmp: [ <expression1>, <expression2> ] }
```

The arguments can be any valid *expression* (page 747) as long as they resolve to strings. For more information on expressions, see *Expressions* (page 747).

Behavior `$strcasecmp` (page 699) only has a well-defined behavior for strings of ASCII characters.

For a case sensitive comparison, see `$cmp` (page 673).

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 699) operator to perform case-insensitive comparison of the `quarter` field value to the string `"13q4"`:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            comparisonResult: { $strcasecmp: [ "$quarter", "13q4" ] }
          }
      }
   ]
)
```

The operation returns the following results:

```
{ "_id" : 1, "item" : "ABC1", "comparisonResult" : -1 }
{ "_id" : 2, "item" : "ABC2", "comparisonResult" : 0 }
{ "_id" : 3, "item" : "XYZ1", "comparisonResult" : 1 }
```

Text Search Operators

Text Search Aggregation Operators

Name	Description
`$meta` (page 701)	Access text search metadata.

$meta (aggregation)

On this page

- Definition (page 701)
- Behavior (page 701)
- Examples (page 701)

Definition

$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 701) expression has the following syntax:

```
{ $meta: <metaDataKeyword> }
```

The $meta (page 701) expression can specify the following keyword as the <metaDataKeyword>:

Keyword	Description	Sort Order
"textScore"	Returns the score associated with the corresponding $text (page 549) query for each matching document. The text score signifies how well the document matched the *search term or terms* (page 551). If not used in conjunction with a $text (page 549) query, returns a score of null.	Descending

Behavior The { $meta: "textScore" } expression is the only *expression* (page 747) that the $sort (page 649) 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 635) stage with a $text (page 549) 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 701) 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 https://docs.mongodb.org/manual/tutorial/text-search-in-aggregation.

Array Operators

Array Aggregation Operators

Name	Description
$arrayElemAt (page 702)	Returns the element at the specified array index.
$concatArrays (page 703)	Concatenates arrays to return the concatenated array.
$filter (page 704)	Selects a subset of the array to return an array with only the elements t filter condition.
$isArray (page 706)	Determines if the operand is an array. Returns a boolean.
$size (page 707)	Returns the number of elements in the array. Accepts a single expressi argument.
$slice (page 707)	Returns a subset of an array.

$arrayElemAt (aggregation)

On this page

- Definition (page 702)
- Behavior (page 702)
- Example (page 702)

Definition

`$arrayElemAt`

New in version 3.2.

Returns the element at the specified array index.

$arrayElemAt (page 702) has the following syntax:

```
{ $arrayElemAt: [ <array>, <idx> ] }
```

The `<array>` expression can be any valid *expression* (page 747) as long as it resolves to an array.

The `<idx>` expression can be any valid *expression* (page 747) as long as it resolves to an integer.

- If positive, $arrayElemAt (page 702) returns the element at the `idx` position, counting from the start of the array.
- If negative, $arrayElemAt (page 702) returns the element at the `idx` position, counting from the end of the array.

If the `idx` exceeds the array bounds, $arrayElemAt (page 702) does not return any result.

For more information on expressions, see *Expressions* (page 747).

Behavior For more information on expressions, see *Expressions* (page 747).

Example	Results
`{ $arrayElemAt: [ [ 1, 2, 3 ], 0 ] }`	1
`{ $arrayElemAt: [ [ 1, 2, 3 ], -2 ] }`	2
`{ $arrayElemAt: [ [ 1, 2, 3 ], 15 ] }`	

Example A collection named `users` contains the following documents:

```
{ "_id" : 1, "name" : "dave123", favorites: [ "chocolate", "cake", "butter", "apples" ] }
{ "_id" : 2, "name" : "li", favorites: [ "apples", "pudding", "pie" ] }
{ "_id" : 3, "name" : "ahn", favorites: [ "pears", "pecans", "chocolate", "cherries" ] }
{ "_id" : 4, "name" : "ty", favorites: [ "ice cream" ] }
```

The following example returns the first and last element in the `favorites` array:

```
db.users.aggregate([
   {
     $project:
      {
         name: 1,
         first: { $arrayElemAt: [ "$favorites", 0 ] },
         last: { $arrayElemAt: [ "$favorites", -1 ] }
      }
   }
])
```

The operation returns the following results:

```
{ "_id" : 1, "name" : "dave123", "first" : "chocolate", "last" : "apples" }
{ "_id" : 2, "name" : "li", "first" : "apples", "last" : "pie" }
{ "_id" : 3, "name" : "ahn", "first" : "pears", "last" : "cherries" }
{ "_id" : 4, "name" : "ty", "first" : "ice cream", "last" : "ice cream" }
```

$concatArrays (aggregation)

On this page

- Definition (page 703)
- Behavior (page 703)
- Example (page 703)

Definition

`$concatArrays`

New in version 3.2.

Concatenates arrays to return the concatenated array.

`$concatArrays` (page 703) has the following syntax:

```
{ $concatArrays: [ <array1>, <array2>, ... ] }
```

The `<array>` expressions can be any valid *expression* (page 747) as long as they resolve to an array. For more information on expressions, see *Expressions* (page 747).

If any argument resolves to a value of `null` or refers to a field that is missing, `$concatArrays` (page 703) returns `null`.

Behavior

Example	Results
`{ $concatArrays: [ [ "hello", " "], [ "world" ] ] }`	`[ "hello", " ", "world" ]`
`{ $concatArrays: [ [ "hello", " "], [ [ "world" ], "again"] ] }`	`[ "hello", " ", [ "world" ], "again" ]`

Example

A collection named `warehouses` contains the following documents:

```
{ "_id" : 1, instock: [ "chocolate" ], ordered: [ "butter", "apples" ] }
{ "_id" : 2, instock: [ "apples", "pudding", "pie" ] }
{ "_id" : 3, instock: [ "pears", "pecans"], ordered: [ "cherries" ] }
{ "_id" : 4, instock: [ "ice cream" ], ordered: [ ] }
```

The following example concatenates the instock and the ordered arrays:

```
db.warehouses.aggregate([
   { $project: { items: { $concatArrays: [ "$instock", "$ordered" ] } } }
])
```

```
{ "_id" : 1, "items" : [ "chocolate", "butter", "apples" ] }
{ "_id" : 2, "items" : null }
{ "_id" : 3, "items" : [ "pears", "pecans", "cherries" ] }
{ "_id" : 4, "items" : [ "ice cream" ] }
```

See also:

$push (page 740)

$filter (aggregation)

On this page

- Definition (page 704)
- Behavior (page 704)
- Example (page 705)

Definition

$filter

New in version 3.2.

Selects a subset of the array to return based on the specified condition. Returns an array with only those elements that match the condition. The returned elements are in the original order.

$filter (page 704) has the following syntax:

```
{ $filter: { input: <array>, as: <string>, cond: <expression> } }
```

Field	Specification
input	An *expression* (page 747) that resolves to an array.
as	The variable name for the element in the input array. The as expression accesses each element in the input array by this *variable* (page 755).
cond	The *expression* (page 747) that determines whether to include the element in the resulting array. The expression accesses the element by the variable name specified in as.

For more information on expressions, see *Expressions* (page 747).

Behavior

Example	Results

`{`
`  $filter: {`
`     input: [ 1, "a", 2, null, 3.1, NumberLong(4), "5" ],`
`     as: "num",`
`     cond: { $and: [`
`        { $gte: [ "$$num", NumberLong("-9223372036854775807") ] },`
`        { $lte: [ "$$num", NumberLong("9223372036854775807") ] }`
`      ] }`
`  }`
`}`
	`[ 1, 2, 3.1, NumberLong(4) ]`

Example A collection sales has the following documents:

```
{
   _id: 0,
   items: [
     { item_id: 43, quantity: 2, price: 10 },
     { item_id: 2, quantity: 1, price: 240 }
   ]
}
{
   _id: 1,
   items: [
     { item_id: 23, quantity: 3, price: 110 },
     { item_id: 103, quantity: 4, price: 5 },
     { item_id: 38, quantity: 1, price: 300 }
   ]
}
{
    _id: 2,
    items: [
       { item_id: 4, quantity: 1, price: 23 }
    ]
}
```

The following example filters the items array to only include documents that have a price `` greater than or equal to ``100:

```
db.sales.aggregate([
   {
      $project: {
         items: {
            $filter: {
               input: "$items",
               as: "item",
               cond: { $gte: [ "$$item.price", 100 ] }
            }
         }
      }
   }
])
```

The operation produces the following results:

```
{
   "_id" : 0,
   "items" : [
      { "item_id" : 2, "quantity" : 1, "price" : 240 }
   ]
}
{
   "_id" : 1,
   "items" : [
      { "item_id" : 23, "quantity" : 3, "price" : 110 },
      { "item_id" : 38, "quantity" : 1, "price" : 300 }
   ]
}
{ "_id" : 2, "items" : [ ] }
```

$isArray (aggregation)

On this page

- Definition (page 706)
- Behavior (page 706)
- Example (page 706)

Definition

$isArray

New in version 3.2.

Determines if the operand is an array. Returns a boolean.

$isArray (page 706) has the following syntax:

```
{ $isArray: [ <expression> ] }
```

Behavior

The <expression> can be any valid *expression* (page 747). For more information on expressions, see *Expressions* (page 747).

Example	Results
`{ $isArray: [ "hello" ] }`	`false`
`{ $isArray: [ [ "hello", "world" ] ] }`	`true`

Example

A collection named warehouses contains the following documents:

```
{ "_id" : 1, instock: [ "chocolate" ], ordered: [ "butter", "apples" ] }
{ "_id" : 2, instock: [ "apples", "pudding", "pie" ] }
{ "_id" : 3, instock: [ "pears", "pecans"], ordered: [ "cherries" ] }
{ "_id" : 4, instock: [ "ice cream" ], ordered: [ ] }
```

The following example checks if the instock and the ordered fields are arrays before concatenating the two:

```
db.warehouses.aggregate([
   { $project:
      { items:
          { $cond:
            {
              if: { $and: [ { $isArray: "$instock" }, { $isArray: "$ordered" } ] },
              then: { $concatArrays: [ "$instock", "$ordered" ] },
              else: "One or more fields is not an array."
            }
          }
      }
   }
])
```

```
{ "_id" : 1, "items" : [ "chocolate", "butter", "apples" ] }
{ "_id" : 2, "items" : "One or more fields is not an array." }
{ "_id" : 3, "items" : [ "pears", "pecans", "cherries" ] }
{ "_id" : 4, "items" : [ "ice cream" ] }
```

See also:

$cond (page 726), $concatArrays (page 703)

$size (aggregation)

On this page

- Definition (page 707)
- Example (page 707)

New in version 2.6.

Definition

$size

Counts and returns the total the number of items in an array.

`$size` (page 707) has the following syntax:

```
{ $size: <expression> }
```

The argument for `$size` (page 707) can be any *expression* (page 747) as long as it resolves to an array. For more information on expressions, see *Expressions* (page 747).

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 707) 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 }
```

$slice (aggregation)

On this page

- Definition (page 707)
- Behavior (page 708)
- Example (page 708)

Definition

$slice

New in version 3.2.

Returns a subset of an array.

$slice (page 707) has one of two syntax forms:

The following syntax returns elements from either the start or end of the array:

```
{ $slice: [ <array>, <n> ] }
```

The following syntax returns elements from the specified position in the array:

```
{ $slice: [ <array>, <position>, <n> ] }
```

Operand	Description
`<array>`	Any valid *expression* (page 747) as long as it resolves to an array.
`<position>`	Optional. Any valid *expression* (page 747) as long as it resolves to an integer. •If positive, `$slice` (page 707) determines the starting position from the start of the array. If `<position>` is greater than the number of elements, the `$slice` (page 707) returns an empty array. •If negative, `$slice` (page 707) determines the starting position from the end of the array. If the absolute value of the `<position>` is greater than the number of elements, the starting position is the start of the array.
`<n>`	Any valid *expression* (page 747) as long as it resolves to an integer. If `<position>` is specified, `<n>` must resolve to a positive integer. •If positive, `$slice` (page 707) returns up to the first `n` elements in the array. If the `<position>` is specified, `$slice` (page 707) returns the first `n` elements starting from the position. •If negative, `$slice` (page 707) returns up to the last `n` elements in the array. `n` cannot resolve to a negative number *if* `<position>` is specified.

For more information on expressions, see *Expressions* (page 747).

Behavior

Example	Results
`{ $slice: [ [ 1, 2, 3 ], 1, 1 ] }`	`[ 2 ]`
`{ $slice: [ [ 1, 2, 3 ], -2 ] }`	`[ 2, 3 ]`
`{ $slice: [ [ 1, 2, 3 ], 15, 2 ] }`	`[ ]`
`{ $slice: [ [ 1, 2, 3 ], -15, 2 ] }`	`[ 1, 2 ]`

Example A collection named `users` contains the following documents:

```
{ "_id" : 1, "name" : "dave123", favorites: [ "chocolate", "cake", "butter", "apples" ] }
{ "_id" : 2, "name" : "li", favorites: [ "apples", "pudding", "pie" ] }
{ "_id" : 3, "name" : "ahn", favorites: [ "pears", "pecans", "chocolate", "cherries" ] }
{ "_id" : 4, "name" : "ty", favorites: [ "ice cream" ] }
```

The following example returns at most the first three elements in the `favorites` array for each user:

```
db.users.aggregate([
   { $project: { name: 1, threeFavorites: { $slice: [ "$favorites", 3 ] } } }
])
```

The operation returns the following results:

```
{ "_id" : 1, "name" : "dave123", "threeFavorites" : [ "chocolate", "cake", "butter" ] }
{ "_id" : 2, "name" : "li", "threeFavorites" : [ "apples", "pudding", "pie" ] }
{ "_id" : 3, "name" : "ahn", "threeFavorites" : [ "pears", "pecans", "chocolate" ] }
{ "_id" : 4, "name" : "ty", "threeFavorites" : [ "ice cream" ] }
```

Variable Operators

Aggregation Variable Operators

Name	Description
`$map` (page 709)	Applies a subexpression to each element of an array and returns the array of resul order. Accepts named parameters.
`$let` (page 710)	Defines variables for use within the scope of a subexpression and returns the resu subexpression. Accepts named parameters.

$map (aggregation)

On this page

- Definition (page 709)
- Example (page 709)

Definition

`$map`

Applies an *expression* (page 747) to each item in an array and returns an array with the applied results.

The `$map` (page 709) expression has the following syntax:

```
{ $map: { input: <expression>, as: <string>, in: <expression> } }
```

Field	Specification
`input`	An *expression* (page 747) 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 755).
`in`	The *expression* (page 747) 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 747).

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 631) 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 710)

$let (aggregation)

On this page

- Definition (page 710)
- Behavior (page 711)
- Example (page 711)

Definition

$let

Binds *variables* (page 755) for use in the specified expression, and returns the result of the expression.

The $let (page 710) expression has the following syntax:

```
{
  $let:
     {
       vars: { <var1>: <expression>, ... },
       in: <expression>
     }
}
```

Field	Specification
vars	Assignment block for the *variables* (page 755) 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 747) 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 747). For information on use of variables in the aggregation pipeline, see *Variables in Aggregation Expressions* (page 755).

Behavior `$let` (page 710) can access variables defined outside its expression block, including *system variables* (page 755).

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 710) 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 710) 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 `$let` (page 710) in the `$project` (page 631) 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 709)

Literal Operators

Aggregation Literal Operators

Name	Description
$literal (page 712)	Return a value without parsing. Use for values that the aggregation pipeline may i expression. For example, use a $literal (page 712) expression to a string that s to avoid parsing as a field path.

$literal (aggregation)

On this page

- Definition (page 712)
- Behavior (page 712)
- Examples (page 712)

Definition

`$literal`

Returns a value without parsing. Use for values that the aggregation pipeline may interpret as an expression.

The $literal (page 712) expression has the following syntax:

```
{ $literal: <value> }
```

Behavior If the `<value>` is an *expression* (page 747), $literal (page 712) 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 747), 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 712) 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 `$project` (page 631) stage uses the expression `<field>: 1` to include the `<field>` in the output. The following example uses the `$literal` (page 712) 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 712) 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
`$dayOfYear` (page 714)	Returns the day of the year for a date as a number between 1 and 366 (leap ye
`$dayOfMonth` (page 715)	Returns the day of the month for a date as a number between 1 and 31.
`$dayOfWeek` (page 716)	Returns the day of the week for a date as a number between 1 (Sunday) and 7
`$year` (page 717)	Returns the year for a date as a number (e.g. 2014).
`$month` (page 718)	Returns the month for a date as a number between 1 (January) and 12 (Decem
`$week` (page 719)	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).
`$hour` (page 720)	Returns the hour for a date as a number between 0 and 23.
`$minute` (page 721)	Returns the minute for a date as a number between 0 and 59.
`$second` (page 722)	Returns the seconds for a date as a number between 0 and 60 (leap seconds).
`$millisecond` (page 723)	Returns the milliseconds of a date as a number between 0 and 999.
`$dateToString` (page 724)	Returns the date as a formatted string.

$dayOfYear (aggregation)

On this page

- Definition (page 714)
- Example (page 714)

Definition

$dayOfYear

Returns the day of the year for a date as a number between 1 and 366.

The `$dayOfYear` (page 714) expression has the following syntax:

```
{ $dayOfYear: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

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 714) 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
}
```

$dayOfMonth (aggregation)

On this page

- Definition (page 715)
- Example (page 715)

Definition

`$dayOfMonth`

Returns the day of the month for a date as a number between 1 and 31.

The `$dayOfMonth` (page 715) expression has the following syntax:

```
{ $dayOfMonth: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

Example

Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736
```

The following aggregation uses the `$dayOfMonth` (page 715) 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)

On this page

- Definition (page 716)
- Example (page 716)

Definition

`$dayOfWeek`

Returns the day of the week for a date as a number between 1 (Sunday) and 7 (Saturday).

The `$dayOfWeek` (page 716) expression has the following syntax:

```
{ $dayOfWeek: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

Example Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736
```

The following aggregation uses the `$dayOfWeek` (page 716) 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)

On this page

- Definition (page 717)
- Example (page 717)

Definition

`$year`

Returns the year portion of a date.

The `$year` (page 717) expression has the following syntax:

```
{ $year: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

Example

Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736
```

The following aggregation uses the `$year` (page 717) 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
}
```

$month (aggregation)

On this page

- Definition (page 718)
- Example (page 718)

Definition

`$month`

Returns the month of a date as a number between 1 and 12.

The `$month` (page 718) expression has the following syntax:

```
{ $month: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

Example

Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736
```

The following aggregation uses the `$month` (page 718) 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
}
```

$week (aggregation)

On this page

- Definition (page 719)
- Example (page 719)

Definition

`$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 719) expression has the following syntax:

```
{ $week: <expression> }
```

The argument can be any valid *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

Example Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736
```

The following aggregation uses the `$week` (page 719) 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
}
```

$hour (aggregation)

On this page

- Definition (page 720)
- Example (page 720)

Definition

`$hour`

Returns the hour portion of a date as a number between 0 and 23.

The `$hour` (page 720) expression has the following syntax:

```
{ $hour: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

Example

Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736
```

The following aggregation uses the `$hour` (page 720) 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
}
```

$minute (aggregation)

On this page

- Definition (page 721)
- Example (page 721)

Definition

`$minute`

Returns the minute portion of a date as a number between 0 and 59.

The `$minute` (page 721) expression has the following syntax:

```
{ $minute: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

Example

Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736
```

The following aggregation uses the `$minute` (page 721) 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
}
```

$second (aggregation)

On this page

- Definition (page 722)
- Example (page 722)

Definition

`$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 722) expression has the following syntax:

```
{ $second: <expression> }
```

The argument can be any valid *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

Example

Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736
```

The following aggregation uses the `$second` (page 722) 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
}
```

$millisecond (aggregation)

On this page

- Definition (page 723)
- Example (page 723)

Definition

$millisecond

Returns the millisecond portion of a date as an integer between 0 and 999.

The `$millisecond` (page 723) expression has the following syntax:

```
{ $millisecond: <expression> }
```

The argument can be any *expression* (page 747) as long as it resolves to a date. For more information on expressions, see *Expressions* (page 747).

Example

Consider a `sales` collection with the following document:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:15:39.736
```

The following aggregation uses the $millisecond (page 723) 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
}
```

$dateToString (aggregation)

On this page

- Definition (page 724)
- Format Specifiers (page 725)
- Example (page 725)

Definition

$dateToString

New in version 3.0.

Converts a date object to a string according to a user-specified format.

The $dateToString (page 724) 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 725).

The `<dateExpression>` can be any *expression* (page 747) that evaluates to a date. For more information on expressions, see *Expressions* (page 747).

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 724) 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" }
```

Conditional Expressions

Conditional Aggregation Operators

Name	Description
$cond (page 726)	A ternary operator that evaluates one expression, and depending on the result, one of the other two expressions. Accepts either three expressions in an order named parameters.
$ifNull (page 727)	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 c or missing fields. Accepts two expressions as arguments. The result of the sec be null.

$cond (aggregation)

On this page

- Definition (page 726)
- Example (page 726)

Definition

$cond

Evaluates a boolean expression to return one of the two specified return expressions.

The $cond (page 726) 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 726) evaluates and returns the value of the `<true-case>` expression. Otherwise, $cond (page 726) evaluates and returns the value of the `<false-case>` expression.

The arguments can be any valid *expression* (page 747). For more information on expressions, see *Expressions* (page 747).

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 726) 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 726) expression and returns the same results:

```
db.inventory.aggregate(
   [
      {
         $project:
           {
             item: 1,
             discount:
               {
                 $cond: [ { $gte: [ "$qty", 250 ] }, 30, 20 ]
               }
           }
      }
   ]
)
```

$ifNull (aggregation)

On this page

- Definition (page 727)
- Example (page 727)

Definition

`$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 727) expression has the following syntax:

```
{ $ifNull: [ <expression>, <replacement-expression-if-null> ] }
```

The arguments can be any valid *expression* (page 747). For more information on expressions, see *Expressions* (page 747).

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 727) 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

Changed in version 3.2: Some accumulators are now available in the `$project` (page 631) stage. In previous versions of MongoDB , accumulators are available only for the `$group` (page 644) stage.

Accumulators, when used in the `$group` (page 644) stage, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

When used in the `$group` (page 644) stage, accumulators take as input a single expression, evaluating the expression once for each input document, and maintain their stage for the group of documents that share the same group key.

When used in the `$project` (page 631) stage, the accumulators do not maintain their state. When used in the `$project` (page 631) stage, accumulators take as input either a single argument or multiple arguments.

Group Accumulator Operators

Changed in version 3.2: Some accumulators are now available in the `$project` (page 631) stage. In previous versions of MongoDB , accumulators are available only for the `$group` (page 644) stage.

Accumulators, when used in the `$group` (page 644) stage, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

When used in the `$group` (page 644) stage, accumulators take as input a single expression, evaluating the expression once for each input document, and maintain their stage for the group of documents that share the same group key.

When used in the `$project` (page 631) stage, the accumulators do not maintain their state. When used in the `$project` (page 631) stage, accumulators take as input either a single argument or multiple arguments.

Name	Description
`$sum` (page 729)	Returns a sum of numerical values. Ignores non-numeric values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$avg` (page 732)	Returns an average of numerical values. Ignores non-numeric values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$first` (page 734)	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order. Available in `$group` (page 644) stage only.
`$last` (page 735)	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order. Available in `$group` (page 644) stage only.
`$max` (page 736)	Returns the highest expression value for each group. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$min` (page 738)	Returns the lowest expression value for each group. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$push` (page 740)	Returns an array of expression values for each group. Available in `$group` (page 644) stage only.
`$addToSet` (page 741)	Returns an array of *unique* expression values for each group. Order of the array elements is undefined. Available in `$group` (page 644) stage only.
`$stdDevPop` (page 742)	Returns the population standard deviation of the input values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$stdDevSamp` (page 744)	Returns the sample standard deviation of the input values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.

$sum (aggregation)

On this page

- Definition (page 729)
- Behavior (page 730)
- Examples (page 730)

Definition

`$sum`

Calculates and returns the sum of numeric values. `$sum` (page 729) ignores non-numeric values.

Changed in version 3.2: `$sum` (page 729) is available in the `$group` (page 644) and `$project` (page 631) stages. In previous versions of MongoDB, `$sum` (page 729) is available in the `$group` (page 644) stage only.

When used in the `$group` (page 644) stage, `$sum` has the following syntax and returns the collective 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: <expression> }
```

When used in the `$project` (page 631) stage, `$sum` returns the sum of the specified expression or list of expressions for each document and has one of two syntaxes:

- `$sum` has one specified expression as its operand:

```
{ $sum: <expression> }
```

•`$sum` has a list of specified expressions as its operand:

```
{ $sum: [ <expression1>, <expression2> ... ]  }
```

For more information on expressions, see *Expressions* (page 747).

Behavior

Non-Numeric or Non-Existent Fields If used on a field that contains both numeric and non-numeric values, `$sum` (page 729) ignores the non-numeric values and returns the sum of the numeric values.

If used on a field that does not exist in any document in the collection, `$sum` (page 729) returns `0` for that field.

If all operands are non-numeric, `$sum` (page 729) returns `0`.

Example	Field Values	Results
`{ $sum : <field> }`	Numeric	Sum of Values
`{ $sum : <field> }`	Numeric and Non-Numeric	Sum of Numeric Values
`{ $sum : <field> }`	Non-Numeric or Non-Existent	0

Array Operand In the `$group` (page 644) stage, if the expression resolves to an array, `$sum` (page 729) treats the operand as a non-numerical value.

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$sum` (page 729) traverses into the array to operate on the numerical elements of the array to return a single value.
- With a list of expressions as its operand, if any of the expressions resolves to an array, `$sum` (page 729) does **not** traverse into the array but instead treats the array as a non-numerical value.

Examples

Use in `$group` Stage 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 729) 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 }
```

Using `$sum` (page 729) on a non-existent field returns a value of `0`. The following operation attempts to `$sum` (page 729) on `qty`:

```
db.sales.aggregate(
   [
     {
       $group:
         {
           _id: { day: { $dayOfYear: "$date"}, year: { $year: "$date" } },
           totalAmount: { $sum: "$qty" },
           count: { $sum: 1 }
         }
     }
   ]
)
```

The operation returns:

```
{ "_id" : { "day" : 46, "year" : 2014 }, "totalAmount" : 0, "count" : 2 }
{ "_id" : { "day" : 34, "year" : 2014 }, "totalAmount" : 0, "count" : 2 }
{ "_id" : { "day" : 1, "year" : 2014 }, "totalAmount" : 0, "count" : 1 }
```

Use in `$project` Stage A collection `students` contains the following documents:

```
{ "_id": 1, "quizzes": [ 10, 6, 7 ], "labs": [ 5, 8 ], "final": 80, "midterm": 75 }
{ "_id": 2, "quizzes": [ 9, 10 ], "labs": [ 8, 8 ], "final": 95, "midterm": 80 }
{ "_id": 3, "quizzes": [ 4, 5, 5 ], "labs": [ 6, 5 ], "final": 78, "midterm": 70 }
```

The following example uses the `$sum` (page 729) in the `$project` (page 631) stage to calculate the total quiz scores, the total lab scores, and the total of the final and the midterm:

```
db.students.aggregate([
   {
     $project: {
       quizTotal: { $sum: "$quizzes"},
       labTotal: { $sum: "$labs" },
       examTotal: { $sum: [ "$final", "$midterm" ] }
     }
   }
])
```

The operation results in the following documents:

```
{ "_id" : 1, "quizTotal" : 23, "labTotal" : 13, "examTotal" : 155 }
{ "_id" : 2, "quizTotal" : 19, "labTotal" : 16, "examTotal" : 175 }
{ "_id" : 3, "quizTotal" : 14, "labTotal" : 11, "examTotal" : 148 }
```

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$sum` (page 729) traverses into the array to operate on the numerical elements of the array to return a single value.

- With a list of expressions as its operand, if any of the expressions resolves to an array, `$sum` (page 729) does **not** traverse into the array but instead treats the array as a non-numerical value.

$avg (aggregation)

On this page

- Definition (page 732)
- Behavior (page 732)
- Examples (page 732)

Definition

`$avg`

Returns the average value of the numeric values. `$avg` (page 732) ignores non-numeric values.

Changed in version 3.2: `$avg` (page 732) is available in the `$group` (page 644) and `$project` (page 631) stages. In previous versions of MongoDB, `$avg` (page 732) is available in the `$group` (page 644) stage only.

When used in the `$group` (page 644) stage, `$avg` has the following syntax and returns the collective average 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:

```
{ $avg: <expression> }
```

When used in the `$project` (page 631) stage, `$avg` returns the average of the specified expression or list of expressions for each document and has one of two syntaxes:

- `$avg` has one specified expression as its operand:

```
{ $avg: <expression> }
```

- `$avg` has a list of specified expressions as its operand:

```
{ $avg: [ <expression1>, <expression2> ... ]  }
```

For more information on expressions, see *Expressions* (page 747).

Behavior

Non-numeric Values `$avg` (page 732) ignores non-numeric values. If all operands for the average are non-numeric, `$avg` (page 732) returns `null`.

Array Operand In the `$group` (page 644) stage, if the expression resolves to an array, `$avg` (page 732) treats the operand as a non-numerical value.

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$avg` (page 732) traverses into the array to operate on the numerical elements of the array to return a single value.
- With a list of expressions as its operand, if any of the expressions resolves to an array, `$avg` (page 732) does **not** traverse into the array but instead treats the array as a non-numerical value.

Examples

Use in `$group` Stage 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 732) 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 }
```

Use in `$project` Stage A collection `students` contains the following documents:

```
{ "_id": 1, "quizzes": [ 10, 6, 7 ], "labs": [ 5, 8 ], "final": 80, "midterm": 75 }
{ "_id": 2, "quizzes": [ 9, 10 ], "labs": [ 8, 8 ], "final": 95, "midterm": 80 }
{ "_id": 3, "quizzes": [ 4, 5, 5 ], "labs": [ 6, 5 ], "final": 78, "midterm": 70 }
```

The following example uses the `$avg` (page 732) in the `$project` (page 631) stage to calculate the average quiz scores, the average lab scores, and the average of the final and the midterm:

```
db.students.aggregate([
   {
     $project: {
       quizAvg: { $avg: "$quizzes"},
       labAvg: { $avg: "$labs" },
       examAvg: { $avg: [ "$final", "$midterm" ] }
     }
   }
])
```

The operation results in the following documents:

```
{ "_id" : 1, "quizAvg" : 7.666666666666667, "labAvg" : 6.5, "examAvg" : 77.5 }
{ "_id" : 2, "quizAvg" : 9.5, "labAvg" : 8, "examAvg" : 87.5 }
{ "_id" : 3, "quizAvg" : 4.666666666666667, "labAvg" : 5.5, "examAvg" : 74 }
```

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$avg` (page 732) traverses into the array to operate on the numerical elements of the array to return a single value.

- With a list of expressions as its operand, if any of the expressions resolves to an array, `$avg` (page 732) does **not** traverse into the array but instead treats the array as a non-numerical value.

$first (aggregation)

On this page

- Definition (page 734)
- Behavior (page 734)
- Example (page 734)

Definition

`$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.

`$first` (page 734) is only available in the `$group` (page 644) stage.

`$first` has the following syntax:

```
{ $first: <expression> }
```

For more information on expressions, see *Expressions* (page 747).

Behavior When using `$first` (page 734) in a `$group` (page 644) stage, the `$group` (page 644) stage should follow a `$sort` (page 649) 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 734) 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)

On this page

- Definition (page 735)
- Behavior (page 735)
- Example (page 735)

Definition

`$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.

`$last` (page 735) is only available in the `$group` (page 644) stage.

`$last` has the following syntax:

```
{ $last: <expression> }
```

For more information on expressions, see *Expressions* (page 747).

Behavior When using `$last` (page 735) in a `$group` (page 644) stage, the `$group` (page 644) stage should follow a `$sort` (page 649) stage to have the input documents in a defined order.

Example Consider a `sales` collection with the following documents:

```
{ "_id" : 1, "item" : "abc", "date" : ISODate("2014-01-01T08:00:00Z"), "price" : 10, "quantity" : 2
{ "_id" : 2, "item" : "jkl", "date" : ISODate("2014-02-03T09:00:00Z"), "price" : 20, "quantity" : 1
{ "_id" : 3, "item" : "xyz", "date" : ISODate("2014-02-03T09:05:00Z"), "price" : 5, "quantity" : 5 }
{ "_id" : 4, "item" : "abc", "date" : ISODate("2014-02-15T08:00:00Z"), "price" : 10, "quantity" : 10
{ "_id" : 5, "item" : "xyz", "date" : ISODate("2014-02-15T09:05:00Z"), "price" : 5, "quantity" : 10
{ "_id" : 6, "item" : "xyz", "date" : ISODate("2014-02-15T12:05:10Z"), "price" : 5, "quantity" : 5 }
{ "_id" : 7, "item" : "xyz", "date" : ISODate("2014-02-15T14:12:12Z"), "price" : 5, "quantity" : 10
```

The following operation first sorts the documents by `item` and `date`, and then in the following `$group` (page 644) stage, groups the now sorted documents by the `item` field and uses the `$last` (page 735) 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)

On this page

- Definition (page 736)
- Behavior (page 736)
- Examples (page 737)

Definition

`$max`

Returns the maximum value. `$max` (page 736) compares both value and type, using the *specified BSON comparison order* for values of different types.

Changed in version 3.2: `$max` (page 736) is available in the `$group` (page 644) and `$project` (page 631) stages. In previous versions of MongoDB, `$max` (page 736) is available in the `$group` (page 644) stage only.

When used in the `$group` (page 644) stage, `$max` has the following syntax and returns the maximum value that results from applying an expression to each document in a group of documents that share the same group by key:

```
{ $max: <expression> }
```

When used in the `$project` (page 631) stage, `$max` returns the maximum of the specified expression or list of expressions for each document and has one of two syntaxes:

- `$max` has one specified expression as its operand:

```
{ $max: <expression> }
```

- `$max` has a list of specified expressions as its operand:

```
{ $max: [ <expression1>, <expression2> ... ]  }
```

For more information on expressions, see *Expressions* (page 747).

Behavior

Null or Missing Values If some, **but not all**, documents for the `$max` (page 736) operation have either a `null` value for the field or are missing the field, the `$max` (page 736) operator only considers the non-null and the non-missing values for the field.

If **all** documents for the `$max` (page 736) operation have `null` value for the field or are missing the field, the `$max` (page 736) operator returns `null` for the maximum value.

Array Operand In the `$group` (page 644) stage, if the expression resolves to an array, `$min` (page 738) does not traverse the array and compares the array as a whole.

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$min` (page 738) traverses into the array to operate on the numerical elements of the array to return a single value.
- With a list of expressions as its operand, if any of the expressions resolves to an array, `$min` (page 738) does **not** traverse into the array but instead treats the array as a non-numerical value.

Examples

Use in `$group` Stage 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 736) 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 }
```

Use in `$project` Stage A collection `students` contains the following documents:

```
{ "_id": 1, "quizzes": [ 10, 6, 7 ], "labs": [ 5, 8 ], "final": 80, "midterm": 75 }
{ "_id": 2, "quizzes": [ 9, 10 ], "labs": [ 8, 8 ], "final": 95, "midterm": 80 }
{ "_id": 3, "quizzes": [ 4, 5, 5 ], "labs": [ 6, 5 ], "final": 78, "midterm": 70 }
```

The following example uses the `$max` (page 736) in the `$project` (page 631) stage to calculate the maximum quiz scores, the maximum lab scores, and the maximum of the final and the midterm:

```
db.students.aggregate([
   {
     $project: {
       quizMax: { $max: "$quizzes"},
       labMax: { $max: "$labs" },
       examMax: { $max: [ "$final", "$midterm" ] }
     }
   }
])
```

The operation results in the following documents:

```
{ "_id" : 1, "quizMax" : 10, "labMax" : 8, "examMax" : 80 }
{ "_id" : 2, "quizMax" : 10, "labMax" : 8, "examMax" : 95 }
{ "_id" : 3, "quizMax" : 5, "labMax" : 6, "examMax" : 78 }
```

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$max` (page 736) traverses into the array to operate on the numerical elements of the array to return a single value.
- With a list of expressions as its operand, if any of the expressions resolves to an array, `$max` (page 736) does **not** traverse into the array but instead treats the array as a non-numerical value.

$min (aggregation)

On this page

- Definition (page 738)
- Behavior (page 738)
- Examples (page 739)

Definition

`$min`

Returns the minimum value. `$min` (page 738) compares both value and type, using the *specified BSON comparison order* for values of different types.

Changed in version 3.2: `$min` (page 738) is available in the `$group` (page 644) and `$project` (page 631) stages. In previous versions of MongoDB, `$min` (page 738) is available in the `$group` (page 644) stage only.

When used in the `$group` (page 644) stage, `$min` has the following syntax and returns the minimum value that results from applying an expression to each document in a group of documents that share the same group by key:

```
{ $min: <expression> }
```

When used in the `$project` (page 631) stage, `$min` returns the minimum of the specified expression or list of expressions for each document and has one of two syntaxes:

- `$min` has one specified expression as its operand:

```
{ $min: <expression> }
```

- `$min` has a list of specified expressions as its operand:

```
{ $min: [ <expression1>, <expression2> ... ]  }
```

For more information on expressions, see *Expressions* (page 747).

Behavior

Null or Missing Values If some, **but not all**, documents for the `$min` (page 738) operation have either a `null` value for the field or are missing the field, the `$min` (page 738) operator only considers the non-null and the non-missing values for the field.

If **all** documents for the `$min` (page 738) operation have `null` value for the field or are missing the field, the `$min` (page 738) operator returns `null` for the minimum value.

Array Operand In the `$group` (page 644) stage, if the expression resolves to an array, `$min` (page 738) does not traverse the array and compares the array as a whole.

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$min` (page 738) traverses into the array to operate on the numerical elements of the array to return a single value.
- With a list of expressions as its operand, if any of the expressions resolves to an array, `$min` (page 738) does **not** traverse into the array but instead treats the array as a non-numerical value.

Examples

Use in `$group` Stage 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 738) 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 }
```

Use in `$project` Stage A collection `students` contains the following documents:

```
{ "_id": 1, "quizzes": [ 10, 6, 7 ], "labs": [ 5, 8 ], "final": 80, "midterm": 75 }
{ "_id": 2, "quizzes": [ 9, 10 ], "labs": [ 8, 8 ], "final": 95, "midterm": 80 }
{ "_id": 3, "quizzes": [ 4, 5, 5 ], "labs": [ 6, 5 ], "final": 78, "midterm": 70 }
```

The following example uses the `$min` (page 738) in the `$project` (page 631) stage to calculate the minimum quiz scores, the minimum lab scores, and the minimum of the final and the midterm:

```
db.students.aggregate([
   {
     $project: {
       quizMin: { $min: "$quizzes"},
       labMin: { $min: "$labs" },
       examMin: { $min: [ "$final", "$midterm" ] }
     }
   }
])
```

The operation results in the following documents:

```
{ "_id" : 1, "quizMin" : 6, "labMin" : 5, "examMin" : 75 }
{ "_id" : 2, "quizMin" : 9, "labMin" : 8, "examMin" : 80 }
{ "_id" : 3, "quizMin" : 4, "labMin" : 5, "examMin" : 70 }
```

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$min` (page 738) traverses into the array to operate on the numerical elements of the array to return a single value.
- With a list of expressions as its operand, if any of the expressions resolves to an array, `$min` (page 738) does **not** traverse into the array but instead treats the array as a non-numerical value.

$push (aggregation)

On this page

- Definition (page 740)
- Example (page 740)

Definition

`$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.

`$push` (page 740) is only available in the `$group` (page 644) stage.

`$push` has the following syntax:

```
{ $push: <expression> }
```

For more information on expressions, see *Expressions* (page 747).

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 `$push` 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 } ]
}
```

$addToSet (aggregation)

On this page

- Definition (page 741)
- Behavior (page 741)
- Example (page 742)

Definition

`$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.

`$addToSet` (page 741) is only available in the `$group` (page 644) stage.

`$addToSet` has the following syntax:

```
{ $addToSet: <expression> }
```

For more information on expressions, see *Expressions* (page 747).

Behavior If the value of the expression is an array, `$addToSet` (page 741) 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 741) 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" ] }
```

$stdDevPop (aggregation)

On this page

- Definition (page 742)
- Behavior (page 743)
- Examples (page 743)

Definition

`$stdDevPop`

New in version 3.2.

Calculates the population standard deviation of the input values. Use if the values encompass the entire population of data you want to represent and do not wish to generalize about a larger population. `$stdDevPop` (page 742) ignores non-numeric values.

If the values represent only a sample of a population of data from which to generalize about the population, use `$stdDevSamp` (page 744) instead.

`$stdDevPop` (page 742) is available in the `$group` (page 644) and `$project` (page 631) stages.

When used in the `$group` (page 644) stage, `$stdDevPop` returns the population standard deviation of the specified expression for a group of documents that share the same group by key and has the following syntax:

- `$stdDevPop` has one specified expression as its operand:

```
{ $stdDevPop: <expression> }
```

When used in the `$project` (page 631) stage, `$stdDevPop` returns the standard deviation of the specified expression or list of expressions for each document and has one of two syntaxes:

- `$stdDevPop` has one specified expression as its operand:

  ```
  { $stdDevPop: <expression> }
  ```

- `$stdDevPop` has a list of specified expressions as its operand:

  ```
  { $stdDevPop: [ <expression1>, <expression2> ... ]  }
  ```

The argument for `$stdDevPop` can be any *expression* (page 747) as long as it resolves to an array. For more information on expressions, see *Expressions* (page 747)

Behavior

Non-numeric Values `$stdDevPop` (page 742) ignores non-numeric values. If all operands for a `$stdDevPop` (page 742) are non-numeric, `$stdDevPop` (page 742) returns `null`.

Single Value If the sample consists of a single numeric value, `$stdDevPop` (page 742) returns `0`.

Array Operand In the `$group` (page 644) stage, if the expression resolves to an array, `$stdDevPop` (page 742) treats the operand as a non-numerical value.

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$stdDevPop` (page 742) traverses into the array to operate on the numerical elements of the array to return a single value.
- With a list of expressions as its operand, if any of the expressions resolves to an array, `$stdDevPop` (page 742) does **not** traverse into the array but instead treats the array as a non-numerical value.

Examples

Use in `$group` Stage A collection named `users` contains the following documents:

```
{ "_id" : 1, "name" : "dave123", "quiz" : 1, "score" : 85 }
{ "_id" : 2, "name" : "dave2", "quiz" : 1, "score" : 90 }
{ "_id" : 3, "name" : "ahn", "quiz" : 1, "score" : 71 }
{ "_id" : 4, "name" : "li", "quiz" : 2, "score" : 96 }
{ "_id" : 5, "name" : "annT", "quiz" : 2, "score" : 77 }
{ "_id" : 6, "name" : "ty", "quiz" : 2, "score" : 82 }
```

The following example calculates the standard deviation of each quiz:

```
db.users.aggregate([
   { $group: { _id: "$quiz", stdDev: { $stdDevPop: "$score" } } }
])
```

The operation returns the following results:

```
{ "_id" : 2, "stdDev" : 8.04155872120988 }
{ "_id" : 1, "stdDev" : 8.04155872120988 }
```

Use in `$project` Stage A collection named `quizzes` contains the following documents:

```
{
   "_id" : 1,
   "scores" : [
      {
         "name" : "dave123",
         "score" : 85
      },
      {
         "name" : "dave2",
         "score" : 90
      },
      {
         "name" : "ahn",
         "score" : 71
      }
   ]
}
{
   "_id" : 2,
   "scores" : [
      {
         "name" : "li",
         "quiz" : 2,
         "score" : 96
      },
      {
         "name" : "annT",
         "score" : 77
      },
      {
         "name" : "ty",
         "score" : 82
      }
   ]
}
```

The following example calculates the standard deviation of each quiz:

```
db.quizzes.aggregate([
   { $project: { stdDev: { $stdDevPop: "$scores.score" } } }
])
```

The operation returns the following results:

```
{ "_id" : 1, "stdDev" : 8.04155872120988 }
{ "_id" : 2, "stdDev" : 8.04155872120988 }
```

$stdDevSamp (aggregation)

On this page

- Definition (page 744)
- Behavior (page 745)
- Example (page 745)

Definition

$stdDevSamp

New in version 3.2.

Calculates the sample standard deviation of the input values. Use if the values encompass a sample of a population of data from which to generalize about the population. `$stdDevSamp` (page 744) ignores non-numeric values.

If the values represent the entire population of data or you do not wish to generalize about a larger population, use `$stdDevPop` (page 742) instead.

`$stdDevSamp` (page 744) is available in the `$group` (page 644) and `$project` (page 631) stages.

When used in the `$group` (page 644) stage, `$stdDevSamp` has the following syntax and returns the sample standard deviation of the specified expression for a group of documents that share the same group by key:

```
{ $stdDevSamp: <expression> }
```

When used in the `$project` (page 631) stage, `$stdDevSamp` returns the sample standard deviation of the specified expression or list of expressions for each document and has one of two syntaxes:

- `$stdDevSamp` has one specified expression as its operand:

  ```
  { $stdDevSamp: <expression> }
  ```

- `$stdDevSamp` has a list of specified expressions as its operand:

  ```
  { $stdDevSamp: [ <expression1>, <expression2> ... ]  }
  ```

The argument for `$stdDevSamp` can be any *expression* (page 747) as long as it resolves to an array. For more information on expressions, see *Expressions* (page 747).

Behavior

Non-numeric Values `$stdDevSamp` (page 744) ignores non-numeric values. If all operands for a sum are non-numeric, `$stdDevSamp` (page 744) returns `null`.

Single Value If the sample consists of a single numeric value, `$stdDevSamp` (page 744) returns `null`.

Array Operand In the `$group` (page 644) stage, if the expression resolves to an array, `$stdDevSamp` (page 744) treats the operand as a non-numerical value.

In the `$project` (page 631) stage:

- With a single expression as its operand, if the expression resolves to an array, `$stdDevSamp` (page 744) traverses into the array to operate on the numerical elements of the array to return a single value.
- With a list of expressions as its operand, if any of the expressions resolves to an array, `$stdDevSamp` (page 744) does **not** traverse into the array but instead treats the array as a non-numerical value.

Example A collection `users` contains documents with the following fields:

```
{_id: 0, username: "user0", age: 20}
{_id: 1, username: "user1", age: 42}
{_id: 2, username: "user2", age: 28}
...
```

To calculate the standard deviation of a sample of users, following aggregation operation first uses the `$sample` (page 648) pipeline to sample 100 users, and then uses `$stdDevSamp` (page 744) calculates the standard deviation for the sampled users.

```
db.users.aggregate(
   [
      { $sample: { size: 100 } },
      { $group: { _id: null, ageStdDev: { $stdDevSamp: "$age" } } }
   ]
)
```

The operation returns a result like the following:

```
{ "_id" : null, "ageStdDev" : 7.811258386185771 }
```

2.4 Aggregation Reference

Aggregation Pipeline Quick Reference **(page 746)** Quick reference card for aggregation pipeline.

Aggregation Commands **(page 753)** The reference for the data aggregation commands, which provide the interfaces to MongoDB's aggregation capability.

Aggregation Commands Comparison **(page 753)** A comparison of `group` (page 313), `mapReduce` (page 318) and `aggregate` (page 303) that explores the strengths and limitations of each aggregation modality.

Aggregation Pipeline Operators **(page 629)** Aggregation pipeline operations have a collection of operators available to define and manipulate documents in pipeline stages.

Variables in Aggregation Expressions **(page 755)** Use of variables in aggregation pipeline expressions.

SQL to Aggregation Mapping Chart **(page 765)** An overview common aggregation operations in SQL and MongoDB using the aggregation pipeline and operators in MongoDB and common SQL statements.

2.4.1 Aggregation Pipeline Quick Reference

On this page

- Stages (page 746)
- Expressions (page 747)
- Accumulators (page 752)

Stages

In the `db.collection.aggregate` (page 20) method, pipeline stages appear in an array. Documents pass through the stages in sequence. All except the `$out` (page 656) and `$geoNear` (page 651) stages can appear multiple times in a pipeline.

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

Name	Description
$project (page 631)	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
$match (page 635)	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match (page 635) uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$redact (page 637)	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 631) and $match (page 635). Can be used to implement field level redaction. For each input document, outputs either one or zero document.
$limit (page 640)	Passes the first *n* documents unmodified to the pipeline where *n* is the specified limit. For each input document, outputs either one document (for the first *n* documents) or zero documents (after the first *n* documents).
$skip (page 641)	Skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents).
$unwind (page 641)	Deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array.
$group (page 644)	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.
$sample (page 648)	Randomly selects the specified number of documents from its input.
$sort (page 649)	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
$geoNear (page 651)	Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of $match (page 635), $sort (page 649), and $limit (page 640) for geospatial data. The output documents include an additional distance field and can include a location identifier field.
$lookup (page 654)	Performs a left outer join to another collection in the *same* database to filter in documents from the "joined" collection for processing.
$out (page 656)	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out (page 656) stage, it must be the last stage in the pipeline.
$indexStats (page 657)	Returns statistics regarding the use of each index for the collection.

Expressions

Expressions can include *field paths and system variables* (page 747), *literals* (page 748), *expression objects* (page 748), and *expression operators* (page 748). 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 755) 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 755) can be rebound.

Along with the `CURRENT` (page 755) system variable, other *system variables* (page 755) are also available for use in expressions. To use user-defined variables, use `$let` (page 710) and `$map` (page 709) 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 `$` as a path to a field and numeric/boolean literals in *expression objects* (page 748) as projection flags. To avoid parsing literals, use the `$literal` (page 712) 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 `$project` (page 631) stage. To avoid treating numeric or boolean literals as projection flags, use the `$literal` (page 712) 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 712) expression or keep the outer array that designates the argument list.

Boolean Expressions Boolean expressions evaluate their argument expressions as booleans and return a boolean as the result.

In addition to the `false` boolean value, Boolean expression evaluates as `false` the following: `null`, `0`, and `undefined` values. The Boolean expression evaluates all other values as `true`, including non-zero numeric values and arrays.

Name	Description
`$and` (page 660)	Returns `true` only when *all* its expressions evaluate to `true`. Accepts any number of argument expressions.
`$or` (page 661)	Returns `true` when *any* of its expressions evaluates to `true`. Accepts any number of argument expressions.
`$not` (page 662)	Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression.

Set Expressions Set expressions performs set operation on arrays, treating arrays as sets. Set expressions ignores the duplicate entries in each input array and the order of the elements.

If the set operation returns a set, the operation filters out duplicates in the result to output an array that contains only unique entries. The order of the elements in the output array is unspecified.

If a set contains a nested array element, the set expression does *not* descend into the nested array but evaluates the array at top-level.

Name	Description
`$setEquals` (page 664)	Returns `true` if the input sets have the same distinct elements. Accepts two or more argument expressions.
`$setIntersectio` (page 665)	Returns a set with elements that appear in *all* of the input sets. Accepts any number of argument expressions.
`$setUnion` (page 666)	Returns a set with elements that appear in *any* of the input sets. Accepts any number of argument expressions.
`$setDifference` (page 668)	Returns a set with elements that appear in the first set but not in the second set; i.e. performs a relative complement[32] of the second set relative to the first. Accepts exactly two argument expressions.
`$setIsSubset` (page 669)	Returns `true` if all elements of the first set appear in the second set, including when the first set equals the second set; i.e. not a strict subset[33]. Accepts exactly two argument expressions.
`$anyElementTrue` (page 670)	Returns `true` if *any* elements of a set evaluate to `true`; otherwise, returns `false`. Accepts a single argument expression.
`$allElementsTru` (page 671)	Returns `true` if *no* element of a set evaluates to `false`, otherwise, returns `false`. Accepts a single argument expression.

Comparison Expressions Comparison expressions return a boolean except for `$cmp` (page 673) 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
`$cmp` (page 673)	Returns: `0` if the two values are equivalent, `1` if the first value is greater than the second, and `-1` if the first value is less than the second.
`$eq` (page 674)	Returns `true` if the values are equivalent.
`$gt` (page 675)	Returns `true` if the first value is greater than the second.
`$gte` (page 676)	Returns `true` if the first value is greater than or equal to the second.
`$lt` (page 677)	Returns `true` if the first value is less than the second.
`$lte` (page 678)	Returns `true` if the first value is less than or equal to the second.
`$ne` (page 679)	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.

[32]http://en.wikipedia.org/wiki/Complement_(set_theory)
[33]http://en.wikipedia.org/wiki/Subset

Name	Description
`$abs` (page 681)	Returns the absolute value of a number.
`$add` (page 682)	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.
`$ceil` (page 683)	Returns the smallest integer greater than or equal to the specified number.
`$divide` (page 684)	Returns the result of dividing the first number by the second. Accepts two argument expressions.
`$exp` (page 685)	Raises *e* to the specified exponent.
`$floor` (page 686)	Returns the largest integer less than or equal to the specified number.
`$ln` (page 687)	Calculates the natural log of a number.
`$log` (page 687)	Calculates the log of a number in the specified base.
`$log10` (page 689)	Calculates the log base 10 of a number.
`$mod` (page 689)	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
`$multiply` (page 690)	Multiplies numbers to return the product. Accepts any number of argument expressions.
`$pow` (page 691)	Raises a number to the specified exponent.
`$sqrt` (page 692)	Calculates the square root.
`$subtract` (page 693)	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.
`$trunc` (page 695)	Truncates a number to its integer.

String Expressions String expressions, with the exception of `$concat` (page 696), only have a well-defined behavior for strings of ASCII characters.

`$concat` (page 696) behavior is well-defined regardless of the characters used.

Name	Description
`$concat` (page 696)	Concatenates any number of strings.
`$substr` (page 697)	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.
`$toLower` (page 698)	Converts a string to lowercase. Accepts a single argument expression.
`$toUpper` (page 699)	Converts a string to uppercase. Accepts a single argument expression.
`$strcasecmp` (page 699)	Performs case-insensitive string comparison and returns: `0` if two strings are equivalent, `1` if the first string is greater than the second, and `-1` if the first string is less than the second.

Text Search Expressions

Name	Description
`$meta` (page 701)	Access text search metadata.

Array Expressions

Name	Description
`$arrayElemAt` (page 702)	Returns the element at the specified array index.
`$concatArrays` (page 703)	Concatenates arrays to return the concatenated array.
`$filter` (page 704)	Selects a subset of the array to return an array with only the elements that match th filter condition.
`$isArray` (page 706)	Determines if the operand is an array. Returns a boolean.
`$size` (page 707)	Returns the number of elements in the array. Accepts a single expression as argument.
`$slice` (page 707)	Returns a subset of an array.

Variable Expressions

Name	Description
`$map` (page 709)	Applies a subexpression to each element of an array and returns the array of resulting values order. Accepts named parameters.
`$let` (page 710)	Defines variables for use within the scope of a subexpression and returns the result of the subexpression. Accepts named parameters.

Literal Expressions

Name	Description
`$literal` (page 712)	Return a value without parsing. Use for values that the aggregation pipeline may interpret as a expression. For example, use a `$literal` (page 712) expression to a string that starts with a to avoid parsing as a field path.

Date Expressions

Name	Description
`$dayOfYear` (page 714)	Returns the day of the year for a date as a number between 1 and 366 (leap year).
`$dayOfMonth` (page 715)	Returns the day of the month for a date as a number between 1 and 31.
`$dayOfWeek` (page 716)	Returns the day of the week for a date as a number between 1 (Sunday) and 7 (Saturday).
`$year` (page 717)	Returns the year for a date as a number (e.g. 2014).
`$month` (page 718)	Returns the month for a date as a number between 1 (January) and 12 (December).
`$week` (page 719)	Returns the week number for a date as a number between 0 (the partial week that precede the first Sunday of the year) and 53 (leap year).
`$hour` (page 720)	Returns the hour for a date as a number between 0 and 23.
`$minute` (page 721)	Returns the minute for a date as a number between 0 and 59.
`$second` (page 722)	Returns the seconds for a date as a number between 0 and 60 (leap seconds).
`$millisecond` (page 723)	Returns the milliseconds of a date as a number between 0 and 999.
`$dateToString` (page 724)	Returns the date as a formatted string.

Conditional Expressions

Name	Description
`$cond` (page 726)	A ternary operator that evaluates one expression, and depending on the result, returns the one of the other two expressions. Accepts either three expressions in an ordered list or thr named parameters.
`$ifNull` (page 727)	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

Changed in version 3.2: Some accumulators are now available in the `$project` (page 631) stage. In previous versions of MongoDB , accumulators are available only for the `$group` (page 644) stage.

Accumulators, when used in the `$group` (page 644) stage, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

When used in the `$group` (page 644) stage, accumulators take as input a single expression, evaluating the expression once for each input document, and maintain their stage for the group of documents that share the same group key.

When used in the `$project` (page 631) stage, the accumulators do not maintain their state. When used in the `$project` (page 631) stage, accumulators take as input either a single argument or multiple arguments.

Name	Description
`$sum` (page 729)	Returns a sum of numerical values. Ignores non-numeric values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$avg` (page 732)	Returns an average of numerical values. Ignores non-numeric values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$first` (page 734)	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order. Available in `$group` (page 644) stage only.
`$last` (page 735)	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order. Available in `$group` (page 644) stage only.
`$max` (page 736)	Returns the highest expression value for each group. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$min` (page 738)	Returns the lowest expression value for each group. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$push` (page 740)	Returns an array of expression values for each group. Available in `$group` (page 644) stage only.
`$addToSet` (page 741)	Returns an array of *unique* expression values for each group. Order of the array elements is undefined. Available in `$group` (page 644) stage only.
`$stdDevPop` (page 742)	Returns the population standard deviation of the input values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.
`$stdDevSamp` (page 744)	Returns the sample standard deviation of the input values. Changed in version 3.2: Available in both `$group` (page 644) and `$project` (page 631) stages.

2.4.2 Aggregation Commands

On this page

- Aggregation Commands (page 753)
- Aggregation Methods (page 753)

Aggregation Commands

Name	Description
`aggregate` (page 303)	Performs `aggregation tasks` such as group using the aggregation framework.
`count` (page 307)	Counts the number of documents in a collection.
`distinct` (page 310)	Displays the distinct values found for a specified key in a collection.
`group` (page 313)	Groups documents in a collection by the specified key and performs simple aggregation.
`mapReduce` (page 318)	Performs `map-reduce` aggregation for large data sets.

Aggregation Methods

Name	Description
`db.collection.aggregate()` (page 20)	Provides access to the `aggregation pipeline`.
`db.collection.group()` (page 75)	Groups documents in a collection by the specified key and performs simple aggregation.
`db.collection.mapReduce()` (page 90)	Performs `map-reduce` aggregation for large data sets.

2.4.3 Aggregation Commands Comparison

The following table provides a brief overview of the features of the MongoDB aggregation commands.

	aggregate (page 303)	mapReduce (page 318)	group (page 313)
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 `$group` (page 644), `$match` (page 635), and `$sort` (page 649). See *Aggregation Pipeline Operators* (page 629) for more information on the pipeline operators.	Implements the Map-Reduce aggregation for processing large data sets.	Provides grouping functionality. Is slower than the `aggregate` (page 303) command and has less functionality than the `mapReduce` (page 318) command.
Key Features	Pipeline operators can be repeated as needed. Pipeline operators need not produce one output document for every input document. Can also generate new documents or filter out documents.	In addition to grouping operations, can perform complex aggregation tasks as well as perform incremental aggregation on continuously growing datasets. See `https://docs.mongodb.org/manual/tutorial/map-reduce-examples` and `https://docs.mongodb.org/manual/tutorial/perform-incremental`	Can either group by existing fields or with a custom `keyf` JavaScript function, can group by calculated fields. See `group` (page 313) for information and example using the `keyf` function.
Flexibility	Limited to the operators and expressions supported by the aggregation pipeline. However, can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results by using the `$project` (page 631) pipeline operator. See `$project` (page 631) for more information as well as *Aggregation Pipeline Operators* (page 629) for more information on all the available pipeline operators.	Custom `map`, `reduce` and `finalize` JavaScript functions offer flexibility to aggregation logic. See `mapReduce` (page 318) for details and restrictions on the functions.	Custom `reduce` and `finalize` JavaScript functions offer flexibility to grouping logic. See `group` (page 313) 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 940) 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 318) 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 940). Changed in version 2.2: The returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. Previous versions had a limit of 10,000 elements.
Sharding Notes	Supports non-sharded and sharded input collections.	Supports non-sharded and sharded input collections. Prior to 2.4, JavaScript code executed in a single thread.	Does **not** support sharded collection. Prior to 2.4, JavaScript code executed in a single thread.
More Infor-	See `https://docs.mongodb.org/manual/core/aggregation-pipeline` and `aggregate` (page 303).	See `https://docs.mongodb.org/manual/core/map-reduce` and `mapReduce` (page 318).	See `group` (page 313).

2.4.4 Variables in Aggregation Expressions

On this page

- User Variables (page 755)
- System Variables (page 755)

Aggregation expressions (page 747) 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 755) the same as `ROOT` (page 755). `CURRENT` (page 755) is modifiable. However, since `$<field>` is equivalent to `$$CURRENT.<field>`, rebinding `CURRENT` (page 755) changes the meaning of `$` accesses.
`DESCEND`	One of the allowed results of a `$redact` (page 637) expression.
`PRUNE`	One of the allowed results of a `$redact` (page 637) expression.
`KEEP`	One of the allowed results of a `$redact` (page 637) expression.

See also:

`$let` (page 710), `$redact` (page 637), `$map` (page 709)

2.4.5 SQL to Aggregation Mapping Chart

On this page

- Examples (page 765)
- Additional Resources (page 768)

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 629):

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match` (page 635)
GROUP BY	`$group` (page 644)
HAVING	`$match` (page 635)
SELECT	`$project` (page 631)
ORDER BY	`$sort` (page 649)
LIMIT	`$limit` (page 640)
SUM()	`$sum` (page 729)
COUNT()	`$sum` (page 729)
join	No direct corresponding operator; *however*, the `$unwind` (page 641) operator allows for somewhat similar functionality, but with fields embedded within the document.

Examples

The following table presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume *two* tables, `orders` and `order_lineitem` that join by the `order_lineitem.order_id` and the `orders.id` columns.
- The MongoDB examples assume *one* collection `orders` that contain documents of the following prototype:

```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
           { sku: "yyy", qty: 25, price: 1 } ]
}
```

SQL Example	MongoDB Example	Description

`SELECT COUNT(*) AS count`
`FROM orders`
	`db.orders.aggregate( [`
	`   {`
	`     $group: {`
	`        _id: null,`
	`        count: { $sum: 1 }`
	`     }`
	`   }`
	`] )`
		Count all records from `orders`

`SELECT SUM(price) AS total`
`FROM orders`
	`db.orders.aggregate( [`
	`   {`
	`     $group: {`
	`        _id: null,`
	`        total: { $sum: "$price" }`
	`     }`
	`   }`
	`] )`
		Sum the `price` field from `orders`

`SELECT cust_id,`
`       SUM(price) AS total`
`FROM orders`
`GROUP BY cust_id`
	`db.orders.aggregate( [`
	`   {`
	`     $group: {`
	`        _id: "$cust_id",`
	`        total: { $sum: "$price" }`
	`     }`
	`   }`
	`] )`
		For each unique `cust_id`, sum the `price` field.

`SELECT cust_id,`
`       SUM(price) AS total`
`FROM orders`
`GROUP BY cust_id`
`ORDER BY total`
	`db.orders.aggregate( [`
	`   {`
	`     $group: {`
	`        _id: "$cust_id",`
	`        total: { $sum: "$price" }`
	`     }`
	`   },`
	`   { $sort: { total: 1 } }`
	`] )`
		For each unique `cust_id`, sum the `price` field, results sorted by sum.

`SELECT cust_id,`
`       ord_date,`
`       SUM(price) AS total`
`FROM orders`
`GROUP BY cust_id,`
`         ord_date`
	`db.orders.aggregate( [`
	`   {`
	`     $group: {`
	`        _id: {`
	`           cust_id: "$cust_id",`
	`           ord_date: {`
	`               month: { $month: "$ord_date" },`
	`               day: { $dayOfMonth: "$ord_date" },`
	`               year: { $year: "$ord_date"}`
	`           }`
	`        },`
	`        total: { $sum: "$price" }`
	`     }`
	`   }`
	`] )`
		For each unique `cust_id`, `ord_date` grouping, sum the `price` field. Excludes the time portion of the date.

`SELECT cust_id,`
`       count(*)`
`FROM orders`
	`db.orders.aggregate( [`
	`   {`
	`     $group: {`
		For `cust_id` with multiple records, return the `cust_id` and the corresponding record count.

Additional Resources

- MongoDB and MySQL Compared[34]
- Quick Reference Cards[35]
- MongoDB Database Modernization Consulting Package[36]

[34]http://www.mongodb.com/mongodb-and-mysql-compared?jmp=docs
[35]https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs
[36]https://www.mongodb.com/products/consulting?jmp=docs#database_modernization

CHAPTER 3

MongoDB and SQL Interface Comparisons

3.1 SQL to MongoDB Mapping Chart

On this page

- Terminology and Concepts (page 759)
- Executables (page 759)
- Examples (page 760)
- Additional Resources (page 764)

In addition to the charts that follow, you might want to consider the `https://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 765).

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 770)	mysqld	oracle	IDS	DB2 Server
Database Client	mongo (page 803)	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 79) 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 117) operations can add fields to existing documents using the `$set` (page 600) operator. `db.users.update( { }, { $set: { join_date: new Date() } }, { multi: true } )`
`ALTER TABLE users DROP COLUMN join_date`	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level. However, at the document level, `update()` (page 117) operations can remove fields from documents using the `$unset` (page 602) operator. `db.users.update( { }, { $unset: { join_date: "" } }, { multi: true } )`
`CREATE INDEX idx_user_id_asc ON users(user_id)`	`db.users.createIndex( { user_id: 1 } )`
`CREATE INDEX idx_user_id_asc_age_desc ON users(user_id, age DESC)`	`db.users.createIndex( { user_id: 1, age: -1 } )`
`DROP TABLE users`	`db.users.drop()`

For more information, see `db.collection.insert()` (page 79), `db.createCollection()` (page 167), `db.collection.update()` (page 117), `$set` (page 600), `$unset` (page 602), `db.collection.createIndex()` (page 36), `indexes`, `db.collection.drop()` (page 45), and `https://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 79).

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 51), `db.collection.distinct()` (page 44), `db.collection.findOne()` (page 62), `$ne` (page 531) `$and` (page 535), `$or` (page 534), `$gt` (page 529), `$lt` (page 530), `$exists` (page 538), `$lte` (page 531), `$regex` (page 546), `limit()` (page 144), `skip()` (page 155), `explain()` (page 140), `sort()` (page 156), and `count()` (page 137).

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 117), `$set` (page 600), `$inc` (page 595), and `$gt` (page 529).

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 101).

3.1.4 Additional Resources

- Transitioning from SQL to MongoDB (Presentation)[1]
- Best Practices for Migrating from RDBMS to MongoDB (Webinar)[2]
- SQL vs. MongoDB Day 1-2[3]

[1] http://www.mongodb.com/presentations/webinar-transitioning-sql-mongodb?jmp=docs
[2] http://www.mongodb.com/webinar/best-practices-migration?jmp=docs
[3] http://www.mongodb.com/blog/post/mongodb-vs-sql-day-1-2?jmp=docs

- SQL vs. MongoDB Day 3-5[4]
- MongoDB vs. SQL Day 14[5]
- MongoDB and MySQL Compared[6]
- Quick Reference Cards[7]
- MongoDB Database Modernization Consulting Package[8]

3.2 SQL to Aggregation Mapping Chart

On this page

- Examples (page 765)
- Additional Resources (page 768)

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 629):

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match` (page 635)
GROUP BY	`$group` (page 644)
HAVING	`$match` (page 635)
SELECT	`$project` (page 631)
ORDER BY	`$sort` (page 649)
LIMIT	`$limit` (page 640)
SUM()	`$sum` (page 729)
COUNT()	`$sum` (page 729)
join	No direct corresponding operator; *however*, the `$unwind` (page 641) 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"),
```

[4] http://www.mongodb.com/blog/post/mongodb-vs-sql-day-3-5?jmp=docs
[5] http://www.mongodb.com/blog/post/mongodb-vs-sql-day-14?jmp=docs
[6] http://www.mongodb.com/mongodb-and-mysql-compared?jmp=docs
[7] https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs
[8] https://www.mongodb.com/products/consulting?jmp=docs#database_modernization

```
    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.

3.2.2 Additional Resources

- MongoDB and MySQL Compared[9]
- Quick Reference Cards[10]
- MongoDB Database Modernization Consulting Package[11]

[9] http://www.mongodb.com/mongodb-and-mysql-compared?jmp=docs
[10] https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs
[11] https://www.mongodb.com/products/consulting?jmp=docs#database_modernization

CHAPTER 4

Program and Tool Reference Pages

4.1 MongoDB Package Components

On this page

- Core Processes (page 769)
- Windows Services (page 811)
- Binary Import and Export Tools (page 814)
- Data Import and Export Tools (page 840)
- Diagnostic Tools (page 857)
- GridFS (page 877)

4.1.1 Core Processes

The core components in the MongoDB package are: `mongod` (page 770), the core database process; `mongos` (page 792) the controller and query router for *sharded clusters*; and `mongo` (page 803) the interactive MongoDB Shell.

`mongod`

On this page

- Synopsis (page 769)
- Options (page 770)

Synopsis

`mongod` (page 770) 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 770). These command line options are primarily useful for testing: In common operation, use the *configuration file options* (page 895) 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 770).

command line option!–version

--version

Returns the `mongod` (page 770) 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 770). The options are equivalent to the command-line configuration options. See *Configuration File Options* (page 895) for more information.

Ensure the configuration file uses ASCII encoding. The `mongod` (page 770) 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 770) 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 (`/etc/mongod.conf`) that sets *`--bind_ip`* (page 793) to `127.0.0.1`.

Specifies the IP address that `mongod` (page 770) binds to in order to listen for connections from applications. You may attach `mongod` (page 770) to any interface. When attaching `mongod` (page 770) 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!–ipv6

`--ipv6`

Enables IPv6 support and allows the `mongod` (page 770) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

command line option!–maxConns <number>

`--maxConns` `<number>`

The maximum number of simultaneous connections that `mongod` (page 770) will accept. This setting has no effect if it is higher than your operating system's configured maximum connection tracking threshold.

Do not assign too low of a value to this option, or you will encounter errors during normal application operation.

Note: Changed in version 2.6: MongoDB removed the upward limit on the `maxIncomingConnections` (page 903) setting.

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 will move any existing log file rather than overwrite it. To instead append to the log file, set the *`--logappend`* (page 793) option.

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 793).

The *`--syslog`* (page 793) 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 793) option.

command line option!–logappend

`--logappend`

Appends new entries to the end of the existing log file when the `mongod` (page 770) instance restarts. Without this option, `mongod` (page 770) will back up the existing log and create a new file.

command line option!–logRotate <string>

`--logRotate` `<string>`

Default: rename

New in version 3.0.0.

Determines the behavior for the `logRotate` (page 465) command. Specify either `rename` or `reopen`:

- `rename` renames the log file.

- reopen closes and reopens the log file following the typical Linux/Unix log rotate behavior. Use reopen when using the Linux/Unix logrotate utility to avoid log loss.

  If you specify reopen, you must also use *--logappend* (page 793).

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 772) 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 (page 915) 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 873) 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 770) to stop writing data to the *diagnostic log* file. However, the mongod (page 770) 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 770) 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 770) process where mongod (page 770) will write its PID. This is useful for tracking the `mongod` (page 770) process in combination with the *`--fork`* (page 795) option. Without a specified *`--pidfilepath`* (page 794) 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 794) implies *`--auth`* (page 774). See *inter-process-auth* for more information.

command line option!–setParameter <options>

`--setParameter` `<options>`
Specifies one of the MongoDB parameters described in *MongoDB Server Parameters* (page 927). You can specify multiple `setParameter` fields.

command line option!–httpinterface

`--httpinterface`
Deprecated since version 3.2: HTTP interface for MongoDB

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:

- While MongoDB Enterprise does support Kerberos authentication, Kerberos is not supported in HTTP status interface in any version of MongoDB.

New in version 2.6.

command line option!–nounixsocket

`--nounixsocket`
Disables listening on the UNIX domain socket. *`--nounixsocket`* (page 794) applies only to Unix-based systems.

The `mongod` (page 770) process always listens on the UNIX socket unless one of the following is true:

- *`--nounixsocket`* (page 794) is set
- `net.bindIp` (page 903) is not set
- `net.bindIp` (page 903) does not specify `127.0.0.1`

New in version 2.6: `mongod` (page 770) 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. *`--unixSocketPrefix`* (page 795) applies only to Unix-based systems.

If this option has no value, the `mongod` (page 770) process creates a socket with `https://docs.mongodb.org/manual/tmp` as a prefix. MongoDB creates and listens on a UNIX socket unless one of the following is true:

- `net.unixDomainSocket.enabled` (page 904) is `false`
- *`--nounixsocket`* (page 794) is set

- `net.bindIp` (page 903) is not set
- `net.bindIp` (page 903) does not specify `127.0.0.1`

command line option!–filePermissions <path>

`--filePermissions` `<path>`

Default: `0700`

Sets the permission for the UNIX domain socket file.

`--filePermissions` (page 795) applies only to Unix-based systems.

command line option!–fork

`--fork`

Enables a *daemon* mode that runs the `mongod` (page 770) process in the background. By default `mongod` (page 770) does not run as a daemon: typically you will run `mongod` (page 770) as a daemon, either by using *`--fork`* (page 795) 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 802). 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!–jsonp

`--jsonp`

Permits *JSONP* access via an HTTP interface. Enabling the interface can increase network exposure. The *`--jsonp`* (page 802) option enables the HTTP interface, even if the `HTTP interface` (page 905) option is disabled.

Deprecated since version 3.2: HTTP interface for MongoDB

command line option!–rest

`--rest`

Enables the simple *REST* API. Enabling the *REST* API enables the HTTP interface, even if the `HTTP interface` (page 905) option is disabled, and as a result can increase network exposure.

Deprecated since version 3.2: HTTP interface for MongoDB

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 484) 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 the `system.profile` collection. 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 770) 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!–noscripting

--noscripting
Disables the scripting engine.

command line option!–notablescan

--notablescan
Forbids operations that require a table scan. See `notablescan` (page 932) for additional information.

command line option!–shutdown

--shutdown
The *--shutdown* (page 775) option cleanly and safely terminates the `mongod` (page 770) process. When invoking `mongod` (page 770) with this option you must set the *--dbpath* (page 776) option either directly or by way of the *configuration file* (page 895) and the *--config* (page 792) option.

The *--shutdown* (page 775) option is available only on Linux systems.

Storage Options command line option!–storageEngine string

--storageEngine string
Default: `wiredTiger`

New in version 3.0.

Changed in version 3.2: Starting in MongoDB 3.2, `wiredTiger` is the default.

Specifies the storage engine for the `mongod` (page 770) database. Available values include:

Value	Description
`mmapv1`	To specify the `https://docs.mongodb.org/manual/core/mmapv1`.
`wiredTiger`	To specify the `https://docs.mongodb.org/manual/core/wiredtiger`.
`inMemory`	To specify the `https://docs.mongodb.org/manual/core/inmemory`. The in-memory storage engine is currently in **beta**. Do not use in production. New in version 3.2: Available in MongoDB Enterprise only.

If you attempt to start a `mongod` (page 770) with a *`--dbpath`* (page 776) that contains data files produced by a storage engine other than the one specified by *`--storageEngine`* (page 775), `mongod` (page 770) will refuse to start.

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 770) instance stores its data.

If you installed MongoDB using a package management system, check the `/etc/mongod.conf` file provided by your packages to see the directory is specified.

Changed in version 3.0: The files in *`--dbpath`* (page 776) must correspond to the storage engine specified in *`--storageEngine`* (page 775). If the data files do not correspond to *`--storageEngine`* (page 775), `mongod` (page 770) will refuse to start.

command line option!–directoryperdb

`--directoryperdb`

Uses a separate directory to store data for each database. The directories are under the *`--dbpath`* (page 776) directory, and each subdirectory name corresponds to the database name.

Changed in version 3.0: To change the *`--directoryperdb`* (page 776) option for existing deployments, you must restart the `mongod` (page 770) instances with the new *`--directoryperdb`* (page 776) value **and** a new data directory (*`--dbpath <new path>`* (page 776)), and then repopulate the data.

- For standalone instances, you can use `mongodump` (page 816) on the existing instance, stop the instance, restart with the new *`--directoryperdb`* (page 776) value **and** a new data directory, and use `mongorestore` (page 824) to populate the new data directory.
- For replica sets, you can update in a rolling manner by stopping a secondary member, restart with the new *`--directoryperdb`* (page 776) value **and** a new data directory, and use *initial sync* to populate the new data directory. To update all members, start with the secondary members first. Then step down the primary, and update the stepped-down member.

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 776) option, MongoDB has a maximum of 8 data files per database. Adjust the quota with *`--quotaFiles`* (page 776).

command line option!–quotaFiles <number>

--quotaFiles <number>

Default: 8

Modifies the limit on the number of data files per database. *--quotaFiles* (page 776) option requires that you set *--quota* (page 776).

command line option!–smallfiles

--smallfiles

Sets MongoDB to use a smaller default file size. The *--smallfiles* (page 777) option reduces the initial size for data files and limits the maximum size to 512 megabytes. *--smallfiles* (page 777) also reduces the size of each *journal* file from 1 gigabyte to 128 megabytes. Use *--smallfiles* (page 777) if you have a large number of databases that each holds a small quantity of data.

The *--smallfiles* (page 777) option can lead the `mongod` (page 770) 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 777) to `0`, MongoDB will not sync the memory mapped files to disk.

The `mongod` (page 770) process writes data very quickly to the journal and lazily to the data files. *--syncdelay* (page 777) has no effect on the `journal` (page 915) files or `journaling`.

The `serverStatus` (page 492) command reports the background flush thread's status via the `backgroundFlushing` (page 494) field.

command line option!–upgrade

--upgrade

Upgrades the on-disk data format of the files specified by the *--dbpath* (page 776) to the latest version, if needed.

This option only affects the operation of the `mongod` (page 770) 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 462) database command on all databases.

Warning: During normal operations, only use the `repairDatabase` (page 462) command and wrappers including `db.repairDatabase()` (page 195) in the `mongo` (page 803) 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 462).

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

When using *journaling*, there is almost never any need to run `repairDatabase` (page 462). 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 770) instance refuses to start. In these cases you should start the `mongod` (page 770) without the `--repair` (page 821) option, which allows the `mongod` (page 770) 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 770) cleanly and restart with the `--repair` (page 821) option.

The `--repair` (page 821) option copies data from the source data files into new data files in the `repairPath` (page 915) 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` (page 915) option.

Specifies a working directory that MongoDB will use during the `--repair` (page 821) operation. After `--repair` (page 821) completes, the data files in option:–*dbpath* and the `--repairpath` (page 778) directory is empty.

The `--repairpath` (page 778) must be within the `dbPath` (page 915). You can specify a symlink to `--repairpath` (page 778) 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 776) option. The `mongod` (page 770) enables journaling by default on 64-bit builds of versions after 2.0.

command line option!–nojournal

--nojournal

Disables the durability journaling. The `mongod` (page 770) 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

Changed in version 3.2.

The maximum amount of time in milliseconds that the `mongod` (page 770) process allows between journal operations. Values can range from 1 to 500 milliseconds. Lower values increase the durability of the journal, at the expense of disk performance. The default journal commit interval is 100 milliseconds.

On MMAPv1, if the journal is on a different block device (e.g. physical volume, RAID device, or LVM volume) than the data files, the default journal commit interval is 30 milliseconds. Additionally, on MMAPv1, when a write operation with `j:true` is pending, `mongod` (page 770) will reduce `commitIntervalMs` (page 916) to a third of the set value.

On WiredTiger, the default journal commit interval is 100 milliseconds. Additionally, a write with `j:true` will cause an immediate sync of the journal.

WiredTiger Options command line option!–wiredTigerCacheSizeGB number

`--wiredTigerCacheSizeGB` `number`

New in version 3.0.

Defines the maximum size of the cache that WiredTiger will use for all data.

With WiredTiger, MongoDB utilizes both the WiredTiger cache and the filesystem cache.

Changed in version 3.2: Starting in MongoDB 3.2, the WiredTiger cache, by default, will use the larger of either:

- 60% of RAM minus 1 GB, or
- 1 GB.

For systems with up to 10 GB of RAM, the new default setting is less than or equal to the 3.0 default setting (For MongoDB 3.0, the WiredTiger cache uses either 1 GB or half of the installed physical RAM, whichever is larger).

For systems with more than 10 GB of RAM, the new default setting is greater than the 3.0 setting.

Via the filesystem cache, MongoDB automatically uses all free memory that is not used by the WiredTiger cache or by other processes. Data in the filesystem cache is compressed.

Avoid increasing the WiredTiger cache size above its default value.

Note: The *`--wiredTigerCacheSizeGB`* (page 779) only limits the size of the WiredTiger cache, not the total amount of memory used by `mongod` (page 770). The WiredTiger cache is only one component of the RAM used by MongoDB. MongoDB also automatically uses all free memory on the machine via the filesystem cache (data in the filesystem cache is compressed).

In addition, the operating system will use any free RAM to buffer filesystem blocks.

To accommodate the additional consumers of RAM, you may have to decrease WiredTiger cache size.

The default WiredTiger cache size value assumes that there is a single `mongod` (page 770) instance per node. If a single node contains multiple instances, then you should decrease the setting to accommodate the other `mongod` (page 770) instances.

If you run `mongod` (page 770) in a container (e.g. `lxc`, `cgroups`, Docker, etc.) that does *not* have access to all of the RAM available in a system, you must set *`--wiredTigerCacheSizeGB`* (page 779) to a value less than the amount of RAM available in the container. The exact amount depends on the other processes running in the container.

command line option!–wiredTigerStatisticsLogDelaySecs <seconds>

`--wiredTigerStatisticsLogDelaySecs` `<seconds>`

Default: 0

New in version 3.0.0.

Specifies the interval, in seconds, at which WiredTiger logs statistics to the file specified in the `dbPath` (page 915) or *`--dbpath`* (page 776). When *`--wiredTigerStatisticsLogDelaySecs`* (page 779) is set to `0`, WiredTiger does not log statistics.

command line option!–wiredTigerJournalCompressor <compressor>

`--wiredTigerJournalCompressor` `<compressor>`

Default: snappy

New in version 3.0.0.

Specifies the type of compression to use to compress WiredTiger journal data.

Available compressors are:

- `none`
- *snappy*
- *zlib*

command line option!–wiredTigerDirectoryForIndexes

`--wiredTigerDirectoryForIndexes`
New in version 3.0.0.

When you start `mongod` (page 770) with *`--wiredTigerDirectoryForIndexes`* (page 780), `mongod` (page 770) stores indexes and collections in separate subdirectories under the data (i.e. *`--dbpath`* ') directory. Specifically, `mongod` (page 770) stores the indexes in a subdirectory named `index` and the collection data in a subdirectory named `collection`.

By using a symbolic link, you can specify a different location for the indexes. Specifically, when `mongod` (page 770) instance is **not** running, move the `index` subdirectory to the destination and create a symbolic link named `index` under the data directory to the new destination.

command line option!–wiredTigerCollectionBlockCompressor <compressor>

`--wiredTigerCollectionBlockCompressor` `<compressor>`
Default: snappy

New in version 3.0.0.

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*

`--wiredTigerCollectionBlockCompressor` (page 780) affects all collections created. If you change the value of *`--wiredTigerCollectionBlockCompressor`* (page 780) on an existing MongoDB deployment, all new collections will use the specified compressor. Existing collections will continue to use the compressor specified when they were created, or the default compressor at that time.

command line option!–wiredTigerIndexPrefixCompression <boolean>

`--wiredTigerIndexPrefixCompression` `<boolean>`
Default: true

New in version 3.0.0.

Enables or disables *prefix compression* for index data.

Specify `true` for *`--wiredTigerIndexPrefixCompression`* (page 780) to enable *prefix compression* for index data, or `false` to disable prefix compression for index data.

The *`--wiredTigerIndexPrefixCompression`* (page 780) setting affects all indexes created. If you change the value of *`--wiredTigerIndexPrefixCompression`* (page 780) on an existing MongoDB deployment, all new indexes will use prefix compression. Existing indexes are not affected.

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 770) 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 770) has created the oplog for the first time, changing the *--oplogSize* (page 781) option will not affect the size of the oplog.

See *replica-set-oplog-sizing* for more information.

command line option!–replIndexPrefetch

--replIndexPrefetch
Default: all

Storage Engine Specific Feature

--replIndexPrefetch (page 781) 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.

Set this option to one of the following:

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.

command line option!–enableMajorityReadConcern

--enableMajorityReadConcern
New in version 3.2.

Enables *read concern* level of `"majority"`. By default, `"majority"` level is not enabled.

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 770) to run as a replication *master*.

command line option!–slave

--slave
Configures the `mongod` (page 770) to run as a replication *slave*.

command line option!–source <host><:port>

--source <host><:port>
For use with the *--slave* (page 781) 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 781) option, the `--only` option specifies only a single *database* to replicate.

command line option!–slavedelay <value>

--slavedelay `<value>`
For use with the `--slave` (page 781) option, the `--slavedelay` (page 782) 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 781) option. When set, the `--autoresync` (page 782) 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 781) 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 782), 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` (page 915) of another member of the set. Otherwise the `mongod` (page 770) 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 770) 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 770) 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 770) with this option is `27019` and the default `--dbpath` (page 776) directory is `/data/configdb`, unless specified.

If using `https://docs.mongodb.org/manual/core/mmapv1`, `--configsvr` (page 782) option also sets `--smallfiles` (page 777).

The `--configsvr` (page 782) option creates a local *oplog*.

Do not use the `--configsvr` (page 782) option with `--shardsvr` (page 783). Config servers cannot be a shard server.

Changed in version 3.2: Starting in MongoDB 3.2, config servers for sharded clusters can be deployed as a `replica set`. The replica set config servers must run the `WiredTiger storage engine`. MongoDB 3.2 deprecates the use of three mirrored `mongod` (page 770) instances for config servers.

In previous versions, `--configsvr` (page 782) could not be used in conjunction with `--replSet` (page 780).

command line option!–configsvrMode <string>

--configsvrMode `<string>`
New in version 3.2.

If set to `sccc`, indicates that the config servers are deployed as three mirrored `mongod` (page 770) instances, even if one or more config servers is also a member of a replica set. `configsvrMode` only accepts the value `sccc`.

If unset, config servers running as replica sets expect to use the "config server replica set" protocol for writing to config servers, rather than the "mirrored mongod" write protocol.

command line option!–shardsvr

`--shardsvr`

Configures this `mongod` (page 770) instance as a shard in a partitioned cluster. The default port for these instances is `27018`. The only effect of *`--shardsvr`* (page 783) is to change the port number.

command line option!–moveParanoia

`--moveParanoia`

If specified, during chunk migration, a shard saves, to the `moveChunk` directory of the `--dbpath`, all documents migrated from that shard.

MongoDB does not automatically delete the data saved in the `moveChunk` directory.

command line option!–noMoveParanoia

`--noMoveParanoia`

Changed in version 3.2: Starting in 3.2, MongoDB uses `--noMoveParanoia` as the default.

During chunk migration, a shard does not save documents migrated from the shard.

TLS/SSL Options

See

`https://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 TLS/SSL for `mongod` (page 770).

With *`--sslOnNormalPorts`* (page 797), a `mongod` (page 770) requires TLS/SSL encryption for all connections on the default MongoDB port, or the port specified by *`--port`* (page 868). By default, *`--sslOnNormalPorts`* (page 797) is disabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslMode <mode>

`--sslMode` `<mode>`

New in version 2.6.

Enables TLS/SSL or mixed TLS/SSL used for all network connections. The argument to the *`--sslMode`* (page 797) option can be one of the following:

Value	Description
disabled	The server does not use TLS/SSL.
allowSSL	Connections between servers do not use TLS/SSL. For incoming connections, the server accepts both TLS/SSL and non-TLS/non-SSL.
preferSSL	Connections between servers use TLS/SSL. For incoming connections, the server accepts both TLS/SSL and non-TLS/non-SSL.
requireSSL	The server uses and accepts only TLS/SSL encrypted connections.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile `<filename>`

Specifies the `.pem` file that contains both the TLS/SSL certificate and key. Specify the file name of the `.pem` file using relative or absolute paths.

You must specify *--sslPEMKeyFile* (page 868) when TLS/SSL is enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword `<value>`

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 868)). Use the *--sslPEMKeyPassword* (page 869) option only if the certificate-key file is encrypted. In all cases, the `mongod` (page 770) 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 869) option, the `mongod` (page 770) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 798) does not specify the `.pem` file for internal cluster authentication, the cluster uses the `.pem` file specified in the *`--sslPEMKeyFile`* (page 868) option.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 798) option only if the certificate-key file is encrypted. In all cases, the `mongod` (page 770) 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 798) option, the `mongod` (page 770) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

Warning: If the *`--sslCAFile`* option and its target file are not specified, x.509 client and member authentication will not function. `mongod` (page 770), and `mongos` (page 792) 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 770) 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 the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidCertificates

--sslAllowInvalidCertificates

New in version 2.6.

Bypasses the validation checks for TLS/SSL certificates on other servers in the cluster and allows the use of invalid certificates.

When using the *--sslAllowInvalidCertificates* (page 869) setting, MongoDB logs a warning regarding the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates, when connecting to other `mongod` (page 770) instances for inter-process authentication. This allows `mongod` (page 770) to connect to other `mongod` (page 770) instances if the hostnames in their certificates do not match their configured hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowConnectionsWithoutCertificates

--sslAllowConnectionsWithoutCertificates

New in version 2.4.

Changed in version 3.0.0: `--sslWeakCertificateValidation` became *--sslAllowConnectionsWithoutCertificates* (page 799). For compatibility, MongoDB processes continue to accept `--sslWeakCertificateValidation`, but all users should update their configuration files.

Disables the requirement for TLS/SSL certificate validation that `--sslCAFile` enables. With the *--sslAllowConnectionsWithoutCertificates* (page 799) option, the `mongod` (page 770) 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 770) has *--sslAllowConnectionsWithoutCertificates* (page 799) enabled, the `mongod` (page 770) will validate the certificate using the root certificate chain specified by `--sslCAFile` and reject clients with invalid certificates.

Use the *--sslAllowConnectionsWithoutCertificates* (page 799) option if you have a mixed deployment that includes clients that do not or cannot present certificates to the `mongod` (page 770).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslDisabledProtocols <protocol(s)>

--sslDisabledProtocols `<protocol(s)>`

New in version 3.0.7.

Prevents a MongoDB server running with SSL from accepting incoming connections that use a specific protocol or protocols. *--sslDisabledProtocols* (page 800) recognizes the following protocols: `TLS1_0`, `TLS1_1`, and `TLS1_2`. Specifying an unrecognized protocol will prevent the server from starting.

To specify multiple protocols, use a comma separated list of protocols.

Members of replica sets and sharded clusters must speak at least one protocol in common.

See also:

ssl-disallow-protocols

command line option!–sslFIPSMode

--sslFIPSMode

New in version 2.4.

Directs the `mongod` (page 770) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[2]. See `https://docs.mongodb.org/manual/tutorial/configure-fips` for more information.

Audit Options command line option!–auditDestination

--auditDestination

New in version 2.6.

Enables `auditing` and specifies where `mongod` (page 770) sends all audit events.

--auditDestination (page 801) 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 801) in the format specified in *--auditFormat* (page 801).

Note: Available only in MongoDB Enterprise[3].

command line option!–auditFormat

--auditFormat

New in version 2.6.

[2] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[3] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Specifies the format of the output file for `auditing` if *--auditDestination* (page 801) is `file`. The *--auditFormat* (page 801) 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 801).
BSON	Output the audit events in BSON binary format to the file specified in *--auditPath* (page 801).

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 801) has value of `file`. The *--auditPath* (page 801) 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 527).

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* (page 895), you must use the YAML format of the configuration file.

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 `https://docs.mongodb.org/manual/tutorial/monitor-wi`

command line option!–snmp-master

--snmp-master

Runs SNMP as a master. For more information, see `https://docs.mongodb.org/manual/tutorial/monitor-wit`

inMemory Options command line option!–inMemorySizeGB <integer>

[4]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[5]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[6]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

--inMemorySizeGB <integer>
Default: 60% of physical RAM less 1 GB

New in version 3.2.

Maximum amount of memory to allocate for `https://docs.mongodb.org/manual/core/inmemory` data and indexes.

command line option!–inMemoryStatisticsLogDelaySecs <integer>

--inMemoryStatisticsLogDelaySecs <integer>
Default: 0, do not log statistics.

New in version 3.2.

Seconds to wait between each write to a `https://docs.mongodb.org/manual/core/inmemory` statistics file in the *--dbpath* (page 776).

Encryption Key Management Options command line option!–enableEncryption <boolean>

--enableEncryption <boolean>
Default: False

New in version 3.2.

Enables encryption for the WiredTiger storage engine. You must set to `true` to pass in encryption keys and configurations.

Enterprise Feature

Available in MongoDB Enterprise only.

command line option!–encryptionCipherMode <string>

--encryptionCipherMode <string>
Default: AES256-CBC

New in version 3.2.

The cipher mode to use for encryption at rest:

Mode	Description
`AES256-CBC`	256-bit Advanced Encryption Standard in Cipher Block Chaining Mode
`AES256-GCM`	256-bit Advanced Encryption Standard in Galois/Counter Mode

Enterprise Feature

Available in MongoDB Enterprise only.

command line option!–encryptionKeyFile <string>

--encryptionKeyFile <string>
New in version 3.2.

The path to the local keyfile when managing keys via process *other than* KMIP. Only set when managing keys via process other than KMIP. If data is already encrypted using KMIP, MongoDB will throw an error.

Requires `enableEncryption` to be `true`.

Enterprise Feature

Available in MongoDB Enterprise only.

command line option!–kmipKeyIdentifier <string>

`--kmipKeyIdentifier` `<string>`

New in version 3.2.

Unique KMIP identifier for an existing key within the KMIP server. Include to use the key associated with the identifier as the system key. You can only use the setting the first time you enable encryption for the `mongod` (page 770) instance. Requires `enableEncryption` to be true.

If unspecified, MongoDB will request that the KMIP server create a new key to utilize as the system key.

If the KMIP server cannot locate a key with the specified identifier or the data is already encrypted with a key, MongoDB will throw an error

Enterprise Feature

Available in MongoDB Enterprise only.

command line option!–kmipRotateMasterKey <boolean>

`--kmipRotateMasterKey` `<boolean>`

Default: False

New in version 3.2.

If true, rotate the master key and re-encrypt the internal keystore.

Enterprise Feature

Available in MongoDB Enterprise only.

See also:

kmip-master-key-rotation

command line option!–kmipServerName <string>

`--kmipServerName` `<string>`

New in version 3.2.

Hostname or IP address of key management solution running a KMIP server. Requires `enableEncryption` to be true.

Enterprise Feature

Available in MongoDB Enterprise only.

command line option!–kmipPort <number>

`--kmipPort` `<number>`

Default: 5696

New in version 3.2.

Port number the KMIP server is listening on. Requires that a `kmipServerName` be provided. Requires `enableEncryption` to be true.

Enterprise Feature

Available in MongoDB Enterprise only.

command line option!–kmipClientCertificateFile <string>

--kmipClientCertificateFile <string>
New in version 3.2.

String containing the path to the client certificate used for authenticating MongoDB to the KMIP server. Requires that a `kmipServerName` be provided.

Enterprise Feature

Available in MongoDB Enterprise only.

command line option!–kmipClientCertificatePassword <string>

--kmipClientCertificatePassword <string>
New in version 3.2.

The password (if one exists) for the client certificate passed into `kmipClientCertificateFile`. Is used for authenticating MongoDB to the KMIP server. Requires that a `kmipClientCertificateFile` be provided.

Enterprise Feature

Available in MongoDB Enterprise only.

command line option!–kmipServerCAFile <string>

--kmipServerCAFile <string>
New in version 3.2.

Path to CA File. Used for validating secure client connection to KMIP server.

Text Search Options command line option!–basisTechRootDirectory <path>

--basisTechRootDirectory <path>
New in version 3.2.

Specify the root directory of the Basis Technology Rosette Linguistics Platform installation to support additional languages for text search operations.

Enterprise Feature

Available in MongoDB Enterprise only.

mongos

On this page

- Synopsis (page 791)
- Considerations (page 792)
- Options (page 792)

Synopsis

`mongos` (page 792) 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 792) instance behaves identically to any other MongoDB instance.

Considerations

Never change the name of the `mongos` (page 792) binary.

Options

`mongos`

Core Options

`mongos`

command line option!–help, -h

`--help, -h`
Returns information on the options and use of `mongos` (page 792).

command line option!–version

`--version`
Returns the `mongos` (page 792) 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 792). The options are equivalent to the command-line configuration options. See *Configuration File Options* (page 895) for more information.

Ensure the configuration file uses ASCII encoding. The `mongos` (page 792) 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 792) 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 (/etc/mongod.conf) that sets *--bind_ip* (page 793) to 127.0.0.1.

Specifies the IP address that mongos (page 792) binds to in order to listen for connections from applications. You may attach mongos (page 792) to any interface. When attaching mongos (page 792) 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 mongos (page 792) will accept. This setting has no effect if it is higher than your operating system's configured maximum connection tracking threshold.

Do not assign too low of a value to this option, or you will encounter errors during normal application operation.

This is particularly useful for a mongos (page 792) if you have a client that creates multiple connections and allows them to timeout rather than closing them.

In this case, set maxIncomingConnections (page 903) to a value slightly higher than the maximum number of connections that the client creates, or the maximum size of the connection pool.

This setting prevents the mongos (page 792) from causing connection spikes on the individual *shards*. Spikes like these may disrupt the operation and memory allocation of the *sharded cluster*.

Note: Changed in version 2.6: MongoDB removed the upward limit on the maxIncomingConnections (page 903) 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 793).

The *--syslog* (page 793) 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 793) 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 will move any existing log file rather than overwrite it. To instead append to the log file, set the *--logappend* (page 793) option.

command line option!–logappend

--logappend

Appends new entries to the end of the existing log file when the `mongos` (page 792) instance restarts. Without this option, `mongod` (page 770) will back up the existing log and create a new file.

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 792) process where `mongos` (page 792) will write its PID. This is useful for tracking the `mongos` (page 792) process in combination with the *`--fork`* (page 795) option. Without a specified *`--pidfilepath`* (page 794) 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 794) implies *`--auth`* (page 774). See *inter-process-auth* for more information.

command line option!–setParameter <options>

--setParameter `<options>`

Specifies one of the MongoDB parameters described in *MongoDB Server Parameters* (page 927). You can specify multiple `setParameter` fields.

command line option!–httpinterface

--httpinterface

Deprecated since version 3.2: HTTP interface for MongoDB

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:

- While MongoDB Enterprise does support Kerberos authentication, Kerberos is not supported in HTTP status interface in any version of MongoDB.

New in version 2.6.

command line option!–nounixsocket

--nounixsocket

Disables listening on the UNIX domain socket. *`--nounixsocket`* (page 794) applies only to Unix-based systems.

The `mongos` (page 792) process always listens on the UNIX socket unless one of the following is true:

- `--nounixsocket` (page 794) is set
- `net.bindIp` (page 903) is not set
- `net.bindIp` (page 903) does not specify `127.0.0.1`

New in version 2.6: `mongos` (page 792) 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. *`--unixSocketPrefix`* (page 795) applies only to Unix-based systems.

If this option has no value, the `mongos` (page 792) process creates a socket with `https://docs.mongodb.org/manual/tmp` as a prefix. MongoDB creates and listens on a UNIX socket unless one of the following is true:

- `net.unixDomainSocket.enabled` (page 904) is `false`
- `--nounixsocket` (page 794) is set
- `net.bindIp` (page 903) is not set
- `net.bindIp` (page 903) does not specify `127.0.0.1`

command line option!–filePermissions <path>

`--filePermissions` `<path>`

Default: `0700`

Sets the permission for the UNIX domain socket file.

`--filePermissions` (page 795) applies only to Unix-based systems.

command line option!–fork

`--fork`

Enables a *daemon* mode that runs the `mongos` (page 792) process in the background. By default `mongos` (page 792) does not run as a daemon: typically you will run `mongos` (page 792) as a daemon, either by using *`--fork`* (page 795) 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 <replicasetName>/<config1>,<config2>...

`--configdb` `<replicasetName>/<config1>,<config2>...`

Changed in version 3.2.

Specifies the *configuration servers* for the *sharded cluster*.

Starting in MongoDB 3.2, config servers for sharded clusters can be deployed as a `replica set`. The replica set config servers must run the `WiredTiger storage engine`. MongoDB 3.2 deprecates the use of three mirrored `mongod` (page 770) instances for config servers.

Specify the config server replica set name and the hostname and port of one of the members of the config server replica set.

The `mongos` (page 792) instances for the sharded cluster must specify the same config server replica set name but can specify hostname and port of different members of the replica set.

If using the deprecated mirrored instances, specify the hostnames and ports of the three `mongod` (page 770) instances. The `mongos` (page 792) instances must specify the same config string.

command line option!–localThreshold

--localThreshold

Default: 15

Specifies the ping time, in milliseconds, that `mongos` (page 792) uses to determine which secondary replica set members to pass read operations from clients. The default value of `15` corresponds to the default value in all of the client `drivers`.

When `mongos` (page 792) receives a request that permits reads to *secondary* members, the `mongos` (page 792) 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 796) option, `mongos` (page 792) 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 796) 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 792) 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.

`--chunkSize` (page 796) 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 `https://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

Disables `mongos` (page 792) from automatically splitting chunks for *sharded collections*. If set on all `mongos` (page 792) instances, this prevents MongoDB from creating new chunks as the data in a collection grows.

Because any `mongos` (page 792) in a cluster can create a split, to totally disable splitting in a cluster you must set *`--noAutoSplit`* (page 796) on all `mongos` (page 792).

Warning: With *`--noAutoSplit`* (page 796) specified, the data in your sharded cluster may become imbalanced over time. Use the option with caution.

TLS/SSL Options

See

https://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 TLS/SSL for `mongos` (page 792).

With *--sslOnNormalPorts* (page 797), a `mongos` (page 792) requires TLS/SSL encryption for all connections on the default MongoDB port, or the port specified by *--port* (page 868). By default, *--sslOnNormalPorts* (page 797) is disabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

command line option!–sslMode <mode>

--sslMode `<mode>`

New in version 2.6.

Enables TLS/SSL or mixed TLS/SSL used for all network connections. The argument to the *--sslMode* (page 797) option can be one of the following:

Value	Description
`disabled`	The server does not use TLS/SSL.
`allowSSL`	Connections between servers do not use TLS/SSL. For incoming connections, the server accepts both TLS/SSL and non-TLS/non-SSL.
`preferSSL`	Connections between servers use TLS/SSL. For incoming connections, the server accepts both TLS/SSL and non-TLS/non-SSL.
`requireSSL`	The server uses and accepts only TLS/SSL encrypted connections.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile `<filename>`

Specifies the `.pem` file that contains both the TLS/SSL certificate and key. Specify the file name of the `.pem` file using relative or absolute paths.

You must specify *--sslPEMKeyFile* (page 868) when TLS/SSL is enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword `<value>`

Specifies the password to de-crypt the certificate-key file (i.e. *--sslPEMKeyFile* (page 868)). Use the *--sslPEMKeyPassword* (page 869) option only if the certificate-key file is encrypted. In all cases, the `mongos` (page 792) 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 869) option, the mongos (page 792) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 798) does not specify the `.pem` file for internal cluster authentication, the cluster uses the `.pem` file specified in the *--sslPEMKeyFile* (page 868) option.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 798) option only if the certificate-key file is encrypted. In all cases, the mongos (page 792) 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 798) option, the mongos (page 792) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

Warning: If the *`--sslCAFile`* option and its target file are not specified, x.509 client and member authentication will not function. `mongod` (page 770), and `mongos` (page 792) 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 770) 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 the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowConnectionsWithoutCertificates

`--sslAllowConnectionsWithoutCertificates`

New in version 2.4.

Changed in version 3.0.0: `--sslWeakCertificateValidation` became *`--sslAllowConnectionsWithoutCertificates`* (page 799). For compatibility, MongoDB processes continue to accept `--sslWeakCertificateValidation`, but all users should update their configuration files.

Disables the requirement for TLS/SSL certificate validation that `--sslCAFile` enables. With the *`--sslAllowConnectionsWithoutCertificates`* (page 799) option, the `mongos` (page 792) 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 792) has *`--sslAllowConnectionsWithoutCertificates`* (page 799) enabled, the `mongos` (page 792) will validate the certificate using the root certificate chain specified by `--sslCAFile` and reject clients with invalid certificates.

Use the *`--sslAllowConnectionsWithoutCertificates`* (page 799) option if you have a mixed deployment that includes clients that do not or cannot present certificates to the `mongos` (page 792).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidCertificates

`--sslAllowInvalidCertificates`

New in version 2.6.

Bypasses the validation checks for TLS/SSL certificates on other servers in the cluster and allows the use of invalid certificates.

When using the *`--sslAllowInvalidCertificates`* (page 869) setting, MongoDB logs a warning regarding the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

`--sslAllowInvalidHostnames`

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates, when connecting to other `mongos` (page 792) instances for inter-process authentication. This allows `mongos` (page 792) to connect to other `mongos` (page 792) instances if the hostnames in their certificates do not match their configured hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslDisabledProtocols <protocol(s)>

`--sslDisabledProtocols` `<protocol(s)>`

New in version 3.0.7.

Prevents a MongoDB server running with SSL from accepting incoming connections that use a specific protocol or protocols. *`--sslDisabledProtocols`* (page 800) recognizes the following protocols: `TLS1_0`, `TLS1_1`, and `TLS1_2`. Specifying an unrecognized protocol will prevent the server from starting.

To specify multiple protocols, use a comma separated list of protocols.

Members of replica sets and sharded clusters must speak at least one protocol in common.

See also:

ssl-disallow-protocols

command line option!–sslFIPSMode

`--sslFIPSMode`

New in version 2.4.

Directs the `mongos` (page 792) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[7]. See `https://docs.mongodb.org/manual/tutorial/configure-fips` for more information.

[7] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Audit Options command line option!–auditDestination

--auditDestination

New in version 2.6.

Enables `auditing` and specifies where `mongos` (page 792) sends all audit events.

--auditDestination (page 801) 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 801) in the format specified in *--auditFormat* (page 801).

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 801) is `file`. The *--auditFormat* (page 801) 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 801).
`BSON`	Output the audit events in BSON binary format to the file specified in *--auditPath* (page 801).

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 801) has value of `file`. The *--auditPath* (page 801) 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.

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:

[8] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[9] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[10] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

```
{ <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 527).

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* (page 895), you must use the YAML format of the configuration file.

Note: Available only in MongoDB Enterprise[11].

Text Search Options command line option!–basisTechRootDirectory <path>

`--basisTechRootDirectory` `<path>`

New in version 3.2.

Specify the root directory of the Basis Technology Rosette Linguistics Platform installation to support additional languages for text search operations.

Enterprise Feature

Available in MongoDB Enterprise only.

Additional Options command line option!–ipv6

`--ipv6`

Enables IPv6 support and allows the `mongos` (page 792) 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 802) option enables the HTTP interface, even if the `HTTP interface` (page 905) option is disabled.

Deprecated since version 3.2: HTTP interface for MongoDB

command line option!–noscripting

`--noscripting`

Disables the scripting engine.

mongo

[11] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

On this page

- Description (page 803)
- Options (page 803)
- Files (page 808)
- Environment (page 809)
- Keyboard Shortcuts (page 809)
- Use (page 810)

Description

mongo

`mongo` (page 803) 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 803) also provides a fully functional JavaScript environment for use with a MongoDB. This document addresses the basic invocation of the `mongo` (page 803) 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 803) command and specify a JavaScript file as an argument, or use *`--eval`* (page 804) to specify JavaScript on the command line, the *`--shell`* (page 803) option provides the user with a shell prompt after the file finishes executing.

command line option!–nodb

--nodb
Prevents the shell from connecting to any database instances. Later, to connect to a database within the shell, see *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 770) or `mongos` (page 792) instance is listening. If *`--port`* (page 868) is not specified, `mongo` (page 803) attempts to connect to port `27017`.

command line option!–host <hostname>

--host `<hostname>`
Specifies the name of the host machine where the `mongod` (page 770) or `mongos` (page 792) is running. If this is not specified, `mongo` (page 803) 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 803) 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 870) and `--authenticationDatabase` (page 870) options. To force `mongo` (page 803) to prompt for a password, enter the `--password` (page 870) 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 803).

command line option!–version

--version
Returns the `mongo` (page 803) 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 803) 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 803) 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 `https://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 867) or `--port` (page 868).

command line option!–disableJavaScriptJIT

--disableJavaScriptJIT
New in version 3.2.

Disables use of the JavaScript engine's JIT compiler.

<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 803) to prompt you for a password using *--password* (page 870), pass the filename as the first parameter with *--username* (page 870) and *--password* (page 870) as the last options, as in the following:

```
mongo file.js --username username --password
```

Use the *--shell* (page 803) option to return to a shell after the file finishes running.

Authentication Options command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
Specifies the database in which the user is created. See *user-authentication-database*.

If you do not specify a value for *--authenticationDatabase* (page 870), mongo (page 803) uses the database specified in the connection string.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
Default: SCRAM-SHA-1

Changed in version 2.6: Added support for the PLAIN and MONGODB-X509 authentication mechanisms.

Changed in version 3.0: Added support for the SCRAM-SHA-1 authentication mechanism. Changed default mechanism to SCRAM-SHA-1.

Specifies the authentication mechanism the mongo (page 803) instance uses to authenticate to the mongod (page 770) or mongos (page 792).

Value	Description
SCRAM-SHA-1	RFC 5802[12] standard Salted Challenge Response Authentication Mechanism using the SHA1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[13].
PLAIN (LDAP SASL)	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[14].

command line option!–gssapiHostName

[12]https://tools.ietf.org/html/rfc5802
[13]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[14]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

--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.

TLS/SSL Options command line option!–ssl

--ssl

Enables connection to a `mongod` (page 770) or `mongos` (page 792) that has TLS/SSL support enabled.

Changed in version 3.0: When running `mongo` (page 803) with the `--ssl` option, you must include either `--sslCAFile` or `--sslAllowInvalidCertificates`.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile `<filename>`

New in version 2.4.

Specifies the `.pem` file that contains both the TLS/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 770) or `mongos` (page 792) that has `CAFile` (page 907) enabled *without* `allowConnectionsWithoutCertificates` (page 908).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword `<value>`

New in version 2.4.

Specifies the password to de-crypt the certificate-key file (i.e. `--sslPEMKeyFile`). Use the *`--sslPEMKeyPassword`* (page 869) option only if the certificate-key file is encrypted. In all cases, the `mongo` (page 803) 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 869) option, the `mongo` (page 803) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and

https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: When running mongo (page 803) with the --ssl option, you must include either --sslCAFile or --sslAllowInvalidCertificates.

Warning: For SSL connections (--ssl) to mongod (page 770) and mongos (page 792), if the mongo (page 803) shell runs without the --sslCAFile <CAFile> option (i.e. specifies the --sslAllowInvalidCertificates instead), the mongo (page 803) shell will not attempt to validate the server certificates. This creates a vulnerability to expired mongod (page 770) and mongos (page 792) certificates as well as to foreign processes posing as valid mongod (page 770) or mongos (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates in cases where intrusion is a possibility.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

command line option!–sslFIPSMode

--sslFIPSMode
New in version 2.6.

Directs the mongo (page 803) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[15]. See https://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 (page 908) setting, MongoDB logs as a warning the use of the invalid certificate.

[15] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Changed in version 3.0: When running `mongo` (page 803) with the `--ssl` option, you must include either `--sslCAFile` or `--sslAllowInvalidCertificates`.

> **Warning:** For SSL connections (`--ssl`) to `mongod` (page 770) and `mongos` (page 792), if the `mongo` (page 803) shell runs without the `--sslCAFile <CAFile>` option (i.e. specifies the `--sslAllowInvalidCertificates` instead), the `mongo` (page 803) shell will not attempt to validate the server certificates. This creates a vulnerability to expired `mongod` (page 770) and `mongos` (page 792) certificates as well as to foreign processes posing as valid `mongod` (page 770) or `mongos` (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates in cases where intrusion is a possibility.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

`--sslAllowInvalidHostnames`

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates. Allows `mongo` (page 803) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

Files

`~/.dbshell` `mongo` (page 803) maintains a history of commands in the `.dbshell` file.

> **Note:** `mongo` (page 803) does not recorded interaction related to authentication in the history file, including `authenticate` (page 372) and `db.createUser()` (page 230).

> **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 803) will read the `.mongorc.js` file from the home directory of the user invoking `mongo` (page 803). In the file, users can define variables, customize the `mongo` (page 803) 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 804) or by specifying *a .js file to mongo* (page 805), `mongo` (page 803) will read the `.mongorc.js` file *after* the JavaScript has finished processing.

Specify the *`--norc`* (page 803) option to disable reading `.mongorc.js`.

`/etc/mongorc.js` Global `mongorc.js` file which the `mongo` (page 803) shell evaluates upon start-up. If a user also has a `.mongorc.js` file located in the `HOME` (page 809) directory, the `mongo` (page 803) 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 803) option for `mongo` (page 803) suppresses only the user's `.mongorc.js` file.

On Windows, the global `mongorc.js </etc/mongorc.js>` exists in the `%ProgramData%\MongoDB` directory.

`https://docs.mongodb.org/manual/tmp/mongo_edit<time_t>.js` Created by `mongo` (page 803) when editing a file. If the file exists, `mongo` (page 803) 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 803) 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 809).

`HOME`
Specifies the path to the home directory where `mongo` (page 803) will read the `.mongorc.js` file and write the `.dbshell` file.

`HOMEDRIVE`
On Windows systems, `HOMEDRIVE` (page 809) specifies the path the directory where `mongo` (page 803) will read the `.mongorc.js` file and write the `.dbshell` file.

`HOMEPATH`
Specifies the Windows path to the home directory where `mongo` (page 803) will read the `.mongorc.js` file and write the `.dbshell` file.

Keyboard Shortcuts

The `mongo` (page 803) shell supports the following keyboard shortcuts: [16]

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 803) shell
Ctrl-D	Delete a char (or exit the `mongo` (page 803) 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

Continued on next page

[16] MongoDB accommodates multiple keybinding. Since 2.0, `mongo` (page 803) includes support for basic emacs keybindings.

Table 4.1 – continued from previous page

Keybinding	Function
Ctrl-K	Kill/erase the line
Ctrl-L or type `cls`	Clear the screen
Ctrl-M	Accept/evaluate the line
Ctrl-N	Retrieve next command from history
Ctrl-P	Retrieve previous command from history
Ctrl-R	Reverse-search command history
Ctrl-S	Forward-search command history
Ctrl-T	Transpose characters
Ctrl-U	Perform Unix line-discard
Ctrl-W	Perform Unix word-rubout
Ctrl-Y	Yank
Ctrl-Z	Suspend (job control works in linux)
Ctrl-H	Backward-delete a character
Ctrl-I	Complete, same as Tab
Meta-B	Go backward one word
Meta-C	Capitalize word
Meta-D	Kill word
Meta-F	Go forward one word
Meta-L	Change word to lowercase
Meta-U	Change word to uppercase
Meta-Y	Yank-pop
Meta-Backspace	Backward-kill word
Meta-<	Retrieve the first command in command history
Meta->	Retrieve the last command in command history

Use

Typically users invoke the shell with the `mongo` (page 803) 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 868) 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.

See also:

- `https://docs.mongodb.org/manual/reference/mongo-shell`
- *mongo Shell Methods* (page 19)
- `https://docs.mongodb.org/manual/mongo`

4.1.2 Windows Services

The `mongod.exe` (page 811) and `mongos.exe` (page 813) describe the options available for configuring MongoDB when running as a Windows Service. The `mongod.exe` (page 811) and `mongos.exe` (page 813) binaries provide a superset of the `mongod` (page 770) and `mongos` (page 792) options.

`mongod.exe`

On this page

- Synopsis (page 811)
- Options (page 811)

Synopsis

`mongod.exe` (page 811) is the build of the MongoDB daemon (i.e. `mongod` (page 770)) for the Windows platform. `mongod.exe` (page 811) has all of the features of `mongod` (page 770) on Unix-like platforms and is completely compatible with the other builds of `mongod` (page 770). In addition, `mongod.exe` (page 811) provides several options for interacting with the Windows platform itself.

This document *only* references options that are unique to `mongod.exe` (page 811). All `mongod` (page 770) options are available. See the *mongod* (page 769) and the *Configuration File Options* (page 895) documents for more information regarding `mongod.exe` (page 811).

To install and use `mongod.exe` (page 811), read the `https://docs.mongodb.org/manual/tutorial/install-mongod` document.

Options

mongod**.exe**

mongod**.exe**

command line option!–install

`--install`

Installs `mongod.exe` (page 811) as a Windows Service and exits.

If needed, you can install services for multiple instances of `mongod.exe` (page 811). Install each service with a unique *`--serviceName`* (page 813) and *`--serviceDisplayName`* (page 814). 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 811) Windows Service. If `mongod.exe` (page 811) is running, this operation will stop and then remove the service.

`--remove` (page 813) requires the *`--serviceName`* (page 813) if you configured a non-default *`--serviceName`* (page 813) during the *`--install`* (page 813) operation.

command line option!–reinstall

--reinstall

Removes `mongod.exe` (page 811) and reinstalls `mongod.exe` (page 811) as a Windows Service.

command line option!–serviceName name

--serviceName `name`

Default: MongoDB

Sets the service name of `mongod.exe` (page 811) when running as a Windows Service. Use this name with the `net start <name>` and `net stop <name>` operations.

You must use *`--serviceName`* (page 813) in conjunction with either the *`--install`* (page 813) or *`--remove`* (page 813) 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 811) service description.

You must use *`--serviceDescription`* (page 814) in conjunction with the *`--install`* (page 813) 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 811) service in the context of a certain user. This user must have “Log on as a service” privileges.

You must use *`--serviceUser`* (page 814) in conjunction with the *`--install`* (page 813) option.

command line option!–servicePassword <password>

--servicePassword `<password>`

Sets the password for `<user>` for `mongod.exe` (page 811) when running with the *`--serviceUser`* (page 814) option.

You must use *`--servicePassword`* (page 814) in conjunction with the *`--install`* (page 813) option.

mongos.exe

On this page

- Synopsis (page 813)
- Options (page 813)

Synopsis

`mongos.exe` (page 813) is the build of the MongoDB Shard (i.e. `mongos` (page 792)) for the Windows platform. `mongos.exe` (page 813) has all of the features of `mongos` (page 792) on Unix-like platforms and is completely compatible with the other builds of `mongos` (page 792). In addition, `mongos.exe` (page 813) provides several options for interacting with the Windows platform itself.

This document only references options that are unique to `mongos.exe` (page 813). All `mongos` (page 792) options are available. See the *mongos* (page 791) and the *Configuration File Options* (page 895) documents for more information regarding `mongos.exe` (page 813).

To install and use `mongos.exe` (page 813), read the `https://docs.mongodb.org/manual/tutorial/install-mongod` document.

Options

`mongos`**`.exe`**

`mongos`**`.exe`**

command line option!–install

`--install`

Installs `mongos.exe` (page 813) as a Windows Service and exits.

If needed, you can install services for multiple instances of `mongos.exe` (page 813). Install each service with a unique *`--serviceName`* (page 813) and *`--serviceDisplayName`* (page 814). 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 813) Windows Service. If `mongos.exe` (page 813) is running, this operation will stop and then remove the service.

`--remove` (page 813) requires the *`--serviceName`* (page 813) if you configured a non-default *`--serviceName`* (page 813) during the *`--install`* (page 813) operation.

command line option!–reinstall

`--reinstall`

Removes `mongos.exe` (page 813) and reinstalls `mongos.exe` (page 813) as a Windows Service.

command line option!–serviceName name

`--serviceName` `name`

Default: MongoS

Sets the service name of `mongos.exe` (page 813) when running as a Windows Service. Use this name with the `net start <name>` and `net stop <name>` operations.

You must use *`--serviceName`* (page 813) in conjunction with either the *`--install`* (page 813) or *`--remove`* (page 813) 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 813) service description.

You must use *--serviceDescription* (page 814) in conjunction with the *--install* (page 813) 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 813) service in the context of a certain user. This user must have "Log on as a service" privileges.

You must use *--serviceUser* (page 814) in conjunction with the *--install* (page 813) option.

command line option!–servicePassword <password>

--servicePassword <password>

Sets the password for <user> for mongos.exe (page 813) when running with the *--serviceUser* (page 814) option.

You must use *--servicePassword* (page 814) in conjunction with the *--install* (page 813) option.

4.1.3 Binary Import and Export Tools

mongodump (page 816) provides a method for creating *BSON* dump files from the mongod (page 770) instances, while mongorestore (page 824) makes it possible to restore these dumps. bsondump (page 833) converts BSON dump files into *JSON*. The mongooplog (page 835) utility provides the ability to stream *oplog* entries outside of normal replication.

mongodump

On this page

- Synopsis (page 814)
- Behavior (page 815)
- Required Access (page 815)
- Options (page 816)
- Examples (page 822)

Synopsis

mongodump (page 816) is a utility for creating a binary export of the contents of a database. mongodump (page 816) can export data from either mongod (page 770) or mongos (page 792) instances.

mongodump (page 816) can be a part of a *backup strategy* with mongorestore (page 824) for partial backups based on a query, syncing from production to staging or development environments, or changing the storage engine of a standalone. However, the use of mongodump (page 816) and mongorestore (page 824) as a backup strategy can be problematic for sharded clusters and replica sets.

For an overview of mongodump (page 816) in conjunction with mongorestore (page 824) part of a backup and recovery strategy, see https://docs.mongodb.org/manual/tutorial/backup-and-restore-tools.

See also:

mongorestore (page 824), https://docs.mongodb.org/manual/tutorial/backup-sharded-cluster-with-da and https://docs.mongodb.org/manual/core/backups.

Behavior

Data Exclusion mongodump (page 816) excludes the content of the local database in its output.

mongodump (page 816) only captures the documents in the database in its backup data and does not include index data. mongorestore (page 824) or mongod (page 770) must then rebuild the indexes after restoring data.

Version Compatibility The data format used by mongodump (page 816) from version 2.2 or later is *incompatible* with earlier versions of mongod (page 770). Do not use recent versions of mongodump (page 816) to back up older data stores.

Read Preference Changed in version 3.0.5: For a sharded cluster where the shards are replica sets, mongodump (page 816), when run against the mongos (page 792) instance, no longer prefers reads from secondary members.

Overwrite Files mongodump (page 816) overwrites output files if they exist in the backup data folder. Before running the mongodump (page 816) 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.

Data Compression Handling When run against a mongod (page 770) instance that uses the WiredTiger storage engine, mongodump (page 816) outputs uncompressed data.

Working Set mongodump (page 816) can adversely affect performance of the mongod (page 770). If your data is larger than system memory, the mongodump (page 816) will push the working set out of memory.

Required Access

To run mongodump (page 816) against a MongoDB deployment that has access control enabled, you must have privileges that grant find action for each database to back up. The built-in backup role provides the required privileges to perform backup of any and all databases.

Changed in version 3.2.1: The backup role provides additional privileges to back up the system.profile (page 893) collections that exist when running with *database profiling*. Previously, users required an additional read access on this collection.

Options

Changed in version 3.0.0: `mongodump` (page 816) removed the `--dbpath` as well as related `--directoryperdb` and `--journal` options. You must use `mongodump` (page 816) while connected to a `mongod` (page 770) instance.

`mongodump`

`mongodump`

command line option!–help

`--help`
Returns information on the options and use of `mongodump` (page 816).

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 816) 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 816) 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 770) to which to connect. By default, the `mongodump` (page 816) attempts to connect to a MongoDB instance running on the localhost on port number `27017`.

To connect to a replica set, specify the `replSetName` (page 922) and a seed list of set members, as in the following:

```
<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 3.0.0: 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 816) 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 770) or `mongos` (page 792) that has TLS/SSL support enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

Warning: For SSL connections (`--ssl`) to `mongod` (page 770) and `mongos` (page 792), if the `mongodump` (page 816) runs without the *`--sslCAFile`* (page 868), `mongodump` (page 816) will not attempt to validate the server certificates. This creates a vulnerability to expired `mongod` (page 770) and `mongos` (page 792) certificates as well as to foreign processes posing as valid `mongod` (page 770) or `mongos` (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates 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 TLS/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 868) option to connect to a `mongod` (page 770) or `mongos` (page 792) that has `CAFile` (page 907) enabled *without* `allowConnectionsWithoutCertificates` (page 908).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 868)). Use the *--sslPEMKeyPassword* (page 869) option only if the certificate-key file is encrypted. In all cases, the mongodump (page 816) 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 869) option, the mongodump (page 816) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 (page 908) setting, MongoDB logs as a warning the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates. Allows mongodump (page 816) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslFIPSMode

--sslFIPSMode

New in version 2.6.

Directs the mongodump (page 816) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[17]. See `https://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.

Changed in version 3.0.0: If you do not specify an argument for *`--password`* (page 870), `mongodump` (page 816) returns an error.

Changed in version 3.0.2: If you wish `mongodump` (page 816) to prompt the user for the password, pass the *`--username`* (page 870) option without *`--password`* (page 870) or specify an empty string as the *`--password`* (page 870) value, as in `--password ""`.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase `<dbname>`
If you do not specify an authentication database, `mongodump` (page 816) assumes that the database specified to export holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism `<name>`
Default: SCRAM-SHA-1

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism. Changed default mechanism to `SCRAM-SHA-1`.

Specifies the authentication mechanism the `mongodump` (page 816) instance uses to authenticate to the `mongod` (page 770) or `mongos` (page 792).

Value	Description
SCRAM-SHA-1	RFC 5802[18] standard Salted Challenge Response Authentication Mechanism using the SHA1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[19].
PLAIN (LDAP SASL)	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].

command line option!–gssapiServiceName

--gssapiServiceName
New in version 2.6.

[17]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[18]https://tools.ietf.org/html/rfc5802
[19]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[20]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

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 816) 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>

`--query` `<json>`, **`-q`** `<json>`

Provides a *JSON document* as a query that optionally limits the documents included in the output of `mongodump` (page 816).

You must enclose the query in single quotes (e.g. `'`) to ensure that it does not interact with your shell environment.

command line option!–queryFile <path>

`--queryFile` `<path>`

New in version 3.2.

Specifies the path to a file containing a JSON document as a query filter that limits the documents included in the output of `mongodump` (page 816). *`--queryFile`* (page 820) enables you to create query filters that are too large to fit in your terminal's buffer.

command line option!–forceTableScan

`--forceTableScan`

Forces `mongodump` (page 816) to scan the data store directly: typically, `mongodump` (page 816) saves entries as they appear in the index of the `_id` field. If you specify a query *`--query`* (page 854), `mongodump` (page 816) will use the most appropriate index to support that query.

Use *`--forceTableScan`* (page 855) 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 855), `mongodump` (page 816) does not use `$snapshot`. As a result, the dump produced by `mongodump` (page 816) can reflect the state of the database at many different points in time.

Important: Use *`--forceTableScan`* (page 855) with extreme caution and consideration.

command line option!–gzip

--gzip

New in version 3.2.

Compresses the output. If `mongodump` (page 816) outputs to the dump directory, the new feature compresses the individual files. The files have the suffix `.gz`.

If `mongodump` (page 816) outputs to an archive file or the standard out stream, the new feature compresses the archive file or the data output to the stream.

command line option!–out <path>, -o <path>

--out `<path>`**, -o** `<path>`

Specifies the directory where `mongodump` (page 816) will write *BSON* files for the dumped databases. By default, `mongodump` (page 816) 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 816) does not write the metadata that writes in a `<dbname>.metadata.json` file when writing to files directly.

You cannot use the `--archive` option with the *`--out`* (page 855) option.

command line option!–archive <file or null>

--archive `<file or null>`

New in version 3.2.

Writes the output to a single archive file or to the standard output (`stdout`).

To output the dump to an archive file, run `mongodump` (page 816) with the new `--archive` option and the archive filename.

To output the dump to the standard output stream in order to pipe to another process, run `mongodump` (page 816) with the `archive` option but *omit* the filename.

You cannot use the `--archive` option with the *`--out`* (page 855) option.

command line option!–repair

--repair

Runs a repair option in addition to dumping the database. The repair option changes the behavior of `mongodump` (page 816) to only write valid data and exclude data that may be in an invalid state as a result of an improper shutdown or `mongod` (page 770) crash.

The *`--repair`* (page 821) option uses aggressive data-recovery algorithms that may produce a large amount of duplication.

`--repair` (page 821) is only available for use with `mongod` (page 770) instances using the `mmapv1` storage engine. You cannot run *`--repair`* (page 821) with `mongos` (page 792) or with `mongod` (page 770) instances that use the `wiredTiger` storage engine. To repair data in a `mongod` (page 770) instance using `wiredTiger` use *`mongod --repair`*.

command line option!–oplog

--oplog

Creates a file named `oplog.bson` as part of the `mongodump` (page 816) output. The `oplog.bson` file, located in the top level of the output directory, contains oplog entries that occur during the `mongodump` (page 816) operation. This file provides an effective point-in-time snapshot of the state of a `mongod` (page 770) instance. To restore to a specific point-in-time backup, use the output created with this option in conjunction with *`mongorestore --oplogReplay`* (page 829).

Without *--oplog* (page 821), 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 821) has no effect when running mongodump (page 816) against a mongos (page 792) instance to dump the entire contents of a sharded cluster. However, you can use *--oplog* (page 821) to dump individual shards.

--oplog (page 821) 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 821) does not dump the oplog collection.

command line option!–dumpDbUsersAndRoles

--dumpDbUsersAndRoles
Includes user and role definitions in the database's dump directory when performing mongodump (page 816) on a specific database. This option applies only when you specify a database in the *--db* (page 828) option. MongoDB always includes user and role definitions when mongodump (page 816) 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 3.0.

Specifies collections to exclude from the output of mongodump (page 816) output.

command line option!–excludeCollectionsWithPrefix array of strings

--excludeCollectionsWithPrefix array of strings
New in version 3.0.

Excludes all collections from the output of mongodump (page 816) with a specified prefix.

Examples

mongodump a Collection 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  --db test --collection collection
```

mongodump with Access Control In the next example, mongodump (page 816) 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/
```

Output to an Archive File New in version 3.2.

To output the dump to an archive file, run mongodump (page 816) with the --archive option and the archive filename. For example, the following operation creates a file test.20150715.archive that contains the dump of the test database.

```
mongodump --archive=test.20150715.archive --db test
```

Output an Archive to Standard Output New in version 3.2.

To output the archive to the standard output stream in order to pipe to another process, run mongodump (page 816) with the archive option but *omit* the filename:

```
mongodump --archive --db test --port 27017 | mongorestore --archive --port 27018
```

Note: You cannot use the --archive option with the *--out* (page 855) option.

Compress the Output To compress the files in the output dump directory, run mongodump (page 816) with the new --gzip option. For example, the following operation outputs compressed files into the default dump directory.

```
mongodump --gzip --db test
```

To compress the archive file output by mongodump (page 816), use the --gzip option in conjunction with the *--archive* (page 831) option, specifying the name of the compressed file.

```
mongodump --archive=test.20150715.gz --gzip --db test
```

Additional Resources

- Backup and its Role in Disaster Recovery White Paper[21]
- Cloud Backup through MongoDB Cloud Manager[22]
- Blog Post: Backup vs. Replication, Why you Need Both[23]
- Backup Service with Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[24]

mongorestore

On this page

- Synopsis (page 823)
- Behavior (page 824)
- Required Access (page 824)
- Options (page 824)
- Examples (page 832)

Synopsis

The mongorestore (page 824) program writes data from a binary database dump created by mongodump (page 816) to a MongoDB instance.

New in version 3.0.0: mongorestore (page 824) also accepts data to restore via the standard input.

mongorestore (page 824) can write data to either mongod (page 770) or mongos (page 792) instances.

For an overview of mongorestore (page 824) usage, see https://docs.mongodb.org/manual/tutorial/backup-and

[21] https://www.mongodb.com/lp/white-paper/backup-disaster-recovery?jmp=docs
[22] https://cloud.mongodb.com/?jmp=docs
[23] http://www.mongodb.com/blog/post/backup-vs-replication-why-do-you-need-both?jmp=docs
[24] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

Behavior

Insert Only `mongorestore` (page 824) can create a new database or add data to an existing database. However, `mongorestore` (page 824) performs inserts only and does not perform updates. That is, if restoring documents to an existing database and collection and existing documents have the same value `_id` field as the to-be-restored documents, `mongorestore` (page 824) will *not* overwrite those documents.

Rebuild Indexes `mongorestore` (page 824) recreates indexes recorded by `mongodump` (page 816).

Version Compatibility The data format used by `mongodump` (page 816) from version 2.2 or later is *incompatible* with earlier versions of `mongod` (page 770). Do not use recent versions of `mongodump` (page 816) to back up older data stores.

Exclude `system.profile` Collection `mongorestore` (page 824) does not restore the `system.profile` (page 893) collection data; however, if the backup data includes `system.profile` (page 893) collection data and the collection does not exist in the target database, `mongorestore` (page 824) creates the collection but does not insert any data into the collection.

Required Access

To restore data to a MongoDB deployment that has `access control` enabled, the `restore` role provides access to restore any database if the backup data does not include `system.profile` (page 893) collection data.

If the backup data includes `system.profile` (page 893) collection data and the target database does not contain the `system.profile` (page 893) collection, `mongorestore` (page 824) attempts to create the collection even though the program does not actually restore `system.profile` documents. As such, the user requires additional privileges to perform `createCollection` and `convertToCapped` actions on the `system.profile` (page 893) collection for a database.

If running `mongorestore` (page 824) with *`--oplogReplay`* (page 829), additional privilege *user-defined role* that has `anyAction` on *resource-anyresource* and grant only to users who must run `mongorestore` (page 824) with *`--oplogReplay`* (page 829).

Options

Changed in version 3.0.0: `mongorestore` (page 824) removed the `--filter`, `--dbpath`, and the `--noobjcheck` options.

`mongorestore`

`mongorestore`

command line option!–help

`--help`

Returns information on the options and use of `mongorestore` (page 824).

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 824) 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 824) 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 770) to which to connect. By default, the `mongorestore` (page 824) attempts to connect to a MongoDB instance running on the localhost on port number `27017`.

To connect to a replica set, specify the `replSetName` (page 922) and a seed list of set members, as in the following:

```
<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 3.0.0: 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 824) 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 770) or `mongos` (page 792) that has TLS/SSL support enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

> **Warning:** For SSL connections (`--ssl`) to `mongod` (page 770) and `mongos` (page 792), if the `mongorestore` (page 824) runs without the *`--sslCAFile`* (page 868), `mongorestore` (page 824) will not attempt to validate the server certificates. This creates a vulnerability to expired `mongod` (page 770) and `mongos` (page 792) certificates as well as to foreign processes posing as valid `mongod` (page 770) or `mongos` (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates 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 TLS/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 868) option to connect to a `mongod` (page 770) or `mongos` (page 792) that has `CAFile` (page 907) enabled *without* `allowConnectionsWithoutCertificates` (page 908).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 868)). Use the *`--sslPEMKeyPassword`* (page 869) option only if the certificate-key file is encrypted. In all cases, the `mongorestore` (page 824) 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 869) option, the `mongorestore` (page 824) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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` (page 908) setting, MongoDB logs as a warning the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates. Allows `mongorestore` (page 824) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslFIPSMode

--sslFIPSMode

New in version 2.6.

Directs the `mongorestore` (page 824) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[25]. See `https://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.

Changed in version 3.0.0: If you do not specify an argument for *`--password`* (page 870), `mongorestore` (page 824) returns an error.

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

Changed in version 3.0.2: If you wish `mongorestore` (page 824) to prompt the user for the password, pass the *`--username`* (page 870) option without *`--password`* (page 870) or specify an empty string as the *`--password`* (page 870) value, as in `--password ""` .

command line option!–authenticationDatabase <dbname>

`--authenticationDatabase` `<dbname>`
Specifies the database in which the user is created. See *user-authentication-database*.

command line option!–authenticationMechanism <name>

`--authenticationMechanism` `<name>`
Default: SCRAM-SHA-1

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism. Changed default mechanism to `SCRAM-SHA-1`.

Specifies the authentication mechanism the `mongorestore` (page 824) instance uses to authenticate to the `mongod` (page 770) or `mongos` (page 792).

Value	Description
SCRAM-SHA-1	RFC 5802[26] standard Salted Challenge Response Authentication Mechanism using the SHA1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[27].
PLAIN (LDAP SASL)	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[28].

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 824) to restore data *into*. If the database does not exist, `mongorestore` (page 824) creates the database. If you do not specify a `<db>`, `mongorestore` (page 824)

[26] https://tools.ietf.org/html/rfc5802
[27] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[28] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

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 828) does *not* control which *BSON* files `mongorestore` (page 824) restores. You must use the `mongorestore` (page 824) *path option* (page 831) to limit that restored data.

command line option!–collection <collection>, -c <collection>

`--collection` `<collection>`, **`-c`** `<collection>`
Specifies a single collection for `mongorestore` (page 824) to restore. If you do not specify `--collection` (page 829), `mongorestore` (page 824) 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 824) 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 874) can have a small impact on performance.

Changed in version 2.4: MongoDB enables `--objcheck` (page 874) by default, to prevent any client from inserting malformed or invalid BSON into a MongoDB database.

command line option!–drop

`--drop`
Before restoring the collections from the dumped backup, drops the collections from the target database. `--drop` (page 829) does not drop collections that are not in the backup.

When the restore includes the `admin` database, `mongorestore` (page 824) with `--drop` (page 829) removes all user credentials and replaces them with the users defined in the dump file. Therefore, in systems with `authorization` (page 910) enabled, `mongorestore` (page 824) must be able to authenticate to an existing user *and* to a user defined in the dump file. If `mongorestore` (page 824) 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`
After restoring the database dump, replays the *oplog* entries from the `oplog.bson` file located in the top level of the dump directory. When used in conjunction with `mongodump --oplog` (page 821), `mongorestore --oplogReplay` (page 824) restores the database to the point-in-time backup captured with the `mongodump --oplog` command. For an example of `--oplogReplay` (page 829), see *backup-restore-oplogreplay*.

`mongorestore --oplogReplay` (page 824) replays any valid `oplog.bson` file found in the top level of the dump directory. That is, if you have a bson file that contains valid oplog entries, you can name the file `oplog.bson` and move it to the top level of the dump directory for `mongorestore --oplogReplay` (page 824) to replay.

See also:

mongorestore Required Access (page 824)

command line option!–oplogLimit <timestamp>

`--oplogLimit` `<timestamp>`
New in version 2.2.

Prevents `mongorestore` (page 824) 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 829) in conjunction with the `--oplogReplay` (page 829) option.

command line option!–keepIndexVersion

--keepIndexVersion
Prevents `mongorestore` (page 824) 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 824) from restoring and building indexes as specified in the corresponding `mongodump` (page 816) output.

command line option!–noOptionsRestore

--noOptionsRestore
New in version 2.2.

Prevents `mongorestore` (page 824) from setting the collection options, such as those specified by the `collMod` (page 457) *database command*, on restored collections.

command line option!–restoreDbUsersAndRoles

--restoreDbUsersAndRoles
Restore user and role definitions for the given database. See `https://docs.mongodb.org/manual/reference/syste` and `https://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 824) writes to the target database. By default, `mongorestore` (page 824) does not wait for a `write acknowledgment`.

command line option!–writeConcern <document>

--writeConcern `<document>`
Default: majority

Specifies the *write concern* for each write operation that `mongorestore` (page 824) 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 824) inserts the documents in the order of their appearance in the input source, otherwise `mongorestore` (page 824) 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 824) should restore in parallel.

If you specify `-j` when restoring a *single* collection, `-j` maps to the *`--numInsertionWorkersPerCollection`* (page 831) option rather than *`--numParallelCollections`* (page 830).

command line option!–numInsertionWorkersPerCollection int

--numInsertionWorkersPerCollection `int`

Default: 1

New in version 3.0.0.

Specifies the number of insertion workers to run concurrently per collection.

For large imports, increasing the number of insertion workers may increase the speed of the import.

command line option!–stopOnError

--stopOnError

New in version 3.0.

Forces `mongorestore` (page 824) to halt the restore when it encounters an error.

command line option!–bypassDocumentValidation

--bypassDocumentValidation

Enables `mongorestore` (page 824) to bypass `document validation` during the operation. This lets you insert documents that do not meet the validation requirements.

New in version 3.2.1.

command line option!–gzip

--gzip

New in version 3.2.

Restores from compressed files or data stream created by `mongodump --archive` (page 816)

To restore from a dump directory that contains compressed files, run `mongorestore` (page 824) with the new `--gzip` option.

To restore from a compressed archive file, run `mongorestore` (page 824) with the `--gzip` option in conjunction with the `--archive` option.

<path>

The final argument of the `mongorestore` (page 824) command is a directory path. This argument specifies the location of the database dump from which to restore.

You cannot specify both the `<path>` argument and the `--dir` option, which also specifies the dump directory, to `mongorestore` (page 824).

command line option!–archive <file or null>

--archive `<file or null>`

New in version 3.2.

Restores from an archive file or from the standard input (`stdin`).

To restore from an archive file, run `mongorestore` (page 824) with the `--archive` option and the archive filename.

To restore from the standard input, run `mongorestore` (page 824) with the `archive` option but *omit* the filename.

Note:

- You cannot use the `--archive` option with the `--dir` option.
- `mongorestore` (page 824) still supports the positional `-` parameter to restore a *single* collection from the standard input.

command line option!–dir string

--dir string
Specifies the dump directory.

- You cannot specify both the `--dir` option and the `<path>` argument, which also specifies the dump directory, to `mongorestore` (page 824).
- You cannot use the `--archive` option with the `--dir` option.

Examples

Restore a Collection Consider the following example:

```
mongorestore --collection people --db accounts dump/accounts/people.bson
```

Here, `mongorestore` (page 824) 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 824) restores data to the instance running on the localhost interface on port `27017`.

Restore with Access Control In the following example, `mongorestore` (page 824) 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 824) 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/mon
```

Restore a Collection from Standard Input You can also *pipe* data directly into to `mongorestore` (page 824) through standard input, as in the following example:

```
zcat /opt/backup/mongodump-2014-12-03/accounts.people.bson.gz | mongorestore --collection people --db
```

Restore a Database from an Archive File New in version 3.2.

To restore from an archive file, run `restore` with the new `--archive` option and the archive filename. For example, the following operation restores the `test` database from the file `test.20150715.archive`.

```
mongorestore --archive=test.20150715.archive --db test
```

Restore a Database from Standard Input New in version 3.2.

To restore from the standard input, run `mongorestore` (page 824) with the `archive` option but *omit* the filename. For example:

```
mongodump --archive --db test --port 27017 | mongorestore --archive --port 27018
```

Restore from Compressed Data New in version 3.2: With the `--gzip` option, `mongorestore` (page 824) can restore from compressed files or data stream created by `mongodump` (page 816).

To restore from a dump directory that contains compressed files, run `mongorestore` (page 824) with the new `--gzip` option. For example, the following operation restores the `test` database from the compressed files located in the default `dump` directory:

```
mongorestore --gzip --db test
```

To restore from a compressed archive file, run `mongorestore` (page 824) with the `--gzip` option in conjunction with the new `--archive` option. For example, the following operation restores the `test` database from the archive file `test.20150715.gz`.

```
mongorestore --gzip --archive=test.20150715.gz --db test
```

Additional Resources

- Backup and its Role in Disaster Recovery White Paper[29]
- Cloud Backup through MongoDB Cloud Manager[30]
- Blog Post: Backup vs. Replication, Why you Need Both[31]
- Backup Service with Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[32]

bsondump

On this page

- Synopsis (page 833)
- Options (page 833)
- Use (page 834)

Synopsis

The `bsondump` (page 833) converts *BSON* files into human-readable formats, including *JSON*. For example, `bsondump` (page 833) is useful for reading the output files generated by `mongodump` (page 816).

Important: `bsondump` (page 833) is a diagnostic tool for inspecting BSON files, not a tool for data ingestion or other application use.

Options

Changed in version 3.0.0: `bsondump` (page 833) removed the `--filter`, `--dbpath` and the `--noobjcheck` options.

bsondump

bsondump

command line option!–help

--help
Returns information on the options and use of `bsondump` (page 833).

[29]https://www.mongodb.com/lp/white-paper/backup-disaster-recovery?jmp=docs
[30]https://cloud.mongodb.com/?jmp=docs
[31]http://www.mongodb.com/blog/post/backup-vs-replication-why-do-you-need-both?jmp=docs
[32]https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

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 833) 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 833) release number.

command line option!–objcheck

--objcheck
Validates each *BSON* object before outputting it in *JSON* format. By default, `bsondump` (page 833) enables `--objcheck` (page 874). For objects with a high degree of sub-document nesting, `--objcheck` (page 874) can have a small impact on performance.

Changed in version 2.4: MongoDB enables `--objcheck` (page 874) by default, to prevent any client from inserting malformed or invalid BSON into a MongoDB database.

command line option!–type <=json|=debug>

--type `<=json|=debug>`
Changes the operation of `bsondump` (page 833) from outputting "*JSON*" (the default) to a debugging format.

command line option!–pretty

--pretty
New in version 3.0.0.

Outputs documents in a pretty-printed format JSON.

<bsonFilename>
The final argument to `bsondump` (page 833) is a document containing *BSON*. This data is typically generated by `bsondump` (page 833) or by MongoDB in a *rollback* operation.

Use

By default, `bsondump` (page 833) 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

On this page

- Synopsis (page 835)
- Options (page 835)
- Use (page 840)

New in version 2.2.

Synopsis

`mongooplog` (page 835) 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 770) 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 816) and `mongorestore` (page 824) or another `backup` operation, which may be better suited to your operation.

Note: If the `mongod` (page 770) instance specified by the `--from` argument is running with `authentication`, then `mongooplog` (page 835) will not be able to copy oplog entries.

See also:

`mongodump` (page 816), `mongorestore` (page 824), `https://docs.mongodb.org/manual/core/backups`, `https://docs.mongodb.org/manual/core/replica-set-oplog`.

Options

Changed in version 3.0.0: `mongooplog` (page 835) removed the `--dbpath` as well as related `--directoryperdb` and `--journal` options. You must use `mongooplog` (page 835) while connected to a `mongod` (page 770) instance.

mongooplog

mongooplog

command line option!–help

--help
Returns information on the options and use of `mongooplog` (page 835).

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 835) 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 835) 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 770) instance to which `mongooplog` (page 835) will apply *oplog* operations retrieved from the server specified by the `--from` option.

By default `mongooplog` (page 835) 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 770) instance where `mongooplog` (page 835) 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 835) 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 770) or `mongos` (page 792) that has TLS/SSL support enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and

`https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

> **Warning:** For SSL connections (`--ssl`) to `mongod` (page 770) and `mongos` (page 792), if the `mongooplog` (page 835) runs without the *`--sslCAFile`* (page 868), `mongooplog` (page 835) will not attempt to validate the server certificates. This creates a vulnerability to expired `mongod` (page 770) and `mongos` (page 792) certificates as well as to foreign processes posing as valid `mongod` (page 770) or `mongos` (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates 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 TLS/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 868) option to connect to a `mongod` (page 770) or `mongos` (page 792) that has `CAFile` (page 907) enabled *without* `allowConnectionsWithoutCertificates` (page 908).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 868)). Use the *`--sslPEMKeyPassword`* (page 869) option only if the certificate-key file is encrypted. In all cases, the `mongooplog` (page 835) 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 869) option, the `mongooplog` (page 835) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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` (page 908) setting, MongoDB logs as a warning the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

`--sslAllowInvalidHostnames`

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates. Allows `mongooplog` (page 835) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslFIPSMode

`--sslFIPSMode`

New in version 2.6.

Directs the `mongooplog` (page 835) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[33]. See `https://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.

Changed in version 3.0.0: If you do not specify an argument for *`--password`* (page 870), `mongooplog` (page 835) returns an error.

Changed in version 3.0.2: If you wish `mongooplog` (page 835) to prompt the user for the password, pass the *`--username`* (page 870) option without *`--password`* (page 870) or specify an empty string as the *`--password`* (page 870) value, as in `--password ""`.

command line option!–authenticationDatabase <dbname>

`--authenticationDatabase` `<dbname>`

Specifies the database in which the user is created. See *user-authentication-database*.

command line option!–authenticationMechanism <name>

`--authenticationMechanism` `<name>`

Default: SCRAM-SHA-1

[33] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism. Changed default mechanism to `SCRAM-SHA-1`.

Specifies the authentication mechanism the `mongooplog` (page 835) instance uses to authenticate to the `mongod` (page 770) or `mongos` (page 792).

Value	Description
SCRAM-SHA-1	RFC 5802[34] standard Salted Challenge Response Authentication Mechanism using the SHA1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[35].
PLAIN (LDAP SASL)	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].

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!–seconds <number>, -s <number>

`--seconds` `<number>`, **`-s`** `<number>`
Specify a number of seconds of operations for `mongooplog` (page 835) 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 835) to retrieve *oplog* operations from. `mongooplog` (page 835) *requires* this option.

Unless you specify the `--host` option, `mongooplog` (page 835) will apply the operations collected with this option to the oplog of the `mongod` (page 770) instance running on the localhost interface connected to port `27017`.

command line option!–oplogns <namespace>

[34] https://tools.ietf.org/html/rfc5802
[35] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[36] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

--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 839) to apply oplog entries stored in another location. Namespaces take the form of `[database].[collection]`.

Use

Consider the following prototype `mongooplog` (page 835) command:

```
mongooplog  --from mongodb0.example.net --host mongodb1.example.net
```

Here, entries from the *oplog* of the `mongod` (page 770) 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 835) 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 841) provides a method for taking data in *JSON*, *CSV*, or *TSV* and importing it into a `mongod` (page 770) instance. `mongoexport` (page 850) provides a method to export data from a `mongod` (page 770) instance into JSON, CSV, or TSV.

Note: The conversion between BSON and other formats lacks full type fidelity. Therefore you cannot use `mongoimport` (page 841) and `mongoexport` (page 850) for round-trip import and export operations.

mongoimport

On this page

- Synopsis (page 840)
- Considerations (page 841)
- Required Access (page 841)
- Options (page 841)
- Use (page 848)

Synopsis

The `mongoimport` (page 841) tool imports content from an *Extended JSON* (page 960), CSV, or TSV export created by `mongoexport` (page 850), or potentially, another third-party export tool.

See *human-intelligible-exports* for more in-depth usage overview, and the *mongoexport* (page 849) document for more information regarding `mongoexport` (page 850), which provides the inverse "exporting" capability.

Considerations

> **Warning:** Avoid using mongoimport (page 841) and mongoexport (page 850) for full instance production backups. They do not reliably preserve all rich *BSON* data types, because *JSON* can only represent a subset of the types supported by BSON. Use mongodump (page 816) and mongorestore (page 824) as described in https://docs.mongodb.org/manual/core/backups for this kind of functionality.

To preserve type information, mongoexport (page 850) and mongoimport (page 841) uses the *strict mode representation* (page 960) for certain types.

For example, the following insert operation in the mongo (page 803) shell uses the *shell mode representation* (page 960) for the BSON types data_date (page 961) and data_numberlong (page 963):

```
use test
db.traffic.insert( { _id: 1, volume: NumberLong('2980000'), date: new Date() } )
```

The argument to data_numberlong (page 963) must be quoted to avoid potential loss of accuracy.

Use mongoexport (page 850) to export the data:

```
mongoexport --db test --collection traffic --out traffic.json
```

The exported data is in *strict mode representation* (page 960) to preserve type information:

```
{ "_id" : 1, "volume" : { "$numberLong" : "2980000" }, "date" : { "$date" : "2014-03-13T13:47:42.483-
```

See *MongoDB Extended JSON* (page 960) for a complete list of these types and the representations used.

Required Access

In order to connect to a mongod (page 770) that enforces authorization with the `--auth` option, you must use the `--username` and `--password` options. The connecting user must possess, at a minimum, the `readWrite` role on the database into which they are importing data.

Options

Changed in version 3.0.0: mongoimport (page 841) removed the `--dbpath` as well as related `--directoryperdb` and `--journal` options. You must use mongoimport (page 841) while connected to a mongod (page 770) instance.

`mongoimport`

`mongoimport`

command line option!–help

`--help`

Returns information on the options and use of mongoimport (page 841).

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 841) 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 841) 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 770) to which to connect. By default, the `mongoimport` (page 841) attempts to connect to a MongoDB instance running on the localhost on port number `27017`.

To connect to a replica set, specify the `replSetName` (page 922) and a seed list of set members, as in the following:

```
<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 3.0.0: 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 841) 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 770) or `mongos` (page 792) that has TLS/SSL support enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

> **Warning:** For SSL connections (--ssl) to mongod (page 770) and mongos (page 792), if the mongoimport (page 841) runs without the *--sslCAFile* (page 868), mongoimport (page 841) will not attempt to validate the server certificates. This creates a vulnerability to expired mongod (page 770) and mongos (page 792) certificates as well as to foreign processes posing as valid mongod (page 770) or mongos (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates 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 TLS/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 868) option to connect to a mongod (page 770) or mongos (page 792) that has CAFile (page 907) enabled *without* allowConnectionsWithoutCertificates (page 908).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

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 868)). Use the *--sslPEMKeyPassword* (page 869) option only if the certificate-key file is encrypted. In all cases, the mongoimport (page 841) 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 869) option, the mongoimport (page 841) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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` (page 908) setting, MongoDB logs as a warning the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

`--sslAllowInvalidHostnames`

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates. Allows `mongoimport` (page 841) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslFIPSMode

`--sslFIPSMode`

New in version 2.6.

Directs the `mongoimport` (page 841) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[37]. See `https://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.

Changed in version 3.0.0: If you do not specify an argument for *`--password`* (page 870), `mongoimport` (page 841) returns an error.

[37] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Changed in version 3.0.2: If you wish `mongoimport` (page 841) to prompt the user for the password, pass the *`--username`* (page 870) option without *`--password`* (page 870) or specify an empty string as the *`--password`* (page 870) value, as in `--password ""`.

command line option!–authenticationDatabase <dbname>

`--authenticationDatabase` `<dbname>`
Specifies the database in which the user is created. See *user-authentication-database*.

command line option!–authenticationMechanism <name>

`--authenticationMechanism` `<name>`
Default: SCRAM-SHA-1

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism. Changed default mechanism to `SCRAM-SHA-1`.

Specifies the authentication mechanism the `mongoimport` (page 841) instance uses to authenticate to the `mongod` (page 770) or `mongos` (page 792).

Value	Description
SCRAM-SHA-1	RFC 5802[38] standard Salted Challenge Response Authentication Mechanism using the SHA1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[39].
PLAIN (LDAP SASL)	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[40].

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 `mongoimport` (page 841).

[38]https://tools.ietf.org/html/rfc5802
[39]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[40]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

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 829), `mongoimport` (page 841) 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 846) when importing JSON data, `mongoimport` (page 841) will return an error. *--fields* (page 846) is only for *csv* or *tsv* imports.

command line option!–fieldFile <filename>

--fieldFile `<filename>`
As an alternative to *--fields* (page 846), the *--fieldFile* (page 846) 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 846) when importing JSON data, `mongoimport` (page 841) will return an error. *--fieldFile* (page 846) 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 841) creates fields without values in imported documents.

If you attempt to include *--ignoreBlanks* (page 846) when importing JSON data, `mongoimport` (page 841) will return an error. *--ignoreBlanks* (page 846) 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**[41]. 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>

--file `<filename>`
Specifies the location and name of a file containing the data to import. If you do not specify a file, `mongoimport` (page 841) 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

[41] http://tools.ietf.org/html/rfc4180.html

--headerline

If using *--type csv* or *--type tsv*, uses the first line as field names. Otherwise, mongoimport (page 841) will import the first line as a distinct document.

If you attempt to include *--headerline* (page 846) when importing JSON data, mongoimport (page 841) will return an error. *--headerline* (page 846) 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 847) mongoimport (page 841) will upsert on the basis of the _id field.

Depending on your MongoDB configuration, *--upsert* (page 847) may impact your mongod (page 770)'s performance.

Changed in version 3.0.0: *--upsertFields* (page 847) now implies *--upsert* (page 847). As such, you may prefer to use *--upsertFields* (page 847) instead of *--upsert* (page 847).

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.

Changed in version 3.0.0: Modifies the import process to update existing objects in the database if they match based on the specified fields, while inserting all other objects. You do not need to use *--upsert* (page 847) with *--upsertFields* (page 847).

If you do not specify a field, *--upsertFields* (page 847) 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 841) to halt the insert operation at the first error rather than continuing the operation despite errors.

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 847) in conjunction with *mongoexport --jsonArray*.

command line option!–maintainInsertionOrder

--maintainInsertionOrder

Default: False

If specified, mongoimport (page 841) inserts the documents in the order of their appearance in the input source, otherwise mongoimport (page 841) may perform the insertions in an arbitrary order.

command line option!–numInsertionWorkers int

--numInsertionWorkers int

Default: 1

New in version 3.0.0.

Specifies the number of insertion workers to run concurrently.

For large imports, increasing the number of insertion workers may increase the speed of the import.

command line option!–writeConcern <document>

--writeConcern <document>
Default: majority

Specifies the *write concern* for each write operation that mongoimport (page 841) writes to the target database.

Specify the write concern as a document with *w options*.

command line option!–bypassDocumentValidation

--bypassDocumentValidation
Enables mongoimport (page 841) to bypass document validation during the operation. This lets you insert documents that do not meet the validation requirements.

New in version 3.2.1.

Use

Simple Usage mongoimport (page 841) restores a database from a backup taken with mongoexport (page 850). Most of the arguments to mongoexport (page 850) also exist for mongoimport (page 841).

In the following example, mongoimport (page 841) imports the data in the *JSON* data from the contacts.json file into the collection contacts in the users database.

```
mongoimport --db users --collection contacts --file contacts.json
```

Import JSON to Remote Host Running with Authentication In the following example, mongoimport (page 841) imports data from the file /opt/backups/mdb1-examplenet.json into the contacts collection within the database marketing on a remote MongoDB database with authentication enabled.

mongoimport (page 841) connects to the mongod (page 770) instance running on the host mongodb1.example.net over port 37017. It authenticates with the username user and the password pass.

```
mongoimport --host mongodb1.example.net --port 37017 --username user --password pass --collection co
```

CSV Import In the following example, mongoimport (page 841) imports the *csv* formatted data in the /opt/backups/contacts.csv file into the collection contacts in the users database on the MongoDB instance running on the localhost port numbered 27017.

Specifying *--headerline* instructs mongoimport (page 841) to determine the name of the fields using the first line in the CSV file.

```
mongoimport --db users --collection contacts --type csv --headerline --file /opt/backups/contacts.cs
```

mongoimport (page 841) uses the input file name, without the extension, as the collection name if -c or --collection is unspecified. The following example is therefore equivalent:

```
mongoimport --db users --type csv --headerline --file /opt/backups/contacts.csv
```

Use the "*--ignoreBlanks*" option to ignore blank fields. For *CSV* and *TSV* imports, this option provides the desired functionality in most cases because it avoids inserting fields with null values into your collection.

mongoexport

On this page

- Synopsis (page 849)
- Considerations (page 849)
- Required Access (page 849)
- Options (page 850)
- Use (page 856)

Synopsis

mongoexport (page 850) is a utility that produces a JSON or CSV export of data stored in a MongoDB instance. See *human-intelligible-exports* for a more in-depth usage overview, and the *mongoimport* (page 840) document for more information regarding the mongoimport (page 841) utility, which provides the inverse "importing" capability.

Considerations

Warning: Avoid using mongoimport (page 841) and mongoexport (page 850) for full instance production backups. They do not reliably preserve all rich *BSON* data types, because *JSON* can only represent a subset of the types supported by BSON. Use mongodump (page 816) and mongorestore (page 824) as described in https://docs.mongodb.org/manual/core/backups for this kind of functionality.

To preserve type information, mongoexport (page 850) and mongoimport (page 841) uses the *strict mode representation* (page 960) for certain types.

For example, the following insert operation in the mongo (page 803) shell uses the *shell mode representation* (page 960) for the BSON types data_date (page 961) and data_numberlong (page 963):

```
use test
db.traffic.insert( { _id: 1, volume: NumberLong('2980000'), date: new Date() } )
```

The argument to data_numberlong (page 963) must be quoted to avoid potential loss of accuracy.

Use mongoexport (page 850) to export the data:

```
mongoexport --db test --collection traffic --out traffic.json
```

The exported data is in *strict mode representation* (page 960) to preserve type information:

```
{ "_id" : 1, "volume" : { "$numberLong" : "2980000" }, "date" : { "$date" : "2014-03-13T13:47:42.483-
```

See *MongoDB Extended JSON* (page 960) for a complete list of these types and the representations used.

Required Access

In order to connect to a mongod (page 770) that enforces authorization with the `--auth` option, you must use the `--username` and `--password` options. The connecting user must possess at a minimum, the `read` role on the database that they are exporting.

Options

Changed in version 3.0.0: `mongoexport` (page 850) removed the `--dbpath` as well as related `--directoryperdb` and `--journal` options. You must use `mongoexport` (page 850) while connected to a `mongod` (page 770) instance.

Changed in version 3.0.0: `mongoexport` (page 850) removed the `--csv` option. Use the `--type=csv` (page 846) option to specify CSV format for the output.

mongoexport

mongoexport

command line option!–help

--help
Returns information on the options and use of `mongoexport` (page 850).

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 850) 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 850) 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 770) to which to connect. By default, the `mongoexport` (page 850) attempts to connect to a MongoDB instance running on the localhost on port number `27017`.

To connect to a replica set, specify the `replSetName` (page 922) and a seed list of set members, as in the following:

```
<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 3.0.0: 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 850) 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 770) or `mongos` (page 792) that has TLS/SSL support enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

Warning: For SSL connections (`--ssl`) to `mongod` (page 770) and `mongos` (page 792), if the `mongoexport` (page 850) runs without the *`--sslCAFile`* (page 868), `mongoexport` (page 850) will not attempt to validate the server certificates. This creates a vulnerability to expired `mongod` (page 770) and `mongos` (page 792) certificates as well as to foreign processes posing as valid `mongod` (page 770) or `mongos` (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates 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 TLS/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 868) option to connect to a `mongod` (page 770) or `mongos` (page 792) that has `CAFile` (page 907) enabled *without* `allowConnectionsWithoutCertificates` (page 908).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 868)). Use the *--sslPEMKeyPassword* (page 869) option only if the certificate-key file is encrypted. In all cases, the mongoexport (page 850) 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 869) option, the mongoexport (page 850) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 (page 908) setting, MongoDB logs as a warning the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates. Allows mongoexport (page 850) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslFIPSMode

--sslFIPSMode

New in version 2.6.

Directs the mongoexport (page 850) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *--sslFIPSMode* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[42]. See `https://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.

Changed in version 3.0.0: If you do not specify an argument for *`--password`* (page 870), `mongoexport` (page 850) returns an error.

Changed in version 3.0.2: If you wish `mongoexport` (page 850) to prompt the user for the password, pass the *`--username`* (page 870) option without *`--password`* (page 870) or specify an empty string as the *`--password`* (page 870) value, as in `--password ""` .

command line option!–authenticationDatabase <dbname>

--authenticationDatabase `<dbname>`

If you do not specify an authentication database, `mongoexport` (page 850) assumes that the database specified to export holds the user's credentials.

command line option!–authenticationMechanism <name>

--authenticationMechanism `<name>`

Default: SCRAM-SHA-1

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism. Changed default mechanism to `SCRAM-SHA-1`.

Specifies the authentication mechanism the `mongoexport` (page 850) instance uses to authenticate to the `mongod` (page 770) or `mongos` (page 792).

Value	Description
SCRAM-SHA-1	RFC 5802[43] standard Salted Challenge Response Authentication Mechanism using the SHA1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[44].
PLAIN (LDAP SASL)	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[45].

command line option!–gssapiServiceName

[42]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[43]https://tools.ietf.org/html/rfc5802
[44]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[45]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

--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 850).

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 850) 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 850) 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 850) 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 846) option allows you to specify in a file the field or fields to *include* in the export and is **only valid** with the `--type` option with value `csv`. The file must have only one field per line, and the line(s) must end with the LF character (`0x0A`).

`mongoexport` (page 850) 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* (page 960).

You must enclose the query in single quotes (e.g. `'`) to ensure that it does not interact with your shell environment.

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 850) uses the *-q* (page **??**) option to export only the documents with the field `a` greater than or equal to (`$gte` (page 530)) 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 856) option to `mongoexport` (page 850).

command line option!–type <string>

--type `<string>`

Default: json

New in version 3.0.0.

Specifies the file type to export. Specify `csv` for *CSV* format or `json` for *JSON* format.

If you specify `csv`, then you must also use either the *--fields* (page 846) or the *--fieldFile* (page 846) 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 850) writes data to standard output (e.g. `stdout`).

command line option!–jsonArray

--jsonArray

Modifies the output of `mongoexport` (page 850) to write the entire contents of the export as a single *JSON* array. By default `mongoexport` (page 850) writes data using one JSON document for every MongoDB document.

command line option!–pretty

--pretty

New in version 3.0.0.

Outputs documents in a pretty-printed format JSON.

command line option!–slaveOk, -k

--slaveOk, -k

Allows `mongoexport` (page 850) to read data from secondary or slave nodes when using `mongoexport` (page 850) with a replica set. This option is only available if connected to a `mongod` (page 770) or `mongos` (page 792) 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 850) to scan the data store directly instead of traversing the `_id` field index. Use *--forceTableScan* (page 855) to skip the index. 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 855), mongoexport (page 850) may return a document more than once if a write operation interleaves with the operation to cause the document to move.

> **Warning:** Use *--forceTableScan* (page 855) with extreme caution and consideration.

command line option!–skip <number>

--skip <number>
Use *--skip* (page 856) to control where mongoexport (page 850) begins exporting documents. See skip() (page 155) 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 144) 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 856) conjunction with *--skip* (page 856) and *--limit* (page 856) 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 156) for information about the underlying operation.

Use

Export in CSV Format Changed in version 3.0.0: mongoexport (page 850) removed the --csv option. Use the *--type=csv* (page 846) option to specify CSV format for the output.

In the following example, mongoexport (page 850) exports data from the collection contacts collection in the users database in *CSV* format to the file /opt/backups/contacts.csv.

The mongod (page 770) instance that mongoexport (page 850) connects to is running on the localhost port number 27017.

When you export in CSV format, you must specify the fields in the documents to export. The operation specifies the name and address fields to export.

```
mongoexport --db users --collection contacts --type=csv --fields name,address --out /opt/backups/cont
```

For CSV exports only, you can also specify the fields in a file containing the line-separated list of fields to export. The file must have only one field per line.

For example, you can specify the name and address fields in a file fields.txt:

```
name
address
```

Then, using the *--fieldFile* (page 846) option, specify the fields to export with the file:

```
mongoexport --db users --collection contacts --type=csv --fieldFile fields.txt --out /opt/backups/co
```

Changed in version 3.0.0: mongoexport (page 850) removed the `--csv` option and replaced with the *--type* (page 846) option.

Export in JSON Format This example creates an export of the `contacts` collection from the MongoDB instance running on the localhost port number `27017`. This writes the export to the `contacts.json` file in *JSON* format.

```
mongoexport --db sales --collection contacts --out contacts.json
```

Export from Remote Host Running with Authentication The following example exports the `contacts` collection from the `marketing` database, which requires authentication.

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
```

Export Query Results You can export only the results of a query by supplying a query filter with the *--query* option, and limit the results to a single database using the "*--db*" option.

For instance, this command returns all documents in the `sales` database's `contacts` collection that contain a field named `field` with a value of `1`.

```
mongoexport --db sales --collection contacts --query '{"field": 1}'
```

You must enclose the query in single quotes (e.g. `'`) to ensure that it does not interact with your shell environment.

4.1.5 Diagnostic Tools

mongostat (page 858), mongotop (page 867), and mongosniff (page 873) provide diagnostic information related to the current operation of a mongod (page 770) instance.

Note: Because mongosniff (page 873) depends on *libpcap*, most distributions of MongoDB do *not* include mongosniff (page 873).

mongostat

On this page

- Synopsis (page 858)
- Required Access (page 858)
- Options (page 858)
- Fields (page 863)
- Use (page 866)

Synopsis

The `mongostat` (page 858) utility provides a quick overview of the status of a currently running `mongod` (page 770) or `mongos` (page 792) instance. `mongostat` (page 858) is functionally similar to the UNIX/Linux file system utility `vmstat`, but provides data regarding `mongod` (page 770) and `mongos` (page 792) instances.

See also:

For more information about monitoring MongoDB, see `https://docs.mongodb.org/manual/administration/monitor`

For more background on other MongoDB status outputs see:

- *serverStatus* (page 492)
- *replSetGetStatus* (page 399)
- *dbStats* (page 481)
- *collStats* (page 472)

For an additional utility that provides MongoDB metrics see *mongotop* (page 866).

Required Access

In order to connect to a `mongod` (page 770) that enforces authorization with the `--auth` option, specify the `--username` and `--password` options, and the connecting user 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 858), see *create-role-for-mongostat*.

Options

`mongostat`

`mongostat`

command line option!–help

`--help`
Returns information on the options and use of `mongostat` (page 858).

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 858) 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 770) to which to connect. By default, the `mongostat` (page 858) attempts to connect to a MongoDB instance running on the localhost on port number `27017`.

To connect to a replica set, you can specify the set member or members to report on, as in the following (see also the `--discover` flag):

```
--host <hostname1><:port>,<hostname2><:port>,<...>
```

Changed in version 3.0.0: 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 858) 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 770) or `mongos` (page 792) that has TLS/SSL support enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

Warning: For SSL connections (`--ssl`) to `mongod` (page 770) and `mongos` (page 792), if the `mongostat` (page 858) runs without the *`--sslCAFile`* (page 868), `mongostat` (page 858) will not attempt to validate the server certificates. This creates a vulnerability to expired `mongod` (page 770) and `mongos` (page 792) certificates as well as to foreign processes posing as valid `mongod` (page 770) or `mongos` (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates 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 TLS/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 868) option to connect to a `mongod` (page 770) or `mongos` (page 792) that has `CAFile` (page 907) enabled *without* `allowConnectionsWithoutCertificates` (page 908).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 868)). Use the *--sslPEMKeyPassword* (page 869) option only if the certificate-key file is encrypted. In all cases, the `mongostat` (page 858) 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 869) option, the `mongostat` (page 858) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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` (page 908) setting, MongoDB logs as a warning the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

--sslAllowInvalidHostnames

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates. Allows `mongostat` (page 858) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and

`https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslFIPSMode

`--sslFIPSMode`
New in version 2.6.

Directs the `mongostat` (page 858) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[46]. See `https://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.

Changed in version 3.0.0: If you do not specify an argument for *`--password`* (page 870), `mongostat` (page 858) returns an error.

Changed in version 3.0.2: If you wish `mongostat` (page 858) to prompt the user for the password, pass the *`--username`* (page 870) option without *`--password`* (page 870) or specify an empty string as the *`--password`* (page 870) value, as in `--password ""`.

command line option!–authenticationDatabase <dbname>

`--authenticationDatabase` `<dbname>`
Specifies the database in which the user is created. See *user-authentication-database*.

`--authenticationDatabase` (page 870) is required for `mongod` (page 770) and `mongos` (page 792) instances that use *authentication*.

command line option!–authenticationMechanism <name>

`--authenticationMechanism` `<name>`
Default: SCRAM-SHA-1

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism. Changed default mechanism to `SCRAM-SHA-1`.

Specifies the authentication mechanism the `mongostat` (page 858) instance uses to authenticate to the `mongod` (page 770) or `mongos` (page 792).

[46] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Value	Description
SCRAM-SHA-1	RFC 5802[47] standard Salted Challenge Response Authentication Mechanism using the SHA1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[48].
PLAIN (LDAP SASL)	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[49].

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!–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 858) operation.

Unless *`--rowcount`* (page 871) is specified, `mongostat` (page 858) will return an infinite number of rows (e.g. value of `0`.)

command line option!–http

`--http`

Configures `mongostat` (page 858) to collect data using the HTTP interface rather than a raw database connection.

Deprecated since version 3.2: HTTP interface for MongoDB

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 862) all non-*hidden members* of the replica set. When connected

[47] https://tools.ietf.org/html/rfc5802

[48] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

[49] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

to a `mongos` (page 792), `mongostat` (page 858) will return data from all *shards* in the cluster. If a replica set provides a shard in the sharded cluster, `mongostat` (page 858) 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 862), `mongostat` (page 858) now respects *`--rowcount`* (page 871).

command line option!–all

`--all`

Configures `mongostat` (page 858) to return all optional *fields* (page 863).

command line option!–json

`--json`

New in version 3.0.0.

Returns output for `mongostat` (page 858) in *JSON* format.

`<sleeptime>`

The final argument is the length of time, in seconds, that `mongostat` (page 858) waits in between calls. By default `mongostat` (page 858) returns one call every second.

`mongostat` (page 858) returns values that reflect the operations over a 1 second period. For values of `<sleeptime>` greater than 1, `mongostat` (page 858) averages data to reflect average operations per second.

Fields

`mongostat` (page 858) returns values that reflect the operations over a 1 second period. When **mongostat <sleeptime>** has a value greater than 1, `mongostat` (page 858) averages the statistics to reflect average operations per second.

`mongostat` (page 858) 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 858) presents two values separated by a pipe character (e.g. `|`), in the form of `local|replicated` commands.

`flushes`

Changed in version 3.0.

For the *storage-wiredtiger*, `flushes` refers to the number of WiredTiger checkpoints triggered between each polling interval.

For the *storage-mmapv1*, `flushes` represents the number of *fsync* operations per second.

`dirty`
New in version 3.0.

Only for *storage-wiredtiger*. The percentage of the WiredTiger cache with dirty bytes, calculated by `wiredTiger.cache.tracked dirty bytes in the cache` (page 508) / `wiredTiger.cache.maximum bytes configured` (page 508).

`used`
New in version 3.0.

Only for *storage-wiredtiger*. The percentage of the WiredTiger cache that is in use, calculated by `wiredTiger.cache.bytes currently in the cache` (page 508) / `wiredTiger.cache.maximum bytes configured` (page 508).

`mapped`
Changed in version 3.0.

Only for *storage-mmapv1*. The total amount of data mapped in megabytes. This is the total data size at the time of the last `mongostat` (page 858) call.

`vsize`
The amount of virtual memory in megabytes used by the process at the time of the last `mongostat` (page 858) call.

`non-mapped`
Changed in version 3.0.

Only for *storage-mmapv1*.

Optional. The total amount of virtual memory excluding all mapped memory at the time of the last `mongostat` (page 858) call.

`mongostat` (page 858) only returns this value when started with the `--all` option.

`res`
The amount of resident memory in megabytes used by the process at the time of the last `mongostat` (page 858) call.

`faults`
Changed in version 3.0.

Only for *storage-mmapv1*. The number of page faults per second.

Changed in version 2.1: Before version 2.1, this value was only provided for MongoDB instances running on Linux hosts.

`lr`
New in version 3.2.

Only for *storage-mmapv1*. The percentage of read lock acquisitions that had to wait. `mongostat` (page 858) displays `lr|lw` if a lock acquisition waited.

`lw`
New in version 3.2.

Only for *storage-mmapv1*. The percentage of write lock acquisitions that had to wait. `mongostat` (page 858) displays `lr|lw` if a lock acquisition waited.

`lrt`
New in version 3.2.

Only for *storage-mmapv1*. The average acquire time, in microseconds, of read lock acquisitions that waited. `mongostat` (page 858) displays `lrt|lwt` if a lock acquisition waited.

`lwt`

New in version 3.2.

Only for *storage-mmapv1*. The average acquire time, in microseconds, of write lock acquisitions that waited. `mongostat` (page 858) displays `lrt|lwt` if a lock acquisition waited.

`locked`

Changed in version 3.0: Only appears when `mongostat` (page 858) runs against pre-3.0 versions of MongoDB instances.

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.

`idx miss`

Changed in version 3.0.

Only for *storage-mmapv1*. 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 858) itself.

`netOut`

The amount of network traffic, in *bytes*, sent by the MongoDB instance.

This includes traffic from `mongostat` (page 858) 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”)
ARB	*arbiter*

Use

In the first example, mongostat (page 858) will return data every second for 20 seconds. mongostat (page 858) collects data from the mongod (page 770) 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 858) returns data every 5 minutes (or 300 seconds) for as long as the program runs. mongostat (page 858) collects data from the mongod (page 770) 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 858) returns data every 5 minutes for an hour (12 times.) mongostat (page 858) collects data from the mongod (page 770) 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 792) 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

On this page

- Synopsis (page 866)
- Required Access (page 867)
- Options (page 867)
- Fields (page 872)
- Use (page 872)

Synopsis

mongotop (page 867) provides a method to track the amount of time a MongoDB instance spends reading and writing data. mongotop (page 867) provides statistics on a per-collection level. By default, mongotop (page 867) returns values every second.

See also:

For more information about monitoring MongoDB, see `https://docs.mongodb.org/manual/administration/monitor`

For additional background on various other MongoDB status outputs see:

- *serverStatus* (page 492)

- *replSetGetStatus* (page 399)
- *dbStats* (page 481)
- *collStats* (page 472)

For an additional utility that provides MongoDB metrics see *mongostat* (page 857).

Required Access

In order to connect to a `mongod` (page 770) that enforces authorization with the `--auth` option, you must use the `--username` and `--password` options, and the connecting user must have the `serverStatus` and `top` privileges.

The most appropriate built-in role that has these privileges is `clusterMonitor`.

Options

`mongotop`

`mongotop`

command line option!–help

`--help`
Returns information on the options and use of `mongotop` (page 867).

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 867) 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 867) 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 770) to which to connect. By default, the `mongotop` (page 867) attempts to connect to a MongoDB instance running on the localhost on port number `27017`.

To connect to a replica set, specify the `replSetName` (page 922) and a seed list of set members, as in the following:

`<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 3.0.0: 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>]`).

If connected to a replica set where the *primary* is not reachable, `mongotop` (page 867) returns an error message.

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 867) 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 770) or `mongos` (page 792) that has TLS/SSL support enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

Warning: For SSL connections (`--ssl`) to `mongod` (page 770) and `mongos` (page 792), if the `mongotop` (page 867) runs without the *`--sslCAFile`* (page 868), `mongotop` (page 867) will not attempt to validate the server certificates. This creates a vulnerability to expired `mongod` (page 770) and `mongos` (page 792) certificates as well as to foreign processes posing as valid `mongod` (page 770) or `mongos` (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates 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 TLS/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 868) option to connect to a `mongod` (page 770) or `mongos` (page 792) that has `CAFile` (page 907) enabled *without* `allowConnectionsWithoutCertificates` (page 908).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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 868)). Use the *`--sslPEMKeyPassword`* (page 869) option only if the certificate-key file is encrypted. In all cases, the `mongotop` (page 867) 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 869) option, the `mongotop` (page 867) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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` (page 908) setting, MongoDB logs as a warning the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

`--sslAllowInvalidHostnames`

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates. Allows `mongotop` (page 867) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and

`https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslFIPSMode

`--sslFIPSMode`

New in version 2.6.

Directs the `mongotop` (page 867) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[50]. See `https://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.

Changed in version 3.0.0: If you do not specify an argument for *`--password`* (page 870), `mongotop` (page 867) returns an error.

Changed in version 3.0.2: If you wish `mongotop` (page 867) to prompt the user for the password, pass the *`--username`* (page 870) option without *`--password`* (page 870) or specify an empty string as the *`--password`* (page 870) value, as in `--password ""` .

command line option!–authenticationDatabase <dbname>

`--authenticationDatabase` `<dbname>`

Specifies the database in which the user is created. See *user-authentication-database*.

Changed in version 3.0.0: *`--authenticationDatabase`* (page 870) is required for `mongod` (page 770) and `mongos` (page 792) instances that use *authentication*.

command line option!–authenticationMechanism <name>

`--authenticationMechanism` `<name>`

Default: SCRAM-SHA-1

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism. Changed default mechanism to `SCRAM-SHA-1`.

Specifies the authentication mechanism the `mongotop` (page 867) instance uses to authenticate to the `mongod` (page 770) or `mongos` (page 792).

[50]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Value	Description
SCRAM-SHA-1	RFC 5802[51] standard Salted Challenge Response Authentication Mechanism using the SHA1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[52].
PLAIN (LDAP SASL)	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[53].

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 867) to report on use of per-database *locks* (page 498). These data are useful for measuring concurrent operations and lock percentage.

`--locks` (page 871) returns an error when called against a `mongod` (page 770) instance that does not report lock usage.

command line option!–rowcount int, -n int

`--rowcount` `int`, **`-n`** `int`

Number of lines of data that `mongotop` (page 867) should print. "0 for indefinite"

command line option!–json

`--json`

New in version 3.0.0.

Returns output for `mongotop` (page 867) in *JSON* format.

`<sleeptime>`

The final argument is the length of time, in seconds, that `mongotop` (page 867) waits in between calls. By default `mongotop` (page 867) returns data every second.

[51] https://tools.ietf.org/html/rfc5802

[52] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

[53] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Fields

`mongotop` (page 867) returns time values specified in milliseconds (ms.)

`mongotop` (page 867) only reports active namespaces or databases, depending on the *`--locks`* (page 871) 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 803) shell to generate activity to affect the output of `mongotop` (page 867).

`mongotop.`**`ns`**
Contains the database namespace, which combines the database name and collection.

Changed in version 2.2: If you use the *`mongotop --locks`*, the `ns` (page 872) field does not appear in the `mongotop` (page 867) 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 867) with the *`--locks`* (page 871) option.

`mongotop.`**`total`**
Provides the total amount of time that this `mongod` (page 770) spent operating on this namespace.

`mongotop.`**`read`**
Provides the amount of time that this `mongod` (page 770) spent performing read operations on this namespace.

`mongotop.`**`write`**
Provides the amount of time that this `mongod` (page 770) spent performing write operations on this namespace.

`mongotop.`**`<timestamp>`**
Provides a time stamp for the returned data.

Use

By default `mongotop` (page 867) connects to the MongoDB instance running on the localhost port `27017`. However, `mongotop` (page 867) can optionally connect to remote `mongod` (page 770) instances. See the *mongotop options* (page 867) for more information.

To force `mongotop` (page 867) to return less frequently specify a number, in seconds at the end of the command. In this example, `mongotop` (page 867) will return every 15 seconds.

```
mongotop 15
```

This command produces the following output:

```
                     ns    total    read    write          2014-12-19T15:32:01-05:00
     admin.system.roles      0ms     0ms      0ms
   admin.system.version      0ms     0ms      0ms
               local.me      0ms     0ms      0ms
         local.oplog.rs      0ms     0ms      0ms
 local.replset.minvalid      0ms     0ms      0ms
      local.startup_log      0ms     0ms      0ms
   local.system.indexes      0ms     0ms      0ms
local.system.namespaces      0ms     0ms      0ms
   local.system.replset      0ms     0ms      0ms

                     ns    total    read    write          2014-12-19T15:47:01-05:00
     admin.system.roles       0ms      0ms       0ms
   admin.system.version       0ms      0ms       0ms
               local.me       0ms      0ms       0ms
         local.oplog.rs       0ms      0ms       0ms
 local.replset.minvalid       0ms      0ms       0ms
      local.startup_log       0ms      0ms       0ms
   local.system.indexes       0ms      0ms       0ms
local.system.namespaces       0ms      0ms       0ms
   local.system.replset       0ms      0ms       0ms
```

The output varies depending on your MongoDB setup. For example, `local.system.indexes` and `local.system.namespaces` only appear for `mongod` (page 770) instances using the *MMAPv1* storage engine.

To return a `mongotop` (page 867) report every 5 minutes, use the following command:

```
mongotop 300
```

To report the use of per-database locks, use *`--locks`* (page 871), 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
```

Changed in version 3.0.0: When called against a `mongod` (page 770) that does not report lock usage, *`--locks`* (page 871) will return a `Failed:  Server does not support reporting locking information` error.

mongosniff

On this page

- Synopsis (page 873)
- Options (page 873)
- Use (page 874)

Synopsis

`mongosniff` (page 873) provides a low-level operation tracing/sniffing view into database activity in real time. Think of `mongosniff` (page 873) as a MongoDB-specific analogue of `tcpdump` for TCP/IP network traffic. `mongosniff` (page 873) is most frequently used in driver development.

Note: `mongosniff` (page 873) requires `libpcap` and is only available for Unix-like systems.

As an alternative to `mongosniff` (page 873), 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 873).

command line option!–forward <host><:port>

--forward `<host><:port>`
Declares a host to forward all parsed requests that the `mongosniff` (page 873) intercepts to another `mongod` (page 770) 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 873).

<port>
Specifies alternate ports to sniff for traffic. By default, `mongosniff` (page 873) watches for MongoDB traffic on port `27017`. Append multiple port numbers to the end of `mongosniff` (page 873) to monitor traffic on multiple ports.

Use

Use the following command to connect to a `mongod` (page 770) or `mongos` (page 792) 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 770) or `mongos` (page 792) running on the localhost interface and port 27018, for driver development and troubleshooting:

```
mongosniff --objcheck --source NET lo 27018
```

See also:

To build `mongosniff` (page 873) yourself, see: Build MongoDB From Source[54].

[54] https://www.mongodb.org/about/contributors/tutorial/build-mongodb-from-source

mongoperf

On this page

- Synopsis (page 875)
- Options (page 875)
- Configuration Fields (page 876)
- Use (page 877)

Synopsis

mongoperf (page 875) 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 875) for any case apart from MongoDB. The mmf (page 876) true mode is completely generic. In that mode it is somewhat analogous to tools such as bonnie++[55] (albeit mongoperf is simpler).

Specify options to mongoperf (page 875) using a JavaScript document.

See also:

- bonnie[56]
- bonnie++[57]
- Output from an example run[58]
- Checking Disk Performance with the mongoperf Utility[59]

Options

mongoperf

mongoperf

command line option!–help, -h

--help, -h
Returns information on the options and use of mongoperf (page 875).

<jsonconfig>
mongoperf (page 875) 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 875), 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>,
```

[55] http://sourceforge.net/projects/bonnie/
[56] http://www.textuality.com/bonnie/
[57] http://sourceforge.net/projects/bonnie/
[58] https://gist.github.com/1694664
[59] http://blog.mongodb.org/post/40769806981/checking-disk-performance-with-the-mongoperf-utility

```
    mmf:<bool>,
    r:<bool>,
    w:<bool>,
    recSizeKB:<n>,
    syncDelay:<n>
}
```

See the *Configuration Fields* (page 876) section for documentation of each of these fields.

Configuration Fields

`mongoperf.`**`nThreads`**

Type: Integer.

Default: 1

Defines the number of threads `mongoperf` (page 875) will use in the test. To saturate your system's storage system you will need multiple threads. Consider setting `nThreads` (page 876) 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 875) will pause for the number of specified `sleepMicros` (page 876) divided by the `nThreads` (page 876) between each operation.

`mongoperf.`**`mmf`**

Type: Boolean.

Default: `false`

Set `mmf` (page 876) to `true` to use memory mapped files for the tests.

Generally:

- when `mmf` (page 876) is `false`, `mongoperf` (page 875) 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 876) is `true`, `mongoperf` (page 875) runs tests of the caching system, and can use normal file system cache. Use `mmf` (page 876) in this mode to test file system cache behavior with memory mapped files.

`mongoperf.`**`r`**

Type: Boolean.

Default: `false`

Set `r` (page 876) to `true` to perform reads as part of the tests.

Either `r` (page 876) or `w` (page 876) must be `true`.

`mongoperf.`**`w`**

Type: Boolean.

Default: `false`

Set w (page 876) to true to perform writes as part of the tests.

Either r (page 876) or w (page 876) 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 877) is similar to *--syncdelay* for mongod (page 770).

The syncDelay (page 877) controls how frequently mongoperf (page 875) performs an asynchronous disk flush of the memory mapped file used for testing. By default, mongod (page 770) performs this operation every 60 seconds. Use syncDelay (page 877) to test basic system performance of this type of operation.

Only use syncDelay (page 877) in conjunction with mmf (page 876) set to true.

The default value of 0 disables this.

Use

```
mongoperf < jsonconfigfile
```

Replace jsonconfigfile with the path to the mongoperf (page 875) configuration. You may also invoke mongoperf (page 875) in the following form:

```
echo "{nThreads:16,fileSizeMB:10000,r:true,w:true}" | mongoperf
```

In this operation:

- mongoperf (page 875) tests direct physical random read and write io's, using 16 concurrent reader threads.
- mongoperf (page 875) 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 878) provides a command-line interact to a MongoDB *GridFS* storage system.

mongofiles

On this page

- Synopsis (page 878)
- Required Access (page 878)
- Options (page 878)
- Commands (page 883)
- Examples (page 884)

mongofiles

Synopsis

The `mongofiles` (page 878) 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 878) commands have the following form:

```
mongofiles <options> <commands> <filename>
```

The components of the `mongofiles` (page 878) command are:

1. *Options* (page 878). You may use one or more of these options to control the behavior of `mongofiles` (page 878).
2. *Commands* (page 883). Use one of these commands to determine the action of `mongofiles` (page 878).
3. A filename which is either: the name of a file on your local's file system, or a GridFS object.

`mongofiles` (page 878), like `mongodump` (page 816), `mongoexport` (page 850), `mongoimport` (page 841), and `mongorestore` (page 824), can access data stored in a MongoDB data directory without requiring a running `mongod` (page 770) instance, if no other `mongod` (page 770) is running.

Important: For *replica sets*, `mongofiles` (page 878) can only read from the set's *primary*.

Required Access

In order to connect to a `mongod` (page 770) that enforces authorization with the `--auth` option, you must use the `--username` and `--password` options. The connecting user must possess, at a minimum:

- the `read` role for the accessed database when using the `list`, `search` or `get` commands,
- the `readWrite` role for the accessed database when using the `put` or `delete` commands.

Options

Changed in version 3.0.0: `mongofiles` (page 878) removed the `--dbpath` as well as related `--directoryperdb` and `--journal` options. You must use `mongofiles` (page 878) while connected to a `mongod` (page 770) instance.

mongofiles

command line option!–help

--help
Returns information on the options and use of `mongofiles` (page 878).

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 878) 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 878) release number.

command line option!–host <hostname><:port>

--host `<hostname><:port>`
Specifies a resolvable hostname for the `mongod` (page 770) that holds your GridFS system. By default `mongofiles` (page 878) 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 878) 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 770) or `mongos` (page 792) that has TLS/SSL support enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

Warning: For SSL connections (--ssl) to mongod (page 770) and mongos (page 792), if the mongofiles (page 878) runs without the *--sslCAFile* (page 868), mongofiles (page 878) will not attempt to validate the server certificates. This creates a vulnerability to expired mongod (page 770) and mongos (page 792) certificates as well as to foreign processes posing as valid mongod (page 770) or mongos (page 792) instances. Ensure that you *always* specify the CA file to validate the server certificates 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 TLS/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 868) option to connect to a mongod (page 770) or mongos (page 792) that has CAFile (page 907) enabled *without* allowConnectionsWithoutCertificates (page 908).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

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 868)). Use the *--sslPEMKeyPassword* (page 869) option only if the certificate-key file is encrypted. In all cases, the mongofiles (page 878) 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 869) option, the mongofiles (page 878) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

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` (page 908) setting, MongoDB logs as a warning the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslAllowInvalidHostnames

`--sslAllowInvalidHostnames`

New in version 3.0.

Disables the validation of the hostnames in TLS/SSL certificates. Allows `mongofiles` (page 878) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

command line option!–sslFIPSMode

`--sslFIPSMode`

New in version 2.6.

Directs the `mongofiles` (page 878) to use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use the *`--sslFIPSMode`* (page 870) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[60]. See `https://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.

Changed in version 3.0.0: If you do not specify an argument for *`--password`* (page 870), `mongofiles` (page 878) returns an error.

[60]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Changed in version 3.0.2: If you wish `mongofiles` (page 878) to prompt the user for the password, pass the `--username` (page 870) option without `--password` (page 870) or specify an empty string as the `--password` (page 870) value, as in `--password ""`.

command line option!–authenticationDatabase <dbname>

`--authenticationDatabase` `<dbname>`
Specifies the database in which the user is created. See *user-authentication-database*.

command line option!–authenticationMechanism <name>

`--authenticationMechanism` `<name>`
Default: SCRAM-SHA-1

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism. Changed default mechanism to `SCRAM-SHA-1`.

Specifies the authentication mechanism the `mongofiles` (page 878) instance uses to authenticate to the `mongod` (page 770) or `mongos` (page 792).

Value	Description
SCRAM-SHA-1	RFC 5802[61] standard Salted Challenge Response Authentication Mechanism using the SHA1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[62].
PLAIN (LDAP SASL)	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[63].

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 878).

[61] https://tools.ietf.org/html/rfc5802
[62] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[63] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

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[64] for more information.

command line option!–local <filename>, -l <filename>

--local `<filename>`, **-l** `<filename>`
Specifies the local filesystem name of a file for get and put operations.

In the **mongofiles put** and **mongofiles get** commands, the required `<filename>` modifier refers to the name the object will have in GridFS. `mongofiles` (page 878) 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 878) 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 878) 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 878) 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.

[64] https://jira.mongodb.org/browse/SERVER-4931

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 878) 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 878) instance will connect to the `mongod` (page 770) 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 770) 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

On this page

- Collections (page 885)

The `config` database supports *sharded cluster* operation. See the `https://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 816) to create a full backup of the `config` database.

To access the `config` database, connect to a `mongos` (page 792) 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The changelog (page 885) 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 885) 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 885) 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 792) 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 887) document depends on the type of change.

`config.`**`chunks`**

Internal MongoDB Metadata

The `config` (page 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `chunks` (page 887) 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `collections` (page 887) 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 887) 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `databases` (page 888) collection stores a document for each database in the cluster, and tracks if the database has sharding enabled. `databases` (page 888) 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `lockpings` (page 888) collection keeps track of the active components in the sharded cluster. Given a cluster with a `mongos` (page 792) running on `example.com:30000`, the document in the `lockpings` (page 888) 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `locks` (page 888) collection stores a distributed lock. This ensures that only one `mongos` (page 792) instance can perform administrative tasks on the cluster at once. The `mongos` (page 792) 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 792) 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `mongos` (page 889) collection stores a document for each `mongos` (page 792) instance affiliated with the cluster. `mongos` (page 792) instances send pings to all members of the cluster every 30 seconds so the cluster can verify that the `mongos` (page 792) is active. The `ping` field shows the time of the last ping, while the `up` field reports the uptime of the `mongos` (page 792) as of the last ping. The cluster maintains this collection for reporting purposes.

The following document shows the status of the `mongos` (page 792) 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `settings` (page 889) collection holds the following sharding configuration settings:

- Chunk size. To change chunk size, see `https://docs.mongodb.org/manual/tutorial/modify-chunk-siz`
- 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The shards (page 889) 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The tags (page 890) collection holds documents for each tagged shard key range in the cluster. The documents in the tags (page 890) 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 885) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The version (page 890) collection holds the current metadata version number. This collection contains only one document:

To access the version (page 890) collection you must use the db.getCollection() (page 181) method. For example, to display the collection's document:

```
mongos> db.getCollection("version").find()
{ "_id" : 1, "version" : 3 }
```

5.2 The local Database

On this page

- Overview (page 891)
- Collection on all mongod Instances (page 891)
- Collections on Replica Set Members (page 892)
- Collections used in Master/Slave Replication (page 892)

5.2.1 Overview

Every mongod (page 770) 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 (page 910)), 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 770) instance inserts a document into startup_log (page 891) with diagnostic information about the mongod (page 770) instance itself and host information. startup_log (page 891) 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 891) 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 891) 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 891) in the system's local time zone.

`local.startup_log.`**`cmdLine`**
An embedded document that reports the `mongod` (page 770) runtime options and their values.

`local.startup_log.`**`pid`**
The process identifier for this process.

`local.startup_log.`**`buildinfo`**
An embedded document that reports information about the build environment and settings used to compile this `mongod` (page 770). This is the same output as `buildInfo` (page 471). See `buildInfo` (page 471).

5.2.3 Collections on Replica Set Members

`local.system.`**`replset`**
`local.system.replset` (page 892) holds the replica set's configuration object as its single document. To view the object's configuration information, issue `rs.conf()` (page 257) from the `mongo` (page 803) shell. You can also query this collection directly.

`local.oplog.`**`rs`**
`local.oplog.rs` (page 892) is the capped collection that holds the *oplog*. You set its size at creation using the `oplogSizeMB` (page 921) setting. To resize the oplog after replica set initiation, use the `https://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`**
Removed in version 3.0: Replica set members no longer mirror replication status of the set to the `local.slaves` (page 892) collection. Use `rs.status()` (page 262) instead.

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`**
  Removed in version 3.0: MongoDB no longer stores information about each slave in the `local.slaves` (page 892) collection. Use `db.serverStatus( { repl: 1 } )` (page 197) instead.

- On each slave:

  `local.`**`sources`**
  This contains information about the slave's master server.

5.3 System Collections

On this page

- Synopsis (page 893)
- Collections (page 893)

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 890), 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 893) 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 893) 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 893) collection.

`admin.system.`**`version`**

New in version 2.6.

Stores the schema version of the user credential documents.

System collections also include these collections stored directly in each database:

`<database>.system.`**`namespaces`**

Deprecated since version 3.0: Access this data using `listCollections` (page 438).

The `<database>.system.namespaces` (page 893) collection contains information about all of the database's collections.

`<database>.system.`**`indexes`**

Deprecated since version 3.0: Access this data using `listIndexes` (page 450).

The `<database>.system.indexes` (page 893) collection lists all the indexes in the database.

`<database>.system.`**`profile`**

The `<database>.system.profile` (page 893) collection stores database profiling information. For information on profiling, see *database-profiling*.

`<database>.system.`**`js`**

The `<database>.system.js` (page 893) collection holds special JavaScript code for use in `server side JavaScript`. See `https://docs.mongodb.org/manual/tutorial/store-javascript-function-o` for more information.

CHAPTER 6

General System Reference

6.1 Configuration File Options

On this page

- Configuration File (page 895)
- Core Options (page 896)
- mongos-only Options (page 925)
- Windows Service Options (page 926)

The following page describes the configuration options available in MongoDB 3.2. For configuration file options for other versions of MongoDB, see the appropriate version of the MongoDB Manual.

6.1.1 Configuration File

You can configure mongod (page 770) and mongos (page 792) instances at startup using a configuration file. The configuration file contains settings that are equivalent to the mongod (page 770) and mongos (page 792) command-line options.

Using a configuration file makes managing mongod (page 770) and mongos (page 792) options easier, especially for large-scale deployments. You can also add comments to the configuration file to explain the server's settings.

If you installed from a package and have started MongoDB using your system's *init script*, you are already using a configuration file.

File Format

Important: Changed in version 2.6: MongoDB 2.6 introduces a YAML-based configuration file format. The 2.4 configuration file format[1] remains for backward compatibility.

MongoDB configuration files use the YAML[2] format [3].

The following sample configuration file contains several mongod (page 770) settings that you may adapt to your local configuration:

[1]https://docs.mongodb.org/v2.4/reference/configuration-options
[2]http://www.yaml.org
[3] YAML is a superset of *JSON*.

Note: YAML does not support tab characters for indention: use spaces instead.

```
systemLog:
   destination: file
   path: "/var/log/mongodb/mongod.log"
   logAppend: true
storage:
   journal:
      enabled: true
processManagement:
   fork: true
net:
   bindIp: 127.0.0.1
   port: 27017
setParameter:
   enableLocalhostAuthBypass: false
...
```

The Linux package init scripts included in the official MongoDB packages depend on specific values for `systemLog.path` (page 897), `storage.dbpath`, and `processManagement.fork` (page 902). If you modify these settings in the default configuration file, `mongod` (page 770) may not start.

Use the Configuration File

To start `mongod` (page 770) or `mongos` (page 792) using a config file, specify the config file with the *`--config`* (page 792) option or the *`-f`* (page **??**) option, as in the following examples:

The following examples use the *`--config`* (page 792) option for `mongod` (page 770) and `mongos` (page 792):

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

mongos --config /etc/mongos.conf
```

You can also use the *`-f`* (page **??**) alias to specify the configuration file, as in the following:

```
mongod -f /etc/mongod.conf

mongos -f /etc/mongos.conf
```

If you installed from a package and have started MongoDB using your system's *init script*, you are already using a configuration file.

6.1.2 Core Options

`systemLog` Options

```
systemLog:
   verbosity: <int>
   quiet: <boolean>
   traceAllExceptions: <boolean>
   syslogFacility: <string>
   path: <string>
   logAppend: <boolean>
   logRotate: <string>
   destination: <string>
   timeStampFormat: <string>
   component:
      accessControl:
         verbosity: <int>
      command:
         verbosity: <int>

      # COMMENT additional component verbosity settings omitted for brevity
```

`systemLog.`**`verbosity`**

Type: integer

Default: 0

Changed in version 3.0.

The default *log message* (page 964) verbosity level for *components* (page 964). The verbosity level determines the amount of *Informational and Debug* (page 964) messages MongoDB outputs.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

To use a different verbosity level for a named component, use the component's verbosity setting. For example, use the `systemLog.component.accessControl.verbosity` (page 899) to set the verbosity level specifically for `ACCESS` (page 964) components.

See the `systemLog.component.<name>.verbosity` settings for specific component verbosity settings.

For various ways to set the log verbosity level, see *Configure Log Verbosity Levels* (page 966).

`systemLog.`**`quiet`**

Type: boolean

Run the `mongos` (page 792) or `mongod` (page 770) in a quiet mode that attempts to limit the amount of output.

`systemLog.quiet` (page 897) is **not** recommended for production systems as it may make tracking problems during particular connections much more difficult.

`systemLog.`**`traceAllExceptions`**

Type: boolean

Print verbose information for debugging. Use for additional logging for support-related troubleshooting.

`systemLog.`**`syslogFacility`**

Type: string

Default: user

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 793) option.

`systemLog.`**`path`**

Type: string

The path of the log file to which `mongod` (page 770) or `mongos` (page 792) should send all diagnostic logging information, rather than the standard output or the host's *syslog*. MongoDB creates the log file at the specified path.

The Linux package init scripts do not expect `systemLog.path` (page 897) to change from the defaults. If you use the Linux packages and change `systemLog.path` (page 897), you will have to use your own init scripts and disable the built-in scripts.

`systemLog.`**`logAppend`**

Type: boolean

Default: False

When `true`, `mongos` (page 792) or `mongod` (page 770) appends new entries to the end of the existing log file when the `mongos` (page 792) or `mongod` (page 770) instance restarts. Without this option, `mongod` (page 770) will back up the existing log and create a new file.

`systemLog.`**`logRotate`**

Type: string

Default: rename

New in version 3.0.0.

The behavior for the `logRotate` (page 465) command. Specify either `rename` or `reopen`:

- `rename` renames the log file.
- `reopen` closes and reopens the log file following the typical Linux/Unix log rotate behavior. Use `reopen` when using the Linux/Unix logrotate utility to avoid log loss.

  If you specify `reopen`, you must also set `systemLog.logAppend` (page 898) to `true`.

`systemLog.`**`destination`**

Type: string

The destination to which MongoDB sends all log output. Specify either `file` or `syslog`. If you specify `file`, you must also specify `systemLog.path` (page 897).

If you do not specify `systemLog.destination` (page 898), MongoDB sends all log output to standard output.

`systemLog.`**`timeStampFormat`**

Type: 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`

`systemLog.component` Options

```
systemLog:
   component:
      accessControl:
         verbosity: <int>
      command:
         verbosity: <int>

      # COMMENT some component verbosity settings omitted for brevity
      storage:
         verbosity: <int>
         journal:
            verbosity: <int>
      write:
         verbosity: <int>
```

systemLog.component.accessControl.**verbosity**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to access control. See ACCESS (page 964) components.

The verbosity level can range from 0 to 5:

- 0 is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- 1 to 5 increases the verbosity level to include *Debug* (page 964) messages.

systemLog.component.command.**verbosity**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to commands. See COMMAND (page 964) components.

The verbosity level can range from 0 to 5:

- 0 is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- 1 to 5 increases the verbosity level to include *Debug* (page 964) messages.

systemLog.component.control.**verbosity**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to control operations. See CONTROL (page 964) components.

The verbosity level can range from 0 to 5:

- 0 is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- 1 to 5 increases the verbosity level to include *Debug* (page 964) messages.

systemLog.component.geo.**verbosity**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to geospatial parsing operations. See GEO (page 965) components.

The verbosity level can range from 0 to 5:

- 0 is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.

- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

`systemLog.component.index.`**`verbosity`**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to indexing operations. See `INDEX` (page 965) components.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

`systemLog.component.network.`**`verbosity`**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to networking operations. See `NETWORK` (page 965) components.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

`systemLog.component.query.`**`verbosity`**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to query operations. See `QUERY` (page 965) components.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

`systemLog.component.replication.`**`verbosity`**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to replication. See `REPL` (page 965) components.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

`systemLog.component.sharding.`**`verbosity`**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to sharding. See `SHARDING` (page 965) components.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

`systemLog.component.storage.`**`verbosity`**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to storage. See `STORAGE` (page 965) components.

If `systemLog.component.storage.journal.verbosity` (page 901) is unset, `systemLog.component.storage.verbosity` (page 901) level also applies to journaling components.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

`systemLog.component.storage.journal.`**`verbosity`**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to journaling. See `JOURNAL` (page 965) components.

If `systemLog.component.storage.journal.verbosity` (page 901) is unset, the journaling components have the same verbosity level as the parent storage components: i.e. either the `systemLog.component.storage.verbosity` (page 901) level if set or the default verbosity level.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

`systemLog.component.write.`**`verbosity`**

Type: integer

Default: 0

New in version 3.0.

The log message verbosity level for components related to write operations. See `WRITE` (page 965) components.

The verbosity level can range from `0` to `5`:

- `0` is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- `1` to `5` increases the verbosity level to include *Debug* (page 964) messages.

processManagement Options

```
processManagement:
   fork: <boolean>
   pidFilePath: <string>
```

processManagement.**fork**

Type: boolean

Default: False

Enable a *daemon* mode that runs the mongos (page 792) or mongod (page 770) process in the background. By default mongos (page 792) or mongod (page 770) does not run as a daemon: typically you will run mongos (page 792) or mongod (page 770) as a daemon, either by using processManagement.fork (page 902) or by using a controlling process that handles the daemonization process (e.g. as with upstart and systemd).

The Linux package init scripts do not expect processManagement.fork (page 902) to change from the defaults. If you use the Linux packages and change processManagement.fork (page 902), you will have to use your own init scripts and disable the built-in scripts.

processManagement.**pidFilePath**

Type: string

Specifies a file location to hold the process ID of the mongos (page 792) or mongod (page 770) process where mongos (page 792) or mongod (page 770) will write its PID. This is useful for tracking the mongos (page 792) or mongod (page 770) process in combination with the *--fork* (page 795) option. Without a specified processManagement.pidFilePath (page 902) option, the process creates no PID file.

net Options

```
net:
   port: <int>
   bindIp: <string>
   maxIncomingConnections: <int>
   wireObjectCheck: <boolean>
   ipv6: <boolean>
   unixDomainSocket:
      enabled: <boolean>
      pathPrefix: <string>
      filePermissions: <int>
   http:
      enabled: <boolean>
      JSONPEnabled: <boolean>
      RESTInterfaceEnabled: <boolean>
   ssl:
      sslOnNormalPorts: <boolean>  # deprecated since 2.6
      mode: <string>
      PEMKeyFile: <string>
      PEMKeyPassword: <string>
      clusterFile: <string>
      clusterPassword: <string>
      CAFile: <string>
      CRLFile: <string>
      allowConnectionsWithoutCertificates: <boolean>
      allowInvalidCertificates: <boolean>
      allowInvalidHostnames: <boolean>
      disabledProtocols: <string>
      FIPSMode: <boolean>
```

`net.`**`port`**

Type: integer

Default: 27017

The TCP port on which the MongoDB instance listens for client connections.

`net.`**`bindIp`**

Type: string

Default: All interfaces.

Changed in version 2.6.0: The `deb` and `rpm` packages include a default configuration file (`/etc/mongod.conf`) that sets `net.bindIp` (page 903) to `127.0.0.1`.

The IP address that `mongos` (page 792) or `mongod` (page 770) binds to in order to listen for connections from applications. You may attach `mongos` (page 792) or `mongod` (page 770) to any interface. When attaching `mongos` (page 792) or `mongod` (page 770) to a publicly accessible interface, ensure that you have implemented proper authentication and firewall restrictions to protect the integrity of your database.

To bind to multiple IP addresses, enter a list of comma separated values.

`net.`**`maxIncomingConnections`**

Type: integer

Default: 65536

The maximum number of simultaneous connections that `mongos` (page 792) or `mongod` (page 770) will accept. This setting has no effect if it is higher than your operating system's configured maximum connection tracking threshold.

Do not assign too low of a value to this option, or you will encounter errors during normal application operation.

This is particularly useful for a `mongos` (page 792) if you have a client that creates multiple connections and allows them to timeout rather than closing them.

In this case, set `maxIncomingConnections` (page 903) to a value slightly higher than the maximum number of connections that the client creates, or the maximum size of the connection pool.

This setting prevents the `mongos` (page 792) from causing connection spikes on the individual *shards*. Spikes like these may disrupt the operation and memory allocation of the *sharded cluster*.

`net.`**`wireObjectCheck`**

Type: boolean

Default: True

When `true`, the `mongod` (page 770) or `mongos` (page 792) instance validates all requests from clients upon receipt to prevent clients from inserting malformed or invalid BSON into a MongoDB database.

For objects with a high degree of sub-document nesting, `net.wireObjectCheck` (page 903) can have a small impact on performance.

`net.`**`ipv6`**

Type: boolean

Default: False

Enable or disable IPv6 support and allows the `mongos` (page 792) or `mongod` (page 770) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes disable IPv6 support by default.

net.unixDomainSocket Options

```
net:
   unixDomainSocket:
      enabled: <boolean>
      pathPrefix: <string>
      filePermissions: <int>
```

net.unixDomainSocket.**enabled**

Type: boolean

Default: True

Enable or disable listening on the UNIX domain socket. net.unixDomainSocket.enabled (page 904) applies only to Unix-based systems.

When net.unixDomainSocket.enabled (page 904) is true, mongos (page 792) or mongod (page 770) listens on the UNIX socket.

The mongos (page 792) or mongod (page 770) process always listens on the UNIX socket unless one of the following is true:

- net.unixDomainSocket.enabled (page 904) is false
- *--nounixsocket* (page 794) is set. The command line option takes precedence over the configuration file setting.
- net.bindIp (page 903) is not set
- net.bindIp (page 903) does not specify 127.0.0.1

New in version 2.6: mongos (page 792) or mongod (page 770) installed from official .deb and .rpm packages have the bind_ip configuration set to 127.0.0.1 by default.

net.unixDomainSocket.**pathPrefix**

Type: string

Default: /tmp

The path for the UNIX socket. net.unixDomainSocket.pathPrefix (page 904) applies only to Unix-based systems.

If this option has no value, the mongos (page 792) or mongod (page 770) process creates a socket with https://docs.mongodb.org/manual/tmp as a prefix. MongoDB creates and listens on a UNIX socket unless one of the following is true:

- net.unixDomainSocket.enabled (page 904) is false
- *--nounixsocket* (page 794) is set
- net.bindIp (page 903) is not set
- net.bindIp (page 903) does not specify 127.0.0.1

net.unixDomainSocket.**filePermissions**

Type: int

Default: 0700

Sets the permission for the UNIX domain socket file.

net.unixDomainSocket.filePermissions (page 904) applies only to Unix-based systems.

`net.http` Options

```
net:
   http:
      enabled: <boolean>
      JSONPEnabled: <boolean>
      RESTInterfaceEnabled: <boolean>
```

Warning: Ensure that the HTTP status interface, the REST API, and the JSON API are all disabled in production environments to prevent potential data exposure and vulnerability to attackers.

`net.http.`**`enabled`**

Type: boolean

Default: False

Deprecated since version 3.2: HTTP interface for MongoDB

Enable or disable 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:

- While MongoDB Enterprise does support Kerberos authentication, Kerberos is not supported in HTTP status interface in any version of MongoDB.

New in version 2.6.

`net.http.`**`JSONPEnabled`**

Type: boolean

Default: False

Enable or disable *JSONP* access via an HTTP interface. Enabling the interface can increase network exposure. The `net.http.JSONPEnabled` (page 905) option enables the HTTP interface, even if the `HTTP interface` (page 905) option is disabled.

Deprecated since version 3.2: HTTP interface for MongoDB

The `net.http.JSONPEnabled` (page 905) setting is available only for `mongod` (page 770).

`net.http.`**`RESTInterfaceEnabled`**

Type: boolean

Default: False

Enable or disable the simple *REST* API. Enabling the *REST* API enables the HTTP interface, even if the `HTTP interface` (page 905) option is disabled, and as a result can increase network exposure.

Deprecated since version 3.2: HTTP interface for MongoDB

The `net.http.RESTInterfaceEnabled` (page 905) setting is available only for `mongod` (page 770).

`net.ssl` Options

```
net:
   ssl:
      sslOnNormalPorts: <boolean>   # deprecated since 2.6
```

```
mode: <string>
PEMKeyFile: <string>
PEMKeyPassword: <string>
clusterFile: <string>
clusterPassword: <string>
CAFile: <string>
CRLFile: <string>
allowConnectionsWithoutCertificates: <boolean>
allowInvalidCertificates: <boolean>
allowInvalidHostnames: <boolean>
disabledProtocols: <string>
FIPSMode: <boolean>
```

`net.ssl.`**`sslOnNormalPorts`**

Type: boolean

Deprecated since version 2.6.

Enable or disable TLS/SSL for `mongos` (page 792) or `mongod` (page 770).

With `net.ssl.sslOnNormalPorts` (page 906), a `mongos` (page 792) or `mongod` (page 770) requires TLS/SSL encryption for all connections on the default MongoDB port, or the port specified by *`--port`* (page 868). By default, *`--sslOnNormalPorts`* (page 797) is disabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`mode`**

Type: string

New in version 2.6.

Enable or disable TLS/SSL or mixed TLS/SSL used for all network connections. The argument to the `net.ssl.mode` (page 906) setting can be one of the following:

Value	Description
`disabled`	The server does not use TLS/SSL.
`allowSSL`	Connections between servers do not use TLS/SSL. For incoming connections, the server accepts both TLS/SSL and non-TLS/non-SSL.
`preferSSL`	Connections between servers use TLS/SSL. For incoming connections, the server accepts both TLS/SSL and non-TLS/non-SSL.
`requireSSL`	The server uses and accepts only TLS/SSL encrypted connections.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`PEMKeyFile`**

Type: string

The `.pem` file that contains both the TLS/SSL certificate and key. Specify the file name of the `.pem` file using relative or absolute paths.

You must specify `net.ssl.PEMKeyFile` (page 906) when TLS/SSL is enabled.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and

`https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`PEMKeyPassword`**
Type: string

The password to de-crypt the certificate-key file (i.e. `PEMKeyFile` (page 906)). Use the `net.ssl.PEMKeyPassword` (page 907) option only if the certificate-key file is encrypted. In all cases, the `mongos` (page 792) or `mongod` (page 770) 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 `net.ssl.PEMKeyPassword` (page 907) option, the `mongos` (page 792) or `mongod` (page 770) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`clusterFile`**
Type: string

New in version 2.6.

The `.pem` file that contains the x.509 certificate-key file for *membership authentication* for the cluster or replica set.

If `net.ssl.clusterFile` (page 907) does not specify the `.pem` file for internal cluster authentication, the cluster uses the `.pem` file specified in the `PEMKeyFile` (page 906) setting.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`clusterPassword`**
Type: string

New in version 2.6.

The password to de-crypt the x.509 certificate-key file specified with `--sslClusterFile`. Use the `net.ssl.clusterPassword` (page 907) option only if the certificate-key file is encrypted. In all cases, the `mongos` (page 792) or `mongod` (page 770) will redact the password from all logging and reporting output.

If the x.509 key file is encrypted and you do not specify the `net.ssl.clusterPassword` (page 907) option, the `mongos` (page 792) or `mongod` (page 770) will prompt for a passphrase. See *ssl-certificate-password*.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`CAFile`**
Type: string

New in version 2.4.

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and

`https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

> **Warning:** If the *`--sslCAFile`* option and its target file are not specified, x.509 client and member authentication will not function. `mongod` (page 770), and `mongos` (page 792) 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 770) will not start with x.509 authentication enabled if the CA file is not specified.

`net.ssl.`**`CRLFile`**

Type: string

New in version 2.4.

The the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`allowConnectionsWithoutCertificates`**

Type: boolean

New in version 2.4.

Changed in version 3.0.0: `net.ssl.weakCertificateValidation` became `net.ssl.allowConnectionsWithoutCertificates` (page 908). For compatibility, MongoDB processes continue to accept `net.ssl.weakCertificateValidation`, but all users should update their configuration files.

Enable or disable the requirement for TLS/SSL certificate validation that `CAFile` (page 907) enables. With the `net.ssl.allowConnectionsWithoutCertificates` (page 908) option, the `mongos` (page 792) or `mongod` (page 770) 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 792) or `mongod` (page 770) has `net.ssl.allowConnectionsWithoutCertificates` (page 908) enabled, the `mongos` (page 792) or `mongod` (page 770) will validate the certificate using the root certificate chain specified by `CAFile` (page 907) and reject clients with invalid certificates.

Use the `net.ssl.allowConnectionsWithoutCertificates` (page 908) option if you have a mixed deployment that includes clients that do not or cannot present certificates to the `mongos` (page 792) or `mongod` (page 770).

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`allowInvalidCertificates`**

Type: boolean

New in version 2.6.

Enable or disable the validation checks for TLS/SSL certificates on other servers in the cluster and allows the use of invalid certificates.

When using the `net.ssl.allowInvalidCertificates` (page 908) setting, MongoDB logs a warning regarding the use of the invalid certificate.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`allowInvalidHostnames`**

Type: boolean

Default: False

New in version 3.0.

When `net.ssl.allowInvalidHostnames` (page 909) is `true`, MongoDB disables the validation of the hostnames in TLS/SSL certificates, allowing `mongod` (page 770) to connect to MongoDB instances if the hostname their certificates do not match the specified hostname.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

`net.ssl.`**`disabledProtocols`**

Type: string

New in version 3.0.7.

Prevents a MongoDB server running with SSL from accepting incoming connections that use a specific protocol or protocols. `net.ssl.disabledProtocols` (page 909) recognizes the following protocols: `TLS1_0`, `TLS1_1`, and `TLS1_2`. Specifying an unrecognized protocol will prevent the server from starting.

To specify multiple protocols, use a comma separated list of protocols.

Members of replica sets and sharded clusters must speak at least one protocol in common.

See also:

ssl-disallow-protocols

`net.ssl.`**`FIPSMode`**

Type: boolean

New in version 2.4.

Enable or disable the use of the FIPS mode of the installed OpenSSL library for the `mongos` (page 792) or `mongod` (page 770). Your system must have a FIPS compliant OpenSSL library to use the `net.ssl.FIPSMode` (page 909) option.

Note: FIPS-compatible SSL is available only in MongoDB Enterprise[4]. See `https://docs.mongodb.org/manual/tutorial/configure-fips` for more information.

`security` Options

```
security:
   keyFile: <string>
   clusterAuthMode: <string>
   authorization: <string>
```

[4]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

```
   javascriptEnabled:  <boolean>
   sasl:
      hostName: <string>
      serviceName: <string>
      saslauthdSocketPath: <string>
   enableEncryption: <boolean>
   encryptionCipherMode: <string>
   encryptionKeyFile: <string>
   kmip:
      keyIdentifier: <string>
      rotateMasterKey: <boolean>
      serverName: <string>
      port: <string>
      clientCertificateFile: <string>
      clientCertificatePassword: <string>
      serverCAFile: <string>
```

security.**keyFile**

Type: string

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 910) implies `security.authorization` (page 910). See *inter-process-auth* for more information.

security.**clusterAuthMode**

Type: string

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.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

security.**authorization**

Type: string

Default: disabled

Enable or disable Role-Based Access Control (RBAC) to govern each user's access to database resources and operations.

Set this option to one of the following:

Value	Description
`enabled`	A user can access only the database resources and actions for which they have been granted privileges.
`disabled`	A user can access any database and perform any action.

See `https://docs.mongodb.org/manual/core/authorization` for more information.

The `security.authorization` (page 910) setting is available only for `mongod` (page 770).

`security.`**`javascriptEnabled`**

Type: boolean

Default: True

Enables or disables the `server-side JavaScript execution`. When disabled, you cannot use operations that perform server-side execution of JavaScript code, such as the `$where` (page 558) query operator, `mapReduce` (page 318) command and the `db.collection.mapReduce()` (page 90) method, `group` (page 313) command and the `db.collection.group()` (page 75) method.

Key Management Configuration Options

```
security:
   enableEncryption: <boolean>,
   encryptionCipherMode: <string>,
   encryptionKeyFile: <string>,
   kmip:
      keyIdentifier: <string>,
      rotateMasterKey: <boolean>
      serverName: <string>,
      port: <string>,
      clientCertificateFile: <string>,
      clientCertificatePassword: <string>,
      serverCAFile: <string>
```

`security.`**`enableEncryption`**

Type: boolean

Default: False

New in version 3.2: Enables encryption for the WiredTiger storage engine. You must set to `true` to pass in encryption keys and configurations.

Enterprise Feature

Available in MongoDB Enterprise only.

`security.`**`encryptionCipherMode`**

Type: string

Default: `AES256-CBC`

New in version 3.2.

The cipher mode to use for encryption at rest:

Mode	Description
`AES256-CBC`	256-bit Advanced Encryption Standard in Cipher Block Chaining Mode
`AES256-GCM`	256-bit Advanced Encryption Standard in Galois/Counter Mode

Enterprise Feature

Available in MongoDB Enterprise only.

`security.`**`encryptionKeyFile`**

Type: string

New in version 3.2.

The path to the local keyfile when managing keys via process *other than* KMIP. Only set when managing keys via process other than KMIP. If data is already encrypted using KMIP, MongoDB will throw an error.

Requires `security.enableEncryption` (page 911) to be `true`.

Enterprise Feature

Available in MongoDB Enterprise only.

`security.kmip.`**`keyIdentifier`**

Type: string

New in version 3.2.

Unique KMIP identifier for an existing key within the KMIP server. Include to use the key associated with the identifier as the system key. You can only use the setting the first time you enable encryption for the `mongod` (page 770) instance. Requires `security.enableEncryption` (page 911) to be true.

If unspecified, MongoDB will request that the KMIP server create a new key to utilize as the system key.

If the KMIP server cannot locate a key with the specified identifier or the data is already encrypted with a key, MongoDB will throw an error.

Enterprise Feature

Available in MongoDB Enterprise only.

`security.kmip.`**`rotateMasterKey`**

Type: boolean

Default: False

New in version 3.2.

If true, rotate the master key and re-encrypt the internal keystore.

Enterprise Feature

Available in MongoDB Enterprise only.

See also:

kmip-master-key-rotation

`security.kmip.`**`serverName`**

Type: string

New in version 3.2.

Hostname or IP address of key management solution running a KMIP server. Requires `security.enableEncryption` (page 911) to be true.

Enterprise Feature

Available in MongoDB Enterprise only.

`security.kmip.port`

Type: string

Default: 5696

New in version 3.2.

Port number the KMIP server is listening on. Requires that a `security.kmip.serverName` (page 912) be provided. Requires `security.enableEncryption` (page 911) to be true.

Enterprise Feature

Available in MongoDB Enterprise only.

`security.kmip.clientCertificateFile`

Type: string

New in version 3.2.

String containing the path to the client certificate used for authenticating MongoDB to the KMIP server. Requires that a `security.kmip.serverName` (page 912) be provided.

Enterprise Feature

Available in MongoDB Enterprise only.

`security.kmip.clientCertificatePassword`

Type: string

New in version 3.2.

The password to decrypt the client certificate (i.e. `security.kmip.clientCertificateFile` (page 913)), used to authenticate MongoDB to the KMIP server. Use the option only if the certificate is encrypted.

Enterprise Feature

Available in MongoDB Enterprise only.

`security.kmip.serverCAFile`

Type: string

New in version 3.2.

Path to CA File. Used for validating secure client connection to KMIP server.

Enterprise Feature

Available in MongoDB Enterprise only.

`security.sasl` Options

```
security:
   sasl:
      hostName: <string>
      serviceName: <string>
      saslauthdSocketPath: <string>
```

security.sasl.**hostName**

Type: string

A fully qualified server domain name for the purpose of configuring SASL and Kerberos authentication. The SASL hostname overrides the hostname only for the configuration of SASL and Kerberos.

For `mongo` (page 803) shell and other MongoDB tools to connect to the new `hostName` (page 913), see the `gssapiHostName` option in the `mongo` (page 803) shell and other tools.

security.sasl.**serviceName**

Type: string

Registered name of the service using SASL. This option allows you to override the default `Kerberos` service name component of the `Kerberos` principal name, on a per-instance basis. If unspecified, the default value is `mongodb`.

MongoDB permits setting this option only at startup. The `setParameter` (page 460) can not change this setting.

This option is available only in MongoDB Enterprise.

Important: Ensure that your driver supports alternate service names. For `mongo` (page 803) shell and other MongoDB tools to connect to the new `serviceName` (page 914), see the `gssapiServiceName` option.

security.sasl.**saslauthdSocketPath**

Type: string

The path to the UNIX domain socket file for `saslauthd`.

`setParameter` Option

setParameter

Set MongoDB parameter or parameters described in *MongoDB Server Parameters* (page 927)

To set parameters in the YAML configuration file, use the following format:

```
setParameter:
   <parameter1>: <value1>
   <parameter2>: <value2>
```

For example, to specify the `enableLocalhostAuthBypass` (page 928) in the configuration file:

```
setParameter:
   enableLocalhostAuthBypass: false
```

`storage` Options

```
storage:
   dbPath: <string>
   indexBuildRetry: <boolean>
   repairPath: <string>
   journal:
      enabled: <boolean>
      commitIntervalMs: <num>
   directoryPerDB: <boolean>
   syncPeriodSecs: <int>
   engine: <string>
   mmapv1:
      preallocDataFiles: <boolean>
      nsSize: <int>
      quota:
         enforced: <boolean>
         maxFilesPerDB: <int>
      smallFiles: <boolean>
      journal:
         debugFlags: <int>
         commitIntervalMs: <num>
   wiredTiger:
      engineConfig:
         cacheSizeGB: <number>
         statisticsLogDelaySecs: <number>
         journalCompressor: <string>
         directoryForIndexes: <boolean>
      collectionConfig:
         blockCompressor: <string>
      indexConfig:
         prefixCompression: <boolean>
```

`storage.`**`dbPath`**

Type: string

Default: `/data/db` on Linux and OS X, `\data\db` on Windows

The directory where the `mongod` (page 770) instance stores its data.

If you installed MongoDB using a package management system, check the `/etc/mongod.conf` file provided by your packages to see the directory is specified.

The `storage.dbPath` (page 915) setting is available only for `mongod` (page 770).

The Linux package init scripts do not expect `storage.dbPath` (page 915) to change from the defaults. If you use the Linux packages and change `storage.dbPath` (page 915), you will have to use your own init scripts and disable the built-in scripts.

`storage.`**`indexBuildRetry`**

Type: boolean

Default: True

Specifies whether `mongod` (page 770) rebuilds incomplete indexes on the next start up. This applies in cases where `mongod` (page 770) restarts after it has shut down or stopped in the middle of an index build. In such cases, `mongod` (page 770) always removes any incomplete indexes, and then, by default, attempts to rebuild them. To stop `mongod` (page 770) from rebuilding indexes, set this option to `false`.

The `storage.indexBuildRetry` (page 915) setting is available only for `mongod` (page 770).

`storage.`**`repairPath`**

Type: string

Default: A `_tmp` directory within the path specified by the `dbPath` (page 915) option.

The working directory that MongoDB will use during the *`--repair`* (page 821) operation. After *`--repair`* (page 821) completes, the data files in `dbPath` (page 915) and the `storage.repairPath` (page 915) directory is empty.

The `storage.repairPath` (page 915) setting is available only for `mongod` (page 770).

`storage.journal.`**`enabled`**

Type: boolean

Default: `true` on 64-bit systems, `false` on 32-bit systems

Enable or disable the durability *journal* to ensure data files remain valid and recoverable. This option applies only when you specify the `--dbpath` (page 776) option. The `mongod` (page 770) enables journaling by default on 64-bit builds of versions after 2.0.

The `storage.journal.enabled` (page 915) setting is available only for `mongod` (page 770).

`storage.journal.`**`commitIntervalMs`**

Type: number

Default: 100 or 30

New in version 3.2.

The maximum amount of time in milliseconds that the `mongod` (page 770) process allows between journal operations. Values can range from 1 to 500 milliseconds. Lower values increase the durability of the journal, at the expense of disk performance. The default journal commit interval is 100 milliseconds.

On MMAPv1, if the journal is on a different block device (e.g. physical volume, RAID device, or LVM volume) than the data files, the default journal commit interval is 30 milliseconds. Additionally, on MMAPv1, when a write operation with `j:true` is pending, `mongod` (page 770) will reduce `commitIntervalMs` (page 916) to a third of the set value.

On WiredTiger, the default journal commit interval is 100 milliseconds. Additionally, a write with `j:true` will cause an immediate sync of the journal.

The `storage.journal.commitIntervalMs` (page 916) setting is available only for `mongod` (page 770).

`storage.`**`directoryPerDB`**

Type: boolean

Default: False

When `true`, MongoDB uses a separate directory to store data for each database. The directories are under the `storage.dbPath` (page 915) directory, and each subdirectory name corresponds to the database name.

Changed in version 3.0: To change the `storage.directoryPerDB` (page 916) option for existing deployments, you must restart the `mongod` (page 770) instances with the new `storage.directoryPerDB` (page 916) value **and** a new data directory (`storage.dbPath` (page 915) value), and then repopulate the data.

- For standalone instances, you can use `mongodump` (page 816) on the existing instance, stop the instance, restart with the new `storage.directoryPerDB` (page 916) value **and** a new data directory, and use `mongorestore` (page 824) to populate the new data directory.
- For replica sets, you can update in a rolling manner by stopping a secondary member, restart with the new `storage.directoryPerDB` (page 916) value **and** a new data directory, and use *initial sync* to populate the new data directory. To update all members, start with the secondary members first. Then step down the primary, and update the stepped-down member.

The `storage.directoryPerDB` (page 916) setting is available only for `mongod` (page 770).

`storage.`**`syncPeriodSecs`**

Type: number

Default: 60

The amount of time that 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 `storage.syncPeriodSecs` (page 916) to `0`, MongoDB will not sync the memory mapped files to disk.

The `mongod` (page 770) process writes data very quickly to the journal and lazily to the data files. `storage.syncPeriodSecs` (page 916) has no effect on the `journal` (page 915) files or `journaling`.

The `serverStatus` (page 492) command reports the background flush thread's status via the `backgroundFlushing` (page 494) field.

The `storage.syncPeriodSecs` (page 916) setting is available only for `mongod` (page 770).

`storage.`**`engine`**

Default: `wiredTiger`

New in version 3.0.

Changed in version 3.2: Starting in MongoDB 3.2, `wiredTiger` is the default.

The storage engine for the `mongod` (page 770) database. Available values include:

Value	Description
`mmapv1`	To specify the `https://docs.mongodb.org/manual/core/mmapv1`.
`wiredTiger`	To specify the `https://docs.mongodb.org/manual/core/wiredtiger`.
`inMemory`	To specify the `https://docs.mongodb.org/manual/core/inmemory`. The in-memory storage engine is currently in **beta**. Do not use in production. New in version 3.2: Available in MongoDB Enterprise only.

If you attempt to start a `mongod` (page 770) with a `storage.dbPath` (page 915) that contains data files produced by a storage engine other than the one specified by `storage.engine` (page 917), `mongod` (page 770) will refuse to start.

`storage.mmapv1` Options

```
storage:
   mmapv1:
      preallocDataFiles: <boolean>
      nsSize: <int>
      quota:
         enforced: <boolean>
         maxFilesPerDB: <int>
      smallFiles: <boolean>
      journal:
         debugFlags: <int>
         commitIntervalMs: <num>
```

`storage.mmapv1.`**`preallocDataFiles`**

Type: boolean

Default: True

Deprecated since version 2.6.

Enable or disable the preallocation of data files. Currently the default. Exists for future compatibility and clarity.

The `storage.mmapv1.preallocDataFiles` (page 917) setting is available only for `mongod` (page 770).

`storage.mmapv1.`**`nsSize`**

Type: integer

Default: 16

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.

The `storage.mmapv1.nsSize` (page 917) setting is available only for `mongod` (page 770).

`storage.mmapv1.quota.`**`enforced`**

Type: Boolean

Default: false

Enable or disable the enforcement of a maximum limit for the number data files each database can have. When running with the `storage.mmapv1.quota.enforced` (page 918) option, MongoDB has a maximum of 8 data files per database. Adjust the quota with `storage.quota.maxFilesPerDB`.

The `storage.mmapv1.quota.enforced` (page 918) setting is available only for `mongod` (page 770).

`storage.mmapv1.quota.`**`maxFilesPerDB`**

Type: integer

Default: 8

The limit on the number of data files per database. `storage.mmapv1.quota.maxFilesPerDB` (page 918) option requires that you set `storage.quota.enforced`.

The `storage.mmapv1.quota.maxFilesPerDB` (page 918) setting is available only for `mongod` (page 770).

`storage.mmapv1.`**`smallFiles`**

Type: boolean

Default: False

When `true`, MongoDB uses a smaller default file size. The `storage.mmapv1.smallFiles` (page 918) option reduces the initial size for data files and limits the maximum size to 512 megabytes. `storage.mmapv1.smallFiles` (page 918) also reduces the size of each *journal* file from 1 gigabyte to 128 megabytes. Use `storage.mmapv1.smallFiles` (page 918) if you have a large number of databases that each holds a small quantity of data.

The `storage.mmapv1.smallFiles` (page 918) option can lead the `mongod` (page 770) instance to create a large number of files, which can affect performance for larger databases.

The `storage.mmapv1.smallFiles` (page 918) setting is available only for `mongod` (page 770).

`storage.mmapv1.journal.`**`debugFlags`**

Type: integer

Provides functionality for testing. Not for general use, and will affect data file integrity in the case of abnormal system shutdown.

The `storage.mmapv1.journal.debugFlags` (page 918) option is available only for `mongod` (page 770).

`storage.mmapv1.journal.`**`commitIntervalMs`**

Type: number

Deprecated since version 3.2: MongoDB 3.2 deprecates the `storage.mmapv1.journal.commitIntervalMs` (page 918) setting. Use `storage.journal.commitIntervalMs` (page 916) instead.

The deprecated setting acts as an alias to the new `storage.journal.commitIntervalMS` setting and applies to either the MMAPv1 or the WiredTiger storage engine.

`storage.wiredTiger` Options

```
storage:
   wiredTiger:
      engineConfig:
         cacheSizeGB: <number>
         statisticsLogDelaySecs: <number>
         journalCompressor: <string>
         directoryForIndexes: <boolean>
      collectionConfig:
         blockCompressor: <string>
      indexConfig:
         prefixCompression: <boolean>
```

`storage.wiredTiger.engineConfig.`**`cacheSizeGB`**

New in version 3.0.

The maximum size of the cache that WiredTiger will use for all data.

With WiredTiger, MongoDB utilizes both the WiredTiger cache and the filesystem cache.

Changed in version 3.2: Starting in MongoDB 3.2, the WiredTiger cache, by default, will use the larger of either:

- 60% of RAM minus 1 GB, or
- 1 GB.

For systems with up to 10 GB of RAM, the new default setting is less than or equal to the 3.0 default setting (For MongoDB 3.0, the WiredTiger cache uses either 1 GB or half of the installed physical RAM, whichever is larger).

For systems with more than 10 GB of RAM, the new default setting is greater than the 3.0 setting.

Via the filesystem cache, MongoDB automatically uses all free memory that is not used by the WiredTiger cache or by other processes. Data in the filesystem cache is compressed.

Avoid increasing the WiredTiger cache size above its default value.

Note: The `storage.wiredTiger.engineConfig.cacheSizeGB` (page 919) only limits the size of the WiredTiger cache, not the total amount of memory used by `mongod` (page 770). The WiredTiger cache is only one component of the RAM used by MongoDB. MongoDB also automatically uses all free memory on the machine via the filesystem cache (data in the filesystem cache is compressed).

In addition, the operating system will use any free RAM to buffer filesystem blocks.

To accommodate the additional consumers of RAM, you may have to decrease WiredTiger cache size.

The default WiredTiger cache size value assumes that there is a single `mongod` (page 770) instance per node. If a single node contains multiple instances, then you should decrease the setting to accommodate the other `mongod` (page 770) instances.

If you run `mongod` (page 770) in a container (e.g. `lxc`, `cgroups`, Docker, etc.) that does *not* have access to all of the RAM available in a system, you must set `storage.wiredTiger.engineConfig.cacheSizeGB` (page 919) to a value less than the amount of RAM available in the container. The exact amount depends on the other processes running in the container.

`storage.wiredTiger.engineConfig.`**`statisticsLogDelaySecs`**

Default: 0

New in version 3.0.0.

The interval, in seconds, at which WiredTiger logs statistics to the file specified in the `dbPath` (page 915) or *--dbpath* (page 776). When `storage.wiredTiger.engineConfig.statisticsLogDelaySecs` (page 919) is set to `0`, WiredTiger does not log statistics.

`storage.wiredTiger.engineConfig.`**`journalCompressor`**

Default: snappy

New in version 3.0.0.

The type of compression to use to compress WiredTiger journal data.

Available compressors are:

- `none`
- *snappy*
- *zlib*

`storage.wiredTiger.engineConfig.`**`directoryForIndexes`**

Type: boolean

Default: false

New in version 3.0.0.

When `storage.wiredTiger.engineConfig.directoryForIndexes` (page 920) is `true`, `mongod` (page 770) stores indexes and collections in separate subdirectories under the data (i.e. `storage.dbPath` (page 915)) directory. Specifically, `mongod` (page 770) stores the indexes in a subdirectory named `index` and the collection data in a subdirectory named `collection`.

By using a symbolic link, you can specify a different location for the indexes. Specifically, when `mongod` (page 770) instance is **not** running, move the `index` subdirectory to the destination and create a symbolic link named `index` under the data directory to the new destination.

`storage.wiredTiger.collectionConfig.`**`blockCompressor`**

Default: snappy

New in version 3.0.0.

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*

`storage.wiredTiger.collectionConfig.blockCompressor` (page 920) affects all collections created. If you change the value of `storage.wiredTiger.collectionConfig.blockCompressor` (page 920) on an existing MongoDB deployment, all new collections will use the specified compressor. Existing collections will continue to use the compressor specified when they were created, or the default compressor at that time.

`storage.wiredTiger.indexConfig.`**`prefixCompression`**

Default: true

New in version 3.0.0.

Enables or disables *prefix compression* for index data.

Specify `true` for `storage.wiredTiger.indexConfig.prefixCompression` (page 920) to enable *prefix compression* for index data, or `false` to disable prefix compression for index data.

The `storage.wiredTiger.indexConfig.prefixCompression` (page 920) setting affects all indexes created. If you change the value of `storage.wiredTiger.indexConfig.prefixCompression` (page 920) on an existing MongoDB deployment, all new indexes will use prefix compression. Existing indexes are not affected.

`operationProfiling` Options

```
operationProfiling:
   slowOpThresholdMs: <int>
   mode: <string>
```

operationProfiling.**slowOpThresholdMs**

Type: 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 484) command for more information on the database profiler.

The `operationProfiling.slowOpThresholdMs` (page 921) setting is available only for `mongod` (page 770).

operationProfiling.**mode**

Type: string

Default: off

The level of database profiling, which inserts information about operation performance into the `system.profile` collection. Specify one of the following levels:

Level	Setting
`off`	Off. No profiling.
`slowOp`	On. Only includes slow operations.
`all`	On. Includes all operations.

Database profiling can impact database performance. Enable this option only after careful consideration.

The `operationProfiling.mode` (page 921) setting is available only for `mongod` (page 770).

`replication` Options

```
replication:
   oplogSizeMB: <int>
   replSetName: <string>
   secondaryIndexPrefetch: <string>
   enableMajorityReadConcern: <boolean>
```

replication.**oplogSizeMB**

Type: integer

The maximum size in megabytes for the replication operation log (i.e., the *oplog*). The `mongod` (page 770) 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 770) has created the oplog for the first time, changing the `replication.oplogSizeMB` (page 921) option will not affect the size of the oplog.

See *replica-set-oplog-sizing* for more information.

The `replication.oplogSizeMB` (page 921) setting is available only for `mongod` (page 770).

`replication.`**`replSetName`**

Type: string

The name of the replica set that the `mongod` (page 770) is part of. 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.

The `replication.replSetName` (page 922) setting is available only for `mongod` (page 770).

`replication.`**`secondaryIndexPrefetch`**

Type: string

Default: all

Storage Engine Specific Feature

`replication.secondaryIndexPrefetch` (page 922) is only available with the `mmapv1` storage engine.

The indexes that *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.

Set this setting to one of the following:

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.

The `replication.secondaryIndexPrefetch` (page 922) setting is available only for `mongod` (page 770).

`replication.`**`enableMajorityReadConcern`**

Type: boolean

Default: False

New in version 3.2.

Enables *read concern* level of `"majority"`.

`sharding` Options

```
sharding:
   clusterRole: <string>
   archiveMovedChunks: <boolean>
```

`sharding.`**`clusterRole`**

Type: string

The role that the `mongod` (page 770) instance has in the sharded cluster. Set this setting to one of the following:

Value	Description
`configsvr`	Start this instance as a *config server*. The instance starts on port `27019` by default.
`shardsvr`	Start this instance as a *shard*. The instance starts on port `27018` by default.

The `sharding.clusterRole` (page 922) setting is available only for `mongod` (page 770).

`sharding.archiveMovedChunks`

Type: boolean

Changed in version 3.2: Starting in 3.2, MongoDB uses `false` as the default.

During chunk migration, a shard does not save documents migrated from the shard.

`auditLog` Options

Note: Available only in MongoDB Enterprise[5].

```
auditLog:
   destination: <string>
   format: <string>
   path: <string>
   filter: <string>
```

`auditLog.destination`

Type: string

New in version 2.6.

When set, `auditLog.destination` (page 923) enables `auditing` and specifies where `mongos` (page 792) or `mongod` (page 770) sends all audit events.

`auditLog.destination` (page 923) 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 801) in the format specified in *`--auditFormat`* (page 801).

Note: Available only in MongoDB Enterprise[6].

`auditLog.format`

Type: string

New in version 2.6.

The format of the output file for `auditing` if `destination` (page 923) is `file`. The `auditLog.format` (page 923) 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 801).
`BSON`	Output the audit events in BSON binary format to the file specified in *`--auditPath`* (page 801).

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[7].

[5] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[6] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[7] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

auditLog.**path**

Type: string

New in version 2.6.

The output file for auditing if destination (page 923) has value of file. The auditLog.path (page 924) option can take either a full path name or a relative path name.

Note: Available only in MongoDB Enterprise[8].

auditLog.**filter**

Type: string representation of a document

New in version 2.6.

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 527).

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* (page 895), you must use the YAML format of the configuration file.

Note: Available only in MongoDB Enterprise[9].

snmp Options

```
snmp:
   subagent: <boolean>
   master: <boolean>
```

snmp.**subagent**

Type: boolean

When snmp.subagent (page 924) is true, SNMP runs as a subagent. For more information, see https://docs.mongodb.org/manual/tutorial/monitor-with-snmp.

The snmp.subagent (page 924) setting is available only for mongod (page 770).

snmp.**master**

Type: boolean

When snmp.master (page 924) is true, SNMP runs as a master. For more information, see https://docs.mongodb.org/manual/tutorial/monitor-with-snmp.

The snmp.master (page 924) setting is available only for mongod (page 770).

[8]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[9]http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Text Search Options

`basisTech`

`basisTech.`**`rootDirectory`**

Type: string

New in version 3.2.

Specify the path to the root directory of the Basis Technology Rosette Linguistics Platform installation to support additional languages for text search operations.

Enterprise Feature

Available in MongoDB Enterprise only.

6.1.3 `mongos`-only Options

```
replication:
   localPingThresholdMs: <boolean>

sharding:
   autoSplit: <boolean>
   configDB: <string>
   chunkSize: <int>
```

`replication.`**`localPingThresholdMs`**

Type: integer

Default: 15

The ping time, in milliseconds, that `mongos` (page 792) uses to determine which secondary replica set members to pass read operations from clients. The default value of `15` corresponds to the default value in all of the client `drivers`.

When `mongos` (page 792) receives a request that permits reads to *secondary* members, the `mongos` (page 792) 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 `replication.localPingThresholdMs` (page 925) option, `mongos` (page 792) 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 `replication.localPingThresholdMs` (page 925) 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 792) recalculates the average.

See the *replica-set-read-preference-behavior-member-selection* section of the `read preference` documentation for more information.

`sharding.`**`autoSplit`**

Type: boolean

Default: True

Enables or disables the automatic splitting of chunks for *sharded collections*. If `sharding.autoSplit` (page 925) is `false` on all `mongos` (page 792) instances, MongoDB does not create new chunks as the data in a collection grows.

Because any `mongos` (page 792) in a cluster can create a split, to totally disable splitting in a cluster, you must set `sharding.autoSplit` (page 925) to `false` on all `mongos` (page 792).

> **Warning:** With auto-splitting disabled, the data in your sharded cluster may become imbalanced over time. Disable with caution.

`sharding.`**`configDB`**

Type: string

Changed in version 3.2.

The *configuration servers* for the *sharded cluster*.

Starting in MongoDB 3.2, config servers for sharded clusters can be deployed as a `replica set`. The replica set config servers must run the `WiredTiger storage engine`. MongoDB 3.2 deprecates the use of three mirrored `mongod` (page 770) instances for config servers.

Specify the config server replica set name and the hostname and port of one of the members of the config server replica set.

The `mongos` (page 792) instances for the sharded cluster must specify the same config server replica set name but can specify hostname and port of different members of the replica set.

If using the deprecated mirrored instances, specify the hostnames and ports of the three `mongod` (page 770) instances. The `mongos` (page 792) instances must specify the same config string.

`sharding.`**`chunkSize`**

Type: integer

Default: 64

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.

`sharding.chunkSize` (page 926) 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 `https://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.

6.1.4 Windows Service Options

```
processManagement:
   windowsService:
      serviceName: <string>
      displayName: <string>
      description: <string>
      serviceUser: <string>
      servicePassword: <string>
```

`processManagement.windowsService.`**`serviceName`**

Type: string

Default: MongoDB

The service name of `mongos` (page 792) or `mongod` (page 770) when running as a Windows Service. Use this name with the `net start <name>` and `net stop <name>` operations.

You must use `processManagement.windowsService.serviceName` (page 926) in conjunction with either the `--install` (page 813) or `--remove` (page 813) install option.

`processManagement.windowsService.`**`displayName`**

Type: string

Default: MongoDB

The name listed for MongoDB on the Services administrative application.

`processManagement.windowsService.`**`description`**

Type: string

Default: MongoDB Server

Run `mongos` (page 792) or `mongod` (page 770) service description.

You must use `processManagement.windowsService.description` (page 927) in conjunction with the `--install` (page 813) option.

For descriptions that contain spaces, you must enclose the description in quotes.

`processManagement.windowsService.`**`serviceUser`**

Type: string

The `mongos` (page 792) or `mongod` (page 770) service in the context of a certain user. This user must have "Log on as a service" privileges.

You must use `processManagement.windowsService.serviceUser` (page 927) in conjunction with the `--install` (page 813) option.

`processManagement.windowsService.`**`servicePassword`**

Type: string

The password for `<user>` for `mongos` (page 792) or `mongod` (page 770) when running with the `--serviceUser` (page 814) option.

You must use `processManagement.windowsService.servicePassword` (page 927) in conjunction with the `--install` (page 813) option.

6.2 MongoDB Server Parameters

On this page

- Synopsis (page 927)
- Parameters (page 928)
- Diagnostic Parameters (page 936)

6.2.1 Synopsis

MongoDB provides a number of configuration options that are accessible via the `setParameter` (page 460) command or the `--setParameter` (page 794) option.

For additional configuration options, see *Configuration File Options* (page 895) and *Manual Page for mongod* (page 769).

6.2.2 Parameters

Authentication Parameters

authenticationMechanisms

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism.

Available for both `mongod` (page 770) and `mongos` (page 792).

Specifies the list of authentication mechanisms the server accepts. Set this to one or more of the following values. If you specify multiple values, use a comma-separated list and no spaces. For descriptions of the authentication mechanisms, see `https://docs.mongodb.org/manual/core/authentication`.

Value	Description
SCRAM-SHA-1	RFC 5802[10] standard Salted Challenge Response Authentication Mechanism using the SHA-1 hash function.
MONGODB-CR	MongoDB challenge/response authentication.
MONGODB-X509	MongoDB TLS/SSL certificate authentication.
GSSAPI (Kerberos)	External authentication using Kerberos. This mechanism is available only in MongoDB Enterprise[11].
PLAIN (LDAP SASL)	External authentication using LDAP. `PLAIN` transmits passwords in plain text. This mechanism is available only in MongoDB Enterprise[12].

For example, to specify `PLAIN` as the authentication mechanism, use the following command:

```
mongod --setParameter authenticationMechanisms=PLAIN --auth
```

clusterAuthMode

New in version 2.6.

Available for both `mongod` (page 770) and `mongos` (page 792).

Set the `clusterAuthMode` (page 910) to either `sendX509` or `x509`. Useful during *rolling upgrade to use x509 for membership authentication* to minimize downtime.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See `https://docs.mongodb.org/manual/tutorial/configure-ssl` and `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information about TLS/SSL and MongoDB.

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, clusterAuthMode: "sendX509" } )
```

enableLocalhostAuthBypass

New in version 2.4.

Available for both `mongod` (page 770) and `mongos` (page 792).

Specify `0` or `false` to disable localhost authentication bypass. Enabled by default.

`enableLocalhostAuthBypass` (page 928) is not available using `setParameter` (page 460) database command. Use the `setParameter` (page 914) option in the configuration file or the `--setParameter` option on the command line.

[10] https://tools.ietf.org/html/rfc5802
[11] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
[12] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

See *localhost-exception* for more information.

saslauthdPath

Note: Available only in MongoDB Enterprise (except MongoDB Enterprise for Windows).

Available for both `mongod` (page 770) and `mongos` (page 792).

Specify the path to the Unix Domain Socket of the `saslauthd` instance to use for proxy authentication.

saslHostName

New in version 2.4.

Available for both `mongod` (page 770) and `mongos` (page 792).

`saslHostName` (page 929) overrides MongoDB's default hostname detection for the purpose of configuring SASL and Kerberos authentication.

`saslHostName` (page 929) does not affect the hostname of the `mongod` (page 770) or `mongos` (page 792) instance for any purpose beyond the configuration of SASL and Kerberos.

You can only set `saslHostName` (page 929) during start-up, and cannot change this setting using the `setParameter` (page 460) database command.

Note: `saslHostName` (page 929) supports Kerberos authentication and is only included in MongoDB Enterprise. For Linux systems, see `https://docs.mongodb.org/manual/tutorial/control-access-to-mongodb-with-kerberos-authe` for more information.

saslServiceName

New in version 2.4.6.

Available for both `mongod` (page 770) and `mongos` (page 792).

Allows users to override the default `Kerberos` service name component of the `Kerberos` principal name, on a per-instance basis. If unspecified, the default value is `mongodb`.

MongoDB only permits setting `saslServiceName` (page 929) at startup. The `setParameter` (page 460) command can not change this setting.

`saslServiceName` (page 929) is only available in MongoDB Enterprise.

Important: Ensure that your driver supports alternate service names.

scramIterationCount

New in version 3.0.0.

Default: `10000`

Available for both `mongod` (page 770) and `mongos` (page 792).

Changes the number of hashing iterations used for all new stored passwords. More iterations increase the amount of time required for clients to authenticate to MongoDB, but makes passwords less susceptible to brute-force attempts. The default value is ideal for most common use cases and requirements. If you modify this value, it does not change the number of iterations for existing passwords.

You can set `scramIterationCount` (page 929) when starting MongoDB or on running `mongod` (page 770) instances.

sslMode

New in version 2.6.

Available for both mongod (page 770) and mongos (page 792).

Set the net.ssl.mode (page 906) to either preferSSL or requireSSL. Useful during rolling upgrade to TLS/SSL to minimize downtime.

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See https://docs.mongodb.org/manual/tutorial/configure-ssl and https://docs.mongodb.org/manual/tutorial/configure-ssl-clients for more information about TLS/SSL and MongoDB.

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, sslMode: "preferSSL" } )
```

supportCompatibilityFormPrivilegeDocuments

Changed in version 3.0: Removed in MongoDB 3.0

Deprecated since version 2.6: supportCompatibilityFormPrivilegeDocuments (page 930) has no effect in 2.6 and will be removed in 3.0.

New in version 2.4.

userCacheInvalidationIntervalSecs

Default: 30.

Available for mongos (page 792) only.

On a mongos (page 792) instance, specifies the interval (in seconds) at which the mongos (page 792) instance checks to determine whether the in-memory cache of user objects has stale data, and if so, clears the cache. If there are no changes to user objects, mongos (page 792) will not clear the cache.

This parameter has a minimum value of 1 second and a maximum value of 86400 seconds (24 hours).

Changed in version 3.0: Default value has changed to 30 seconds, and the minimum value allowed has changed to 1 second. mongos (page 792) only clears the user cache if there are changes.

General Parameters

connPoolMaxShardedConnsPerHost

New in version 2.6.

Default: 200

Available for both mongod (page 770) and mongos (page 792).

Set the maximum size of the connection pools for communication to the shards. The size of a pool does not prevent the creation of additional connections, but *does* prevent the connection pools from retaining connections above this limit.

Increase the connPoolMaxShardedConnsPerHost (page 930) value **only** if the number of connections in a connection pool has a high level of churn or if the total number of created connections increase.

You can only set connPoolMaxShardedConnsPerHost (page 930) during startup in the config file or on the command line, as follows to increase the size of the connection pool:

```
mongos --setParameter connPoolMaxShardedConnsPerHost=250
```

connPoolMaxConnsPerHost

New in version 2.6.

Default: 200

Available for both mongod (page 770) and mongos (page 792).

Set the maximum size of the connection pools for outgoing connections to other mongod (page 770) instances. The size of a pool does not prevent the creation of additional connections, but *does* prevent a connection pool from retaining connections in excess of the value of connPoolMaxConnsPerHost (page 930).

Only adjust this setting if your driver does *not* pool connections and you're using authentication in the context of a sharded cluster.

You can only set connPoolMaxConnsPerHost (page 930) during startup in the config file or on the command line, as in the following example:

```
mongod --setParameter connPoolMaxConnsPerHost=250
```

cursorTimeoutMillis

New in version 3.0.2.

Default: 600000 (i.e. 10 minutes)

Available for both mongod (page 770) and mongos (page 792).

Sets the expiration threshold in milliseconds for idle cursors before MongoDB removes them; i.e. MongoDB removes cursors that have been idle for the specified cursorTimeoutMillis (page 931).

For example, the following sets the cursorTimeoutMillis (page 931) to 300000 milliseconds (i.e. 5 minutes).

```
mongod --setParameter cursorTimeoutMillis=300000
```

Or, if using the setParameter (page 460) command within the mongo (page 803) shell:

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, cursorTimeoutMillis: 300000 } )
```

enableTestCommands

New in version 2.4.

Available for both mongod (page 770) and mongos (page 792).

enableTestCommands (page 931) enables a set of internal commands useful for internal testing operations. enableTestCommands (page 931) is only available at startup and you cannot use setParameter (page 460) to modify this parameter. Consider the following mongod (page 770) invocation, which sets enableTestCommands (page 931):

```
mongod --setParameter enableTestCommands=1
```

enableTestCommands (page 931) provides access to the following internal commands:

- captrunc (page 522)
- configureFailPoint (page 526)
- emptycapped (page 522)
- godinsert (page 522)
- _hashBSONElement (page 523)
- journalLatencyTest (page 524)
- replSetTest (page 525)
- _skewClockCommand (page 525)
- sleep (page 524)
- _testDistLockWithSkew (page 521)

- _testDistLockWithSyncCluster (page 521)

failIndexKeyTooLong

New in version 2.6.

Available for mongod (page 770) only.

In MongoDB 2.6, if you attempt to insert or update a document so that the value of an indexed field is longer than the Index Key Length Limit (page 941), the operation will fail and return an error to the client. In previous versions of MongoDB, these operations would successfully insert or modify a document but the index or indexes would not include references to the document.

To avoid this issue, consider using hashed indexes or indexing a computed value. If you have an existing data set and want to disable this behavior so you can upgrade and then gradually resolve these indexing issues, you can use failIndexKeyTooLong (page 932) to disable this behavior.

failIndexKeyTooLong (page 932) defaults to true. When false, a 2.6 mongod (page 770) instance will provide the 2.4 behavior.

Issue the following command to disable the index key length validation: for a running:program:*mongod* instance:

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, failIndexKeyTooLong: false } )
```

You can also set failIndexKeyTooLong (page 932) at startup time with the following option:

```
mongod --setParameter failIndexKeyTooLong=false
```

newCollectionsUsePowerOf2Sizes

Deprecated since version 3.0.0: MongoDB deprecates the newCollectionsUsePowerOf2Sizes (page 932) parameter such that you cannot set the newCollectionsUsePowerOf2Sizes (page 932) to false and newCollectionsUsePowerOf2Sizes (page 932) set to true is a no-op. To disable the *power of 2 allocation* for a collection, use the collMod (page 457) command with the noPadding (page 458) flag or the db.createCollection() (page 167) method with the noPadding option.

Default: true.

Available for mongod (page 770) only.

Available for the MMAPv1 storage engine only.

notablescan

Available for mongod (page 770) only.

Specify whether **all** queries must use indexes. If 1, MongoDB will not execute queries that require a table scan and will return an error.

Consider the following example which sets notablescan (page 932) to 1 or true:

```
db.getSiblingDB("admin").runCommand( { setParameter: 1, notablescan: 1 } )
```

Setting notablescan (page 932) to 1 can be useful for testing application queries, for example, to identify queries that scan an entire collection and cannot use an index.

To detect unindexed queries without notablescan, consider reading the https://docs.mongodb.org/manual/tutorial/evaluate-operation-performance and https://docs.mongodb.org/manual/tutorial/optimize-query-performance-with-indexes-and-p
sections and using the logLevel (page 933) parameter, *mongostat* (page 857) and *profiling*.

Don't run production mongod (page 770) instances with notablescan (page 932) because preventing table scans can potentially affect queries in all databases, including administrative queries.

textSearchEnabled

Deprecated since version 2.6: MongoDB enables the text search feature by default. Manual enabling of this feature is unnecessary.

Available for both mongod (page 770) and mongos (page 792).

Enables the text search feature. When manually enabling, you must enable on **each and every** mongod (page 770) for replica sets.

ttlMonitorEnabled

New in version 2.4.6.

Available for mongod (page 770) only.

To support TTL Indexes, mongod (page 770) instances have a background thread that is responsible for deleting documents from collections with TTL indexes.

To disable this worker thread for a mongod (page 770), set ttlMonitorEnabled (page 933) to false, as in the following operations:

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, ttlMonitorEnabled: false } )
```

Alternately, you may disable the thread at startup time by starting the mongod (page 770) instance with the following option:

```
mongod --setParameter ttlMonitorEnabled=false
```

disableJavaScriptJIT

New in version 3.2.

Available for mongod (page 770) only.

The MongoDB JavaScript engine uses SpiderMonkey, which implements Just-in-Time (JIT) compilation for improved performance when running scripts.

To disable the JIT, set disableJavaScriptJIT (page 933) to true, as in the following example:

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, disableJavaScriptJIT: true } )
```

Be aware that group (page 313) and $where (page 558) will reuse existing JavaScript interpreter contexts, so changes to disableJavaScriptJIT (page 933) may not take effect immediately for these operations.

Alternately, you may disable the JIT at startup time by starting the mongod (page 770) instance with the following option:

```
mongod --setParameter disableJavaScriptJIT=true
```

Logging Parameters

logLevel

Available for both mongod (page 770) and mongos (page 792).

Specify an integer between 0 and 5 signifying the verbosity of the logging, where 5 is the most verbose.

Consider the following example which sets the logLevel (page 933) to 2:

```
use admin
db.runCommand( { setParameter: 1, logLevel: 2 } )
```

The default logLevel (page 933) is 0.

See also:

verbosity (page 897).

logComponentVerbosity

New in version 3.0.0.

Available for both mongod (page 770) and mongos (page 792).

Sets the verbosity levels of various *components* (page 964) for *log messages* (page 964). The verbosity level determines the amount of *Informational and Debug* (page 964) messages MongoDB outputs.

The verbosity level can range from 0 to 5:

- 0 is the MongoDB's default log verbosity level, to include *Informational* (page 964) messages.
- 1 to 5 increases the verbosity level to include *Debug* (page 964) messages.

For a component, you can also specify -1 to inherit the parent's verbosity level.

To specify the verbosity level, use a document similar to the following:

```
{
  verbosity: <int>,
  <component1>: { verbosity: <int> },
  <component2>: {
     verbosity: <int>,
     <component3>: { verbosity: <int> }
  },
  ...
}
```

For the components, you can specify just the `<component>:  <int>` in the document, unless you are setting both the parent verbosity level and that of the child component(s) as well:

```
{
  verbosity: <int>,
  <component1>: <int> ,
  <component2>: {
     verbosity: <int>,
     <component3>: <int>
  }
  ...
}
```

The top-level verbosity field corresponds to systemLog.verbosity (page 897) which sets the default level for all components. The default value of systemLog.verbosity (page 897) is 0.

The components correspond to the following settings:

- accessControl (page 899)
- command (page 899)
- control (page 899)
- geo (page 899)
- index (page 900)
- network (page 900)
- query (page 900)
- replication (page 900)
- sharding (page 900)

- `storage` (page 901)
- `storage.journal` (page 901)
- `write` (page 901)

Unless explicitly set, the component has the verbosity level of its parent. For example, `storage` is the parent of `storage.journal`. That is, if you specify a `storage` (page 901) verbosity level, this level also applies to `storage.journal` (page 901) components *unless* you specify the verbosity level for `storage.journal` (page 901).

For example, the following sets the `default verbosity level` (page 897) to `1`, the `query` (page 900) to `2`, the `storage` (page 901) to `2`, and the `storage.journal` (page 901) to `1`.

```
use admin
db.runCommand( {
   setParameter: 1,
   logComponentVerbosity: {
      verbosity: 1,
      query: { verbosity: 2 },
      storage: {
         verbosity: 2,
         journal: {
            verbosity: 1
         }
      }
   }
} )
```

You can also set parameter `logComponentVerbosity` (page 934) at startup time, passing the verbosity level document as a string.

`mongo` (page 803) shell also provides the `db.setLogLevel()` (page 197) to set the log level for a single component. For various ways to set the log verbosity level, see *Configure Log Verbosity Levels* (page 966).

`logUserIds`

New in version 2.4.

Available for both `mongod` (page 770) and `mongos` (page 792).

Specify `1` to enable logging of userids.

Disabled by default.

`quiet`

Available for both `mongod` (page 770) and `mongos` (page 792).

Sets quiet logging mode. If `1`, `mongod` (page 770) will go into a quiet logging mode which will not log the following events/activities:

- connection events;
- the `drop` (page 439) command, the `dropIndexes` (page 450) command, the `diagLogging` (page 483) command, the `validate` (page 485) command, and the `clean` (page 453) command; and
- replication synchronization activities.

Consider the following example which sets the `quiet` to `1`:

```
db = db.getSiblingDB("admin")
db.runCommand( { setParameter: 1, quiet: 1 } )
```

See also:

quiet (page 897)

traceExceptions

Available for both mongod (page 770) and mongos (page 792).

Configures mongod (page 770) to log full source code stack traces for every database and socket C++ exception, for use with debugging. If true, mongod (page 770) will log full stack traces.

Consider the following example which sets the traceExceptions to true:

```
db.getSiblingDB("admin").runCommand( { setParameter: 1, traceExceptions: true } )
```

See also:

systemLog.traceAllExceptions (page 897)

6.2.3 Diagnostic Parameters

To support the *diagnostic data capture* (page 999), MongoDB introduces the following parameters:

Note: The default values for the diagnostic data capture interval and the maximum sizes are chosen to provide useful data to MongoDB engineers with minimal impact on performance and storage size. Typically, these values will only need modifications as requested by MongoDB engineers for specific diagnostic purposes.

Diagnostic data files are stored in the diagnostic.data directory under the mongod (page 770) instance's --dbpath or storage.dbPath (page 915).

diagnosticDataCollectionEnabled

New in version 3.2.

Type: boolean

Default: true

Available for mongod (page 770) only.

Determines whether to enable the collecting and logging of data for diagnostic purposes. Diagnostic logging is enabled by default.

For example, the following disables the diagnostic collection:

```
mongod --setParameter diagnosticDataCollectionEnabled=false
```

diagnosticDataCollectionDirectorySizeMB

New in version 3.2.

Type: integer

Default: 100

Available for mongod (page 770) only.

Specifies the maximum size, in megabytes, of the diagnostic.data directory . If directory size exceeds this number, the oldest *diagnostic files in the directory* (page 999) are automatically deleted based on the timestamp in the file name.

For example, the following sets the maximum size of the directory to 200 megabytes:

```
mongod --setParameter diagnosticDataCollectionDirectorySizeMB=200
```

The minimum value for `diagnosticDataCollectionDirectorySizeMB` (page 936) is `10` megabytes. `diagnosticDataCollectionDirectorySizeMB` (page 936) must be greater than maximum diagnostic file size `diagnosticDataCollectionFileSizeMB` (page 937).

diagnosticDataCollectionFileSizeMB

New in version 3.2.

Type: integer

Default: 10

Available for `mongod` (page 770) only.

Specifies the maximum size, in megabytes, of each *diagnostic file* (page 999). If the file exceeds the maximum file size, MongoDB creates a new file.

For example, the following sets the maximum size of each diagnostic file to `20` megabytes:

```
mongod --setParameter diagnosticDataCollectionFileSizeMB=20
```

The minimum value for `diagnosticDataCollectionFileSizeMB` (page 937) is `1` megabyte.

diagnosticDataCollectionPeriodMillis

New in version 3.2.

Type: integer

Default: 1000

Available for `mongod` (page 770) only.

Specifies the interval, in milliseconds, at which to collect diagnostic data.

For example, the following sets the interval to `5000` milliseconds or 5 seconds:

```
mongod --setParameter diagnosticDataCollectionPeriodMillis=5000
```

The minimum value for `diagnosticDataCollectionPeriodMillis` (page 937) is `100` milliseconds.

Replication Parameters

replApplyBatchSize

Available for `mongod` (page 770) only.

Specify the number of oplog entries to apply as a single batch. `replApplyBatchSize` (page 937) must be an integer between 1 and 1024. The default value is 1. This option only applies to master/slave configurations and is valid only on a `mongod` (page 770) started with the `--slave` command line option.

Batch sizes must be `1` for members with *`slavedelay`* configured.

replIndexPrefetch

Available for `mongod` (page 770) only.

Use `replIndexPrefetch` (page 937) in conjunction with `replSetName` (page 922) when configuring a replica set. The default value is `all` and available options are:

- `none`
- `all`
- `_id_only`

By default *secondary* members of a *replica set* will load all indexes related to an operation into memory before applying operations from the oplog. You can modify this behavior so that the secondaries will only load the `_id` index. Specify `_id_only` or `none` to prevent the `mongod` (page 770) from loading *any* index into memory.

replWriterThreadCount

New in version 3.2.

Type: integer

Default: 16

Available for mongod (page 770) only.

Number of threads to use to apply replicated operations in parallel. Values can range from 1 to 256 inclusive. You can only set replWriterThreadCount (page 938) at startup and cannot change this setting with the setParameter (page 460) command.

Sharding Parameters

recoverShardingState

Available for mongod (page 770) only.

Specify a boolean to check or ignore sharding state recovery information. Default is `true` to check the sharding state recovery information.

Storage Parameters

journalCommitInterval

Available for mongod (page 770) only.

Specify an integer between `1` and `500` signifying the number of milliseconds (ms) between journal commits.

Consider the following example which sets the journalCommitInterval (page 938) to `200` ms:

```
db.getSiblingDB("admin").runCommand( { setParameter: 1, journalCommitInterval: 200 } )
```

See also:

storage.journal.commitIntervalMs (page 916)

syncdelay

Available for mongod (page 770) only.

Specify the interval in seconds between *fsync* operations where mongod (page 770) flushes its working memory to disk. By default, mongod (page 770) flushes memory to disk every 60 seconds. In almost every situation you should not set this value and use the default setting.

Consider the following example which sets the `syncdelay` to `60` seconds:

```
db.getSiblingDB("admin").runCommand( { setParameter: 1, syncdelay: 60 } )
```

See also:

syncPeriodSecs (page 916) and journalCommitInterval (page 938).

WiredTiger Parameters

wiredTigerConcurrentReadTransactions

New in version 3.0.0.

Available for mongod (page 770) only.

Available for the WiredTiger storage engine only.

Specify the maximum number of concurrent read transactions allowed into the WiredTiger storage engine.

```
db.adminCommand( { setParameter: 1, wiredTigerConcurrentReadTransactions: <num> } )
```

See also:

`wiredTiger.concurrentTransactions` (page 509)

`wiredTigerConcurrentWriteTransactions`

New in version 3.0.0.

Available for `mongod` (page 770) only.

Available for the WiredTiger storage engine only.

Specify the maximum number of concurrent write transactions allowed into the WiredTiger storage engine.

```
db.adminCommand( { setParameter: 1, wiredTigerConcurrentWriteTransactions: <num> } )
```

See also:

`wiredTiger.concurrentTransactions` (page 509)

`wiredTigerEngineRuntimeConfig`

New in version 3.0.0.

Available for `mongod` (page 770) only.

Specify `wiredTiger` storage engine configuration options for a running `mongod` (page 770) instance. You can *only* set this parameter using the `setParameter` (page 460) command and *not* using the command line or configuration file option.

Consider the following operation prototype:

```
db.adminCommand({
   "setParameter": 1,
   "wiredTigerEngineRuntimeConfig": "<option>=<setting>,<option>=<setting>"
})
```

See the WiredTiger documentation for all available WiredTiger configuration options[13].

Auditing Parameters

`auditAuthorizationSuccess`

New in version 2.6.5.

Default: `false`

Note: Available only in MongoDB Enterprise[14].

Available for both `mongod` (page 770) and `mongos` (page 792).

Enables the `auditing` of authorization successes for the *authCheck* action.

When `auditAuthorizationSuccess` (page 939) is `false`, the `audit system` only logs the authorization failures for `authCheck`.

To enable the audit of authorization successes, issue the following command:

```
db.adminCommand( { setParameter: 1, auditAuthorizationSuccess: true } )
```

[13] http://source.wiredtiger.com/2.4.1/struct_w_t___c_o_n_n_e_c_t_i_o_n.html#a579141678af06217b22869cbc604c6d4
[14] http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

Enabling `auditAuthorizationSuccess` (page 939) degrades performance more than logging only the authorization failures.

6.3 MongoDB Limits and Thresholds

On this page

- BSON Documents (page 940)
- Namespaces (page 940)
- Indexes (page 941)
- Data (page 942)
- Replica Sets (page 943)
- Sharded Clusters (page 943)
- Operations (page 945)
- Naming Restrictions (page 946)

This document provides a collection of hard and soft limitations of the MongoDB system.

6.3.1 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 878) 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*.

6.3.2 Namespaces

`Namespace Length`

The maximum length of the collection namespace, which includes the database name, the dot (`.`) separator, and the collection name (i.e. `<database>.<collection>`), is 120 bytes.

`Number of Namespaces`

Changed in version 3.0.

For the MMAPv1 the number of namespaces is limited to 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.

The WiredTiger storage engine is *not* subject to this limitation.

`Size of Namespace File`

Changed in version 3.0.

For the MMAPv1 storage engine, namespace files can be no larger than 2047 megabytes.

By default namespace files are 16 megabytes. You can configure the size using the `nsSize` (page 917) option.

The WiredTiger storage engine is *not* subject to this limitation.

6.3.3 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 versions 2.6 and greater implement a stronger enforcement of the limit on `index key` (page 941):

- MongoDB will **not** `create an index` (page 36) on a collection if the index entry for an existing document exceeds the `index key limit` (page 941). 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 941). Reindexing operations occur as part of `compact` (page 454) and `repairDatabase` (page 462) commands as well as the `db.collection.reIndex()` (page 97) method.

  Because these operations drop *all* the indexes from a collection and then recreate them sequentially, the error from the `index key limit` (page 941) prevents these operations from rebuilding any remaining indexes for the collection and, in the case of the `repairDatabase` (page 462) 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 941), 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 941).

  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 824) and `mongoimport` (page 841) will not insert documents that contain an indexed field whose corresponding index entry would exceed the `index key limit` (page 941).
- 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 941) 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 941) 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 941).
- 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 941).

`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 `createIndex()` (page 36) 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 `$text` (page 549) query, which requires a special *text index*, with a query operator that requires a different type of special index. For example you cannot combine `$text` (page 549) query with the `$near` (page 565) 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 943).

`NaN values returned from Covered Queries by the WiredTiger Storage Engine are always of typ`
If the value of a field returned from a query that is *covered by an index* is `NaN`, the type of that `NaN` value is *always* `double`.

`Multikey Index`
A *multikey index* cannot support a *covered query*.

6.3.4 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 439), 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.

`Database Size`
The MMAPv1 storage engine limits each database to no more than 16000 data files. This means that a single MMAPv1 database has a maximum size of 32TB. Setting the `storage.mmapv1.smallFiles` (page 918) option reduces this limit to 8TB.

`Data Size`
Changed in version 3.0.

Using the MMAPv1 storage engine, a single `mongod` (page 770) 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

The WiredTiger storage engine is *not* subject to this limitation.

`Number of Collections in a Database`
Changed in version 3.0.

For the MMAPv1 storage engine, 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.

The WiredTiger storage engine is *not* subject to this limitation.

See `Number of Namespaces` (page 940) for more information.

6.3.5 Replica Sets

`Number of Members of a Replica Set`

Changed in version 3.0.0.

Replica sets can have up to 50 members. See *Increased Number of Replica Set Members* (page 1043) for more information about specific driver compatibility with large replica sets.

`Number of Voting Members of a Replica Set`

Replica sets can have up to 7 voting members. For replica sets with more than 7 total members, see *replica-set-non-voting-members*.

`Maximum Size of Auto-Created Oplog`

Changed in version 2.6.

If you do not explicitly specify an oplog size (i.e. with `oplogSizeMB` (page 921) or `--oplogSize`) MongoDB will create an oplog that is no larger than 50 gigabytes.

6.3.6 Sharded Clusters

Sharded clusters have the restrictions and thresholds described here.

Sharding Operational Restrictions

`Operations Unavailable in Sharded Environments`

The `group` (page 313) does not work with sharding. Use `mapReduce` (page 318) or `aggregate` (page 303) instead.

`db.eval()` (page 178) is incompatible with sharded collections. You may use `db.eval()` (page 178) with un-sharded collections in a shard cluster.

`$where` (page 558) does not permit references to the `db` object from the `$where` (page 558) function. This is uncommon in un-sharded collections.

The `$isolated` (page 629) update modifier does not work in sharded environments.

`$snapshot` queries do not work in sharded environments.

The `geoSearch` (page 332) command is not supported in sharded environments.

`Covered Queries in Sharded Clusters`

An index cannot *cover* a query on a *sharded* collection when run against a `mongos` (page 792) if the index does not contain the shard key, with the following exception for the `_id` index: If a query on a sharded collection only specifies a condition on the `_id` field and returns only the `_id` field, the `_id` index can cover the query when run against a `mongos` (page 792) even if the `_id` field is not the shard key.

Changed in version 3.0: In previous versions, an index cannot *cover* a query on a *sharded* collection when run against a `mongos` (page 792).

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
Max Collection Size (256 MB Chunk Size)	7.6 TB	21.2 TB
Max Collection Size (512 MB Chunk Size)	15.2 TB	42.4 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 117) and `remove()` (page 101) 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 117) and `remove()` (page 101) 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

`https://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 117) 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.

6.3.7 Operations

`Sort Operations`

If MongoDB cannot use an index to get documents in the requested sort order, the combined size of all documents in the sort operation, plus a small overhead, must be 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:

$sort and Memory Restrictions (page 651) and *$group Operator and Memory* (page 645).

`2d Geospatial queries cannot use the $or operator`

See

`$or` (page 534) and `https://docs.mongodb.org/manual/core/geospatial-indexes`.

`Area of GeoJSON Polygons`

For `$geoIntersects` (page 562) or `$geoWithin` (page 560), 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 $geometry` (page 569) expression; otherwise, `$geoIntersects` (page 562) or `$geoWithin` (page 560) queries for the complementary geometry. For all other GeoJSON polygons with areas greater than a hemisphere, `$geoIntersects` (page 562) or `$geoWithin` (page 560) queries for the complementary geometry.

Write Command Operation Limit Size

Write commands (page 334) can accept no more than 1000 operations. The `Bulk()` (page 210) operations in the `mongo` (page 803) shell and comparable methods in the drivers do not have this limit.

6.3.8 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 1151).

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 803) shell, use `db.getCollection()` (page 181) 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.

6.4 Explain Results

On this page

- Explain Output (page 947)
- Sharded Collection (page 951)
- Compatibility Changes (page 952)

Changed in version 3.0.

MongoDB provides the `db.collection.explain()` (page 48) method, the `cursor.explain()` (page 140) method, and the `explain` (page 467) command to return information on query plans and execution statistics of the query plans.

The `explain` results present the query plans as a tree of stages. Each stage passes its results (i.e. documents or index keys) to the parent node. The leaf nodes access the collection or the indices. The internal nodes manipulate the documents or the index keys that result from the child nodes. The root node is the final stage from which MongoDB derives the result set.

Stages are descriptive of the operation; e.g.

- `COLLSCAN` for a collection scan
- `IXSCAN` for scanning index keys
- `FETCH` for retrieving documents
- `SHARD_MERGE` for merging results from shards

6.4.1 Explain Output

The following sections presents a list of some key fields returned by the `explain` operation.

Note:

- The list of fields is not meant to be exhaustive, but is meant to highlight some key field changes from earlier versions of explain.
- The output format is subject to change between releases.

queryPlanner

`queryPlanner` (page 948) information details the plan selected by the `query optimizer`.

For unsharded collections, `explain` returns the following information:

```
{
   "queryPlanner" : {
      "plannerVersion" : <int>,
      "namespace" : <string>,
      "indexFilterSet" : <boolean>,
      "parsedQuery" : {
         ...
      },
      "winningPlan" : {
         "stage" : <STAGE1>,
         ...
         "inputStage" : {
            "stage" : <STAGE2>,
            ...
            "inputStage" : {
               ...
            }
         }
      },
      "rejectedPlans" : [
         <candidate plan 1>,
         ...
      ]
   }
```

explain.**queryPlanner**

Contains information on the selection of the query plan by the `query optimizer`.

explain.queryPlanner.**namespace**

A string that specifies the namespace (i.e., `<database>.<collection>`) against which the query is run.

explain.queryPlanner.**indexFilterSet**

A boolean that specifies whether MongoDB applied an *index filter* for the *query shape*.

explain.queryPlanner.**winningPlan**

A document that details the plan selected by the `query optimizer`. MongoDB presents the plan as a tree of stages; i.e. a stage can have an `inputStage` (page 948) or, if the stage has multiple child stages, `inputStages` (page 948).

explain.queryPlanner.winningPlan.**stage**

A string that denotes the name of the stage.

Each stage consists of information specific to the stage. For instance, an `IXSCAN` stage will include the index bounds along with other data specific to the index scan. If a stage has a child stage or multiple child stages, the stage will have an inputStage or inputStages.

explain.queryPlanner.winningPlan.**inputStage**

A document that describes the child stage, which provides the documents or index keys to its parent. The field is present *if* the parent stage has only one child.

explain.queryPlanner.winningPlan.**inputStages**

An array of documents describing the child stages. Child stages provide the documents or index keys to the parent stage. The field is present *if* the parent stage has multiple child nodes. For example, stages for *$or expressions* (page 953) or *index intersection* (page 952) consume input from multiple sources.

explain.queryPlanner.**rejectedPlans**

Array of candidate plans considered and rejected by the query optimizer. The array can be empty if there were no other candidate plans.

For sharded collections, the winning plan includes the `shards` (page 951) array which contains the plan information for each accessed shard. For details, see *Sharded Collection* (page 951).

executionStats

The returned `executionStats` (page 949) information details the execution of the winning plan.

You must run the explain in *executionStats* (page 49) or *allPlansExecution* (page 49) verbosity mode in order for `executionStats` to be present in the results.

To include partial execution data captured during *plan selection*, you must run in `allPlansExecution` mode.

For unsharded collections, `explain` returns the following information:

```
"executionStats" : {
   "executionSuccess" : <boolean>,
   "nReturned" : <int>,
   "executionTimeMillis" : <int>,
   "totalKeysExamined" : <int>,
   "totalDocsExamined" : <int>,
   "executionStages" : {
      "stage" : <STAGE1>
      "nReturned" : <int>,
      "executionTimeMillisEstimate" : <int>,
      "works" : <int>,
      "advanced" : <int>,
      "needTime" : <int>,
      "needYield" : <int>,
      "isEOF" : <boolean>,
      ...
      "inputStage" : {
         "stage" : <STAGE2>,
         ...
         "nReturned" : <int>,
         "executionTimeMillisEstimate" : <int>,
         "keysExamined" : <int>,
         "docsExamined" : <int>,
         ...
         "inputStage" : {
            ...
         }
      }
   },
   "allPlansExecution" : [
      { <partial executionStats1> },
      { <partial executionStats2> },
      ...
   ]
}
```

explain.**executionStats**

Contains statistics that describe the completed query execution for the winning plan. For write operations, completed query execution refers to the modifications that *would* be performed, but does *not* apply the modifications to the database.

explain.executionStats.**nReturned**

Number of documents that match the query condition. nReturned corresponds to the n field returned by cursor.explain() in earlier versions of MongoDB.

explain.executionStats.**executionTimeMillis**

Total time in milliseconds required for query plan selection and query execution. executionTimeMillis corresponds to the millis field returned by cursor.explain() in earlier versions of MongoDB.

explain.executionStats.**totalKeysExamined**

Number of index entries scanned. totalKeysExamined (page 949) corresponds to the nscanned field returned by cursor.explain() in earlier versions of MongoDB.

explain.executionStats.**totalDocsExamined**

Number of documents scanned. In earlier versions of MongoDB, totalDocsExamined (page 949) corresponds to the nscannedObjects field returned by cursor.explain() in earlier versions of MongoDB.

explain.executionStats.**executionStages**

Details the completed execution of the winning plan as a tree of stages; i.e. a stage can have an inputStage or multiple inputStages.

Each stage consists of execution information specific to the stage.

explain.executionStats.executionStages.**works**

Specifies the number of "work units" performed by the query execution stage. Query execution divides its work into small units. A "work unit" might consist of examining a single index key, fetching a single document from the collection, applying a projection to a single document, or doing a piece of internal bookkeeping.

`explain.executionStats.executionStages.`**`advanced`**

The number of intermediate results returned, or *advanced*, by this stage to its parent stage.

`explain.executionStats.executionStages.`**`needTime`**

The number of work cycles that did not advance an intermediate result to its parent stage (see `explain.executionStats.executionStages.advanced` (page 950)). For instance, an index scan stage may spend a work cycle seeking to a new position in the index as opposed to returning an index key; this work cycle would count towards `explain.executionStats.executionStages.needTime` (page 950) rather than `explain.executionStats.executionStages.advanced` (page 950).

`explain.executionStats.executionStages.`**`needYield`**

The number of times that the storage layer requested that the query system yield its locks.

`explain.executionStats.executionStages.`**`isEOF`**

Specifies whether the execution stage has reached end of stream:

- If `true` or `1`, the execution stage has reached end-of-stream.
- If `false` or `0`, the stage may still have results to return. For example, consider a query with a limit whose execution stages consists of a `LIMIT` stage with an input stage of `IXSCAN` for the query. If the query returns more than the specified limit, the `LIMIT` stage will report `isEOF: 1`, but its underlying `IXSCAN` stage will report `isEOF: 0`.

`explain.executionStats.executionStages.inputStage.`**`keysExamined`**

For query execution stages that scan an index (e.g. IXSCAN), `keysExamined` is the total number of in-bounds and out-of-bounds keys that are examined in the process of the index scan. If the index scan consists of a single contiguous range of keys, only in-bounds keys need to be examined. If the index bounds consists of several key ranges, the index scan execution process may examine out-of-bounds keys in order to skip from the end of one range to the beginning of the next.

Consider the following example, where there is an index of field `x` and the collection contains 100 documents with `x` values 1 through 100:

```
db.keys.find( { x : $in : [ 3, 4, 50, 74, 75, 90 ] } ).explain( "executionStats" )
```

The query will scan keys `3` and `4`. It will then scan the key `5`, detect that it is out-of-bounds, and skip to the next key `50`.

Continuing this process, the query scans keys 3, 4, 5, 50, 51, 74, 75, 76, 90, and 91. Keys `5`, `51`, `76`, and `91` are out-of-bounds keys that are still examined. The value of `keysExamined` is 10.

`explain.executionStats.executionStages.inputStage.`**`docsExamined`**

Specifies the number of documents scanned during the query execution stage.

Present for the `COLLSCAN` stage, as well as for stages that retrieve documents from the collection (e.g. `FETCH`)

`explain.executionStats.`**`allPlansExecution`**

Contains *partial* execution information captured during the *plan selection phase* for both the winning and rejected plans. The field is present only if `explain` runs in `allPlansExecution` verbosity mode.

For sharded collections, `explain` also includes the execution statistics for each accessed shard. For details, see *Sharded Collection* (page 951).

serverInfo

For unsharded collections, `explain` returns the following information for the MongoDB instance:

```
"serverInfo" : {
   "host" : <string>,
   "port" : <int>,
   "version" : <string>,
   "gitVersion" : <string>
}
```

For sharded collections, `explain` returns the `serverInfo` for each accessed shard. For details, see *Sharded Collection* (page 951).

6.4.2 Sharded Collection

For sharded collections, `explain` returns the core query planner and server information for each accessed shard in the `shards` field:

```
{
   "queryPlanner" : {
      ...
      "winningPlan" : {
         ...
         "shards" : [
            {
              "shardName" : <shard>,
              <queryPlanner information for shard>,
              <serverInfo for shard>
            },
            ...
         ],
      },
   },
   "executionStats" : {
      ...
      "executionStages" : {
         ...
         "shards" : [
            {
               "shardName" : <shard>,
               <executionStats for shard>
            },
            ...
         ]
      },
      "allPlansExecution" : [
         {
            "shardName" : <string>,
            "allPlans" : [ ... ]
         },
         ...
      ]
   }
}
```

`explain.queryPlanner.winningPlan.`**`shards`**
    Array of documents that contain `queryPlanner` (page 948) and `serverInfo` for each accessed shard.

explain.executionStats.executionStages.**shards**

Array of documents that contain executionStats (page 949) for each accessed shard.

6.4.3 Compatibility Changes

Changed in version 3.0.

The format and fields of the explain results have changed from previous versions. The following lists some key differences.

Collection Scan vs. Index Use

If the query planner selects a collection scan, the explain result includes a COLLSCAN stage.

If the query planner selects an index, the explain result includes a IXSCAN stage. The stage includes information such as the index key pattern, direction of traversal, and index bounds.

In previous versions of MongoDB, cursor.explain() returned the cursor field with the value of:

- BasicCursor for collection scans, and
- BtreeCursor <index name> [<direction>] for index scans.

For more information on execution statistics of collection scans versus index scans, see https://docs.mongodb.org/manual/tutorial/analyze-query-plan.

Covered Queries

When an index covers a query, MongoDB can both match the query conditions **and** return the results using only the index keys; i.e. MongoDB does not need to examine documents from the collection to return the results.

When an index covers a query, the explain result has an IXSCAN stage that is **not** a descendant of a FETCH stage, and in the *executionStats* (page 948), the totalDocsExamined is 0.

In earlier versions of MongoDB, cursor.explain() returned the indexOnly field to indicate whether the index covered a query.

Index Intersection

For an index intersection plan, the result will include either an AND_SORTED stage or an AND_HASH stage with an inputStages (page 948) array that details the indexes; e.g.:

```
{
   "stage" : "AND_SORTED",
   "inputStages" : [
      {
         "stage" : "IXSCAN",
         ...
      },
      {
         "stage" : "IXSCAN",
         ...
      }
   ]
}
```

In previous versions of MongoDB, `cursor.explain()` returned the `cursor` field with the value of `Complex Plan` for index intersections.

`$or` Expression

If MongoDB uses indexes for an `$or` (page 534) expression, the result will include the `OR` stage with an `inputStages` array that details the indexes; e.g.:

```
{
   "stage" : "OR",
   "inputStages" : [
      {
         "stage" : "IXSCAN",
         ...
      },
      {
         "stage" : "IXSCAN",
         ...
      },
      ...
   ]
}
```

In previous versions of MongoDB, `cursor.explain()` returned the `clauses` array that detailed the indexes.

Sort Stage

If MongoDB can use an index scan to obtain the requested sort order, the result will **not** include a `SORT` stage. Otherwise, if MongoDB cannot use the index to sort, the `explain` result will include a `SORT` stage.

In previous versions of MongoDB, `cursor.explain()` returned the `scanAndOrder` field to specify whether MongoDB could use the index order to return sorted results.

6.5 Connection String URI Format

On this page

- Standard Connection String Format (page 953)
- Connection String Options (page 954)
- Examples (page 958)

This document describes the URI format for defining connections between applications and MongoDB instances in the official MongoDB `drivers`.

6.5.1 Standard Connection String Format

This section describes the standard format of the MongoDB connection URI used to connect to a MongoDB database server. The format is the same for all official MongoDB drivers. For a list of drivers and links to driver documentation, see `https://docs.mongodb.org/manual/applications/drivers`.

The following is the standard URI connection scheme:

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options
```

The components of this string are:

1. `mongodb://`

   A required prefix to identify that this is a string in the standard connection format.

2. `username:password@`

   Optional. If specified, the client will attempt to log in to the specific database using these credentials after connecting to the `mongod` (page 770) instance.

3. `host1`

   This the only required part of the URI. It identifies a server address to connect to. It identifies either a hostname, IP address, or UNIX domain socket.

4. `:port1`

   Optional. The default value is `:27017` if not specified.

5. `hostX`

   Optional. You can specify as many hosts as necessary. You would specify multiple hosts, for example, for connections to replica sets.

6. `:portX`

   Optional. The default value is `:27017` if not specified.

7. `/database`

   Optional. The name of the database to authenticate if the connection string includes authentication credentials in the form of `username:password@`. If `/database` is not specified and the connection string includes credentials, the driver will authenticate to the `admin` database.

8. `?options`

   Connection specific options. See *Connection String Options* (page 954) for a full description of these options.

   If the connection string does not specify a database/ you must specify a slash (i.e. `https://docs.mongodb.org/manual/`) between the last `hostN` and the question mark that begins the string of options.

Example

To describe a connection to a replica set named `test`, with the following `mongod` (page 770) hosts:

- `db1.example.net` on port `27017` and
- `db2.example.net` on port `2500`.

You would use a connection string that resembles the following:

```
mongodb://db1.example.net,db2.example.net:2500/?replicaSet=test
```

6.5.2 Connection String Options

This section lists all connection options used in the *Standard Connection String Format* (page 953).

Connection options are pairs in the following form: `name=value`. The `value` is always case sensitive. Separate options with the ampersand (i.e. `&`) character. In the following example, a connection uses the `replicaSet` and `connectTimeoutMS` options:

```
mongodb://db1.example.net,db2.example.net:2500/?replicaSet=test&connectTimeoutMS=300000
```

Semi-colon separator for connection string arguments

To provide backwards compatibility, drivers currently accept semi-colons (i.e. `;`) as option separators.

Replica Set Option

`uri.`**`replicaSet`**

Specifies the name of the *replica set*, if the `mongod` (page 770) is a member of a replica set.

When connecting to a replica set it is important to give a seed list of at least two `mongod` (page 770) instances. If you only provide the connection point of a single `mongod` (page 770) instance, and omit the `replicaSet` (page 955), the client will create a *standalone* connection.

Connection Options

`uri.`**`ssl`**

`true`: Initiate the connection with TLS/SSL.

`false`: Initiate the connection without TLS/SSL.

The default value is `false`.

Note: The `ssl` (page 955) option is not supported by all drivers. See your `driver` documentation and the `https://docs.mongodb.org/manual/tutorial/configure-ssl` document.

`uri.`**`connectTimeoutMS`**

The time in milliseconds to attempt a connection before timing out. The default is never to timeout, though different drivers might vary. See the `driver` documentation.

`uri.`**`socketTimeoutMS`**

The time in milliseconds to attempt a send or receive on a socket before the attempt times out. The default is never to timeout, though different drivers might vary. See the `driver` documentation.

Connection Pool Options

Most drivers implement some kind of connection pooling handle this for you behind the scenes. Some drivers do not support connection pools. See your `driver` documentation for more information on the connection pooling implementation. These options allow applications to configure the connection pool when connecting to the MongoDB deployment.

`uri.`**`maxPoolSize`**

The maximum number of connections in the connection pool. The default value is `100`.

`uri.`**`minPoolSize`**

The minimum number of connections in the connection pool. The default value is `0`.

Note: The `minPoolSize` (page 955) option is not supported by all drivers. For information on your driver, see the `drivers` documentation.

`uri.`**`maxIdleTimeMS`**

The maximum number of milliseconds that a connection can remain idle in the pool before being removed and closed.

This option is not supported by all drivers.

`uri.`**`waitQueueMultiple`**

A number that the driver multiples the `maxPoolSize` (page 955) value to, to provide the maximum number of threads allowed to wait for a connection to become available from the pool. For default values, see the `https://docs.mongodb.org/manual/applications/drivers` documentation.

`uri.`**`waitQueueTimeoutMS`**

The maximum time in milliseconds that a thread can wait for a connection to become available. For default values, see the `https://docs.mongodb.org/manual/applications/drivers` documentation.

Write Concern Options

Write concern describes the kind of assurances that the `mongod` (page 770) and the driver provide to the application regarding the success and durability of the write operation. For a full explanation of write concern and write operations in general, see `https://docs.mongodb.org/manual/reference/write-concern`.

Note: You can specify the write concern both in the connection string and as a parameter to method calls like `insert` or `update`. If the write concern is specified in both places, the method parameter overrides the connection-string setting.

`uri.`**`w`**

Corresponds to the write concern *wc-w*. The `w` option requests acknowledgement that the write operation has propagated to a specified number of `mongod` (page 770) instances or to `mongod` (page 770) instances with specified tags.

You can specify a `number`, the string `majority`, or a `tag set`.

For details, see *wc-w*.

`uri.`**`wtimeoutMS`**

Corresponds to the write concern *wc-wtimeout*. `wtimeoutMS` (page 956) specifies a time limit, in milliseconds, for the write concern.

When `wtimeoutMS` is `0`, write operations will never time out. For more information, see *wc-wtimeout*.

`uri.`**`journal`**

Corresponds to the write concern *wc-j* option. The `journal` (page 956) option requests acknowledgement from MongoDB that the write operation has been written to the `journal`. For details, see *wc-j*.

If you set `journal` (page 956) to `true`, and specify a `w` (page 956) value less than 1, `journal` (page 956) prevails.

Changed in version 2.6: If you set `journal` (page 956) to true, and the `mongod` (page 770) does not have journaling enabled, as with `storage.journal.enabled` (page 915), then MongoDB will error. In previous versions, MongoDB would provide basic receipt acknowledgment (i.e. `w:1`), ignore `journal` (page 956), and include a `jnote` field in its return document.

`readConcern` Options

New in version 3.2: For the WiredTiger storage engine, MongoDB 3.2 introduces the readConcern option for replica sets and replica set shards.

`https://docs.mongodb.org/manual/reference/read-concern` allows clients to choose a level of isolation for their reads from replica sets.

`uri.`**`readConcernLevel`**

The level of isolation. Accepts either `"local"` or `"majority"`.

For details, see `https://docs.mongodb.org/manual/reference/read-concern`.

Read Preference Options

`Read preferences` describe the behavior of read operations with regards to *replica sets*. These parameters allow you to specify read preferences on a per-connection basis in the connection string:

`uri.`**`readPreference`**

Specifies the *replica set* read preference for this connection. The read preference values are the following:

- `primary`
- `primaryPreferred`
- `secondary`
- `secondaryPreferred`
- `nearest`

For descriptions of each value, see *replica-set-read-preference-modes*.

The default value is `primary`, which sends all read operations to the replica set's *primary*.

`uri.`**`readPreferenceTags`**

Specifies a tag set as a comma-separated list of colon-separated key-value pairs. For example:

```
dc:ny,rack:1
```

To specify a *list* of tag sets, use multiple `readPreferenceTags`. The following specifies two tag sets and an empty tag set:

```
readPreferenceTags=dc:ny,rack:1&readPreferenceTags=dc:ny&readPreferenceTags=
```

Order matters when using multiple `readPreferenceTags`.

Authentication Options

`uri.`**`authSource`**

New in version 2.4.

Specify the database name associated with the user's credentials, if the users collection do not exist in the database where the client is connecting. `authSource` (page 957) defaults to the database specified in the connection string.

For authentication mechanisms that delegate credential storage to other services, the `authSource` (page 957) value should be `$external` as with the `PLAIN` (LDAP) and `GSSAPI` (Kerberos) authentication mechanisms.

MongoDB will ignore `authSource` (page 957) values if the connection string specifies no user name.

`uri.`**`authMechanism`**

New in version 2.4.

Changed in version 2.6: Added support for the `PLAIN` and `MONGODB-X509` authentication mechanisms.

Changed in version 3.0: Added support for the `SCRAM-SHA-1` authentication mechanism.

Specify the authentication mechanism that MongoDB will use to authenticate the connection. Possible values include:

- *SCRAM-SHA-1*
- *MONGODB-CR*
- *MONGODB-X509*
- *GSSAPI* (Kerberos)
- *PLAIN* (LDAP SASL)

Only MongoDB Enterprise `mongod` (page 770) and `mongos` (page 792) instances provide `GSSAPI` (Kerberos) and `PLAIN` (LDAP) mechanisms. To use `MONGODB-X509`, you must have TLS/SSL Enabled.

See `https://docs.mongodb.org/manual/core/authentication` for more information about the authentication system in MongoDB. Also consider `https://docs.mongodb.org/manual/tutorial/configure-x509-client-authentication` for more information on x509 authentication.

`uri.`**`gssapiServiceName`**

Set the Kerberos service name when connecting to Kerberized MongoDB instances. This value must match the service name set on MongoDB instances.

`gssapiServiceName` (page 958) defaults to `mongodb` for all clients and for MongoDB instance. If you change `saslServiceName` (page 929) setting on a MongoDB instance, you will need to set `gssapiServiceName` (page 958) to the same value.

Miscellaneous Configuration

`uri.`**`uuidRepresentation`**

option standard The standard binary representation.

option csharpLegacy The default representation for the C# driver.

option javaLegacy The default representation for the Java driver.

option pythonLegacy The default representation for the Python driver.

For the default, see the `drivers` documentation for your driver.

Note: Not all drivers support the `uuidRepresentation` (page 958) option. For information on your driver, see the `drivers` documentation.

6.5.3 Examples

The following provide example URI strings for common connection targets.

Database Server Running Locally

The following connects to a database server running locally on the default port:

```
mongodb://localhost
```

`admin` Database

The following connects and logs in to the `admin` database as user `sysop` with the password `moon`:

```
mongodb://sysop:moon@localhost
```

`records` Database

The following connects and logs in to the `records` database as user `sysop` with the password `moon`:

```
mongodb://sysop:moon@localhost/records
```

UNIX Domain Socket

The following connects to a UNIX domain socket:

```
mongodb:///tmp/mongodb-27017.sock
```

Note: Not all drivers support UNIX domain sockets. For information on your driver, see the `drivers` documentation.

Replica Set with Members on Different Machines

The following connects to a *replica set* with two members, one on `db1.example.net` and the other on `db2.example.net`:

```
mongodb://db1.example.net,db2.example.com
```

Replica Set with Members on `localhost`

The following connects to a replica set with three members running on `localhost` on ports `27017`, `27018`, and `27019`:

```
mongodb://localhost,localhost:27018,localhost:27019
```

Replica Set with Read Distribution

The following connects to a replica set with three members and distributes reads to the *secondaries*:

```
mongodb://example1.com,example2.com,example3.com/?readPreference=secondary
```

Replica Set with a High Level of Write Concern

The following connects to a replica set with write concern configured to wait for replication to succeed on at least two members, with a two-second timeout.

```
mongodb://example1.com,example2.com,example3.com/?w=2&wtimeoutMS=2000
```

6.6 MongoDB Extended JSON

On this page

- Parsers and Supported Format (page 960)
- BSON Data Types and Associated Representations (page 961)

JSON can only represent a subset of the types supported by *BSON*. To preserve type information, MongoDB adds the following extensions to the JSON format:

- *Strict mode*. Strict mode representations of BSON types conform to the JSON RFC[15]. Any JSON parser can parse these strict mode representations as key/value pairs; however, only the MongoDB internal JSON parser recognizes the type information conveyed by the format.
- `mongo` *Shell mode*. The MongoDB internal JSON parser and the `mongo` (page 803) shell can parse this mode.

The representation used for the various data types depends on the context in which the JSON is parsed.

6.6.1 Parsers and Supported Format

Input in Strict Mode

The following can parse representations in strict mode *with* recognition of the type information.

- REST Interfaces[16]
- `mongoimport` (page 841)
- `--query` option of various MongoDB tools

Other JSON parsers, including `mongo` (page 803) shell and `db.eval()` (page 178), can parse strict mode representations as key/value pairs, but *without* recognition of the type information.

Input in `mongo` Shell Mode

The following can parse representations in `mongo` shell mode *with* recognition of the type information.

- REST Interfaces[17]
- `mongoimport` (page 841)
- `--query` option of various MongoDB tools
- `mongo` (page 803) shell

Output in Strict mode

`mongoexport` (page 850) and REST and HTTP Interfaces[18] output data in *Strict mode*.

[15] http://www.json.org
[16] https://docs.mongodb.org/ecosystem/tools/http-interfaces
[17] https://docs.mongodb.org/ecosystem/tools/http-interfaces
[18] https://docs.mongodb.org/ecosystem/tools/http-interfaces

Output in `mongo` Shell Mode

`bsondump` (page 833) outputs in `mongo` *Shell mode*.

6.6.2 BSON Data Types and Associated Representations

The following presents the BSON data types and the associated representations in *Strict mode* and `mongo` *Shell mode*.

Binary

`data_binary`

Strict Mode		mongo Shell Mode
`{ "$binary": "<bindata>",`	`"$type": "<t>" }`	`BinData ( <t>, <bindata> )`

- `<bindata>` is the base64 representation of a binary string.
- `<t>` is a representation of a single byte indicating the data type. In *Strict mode* it is a hexadecimal string, and in *Shell mode* it is an integer. See the extended bson documentation. http://bsonspec.org/spec.html

Date

`data_date`

Strict Mode		mongo Shell Mode
`{ "$date": "<date>" }`		`new Date ( <date> )`

In *Strict mode*, `<date>` is an ISO-8601 date format with a mandatory time zone field following the template `YYYY-MM-DDTHH:mm:ss.mmm<+/-Offset>`.

The MongoDB JSON parser currently does not support loading ISO-8601 strings representing dates prior to the *Unix epoch*. When formatting pre-epoch dates and dates past what your system's `time_t` type can hold, the following format is used:

```
{ "$date" : { "$numberLong" : "<dateAsMilliseconds>" } }
```

In *Shell mode*, `<date>` is the JSON representation of a 64-bit signed integer giving the number of milliseconds since epoch UTC.

Timestamp

`data_timestamp`

Strict Mode		mongo Shell Mode
`{ "$timestamp": { "t": <t`	`>, "i": <i> } }`	`Timestamp( <t>, <i> )`

- `<t>` is the JSON representation of a 32-bit unsigned integer for seconds since epoch.
- `<i>` is a 32-bit unsigned integer for the increment.

Regular Expression

`data_regex`

Strict Mode		mongo Shell Mode
`{ "$regex": "<sRegex>", "$options": "<sOptions>" }`		`/<jRegex>/<jOptions>`

- `<sRegex>` is a string of valid JSON characters.
- `<jRegex>` is a string that may contain valid JSON characters and unescaped double quote (`"`) characters, but may not contain unescaped forward slash (`https://docs.mongodb.org/manual/`) characters.
- `<sOptions>` is a string containing the regex options represented by the letters of the alphabet.
- `<jOptions>` is a string that may contain only the characters 'g', 'i', 'm' and 's' (added in v1.9). Because the `JavaScript` and `mongo Shell` representations support a limited range of options, any nonconforming options will be dropped when converting to this representation.

OID

`data_oid`

Strict Mode		mongo Shell Mode
`{ "$oid": "<id>" }`		`ObjectId( "<id>" )`

`<id>` is a 24-character hexadecimal string.

DB Reference

`data_ref`

Strict Mode		mongo Shell Mode
`{ "$ref": "<name>", "$id": "<id>" }`		`DBRef("<name>", "<id>")`

- `<name>` is a string of valid JSON characters.
- `<id>` is any valid extended JSON type.

Undefined Type

`data_undefined`

Strict Mode		mongo Shell Mode
`{ "$undefined": true }`		`undefined`

The representation for the JavaScript/BSON undefined type.

You *cannot* use `undefined` in query documents. Consider the following document inserted into the `people` collection:

```
db.people.insert( { name : "Sally", age : undefined } )
```

The following queries return an error:

```
db.people.find( { age : undefined } )
db.people.find( { age : { $gte : undefined } } )
```

However, you can query for undefined values using `$type` (page 540), as in:

```
db.people.find( { age : { $type : 6 } } )
```

This query returns all documents for which the `age` field has value `undefined`.

MinKey

`data_minkey`

Strict Mode		mongo Shell Mode
`{ "$minKey": 1 }`		`MinKey`

The representation of the MinKey BSON data type that compares lower than all other types. See *faq-dev-compare-order-for-BSON-types* for more information on comparison order for BSON types.

MaxKey

`data_maxkey`

Strict Mode		mongo Shell Mode
`{ "$maxKey": 1 }`		`MaxKey`

The representation of the MaxKey BSON data type that compares higher than all other types. See *faq-dev-compare-order-for-BSON-types* for more information on comparison order for BSON types.

NumberLong

New in version 2.6.

`data_numberlong`

Strict Mode		mongo Shell Mode
`{ "$numberLong": "<number>" }`		`NumberLong( "<number>" )`

`NumberLong` is a 64 bit signed integer. You must include quotation marks or it will be interpreted as a floating point number, resulting in a loss of accuracy.

For example, the following commands insert `9223372036854775807` as a `NumberLong` with and without quotation marks around the integer value:

```
db.json.insert( { longQuoted : NumberLong("9223372036854775807") } )
db.json.insert( { longUnQuoted : NumberLong(9223372036854775807) } )
```

When you retrieve the documents, the value of `longUnquoted` has changed, while `longQuoted` retains its accuracy:

```
db.json.find()
{ "_id" : ObjectId("54ee1f2d33335326d70987df"), "longQuoted" : NumberLong("9223372036854775807")
{ "_id" : ObjectId("54ee1f7433335326d70987e0"), "longUnquoted" : NumberLong("-922337203685477580
```

6.7 Log Messages

On this page

- Timestamp (page 964)
- Severity Levels (page 964)
- Components (page 964)
- Verbosity Levels (page 966)

Changed in version 3.0.

Starting in MongoDB 3.0, MongoDB includes the *severity level* (page 964) and the *component* (page 964) associated with each log message. The log messages have the form:

```
<timestamp> <severity> <component> [<context>] <message>
```

For example:

```
2014-11-03T18:28:32.450-0500 I NETWORK [initandlisten] waiting for connections on port 27017
```

6.7.1 Timestamp

The default format for the `<timestamp>` is `iso8601-local`. To modify the timestamp format, use the *--timeStampFormat* runtime option or the `systemLog.timeStampFormat` (page 898) setting.

6.7.2 Severity Levels

The following table lists the severity levels associated with each log message:

Level	Description
F	Fatal
E	Error
W	Warning
I	Informational, for Verbosity Level of 0
D	Debug, for All Verbosity Levels > 0

You can specify the verbosity level of various components to determine the amount of Informational and Debug messages MongoDB outputs.

To set verbosity levels, see *Configure Log Verbosity Levels* (page 966).

6.7.3 Components

Log messages now include components, providing functional categorization of the messages:

ACCESS
: Messages related to access control, such as authentication. To specify the log level for `ACCESS` (page 964) components, use the `systemLog.component.accessControl.verbosity` (page 899) setting.

COMMAND
: Messages related to *database commands* (page 303), such as `count` (page 307). To specify the log level for `COMMAND` (page 964) components, use the `systemLog.component.command.verbosity` (page 899) setting.

CONTROL
Messages related to control activities, such as initialization. To specify the log level for `CONTROL` (page 964) components, use the `systemLog.component.control.verbosity` (page 899) setting.

GEO
Messages related to the parsing of geospatial shapes, such as verifying the GeoJSON shapes. To specify the log level for `GEO` (page 965) components, set the `systemLog.component.geo.verbosity` (page 899) parameter.

INDEX
Messages related to indexing operations, such as creating indexes. To specify the log level for `INDEX` (page 965) components, set the `systemLog.component.index.verbosity` (page 900) parameter.

NETWORK
Messages related to network activities, such as accepting connections. To specify the log level for `NETWORK` (page 965) components, set the `systemLog.component.network.verbosity` (page 900) parameter.

QUERY
Messages related to queries, including query planner activities. To specify the log level for `QUERY` (page 965) components, set the `systemLog.component.query.verbosity` (page 900) parameter.

REPL
Messages related to replica sets, such as initial sync and heartbeats. To specify the log level for `REPL` (page 965) components, set the `systemLog.component.replication.verbosity` (page 900) parameter.

SHARDING
Messages related to sharding activities, such as the startup of the `mongos` (page 792). To specify the log level for `SHARDING` (page 965) components, use the `systemLog.component.sharding.verbosity` (page 900) setting.

STORAGE
Messages related to storage activities, such as processes involved in the `fsync` (page 451) command. To specify the log level for `STORAGE` (page 965) components, use the `systemLog.component.storage.verbosity` (page 901) setting.

MongoDB distinguishes `JOURNAL` (page 965) components from `STORAGE` (page 965) components; however, `STORAGE` (page 965) is the parent of `JOURNAL` (page 965). As such, if `systemLog.component.storage.journal.verbosity` (page 901) setting is unset, MongoDB uses the `STORAGE` (page 965) verbosity level for `JOURNAL` (page 965) components

JOURNAL
Messages related specifically to journaling activities. To specify the log level for `JOURNAL` (page 965) components, use the `systemLog.component.storage.journal.verbosity` (page 901) setting.

MongoDB distinguishes `JOURNAL` (page 965) components from `STORAGE` (page 965) components; however, `STORAGE` (page 965) is the parent of `JOURNAL` (page 965). As such, if `systemLog.component.storage.journal.verbosity` (page 901) is unset, MongoDB uses the `STORAGE` (page 965) verbosity level for the `JOURNAL` (page 965) components as well.

WRITE
Messages related to write operations, such as `update` (page 340) commands. To specify the log level for `WRITE` (page 965) components, use the `systemLog.component.write.verbosity` (page 901) setting.

-
Messages not associated with a named component. Unnamed components have the default log level specified in the `systemLog.verbosity` (page 897) setting. The `systemLog.verbosity` (page 897) setting is the default setting for both named and unnamed components.

6.7.4 Verbosity Levels

View Current Log Verbosity Level

To view the current verbosity levels, use the `db.getLogComponents()` (page 187) method.

Configure Log Verbosity Levels

You can configure the verbosity level using: the `systemLog.verbosity` (page 897) and `systemLog.component.<name>.verbosity` settings, the `logComponentVerbosity` (page 934) parameter; the `db.setLogLevel()` (page 197) method.

`systemLog` Verbosity Settings

To configure the default log level for all components, use the `systemLog.verbosity` (page 897) setting. To configure the level of specific components, use the `systemLog.component.<name>.verbosity` settings.

For example, the following configuration sets the `systemLog.verbosity` (page 897) to `1`, the `systemLog.component.query.verbosity` (page 900) to `2`, the `systemLog.component.storage.verbosity` (page 901) to `2`, and the `systemLog.component.storage.journal.verbosity` (page 901) to `1`:

```
systemLog:
   verbosity: 1
   component:
      query:
         verbosity: 2
      storage:
         verbosity: 2
         journal:
            verbosity: 1
```

All components not specified in the configuration have the `systemLog.verbosity` (page 897) of `1`.

`logComponentVerbosity` Parameter

To set the `logComponentVerbosity` (page 934) parameter, pass a document with the verbosity settings to change.

For example, the following sets the `default verbosity level` (page 897) to `1`, the `query` (page 900) to `2`, the `storage` (page 901) to `2`, and the `storage.journal` (page 901) to `1`.

```
use admin
db.runCommand( {
   setParameter: 1,
   logComponentVerbosity: {
      verbosity: 1,
      query: {
         verbosity: 2
      },
      storage: {
         verbosity: 2,
         journal: {
            verbosity: 1
         }
      }
```

```
    }
} )
```

db.setLogLevel()

Use the db.setLogLevel() (page 197) method to update a single component log level. For a component, you can specify verbosity level of 0 to 5, or you can specify -1 to inherit the verbosity of the parent. For example, the following sets the systemLog.component.query.verbosity (page 900) to its parent verbosity (i.e. default verbosity):

```
db.setLogLevel(-1, "query")
```

6.8 Glossary

On this page

• Additional Resources (page 977)

$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 644).

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 431).

aggregation Any of a variety of operations that reduces and summarizes large sets of data. MongoDB's `aggregate()` (page 20) and `mapReduce()` (page 90) methods are two examples of aggregation operations. For more information, see `https://docs.mongodb.org/manual/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 746).

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 `https://docs.mongodb.org/manual/core/authentication`.

authorization Provisioning of access to databases and operations. See `https://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 `https://docs.mongodb.org/manual/reference/bson-types` and *MongoDB Extended JSON* (page 960).

BSON types The set of types supported by the *BSON* serialization format. For a list of BSON types, see `https://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 `https://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 `https://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.

Client can also refer to a single thread or process.

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 `https://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*.

concurrency control Concurrency control ensures that database operations can be executed concurrently without compromising correctness. Pessimistic concurrency control, such as used in systems with *locks*, will block any potentially conflicting operations even if they may not turn out to actually conflict. Optimistic concurrency control, the approach used by *WiredTiger*, will delay checking until after a conflict may have occurred, aborting and retrying one of the operations involved in any *write conflict* that arises.

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 885).

config server A `mongod` (page 770) instance that stores all the metadata associated with a *sharded cluster*. See *sharding-config-server*.

CRUD An acronym for the fundamental operations of a database: Create, Read, Update, and Delete. See `https://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 841).

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 770) stores data files. The `dbPath` (page 915) 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 303). 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 `https://docs.mongodb.org/manual/applications/geospatial-indexes`.

dbpath The location of MongoDB's data file storage. See `dbPath` (page 915).

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 `https://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 `https://docs.mongodb.org/manual/tutorial/remove-shards-from-cluster`.

driver A client library for interacting with MongoDB in a particular language. See `https://docs.mongodb.org/manual/applications/drivers`.

durable A write operation is durable when it will persist across a shutdown (or crash) and restart of one or more server processes. For a single `mongod` (page 770) server, a write operation is considered durable when it has been written to the server's *journal* file. For a `replica set`, a write operation is considerable durable once the write operation is durable on a `majority of voting nodes` in the replica set; i.e. written to a majority of voting nodes' journal files.

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, in a replica set with one primary member [19],

[19] In *some circumstances*, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with `{ w: "majority" }` write concern. The node that can complete `{ w: "majority" }` writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs,

- With `"local"` `readConcern`, reads from the primary reflect the latest writes in absence of a failover;
- With `"majority"` `readConcern`, read operations from the primary or the 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 `https://docs.mongodb.org/manual/core/aggregation-pipeline`.

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 451).

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 `https://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 878) program. See `https://docs.mongodb.org/manual/core/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 `https://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 `https://docs.mongodb.org/manual/core/indexes`.

init 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.

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*.

clients that connect to the former primary may observe stale data despite having requested read preference `primary`, and new writes to the former primary will eventually roll back.

intent lock A *lock* on a resource that indicates that the holder of the lock will read (intent shared) or write (intent exclusive) the resource using *concurrency control* at a finer granularity than that of the resource with the intent lock. Intent locks allow concurrent readers and writers of a resource. See *faq-concurrency-locking*.

interrupt point A point in an operation's lifecycle when it can safely abort. MongoDB only terminates an operation at designated interrupt points. See `https://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 803) 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 `https://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 `https://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 *MongoDB Extended JSON* (page 960).

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 `https://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 that `concurrency` does not affect correctness. MongoDB uses *read locks*, *write locks* and *intent 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 `https://docs.mongodb.org/manual/core/map-reduce`. For all approaches to aggregation, see `https://docs.mongodb.org/manual/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 `https://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 445).

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 878) tool provides an option to specify a MIME type to describe a file inserted into *GridFS* storage.

mongo The MongoDB shell. The `mongo` (page 803) process starts the MongoDB shell as a daemon connected to either a `mongod` (page 770) or `mongos` (page 792) instance. The shell has a JavaScript interface. See *mongo* (page 802) and *mongo Shell Methods* (page 19).

mongod The MongoDB database server. The `mongod` (page 770) process starts the MongoDB server as a daemon. The MongoDB server manages data requests and formats and manages background operations. See *mongod* (page 769).

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[20].

mongos The routing and load balancing process that acts an interface between an application and a MongoDB *sharded cluster*. See *mongos* (page 791).

namespace The canonical name for a collection or index in MongoDB. The namespace is a combination of the database name and the name of the collection or index, like so: `[database-name].[collection-or-index-name]`. All documents belong to a namespace. See *faq-dev-namespace*.

natural order The order in which the database refers to documents on disk. This is the default sort order. See `$natural` and *Return in Natural Order* (page 159).

network partition A network failure that separates a distributed system into partitions such that nodes in one partition cannot communicate with the nodes in the other partition.

Sometimes, partitions are partial or asymmetric. An example of a partial partition would be a division of the nodes of a network into three sets, where members of the first set cannot communicate with members of the second set, and vice versa, but all nodes can communicate with members of the third set. In an asymmetric partition, communication may be possible only when it originates with certain nodes. For example, nodes on one side of the partition can communicate to the other side only if they originate the communications channel.

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 527).

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 `https://docs.mongodb.org/manual/core/replica-set-oplog`.

[20]https://www.mongodb.com/products/mongodb-subscriptions

ordered query plan A query plan that returns results in the order consistent with the `sort()` (page 156) 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 415) 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 117) 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 With the MMAPv1 storage engine, page faults can occur as MongoDB reads from or writes data to parts of its data files that are not currently located in physical memory. In contrast, operating system page faults happen when physical memory is exhausted and pages of physical memory are swapped to disk.

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 `members[n].priority` is `0`. See `https://docs.mongodb.org/manual/core/replica-set-priority-0-member`.

pcap A packet-capture format used by `mongosniff` (page 873) to record packets captured from network interfaces and display them as human-readable MongoDB operations. See *Options* (page 873).

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 `https://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 457) and `usePowerOf2Sizes` (page 458).

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 `https://docs.mongodb.org/manual/tutorial/create-chunks-in-sharded-cluster`.

prefix compression Reduces memory and disk consumption by storing any identical index key prefixes only once, per page of memory. See: *storage-wiredtiger-compression* for more about WiredTiger's compression behavior.

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 `members[n].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 588).

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 803) shell, you can issue queries using the `find()` (page 51) and `findOne()` (page 62) 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 concern Specifies a level of isolation for read operations. For example, you can use read concern to only read data that has propagated to a majority of nodes in a *replica set*. See `https://docs.mongodb.org/manual/reference/read-concern`.

read lock A shared *lock* on a resource such as a collection or database 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 shard replica sets. By default, MongoDB directs reads to *primaries*. 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 454).

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 `https://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 `https://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-api*.

Deprecated since version 3.2: HTTP interface for MongoDB

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 `https://docs.mongodb.org/manual/security`.

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 `https://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` (page 922) setting or the `--replSet` option.

shard A single `mongod` (page 770) 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 `https://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 config servers, shards, and one or more `mongos` (page 792) routing processes. See `https://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 `https://docs.mongodb.org/manual/sharding`.

shell helper A method in the `mongo` shell that provides a more concise syntax for a *database command* (page 303). Shell helpers improve the general interactive experience. See *mongo Shell Methods* (page 19).

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 `https://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 `https://docs.mongodb.org/manual/core/master-slave`.

snappy A compression/decompression library designed to balance efficient computation requirements with reasonable compression rates. Snappy is the default compression library for MongoDB's use of *WiredTiger*. See Snappy[21] and the WiredTiger compression documentation[22] for more information.

split The division between *chunks* in a *sharded cluster*. See `https://docs.mongodb.org/manual/core/sharding-chunk`

[21]https://google.github.io/snappy/
[22]http://source.wiredtiger.com/2.4.1/compression.html

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 `https://docs.mongodb.org/manual/core/replica-set-oplog` and `https://docs.mongodb.org/manual/core/replica-set-sync` for more information.

standalone An instance of `mongod` (page 770) that is running as a single server and not as part of a *replica set*. To convert a standalone into a replica set, see `https://docs.mongodb.org/manual/tutorial/convert-standalone-to-replica-set`.

storage engine The part of a database that is responsible for managing how data is stored and accessed, both in memory and on disk. Different storage engines perform better for specific workloads. See `https://docs.mongodb.org/manual/core/storage-engines` for specific details on the built-in storage engines in MongoDB.

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, in a replica set with one primary member [1],

- With `"local"` `readConcern`, reads from the primary reflect the latest writes in absence of a failover;
- With `"majority"` `readConcern`, read operations from the primary or the secondaries 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 `https://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` (page 897).

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*.

tag set A document containing zero or more *tags*.

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 `https://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 792).)

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 841).

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 `https://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 156) order. See *read-operations-query-optimization*.

upsert An option for update operations; e.g. `update()` (page 117), `findAndModify()` (page 57). If set to true, the update operation will either update the document(s) matched by the specified query or if no documents match, insert a new document. The new document will have the fields indicated in the operation. See *Upsert Option* (page 118).

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 770) instances. For *replica sets*, you can configure write concern to confirm replication to a specified number of members. See `https://docs.mongodb.org/manual/reference/write-concern`.

write conflict A situation in which two concurrent operations, at least one of which is a write, attempt to use a resource in a way that would violate constraints imposed by a storage engine using optimistic *concurrency control*. MongoDB will transparently abort and retry one of the conflicting operations.

write lock An exclusive *lock* on a resource such as a collection or database. When a process writes to a resource, it takes an exclusive write lock to prevent other processes from writing to or reading from that resource. For more information on locks, see `https://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 510).

zlib A data compression library that provides higher compression rates at the cost of more CPU, compared to MongoDB's use of *snappy*. You can configure *WiredTiger* to use zlib as its compression library. See http://www.zlib.net and the WiredTiger compression documentation[23] for more information.

6.8.1 Additional Resources

- Quick Reference Cards[24]

[23]http://source.wiredtiger.com/2.4.1/compression.html
[24]https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs

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 Stable Release

(*3.2-series*)

7.1.1 Release Notes for MongoDB 3.2

On this page

- Minor Releases (page 980)
- WiredTiger as Default (page 990)
- Replication Election Enhancements (page 990)
- Sharded Cluster Enhancements (page 990)
- `readConcern` (page 991)
- Partial Indexes (page 991)
- Document Validation (page 991)
- Aggregation Framework Enhancements (page 992)
- MongoDB Tools Enhancements (page 995)
- Encrypted Storage Engine (page 995)
- Text Search Enhancements (page 995)
- New Storage Engines (page 996)
- General Enhancements (page 997)
- Changes Affecting Compatibility (page 1000)
- Upgrade Process (page 1003)
- Known Issues in 3.2.1 (page 1012)
- Known Issues in 3.2.0 (page 1013)
- Download (page 1013)
- Additional Resources (page 1013)

Dec 8, 2015

MongoDB 3.2 is now available. Key features include WiredTiger as the default storage engine, replication election enhancements, config servers as replica sets, `readConcern`, and document validations.

OpsManager 2.0 is also available. See the Ops Manager documentation[1] and the Ops Manager release notes[2] for more information.

Minor Releases

3.2 Changelog

On this page

- 3.2.3 Changelog (page 980)
- 3.2.1 Changelog (page 986)

3.2.3 Changelog

Sharding

- SERVER-18671[3] SecondaryPreferred can end up using unversioned connections
- SERVER-20030[4] ForwardingCatalogManager::shutdown races with _replaceCatalogManager
- SERVER-20036[5] Add interruption points to operations that hold distributed locks for a long time
- SERVER-20037[6] Transfer responsibility for the release of distributed locks to new catalog manager
- SERVER-20290[7] Recipient shard for migration can continue on retrieving data even after donor shard aborts
- SERVER-20418[8] Make sure mongod and mongos always start the distlock pinger when running in SCCC mode
- SERVER-20422[9] setShardVersion configdb string mismatch during config rs upgrade
- SERVER-20580[10] Failure in csrs_upgrade_during_migrate.js
- SERVER-20694[11] user-initiated finds against the config servers can fail with "need to swap catalog manager" error
- SERVER-21382[12] Sharding migration transfers all document deletions
- SERVER-21789[13] mongos replica set monitor should choose primary based on (rs config version, electionId)
- SERVER-21896[14] Chunk metadata will not get refreshed after shard is removed
- SERVER-21906[15] Race in ShardRegistry::reload and config.shard update can cause shard not found error
- SERVER-21956[16] applyOps does not correctly propagate operation cancellation exceptions

[1] http://docs.opsmanager.mongodb.com/current/
[2] http://docs.opsmanager.mongodb.com/current/release-notes/application/
[3] https://jira.mongodb.org/browse/SERVER-18671
[4] https://jira.mongodb.org/browse/SERVER-20030
[5] https://jira.mongodb.org/browse/SERVER-20036
[6] https://jira.mongodb.org/browse/SERVER-20037
[7] https://jira.mongodb.org/browse/SERVER-20290
[8] https://jira.mongodb.org/browse/SERVER-20418
[9] https://jira.mongodb.org/browse/SERVER-20422
[10] https://jira.mongodb.org/browse/SERVER-20580
[11] https://jira.mongodb.org/browse/SERVER-20694
[12] https://jira.mongodb.org/browse/SERVER-21382
[13] https://jira.mongodb.org/browse/SERVER-21789
[14] https://jira.mongodb.org/browse/SERVER-21896
[15] https://jira.mongodb.org/browse/SERVER-21906
[16] https://jira.mongodb.org/browse/SERVER-21956

- SERVER-21994[17] cleanup_orphaned_basic.js
- SERVER-21995[18] Queries against sharded collections fail after upgrade to CSRS due to caching of config server string in setShardVersion
- SERVER-22010[19] min_optime_recovery.js failure in the sharding continuous config stepdown suite
- SERVER-22016[20] Fatal assertion 28723 trying to rollback applyOps on a CSRS config server
- SERVER-22027[21] AsyncResultMerger should not retry killed operations
- SERVER-22079[22] Make sharding_rs1.js more compact
- SERVER-22112[23] Circular call dependency between CatalogManager and CatalogCache
- SERVER-22113[24] Remove unused sharding-specific getLocsInRange code in dbhelpers
- SERVER-22114[25] Mongos can accumulate multiple copies of ChunkManager when a shard restarts
- SERVER-22169[26] Deadlock during CatalogManager swap from SCCC -> CSRS
- SERVER-22232[27] Increase stability of csrs_upgrade_during_migrate.js test
- SERVER-22247[28] Parsing old config.collection documents fails because of missing 'lastmodEpoch' field
- SERVER-22249[29] stats.js - Not starting chunk migration because another migration is already in progress
- SERVER-22270[30] applyOps to config rs does not wait for majority
- SERVER-22303[31] Wait longer for initial sync to finish in csrs_upgrade_during_migrate.js

Replication

- SERVER-21583[32] ApplyOps background index creation may deadlock
- SERVER-21678[33] fromMigrate flag never set for deletes in oplog
- SERVER-21744[34] Clients may fail to discover new primaries when clock skew between nodes is greater than electionTimeout
- SERVER-21958[35] Eliminate unused flags from Cloner methods
- SERVER-21988[36] Rollback does not wait for applier to finish before starting
- SERVER-22109[37] Invariant failure when running applyOps to create an index with a bad ns field

[17]https://jira.mongodb.org/browse/SERVER-21994
[18]https://jira.mongodb.org/browse/SERVER-21995
[19]https://jira.mongodb.org/browse/SERVER-22010
[20]https://jira.mongodb.org/browse/SERVER-22016
[21]https://jira.mongodb.org/browse/SERVER-22027
[22]https://jira.mongodb.org/browse/SERVER-22079
[23]https://jira.mongodb.org/browse/SERVER-22112
[24]https://jira.mongodb.org/browse/SERVER-22113
[25]https://jira.mongodb.org/browse/SERVER-22114
[26]https://jira.mongodb.org/browse/SERVER-22169
[27]https://jira.mongodb.org/browse/SERVER-22232
[28]https://jira.mongodb.org/browse/SERVER-22247
[29]https://jira.mongodb.org/browse/SERVER-22249
[30]https://jira.mongodb.org/browse/SERVER-22270
[31]https://jira.mongodb.org/browse/SERVER-22303
[32]https://jira.mongodb.org/browse/SERVER-21583
[33]https://jira.mongodb.org/browse/SERVER-21678
[34]https://jira.mongodb.org/browse/SERVER-21744
[35]https://jira.mongodb.org/browse/SERVER-21958
[36]https://jira.mongodb.org/browse/SERVER-21988
[37]https://jira.mongodb.org/browse/SERVER-22109

- SERVER-22152[38] priority_takeover_two_nodes_equal_priority.js fails if default priority node gets elected at beginning of test
- SERVER-22190[39] electionTime field not set in heartbeat response from primary under protocol version 1
- SERVER-22335[40] Do not prepare getmore when un-needed in bgsync fetcher
- SERVER-22362[41] election_timing.js waits for wrong node to become primary
- SERVER-22420[42] priority_takeover_two_nodes_equal_priority.js fails if existing primary's step down period expires
- SERVER-22456[43] The oplog find query timeout is too low

Query

- SERVER-17011[44] Cursor can return objects out of order if updated during query ("legacy" readMode only)
- SERVER-18115[45] The planner can add an unnecessary in-memory sort stage for .min()/.max() queries
- SERVER-20083[46] Add log statement at default log level for when an index filter is set or cleared successfully
- SERVER-21776[47] Move per-operation log lines for queries out of the QUERY log component
- SERVER-21869[48] Avoid wrapping of spherical queries in geo_full.js
- SERVER-22002[49] Do not retry findAndModify operations on MMAPv1
- SERVER-22100[50] memory pressure from find/getMore buffer preallocation causes concurrency suite slowness on Windows DEBUG
- SERVER-22448[51] Query planner does not filter 2dsphere Index Version 3 correctly

Write Operations

- SERVER-11983[52] Update on document without _id, in capped collection without _id index, creates an _id field
- SERVER-21647[53] $rename changes field ordering

Aggregation

- SERVER-21887[54] $sample takes disproportionately long time on newly created collection
- SERVER-22048[55] Index access stats should be recorded for $match & mapReduce

[38]https://jira.mongodb.org/browse/SERVER-22152
[39]https://jira.mongodb.org/browse/SERVER-22190
[40]https://jira.mongodb.org/browse/SERVER-22335
[41]https://jira.mongodb.org/browse/SERVER-22362
[42]https://jira.mongodb.org/browse/SERVER-22420
[43]https://jira.mongodb.org/browse/SERVER-22456
[44]https://jira.mongodb.org/browse/SERVER-17011
[45]https://jira.mongodb.org/browse/SERVER-18115
[46]https://jira.mongodb.org/browse/SERVER-20083
[47]https://jira.mongodb.org/browse/SERVER-21776
[48]https://jira.mongodb.org/browse/SERVER-21869
[49]https://jira.mongodb.org/browse/SERVER-22002
[50]https://jira.mongodb.org/browse/SERVER-22100
[51]https://jira.mongodb.org/browse/SERVER-22448
[52]https://jira.mongodb.org/browse/SERVER-11983
[53]https://jira.mongodb.org/browse/SERVER-21647
[54]https://jira.mongodb.org/browse/SERVER-21887
[55]https://jira.mongodb.org/browse/SERVER-22048

JavaScript SERVER-21528[56] Clean up core/capped6.js

Storage

- SERVER-21388[57] Invariant Failure in CappedRecordStoreV1::cappedTruncateAfter
- SERVER-22011[58] Direct writes to the local database can cause deadlock involving the WiredTiger write throttle
- SERVER-22058[59] 'not all control paths return a value' warning in non-MMAP V1 implementations of '::writingPtr'
- SERVER-22167[60] Failed to insert document larger than 256k
- SERVER-22199[61] Collection drop command during checkpoint causes complete stall until end of checkpoint

WiredTiger

- SERVER-21833[62] Compact does not release space to the system with WiredTiger
- SERVER-21944[63] WiredTiger changes for 3.2.3
- SERVER-22064[64] Coverity analysis defect 77699: Unchecked return value
- SERVER-22279[65] SubplanStage fails to register its MultiPlanStage

MMAP

- SERVER-21997[66] kill_cursors.js deadlocks
- SERVER-22261[67] MMAPv1 LSNFile may be updated ahead of what is synced to data files

Operations

- SERVER-20358[68] Usernames can contain NULL characters
- SERVER-22007[69] List all commands crashes server
- SERVER-22075[70] election_timing.js election timed out

Build and Packaging

- SERVER-21905[71] Can't compile Mongo 3.2
- SERVER-22042[72] If ssl libraries not present, configure fails with a misleading error about boost

[56] https://jira.mongodb.org/browse/SERVER-21528
[57] https://jira.mongodb.org/browse/SERVER-21388
[58] https://jira.mongodb.org/browse/SERVER-22011
[59] https://jira.mongodb.org/browse/SERVER-22058
[60] https://jira.mongodb.org/browse/SERVER-22167
[61] https://jira.mongodb.org/browse/SERVER-22199
[62] https://jira.mongodb.org/browse/SERVER-21833
[63] https://jira.mongodb.org/browse/SERVER-21944
[64] https://jira.mongodb.org/browse/SERVER-22064
[65] https://jira.mongodb.org/browse/SERVER-22279
[66] https://jira.mongodb.org/browse/SERVER-21997
[67] https://jira.mongodb.org/browse/SERVER-22261
[68] https://jira.mongodb.org/browse/SERVER-20358
[69] https://jira.mongodb.org/browse/SERVER-22007
[70] https://jira.mongodb.org/browse/SERVER-22075
[71] https://jira.mongodb.org/browse/SERVER-21905
[72] https://jira.mongodb.org/browse/SERVER-22042

- SERVER-22350[73] Package generation failure doesn't fail compile tasks

Tools TOOLS-1039[74] mongoexport chokes on data with quotes

Internals

- SERVER-12108[75] setup_multiversion_mongodb.py script should support downloading windows binaries
- SERVER-20409[76] Negative scaling with more than 10K connections
- SERVER-21035[77] Delete the disabled fsm_all_sharded.js test runner
- SERVER-21050[78] Add a failover workload to cause CSRS config server primary failovers
- SERVER-21309[79] Remove Install step from jstestfuzz in evergreen
- SERVER-21421[80] Update concurrency suite's ThreadManager constructor to provide default executionMode
- SERVER-21499[81] Enable fsm_all_simultaneous.js (FSM parallel mode)
- SERVER-21565[82] resmoke.py can not start replica sets with more than 7 nodes
- SERVER-21597[83] Fix connPoolStats command to work with many TaskExecutor-NetworkInterface pairs
- SERVER-21747[84] CheckReplDBHash should not print error message when the system collections differ in the presence of other errors
- SERVER-21801[85] CheckReplDBHash testing hook should check document type (resmoke.py)
- SERVER-21875[86] AttributeError in hang_analyzer.py when sending SIGKILL on Windows
- SERVER-21892[87] Include thread ID in concurrency suite error report
- SERVER-21894[88] Remove unused 'hashed' resmoke.py tags from JS tests
- SERVER-21902[89] Use multiple shard nodes in the jstestfuzz_sharded suite
- SERVER-21916[90] Add missing tasks/suites to ASan Evergreen variant
- SERVER-21917[91] Add the httpinterface test suite to the Enterprise RHEL 6.2 variant
- SERVER-21934[92] Add extra information to OSX stack traces to facilitate addr2line translation
- SERVER-21940[93] Workload connection cache in FSM suite is not nulled out properly

[73]https://jira.mongodb.org/browse/SERVER-22350
[74]https://jira.mongodb.org/browse/TOOLS-1039
[75]https://jira.mongodb.org/browse/SERVER-12108
[76]https://jira.mongodb.org/browse/SERVER-20409
[77]https://jira.mongodb.org/browse/SERVER-21035
[78]https://jira.mongodb.org/browse/SERVER-21050
[79]https://jira.mongodb.org/browse/SERVER-21309
[80]https://jira.mongodb.org/browse/SERVER-21421
[81]https://jira.mongodb.org/browse/SERVER-21499
[82]https://jira.mongodb.org/browse/SERVER-21565
[83]https://jira.mongodb.org/browse/SERVER-21597
[84]https://jira.mongodb.org/browse/SERVER-21747
[85]https://jira.mongodb.org/browse/SERVER-21801
[86]https://jira.mongodb.org/browse/SERVER-21875
[87]https://jira.mongodb.org/browse/SERVER-21892
[88]https://jira.mongodb.org/browse/SERVER-21894
[89]https://jira.mongodb.org/browse/SERVER-21902
[90]https://jira.mongodb.org/browse/SERVER-21916
[91]https://jira.mongodb.org/browse/SERVER-21917
[92]https://jira.mongodb.org/browse/SERVER-21934
[93]https://jira.mongodb.org/browse/SERVER-21940

- SERVER-21949[94] Add validation testing hook to resmoke.py
- SERVER-21952[95] jstestfuzz tasks should not run with –continueOnFailure
- SERVER-21959[96] Do not truncate stack traces in log messages
- SERVER-21960[97] Include symbol name in stacktrace json when available
- SERVER-21964[98] Remove startPort option from ReplSetTest options in jstests/replsets/auth1.js
- SERVER-21978[99] move_primary_basic.js should always set a fixed primary shard
- SERVER-21990[100] Deprecation warning from resmoke.py - replicaset.py insert is deprecated
- SERVER-22028[101] hang_analyzer should fail when run against unsupported lldb
- SERVER-22034[102] Server presents clusterFile certificate for incoming connections
- SERVER-22054[103] Authentication failure reports incorrect IP address
- SERVER-22055[104] Cleanup unused legacy client functionality from the server code
- SERVER-22059[105] Add the authSchemaUpgrade command to the readConcern passthrough
- SERVER-22066[106] range_deleter_test:ImmediateDelete is flaky
- SERVER-22083[107] Delete the disabled fsm_all_master_slave.js test runner
- SERVER-22098[108] Split FSM sharded tests for SCCC into a separate suite
- SERVER-22099[109] Remove unreliable check in cleanup_orphaned_basic.js
- SERVER-22120[110] No data found after force sync in no_chaining.js
- SERVER-22121[111] Add resmoke.py validation testing hook to test suites
- SERVER-22142[112] resmoke.py's FlushThread attempts to reference imported members during Python interpreter shutdown
- SERVER-22154[113] csrs_upgrade.js, csrs_upgrade_during_migrate.js should be blacklisted on in-mem
- SERVER-22165[114] Deadlock in resmoke.py between logger pipe and timer thread
- SERVER-22171[115] The lint task is running on 3 Evergreen variants
- SERVER-22219[116] Use the subprocess32 package on POSIX systems in resmoke.py if it's available

[94]https://jira.mongodb.org/browse/SERVER-21949
[95]https://jira.mongodb.org/browse/SERVER-21952
[96]https://jira.mongodb.org/browse/SERVER-21959
[97]https://jira.mongodb.org/browse/SERVER-21960
[98]https://jira.mongodb.org/browse/SERVER-21964
[99]https://jira.mongodb.org/browse/SERVER-21978
[100]https://jira.mongodb.org/browse/SERVER-21990
[101]https://jira.mongodb.org/browse/SERVER-22028
[102]https://jira.mongodb.org/browse/SERVER-22034
[103]https://jira.mongodb.org/browse/SERVER-22054
[104]https://jira.mongodb.org/browse/SERVER-22055
[105]https://jira.mongodb.org/browse/SERVER-22059
[106]https://jira.mongodb.org/browse/SERVER-22066
[107]https://jira.mongodb.org/browse/SERVER-22083
[108]https://jira.mongodb.org/browse/SERVER-22098
[109]https://jira.mongodb.org/browse/SERVER-22099
[110]https://jira.mongodb.org/browse/SERVER-22120
[111]https://jira.mongodb.org/browse/SERVER-22121
[112]https://jira.mongodb.org/browse/SERVER-22142
[113]https://jira.mongodb.org/browse/SERVER-22154
[114]https://jira.mongodb.org/browse/SERVER-22165
[115]https://jira.mongodb.org/browse/SERVER-22171
[116]https://jira.mongodb.org/browse/SERVER-22219

- SERVER-22324[117] Update findAndModify FSM workloads to handle not matching anything
- TOOLS-1028[118] expose qr/qw and ar/aw fields in mongostat JSON output mode.

3.2.1 Changelog

Security

- SERVER-21724[119] Backup role can't read system.profile
- SERVER-21824[120] Disable kmip.js test in ESE suite; re-enable once fixed
- SERVER-21890[121] Create a flag to allow server realm to be specified explicitly on Windows

Sharding

- SERVER-20824[122] Test for sharding state recovery
- SERVER-21076[123] Write tests to ensure that operations using DBDirectClient handle shard versioning properly
- SERVER-21132[124] Add more basic tests for moveChunk
- SERVER-21133[125] Add more basic test for mergeChunk
- SERVER-21134[126] Add more basic tests for shardCollection
- SERVER-21135[127] Add more basic tests for sharded implicit database creation
- SERVER-21136[128] Add more basic tests for enableSharding
- SERVER-21137[129] Add more basic tests for movePrimary
- SERVER-21138[130] Add more basic tests for dropDatabase
- SERVER-21139[131] Add more basic tests for drop collection
- SERVER-21366[132] Long-running transactions in MigrateStatus::apply
- SERVER-21586[133] Investigate v3.0 mongos and v3.2 cluster compatibility issues in jstests/sharding
- SERVER-21704[134] JS Test single_node_config_server_smoke has race condition
- SERVER-21706[135] Certain parameters to mapReduce trigger segmentation fault in a sharded cluster
- SERVER-21786[136] Fix code coverage gaps in s/query directory exposed by code coverage tool

[117] https://jira.mongodb.org/browse/SERVER-22324
[118] https://jira.mongodb.org/browse/TOOLS-1028
[119] https://jira.mongodb.org/browse/SERVER-21724
[120] https://jira.mongodb.org/browse/SERVER-21824
[121] https://jira.mongodb.org/browse/SERVER-21890
[122] https://jira.mongodb.org/browse/SERVER-20824
[123] https://jira.mongodb.org/browse/SERVER-21076
[124] https://jira.mongodb.org/browse/SERVER-21132
[125] https://jira.mongodb.org/browse/SERVER-21133
[126] https://jira.mongodb.org/browse/SERVER-21134
[127] https://jira.mongodb.org/browse/SERVER-21135
[128] https://jira.mongodb.org/browse/SERVER-21136
[129] https://jira.mongodb.org/browse/SERVER-21137
[130] https://jira.mongodb.org/browse/SERVER-21138
[131] https://jira.mongodb.org/browse/SERVER-21139
[132] https://jira.mongodb.org/browse/SERVER-21366
[133] https://jira.mongodb.org/browse/SERVER-21586
[134] https://jira.mongodb.org/browse/SERVER-21704
[135] https://jira.mongodb.org/browse/SERVER-21706
[136] https://jira.mongodb.org/browse/SERVER-21786

- SERVER-21848[137] bulk write operations on config/admin triggers invariant failure

Replication

- SERVER-21248[138] jstests for fast-failover correctness
- SERVER-21667[139] do not set lastop on clients used by replication on secondaries
- SERVER-21795[140] Do not reschedule more than one liveness timeout callback at a time
- SERVER-21847[141] log range of operations read from sync source during replication
- SERVER-21868[142] Shutdown may not be handled correctly on secondary nodes
- SERVER-21930[143] Restart oplog query if oplog entries are not monotonically increasing

Query

- SERVER-21600[144] Increase test coverage for killCursors command and OP_KILLCURSORS
- SERVER-21602[145] Reduce execution time of cursor_timeout.js
- SERVER-21637[146] Add mixed version tests for find/getMore commands
- SERVER-21638[147] Audit and improve logging in new find/getMore commands code
- SERVER-21750[148] getMore command does not set "nreturned" operation counter

Storage

- SERVER-21384[149] Expand testing for in memory storage engines
- SERVER-21545[150] collMod and invalid parameter triggers fassert on dropCollection on mmapv1
- SERVER-21885[151] capped_truncate.js cannot be run with –repeat
- SERVER-21920[152] Use enhanced WiredTiger next_random cursors for oplog stones

WiredTiger

- SERVER-21792[153] 75% performance regression in insert workload under Windows between 3.0.7 and 3.2 with WiredTiger
- SERVER-21872[154] WiredTiger changes for 3.2.1

[137] https://jira.mongodb.org/browse/SERVER-21848
[138] https://jira.mongodb.org/browse/SERVER-21248
[139] https://jira.mongodb.org/browse/SERVER-21667
[140] https://jira.mongodb.org/browse/SERVER-21795
[141] https://jira.mongodb.org/browse/SERVER-21847
[142] https://jira.mongodb.org/browse/SERVER-21868
[143] https://jira.mongodb.org/browse/SERVER-21930
[144] https://jira.mongodb.org/browse/SERVER-21600
[145] https://jira.mongodb.org/browse/SERVER-21602
[146] https://jira.mongodb.org/browse/SERVER-21637
[147] https://jira.mongodb.org/browse/SERVER-21638
[148] https://jira.mongodb.org/browse/SERVER-21750
[149] https://jira.mongodb.org/browse/SERVER-21384
[150] https://jira.mongodb.org/browse/SERVER-21545
[151] https://jira.mongodb.org/browse/SERVER-21885
[152] https://jira.mongodb.org/browse/SERVER-21920
[153] https://jira.mongodb.org/browse/SERVER-21792
[154] https://jira.mongodb.org/browse/SERVER-21872

Operations SERVER-21870[155] Missing space in error message

Build and Packaging

- SERVER-13370[156] Generate Enterprise RPM's for Amazon Linux
- SERVER-21781[157] Nightly packages are in the wrong repo directories
- SERVER-21796[158] fix startup_log.js test to handle git describe versioning
- SERVER-21864[159] streamline artifact signing procedure to support coherent release process

Tools

- TOOLS-954[160] Add bypassDocumentValidation option to mongorestore and mongoimport
- TOOLS-982[161] Missing "from" text in mongorestore status message

Internals

- SERVER-21164[162] Change assert to throw in rslib.js's wait loop
- SERVER-21214[163] Dump config server data when the sharded concurrency suites fail
- SERVER-21426[164] Add writeConcern support to benchRun
- SERVER-21450[165] Modify MongoRunner to add enableMajorityReadConcern flag based on jsTestOptions
- SERVER-21500[166] Include the name of the FSM workload in the WorkloadFailure description
- SERVER-21516[167] Remove dbStats command from readConcern testing override
- SERVER-21665[168] Suppress tar output in jstestfuzz tasks
- SERVER-21714[169] Increase replSetTest.initiate() timeout for FSM tests
- SERVER-21719[170] Add initiateTimeout rsOption for ShardingTest
- SERVER-21725[171] Enable the analysis script move
- SERVER-21737[172] remove deprecated release process configuration from master branch evergreen configuration
- SERVER-21752[173] slow2_wt fails by exhausting host machine's memory

[155]https://jira.mongodb.org/browse/SERVER-21870
[156]https://jira.mongodb.org/browse/SERVER-13370
[157]https://jira.mongodb.org/browse/SERVER-21781
[158]https://jira.mongodb.org/browse/SERVER-21796
[159]https://jira.mongodb.org/browse/SERVER-21864
[160]https://jira.mongodb.org/browse/TOOLS-954
[161]https://jira.mongodb.org/browse/TOOLS-982
[162]https://jira.mongodb.org/browse/SERVER-21164
[163]https://jira.mongodb.org/browse/SERVER-21214
[164]https://jira.mongodb.org/browse/SERVER-21426
[165]https://jira.mongodb.org/browse/SERVER-21450
[166]https://jira.mongodb.org/browse/SERVER-21500
[167]https://jira.mongodb.org/browse/SERVER-21516
[168]https://jira.mongodb.org/browse/SERVER-21665
[169]https://jira.mongodb.org/browse/SERVER-21714
[170]https://jira.mongodb.org/browse/SERVER-21719
[171]https://jira.mongodb.org/browse/SERVER-21725
[172]https://jira.mongodb.org/browse/SERVER-21737
[173]https://jira.mongodb.org/browse/SERVER-21752

- SERVER-21768[174] Remove the 'numCollections' field from dbHash's response
- SERVER-21772[175] findAndModify not captured by Profiler
- SERVER-21793[176] create v3.2 branch and update evergreen configuration
- SERVER-21849[177] Fix timestamp compare in min_optime_recovery.js
- SERVER-21852[178] kill_cursors.js fails in small_oplog* configurations
- SERVER-21871[179] Do not run min_optime_recovery.js on ephemeralForTest storageEngine
- SERVER-21901[180] CheckReplDBHash checks the wrong node when dumping docs from missing collections
- SERVER-21923[181] ReplSetTest.awaitSecondaryNodes does not propagate supplied timeout
- TOOLS-944[182] write concern mongos tests are flaky
- TOOLS-1002[183] oplog_rollover test is flaky

3.2.3 – Feb 17, 2016

- Fixed issue with MMAPv1 journaling where the "last sequence number" file (`lsn` file) may be ahead of what is synced to the data files: SERVER-22261[184].
- Fixed issue where in some cases, insert operations fails to add the `_id` field to large documents: SERVER-22167[185].
- Increased timeout for querying oplog to 1 minute: SERVER-22456[186].
- All issues closed in 3.2.3[187]

3.2.1 – Jan 12, 2016

- Fixed error where during a regular shutdown of a replica set, secondaries may mark certain replicated but yet to be applied operations as successfully applied: SERVER-21868[188].
- Improve insert workload performance with WiredTiger on Windows: SERVER-20262[189].
- Fixed long-running transactions during chunk moves: SERVER-21366[190]
- All issues closed in 3.2.1[191]

[174]https://jira.mongodb.org/browse/SERVER-21768
[175]https://jira.mongodb.org/browse/SERVER-21772
[176]https://jira.mongodb.org/browse/SERVER-21793
[177]https://jira.mongodb.org/browse/SERVER-21849
[178]https://jira.mongodb.org/browse/SERVER-21852
[179]https://jira.mongodb.org/browse/SERVER-21871
[180]https://jira.mongodb.org/browse/SERVER-21901
[181]https://jira.mongodb.org/browse/SERVER-21923
[182]https://jira.mongodb.org/browse/TOOLS-944
[183]https://jira.mongodb.org/browse/TOOLS-1002
[184]https://jira.mongodb.org/browse/SERVER-22261
[185]https://jira.mongodb.org/browse/SERVER-22167
[186]https://jira.mongodb.org/browse/SERVER-22456
[187]https://jira.mongodb.org/issues/?jql=project%20in%20(SERVER%2C%20TOOLS)%20AND%20fixVersion%20%3D%203.2.3%20AND%20resolution%20%3D%2
[188]https://jira.mongodb.org/browse/SERVER-21868
[189]https://jira.mongodb.org/browse/SERVER-20262
[190]https://jira.mongodb.org/browse/SERVER-21366
[191]https://jira.mongodb.org/issues/?jql=project%20in%20(SERVER%2C%20TOOLS)%20AND%20fixVersion%20%3D%203.2.1%20AND%20resolution%20%3D%2

WiredTiger as Default

Starting in 3.2, MongoDB uses the WiredTiger as the default storage engine.

To specify the MMAPv1 storage engine, you must specify the storage engine setting either:

- On the command line with the `--storageEngine` option:

  ```
  mongod --storageEngine mmapv1
  ```

- Or in a *configuration file* (page 895), using the `storage.engine` (page 917) setting:

  ```
  storage:
     engine: mmapv1
  ```

Note: For existing deployments, if you do not specify the `--storageEngine` or the `storage.engine` (page 917) setting, MongoDB 3.2 can automatically determine the storage engine used to create the data files in the `--dbpath` or `storage.dbPath` (page 915).

If specifying `--storageEngine` or `storage.engine` (page 917), `mongod` (page 770) will not start if `dbPath` contains data files created by a storage engine other than the one specified.

See also:

Default Storage Engine Change (page 1000)

Replication Election Enhancements

Starting in MongoDB 3.2, MongoDB reduces replica set failover time and accelerates the detection of multiple simultaneous primaries.

As part of this enhancement, MongoDB introduces a version 1 of the replication protocol. New replica sets will, by default, use `protocolVersion: 1`. Previous versions of MongoDB use version 0 of the protocol.

In addition, MongoDB introduces a new `replica set configuration` option `electionTimeoutMillis`. `electionTimeoutMillis` specifies the time limit in milliseconds for detecting when a replica set's primary is unreachable.

`electionTimeoutMillis` only applies if using the version 1 of the `replication protocol`.

Sharded Cluster Enhancements

MongoDB 3.2 deprecates the use of three mirrored `mongod` (page 770) instances for config servers.

Instead, starting in 3.2, the `config servers` for a sharded cluster can be deployed as a replica set. The replica set config servers must run the WiredTiger storage engine.

This change improves consistency across the config servers, since MongoDB can take advantage of the standard replica set read and write protocols for sharding config data. In addition, this allows a sharded cluster to have more than 3 config servers since a replica set can have up to 50 members.

For more information, see `https://docs.mongodb.org/manual/core/sharded-cluster-config-servers`. To deploy a **new** sharded cluster with replica set config servers, see `https://docs.mongodb.org/manual/tutorial/deploy-shard-cluster`.

`readConcern`

MongoDB 3.2 introduces the `readConcern` query option for replica sets and replica set shards. For the `WiredTiger storage engine`, the `readConcern` option allows clients to choose a level of isolation for their reads. You can specify a `readConcern` of `"majority"` to read data that has been written to a majority of nodes and thus cannot be rolled back. By default, MongoDB uses a `readConcern` of `"local"` to return the most recent data available to the node at the time of the query, even if the data has not been persisted to a majority of nodes and may be rolled back. With the `MMAPv1 storage engine`, you can only specify a `readConcern` of `"local"`.

`readConcern` requires MongoDB drivers updated for MongoDB 3.2.

Only replica sets using `protocol version 1` support `"majority"` read concern. Replica sets running protocol version 0 do not support `"majority"` read concern.

For details on `readConcern`, including operations that support the option, see `https://docs.mongodb.org/manual/reference/read-concern`.

Partial Indexes

MongoDB 3.2 provides the option to create indexes that only index the documents in a collection that meet a specified filter expression. By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance. You can specify a `partialFilterExpression` option for all MongoDB `index types`.

The `partialFilterExpression` option accepts a document that specifies the condition using:

- equality expressions (i.e. `field: value` or using the `$eq` (page 527) operator),
- `$exists: true` (page 538) expression,
- `$gt` (page 529), `$gte` (page 530), `$lt` (page 530), `$lte` (page 531) expressions,
- `$type` (page 540) expressions,
- `$and` (page 535) operator at the top-level only

For details, see `https://docs.mongodb.org/manual/core/index-partial`.

Document Validation

Starting in 3.2, MongoDB provides the capability to validate documents during updates and insertions. Validation rules are specified on a per-collection basis.

To specify document validation on a new collection, use the new `validator` option in the `db.createCollection()` (page 167) method. To add document validation to an existing collection, use the new `validator` option in the `collMod` (page 457) command. For more information, see `https://docs.mongodb.org/manual/core/document-validation`.

To view the validation specifications for a collection, use the `db.getCollectionInfos()` (page 182) method.

The following commands can bypass validation per operation using the new option `bypassDocumentValidation`:

- `applyOps` (page 408) command
- `findAndModify` (page 348) command and `db.collection.findAndModify()` (page 57) method
- `mapReduce` (page 318) command and `db.collection.mapReduce()` (page 90) method
- `insert` (page 337) command

- `update` (page 340) command
- `$out` (page 656) for the `aggregate` (page 303) command and `db.collection.aggregate()` (page 20) method

For deployments that have enabled access control, you must have `bypassDocumentValidation` action. The built-in roles `dbAdmin` and `restore` provide this action.

Aggregation Framework Enhancements

MongoDB introduces:

- New stages, accumulators, and expressions.
- *Availability of accumulator expressions* (page 994) in `$project` (page 631) stage.
- *Performance improvements* (page 994) on sharded clusters.

New Aggregation Stages

Stage	Description	Syntax
`$sample` (page 648)	Randomly selects N documents from its input.	`{ $sample: { size: <positive integer> } }`
`$indexStats` (page 657)	Returns statistics on index usage.	`{ $indexStats: { } }`
`$lookup` (page 654)	Performs a left outer join with another collection.	`{ $lookup: { from: <collection to join>, localField: <fieldA>, foreignField: <fieldB>, as: <output array field> } }`

New Accumulators for `$group` Stage

Accumulator	Description	Syntax
`$stdDevSamp` (page 744)	Calculates standard deviation.	`{ $stdDevSamp: <array> }`
`$stdDevPop` (page 742)	Calculates population standard deviation.	`{ $stdDevPop: <array> }`

New Aggregation Arithmetic Operators

Operator	Description	Syntax
`$sqrt` (page 692)	Calculates the square root.	`{ $sqrt: <number> }`
`$abs` (page 681)	Returns the absolute value of a number.	`{ $abs: <number> }`
`$log` (page 687)	Calculates the log of a number in the specified base.	`{ $log: [ <number>, <base> ] }`
`$log10` (page 689)	Calculates the log base 10 of a number.	`{ $log10: <number> }`
`$ln` (page 687)	Calculates the natural log of a number.	`{ $ln: <number> }`
`$pow` (page 691)	Raises a number to the specified exponent.	`{ $pow: [ <number>, <exponent> ] }`
`$exp` (page 685)	Raises *e* to the specified exponent.	`{ exp: <number> }`
`$trunc` (page 695)	Truncates a number to its integer.	`{ $trunc: <number> }`
`$ceil` (page 683)	Returns the smallest integer greater than or equal to the specified number.	`{ $ceil: <number> }`
`$floor` (page 686)	Returns the largest integer less than or equal to the specified number.	`{ floor: <number> }`

New Aggregation Array Operators

Operator	Description	Syntax

`$slice` (page 707)	Returns a subset of an array.	`{ $slice: [ <array>, <n> ] }`
		or
		`{ $slice: [ <array>, <position>, <n> ] }`

`$arrayElemAt` (page 702)	Returns the element at the specified array index.	`{ $arrayElemAt: [ <array>, <idx> ] }`

`$concatArrays` (page 703)	Concatenates arrays.	`{`
		`  $concatArrays: [ <array1>, <array2`
		`}`

`$isArray` (page 706)	Determines if the operand is an array.	`{ $isArray: [ <expression> ] }`

`$filter` (page 704)	Selects a subset of the array based on the condition.	`{`
		`  $filter:`
		`    {`
		`      input: <array>,`
		`      as: <string>,`
		`      cond: <expression>`
		`    }`
		`}`

Accumulator Expression Availability

Starting in version 3.2, the following accumulator expressions, previously only available in the `$group` (page 644) stage, are now also available in the `$project` (page 631) stage:

- `$avg` (page 732)
- `$min` (page 738)
- `$max` (page 736)
- `$sum` (page 729)
- `$stdDevPop` (page 742)
- `$stdDevSamp` (page 744)

When used as part of the `$project` (page 631) stage, these accumulator expressions can accept either:

- A single argument: `<accumulator> : <arg>`
- Multiple arguments: `<accumulator> : [ <arg1>, <arg2>, ... ]`

General Enhancements

- In MongoDB 3.2, `$project` (page 631) stage supports using the square brackets `[]` to directly create new array fields. For an example, see *Project New Array Fields* (page 634).
- MongoDB 3.2 introduces the `minDistance` option for the `$geoNear` (page 651) stage.
- `$unwind` (page 641) stage no longer errors on non-array operand. If the operand does not resolve to an array but is not missing, null, or an empty array, `$unwind` (page 641) treats the operand as a single element array.

  `$unwind` (page 641) stage can:

  – include the array index of the array element in the output by specifying a new option `includeArrayIndex` in the stage specification.

  – output those documents where the array field is missing, null or an empty array by specifying a new option `preserveNullAndEmptyArrays` in the stage specification.

  To support these new features, `$unwind` (page 641) can now take an alternate syntax. See `$unwind` (page 641) for details.

Optimization

Indexes can *cover* aggregation operations.

MongoDB improves the overall performance of the pipeline in large sharded clusters.

If the pipeline starts with an exact `$match` (page 635) on a shard key, the entire pipeline runs on the matching shard only. Previously, the pipeline would have been split, and the work of merging it would have to be done on the primary shard.

For aggregation operations that run on multiple shards, if the operations do not require running on the database's primary shard, these operations can route the results to any shard to merge the results and avoid overloading the primary shard for that database. Aggregation operations that require running on the database's primary shard are the `$out` (page 656) stage and `$lookup` (page 654) stage.

Compatibility

For compatibility changes, see *Aggregation Compatibility Changes* (page 1001).

MongoDB Tools Enhancements

- `mongodump` (page 816) and `mongorestore` (page 824) add support for archive files and standard output/input streams with a new `--archive` option. This enhancement allows for the streaming of the dump data over a network device via a pipe. For examples, see
  - *mongodump to an Archive File* (page 822) and *mongodump an Archive to Standard Output* (page 823)
  - *Restore a Database from an Archive File* (page 832) and *Restore a Database from Standard Input* (page 832).
- `mongodump` (page 816) and `mongorestore` (page 824) add support for compressed data dumps with a new `--gzip` option. This enhancement reduces storage space for the dump files. For examples, see:
  - *Compress mongodump Output* (page 823)
  - *Restore from Compressed Data* (page 832).

Encrypted Storage Engine

Enterprise Feature

Available in MongoDB Enterprise only.

Important: Available for the WiredTiger Storage Engine only.

Encryption at rest, when used in conjunction with transport encryption and good security policies that protect relevant accounts, passwords, and encryption keys, can help ensure compliance with security and privacy standards, including HIPAA, PCI-DSS, and FERPA.

MongoDB Enterprise 3.2 introduces a native encryption option for the WiredTiger storage engine. This feature allows MongoDB to encrypt data files such that only parties with the decryption key can decode and read the data. For detail, see *encrypted-storage-engine*.

Text Search Enhancements

`text` Index Version 3

MongoDB 3.2 introduces a version 3 of the `text index`. Key features of the new version of the index are:

- Improved *case insensitivity*.
- *Diacritic insensitivity*.
- Additional *delimiters for tokenization*.

Starting in MongoDB 3.2, version 3 is the default version for new `text` indexes.

See also:

Text Index Version 3 Compatibility (page 1001)

`$text` Operator Enhancements

The `$text` (page 549) operator adds support for:

- *case sensitive text search* (page 552) with the new `$caseSensitive` option, and
- *diacritic sensitive text search* (page 552) with the new `$diacriticSensitive` option.

For more information and examples, see the `$text` (page 549) operator reference sections *Case Insensitivity* (page 552) and *Diacritic Insensitivity* (page 552).

Support for Additional Languages

Enterprise Feature

Available in MongoDB Enterprise only.

Starting in 3.2, MongoDB Enterprise provides support for the following languages: Arabic, Farsi (specifically Dari and Iranian Persian dialects), Urdu, Simplified Chinese, and Traditional Chinese.

For details, see `https://docs.mongodb.org/manual/tutorial/text-search-with-rlp`.

New Storage Engines

`inMemory` Storage Engine

Enterprise Feature

Available in MongoDB Enterprise only.

MongoDB Enterprise 3.2 provides an in-memory storage engine. Other than some metadata, the in-memory storage engine does not maintain any on-disk data. By avoiding disk I/O, the in-memory storge engine allows for more predictable latency of database operations.

Warning: Currently in beta. Do not use in production.

To select this storage engine, specify

- `inMemory` for the *`--storageEngine`* (page 775) option or the `storage.engine` (page 917) setting.
- `--dbpath`. Although the in-memory storage engine does not write data to the filesystem, it maintains in the `--dbpath` small metadata files and diagnostic data as well temporary files for building large indexes.

The `inMemory` storage engine uses document-level locking. For more details, see `https://docs.mongodb.org/manual/core/inmemory`.

`ephemeralForTest` Storage Engine

MongoDB 3.2 provides a new for-test storage engine. Other than some metadata, the for-test storage engine does not maintain any on-disk data, removing the need to clean up between test runs. The for-test storage engine is unsupported.

Warning: For test purposes only. Do not use in production.

To select this storage engine, specify

- `ephemeralForTest` for the *`--storageEngine`* (page 775) option or the `storage.engine` (page 917) setting.
- `--dbpath`. Although the for-test storage engine does not write data to the filesystem, it maintains small metadata files in the `--dbpath`.

The `ephemeralForTest` storage engine uses collection-level locking.

General Enhancements

Bit Test Query Operators

MongoDB 3.2 provides new query operators to test bit values:

- `$bitsAllSet` (page 581)
- `$bitsAllClear` (page 584)
- `$bitsAnySet` (page 582)
- `$bitsAnyClear` (page 585)

SpiderMonkey JavaScript Engine

MongoDB 3.2 uses SpiderMonkey as the JavaScript engine for the `mongo` (page 803) shell and `mongod` (page 770) server. SpiderMonkey provides support for additional platforms and has an improved memory management model.

This change affects all JavaScript behavior including the commands `mapReduce` (page 318), `group` (page 313), and the query operator `$where` (page 558); *however*, this change should be completely transparent to the user.

See also:

SpiderMonkey Compatibility Changes (page 1001)

`mongo` Shell and CRUD API

To provide consistency with the MongoDB drivers' CRUD (Create/Read/Update/Delete) API, the `mongo` (page 803) shell introduces additional CRUD methods that are consistent with the drivers' CRUD API:

New API	Description
`db.collection.bulkWrite()` (page 25)	Equivalent to initializing `Bulk()` (page 210) operations builder, using *Bulk methods* (page 211) to add operations, and running `Bulk.execute()` (page 223) to execute the operations. MongoDB 3.2 deprecates `Bulk()` (page 210) and its associated *methods* (page 209).
`db.collection.deleteMany()` (page 42)	Equivalent to `db.collection.remove()` (page 101).
`db.collection.deleteOne()` (page 40)	Equivalent to `db.collection.remove()` (page 101) with the `justOne` set to true; i.e. `db.collection.remove( <query>, true )` (page 101) or `db.collection.remove( <query>, { justOne: true } )` (page 101).
`db.collection.findOneAndDelete()` (page 64)	Equivalent to `db.collection.findAndModify()` (page 57) method with `remove` set to true.
`db.collection.findOneAndReplace()` (page 66)	Equivalent to `db.collection.findAndModify()` (page 57) method with `update` set to a replacement document.
`db.collection.findOneAndUpdate()` (page 69)	Equivalent to `db.collection.findAndModify()` (page 57) method with `update` set to a document that specifies modifications using *update operators* (page 594).
`db.collection.insertMany()` (page 85)	Equivalent to `db.collection.insert()` (page 79) method with an array of documents as the parameter.
`db.collection.insertOne()` (page 82)	Equivalent to `db.collection.insert()` (page 79) method with a single document as the parameter.
`db.collection.replaceOne()` (page 98)	Equivalent to `db.collection.update( <query>, <update> )` (page 117) method with a replacement document as the `<update>` parameter.
`db.collection.updateMany()` (page 128)	Equivalent to `db.collection.update( <query>, <update>, { multi: true, ... })` (page 117) method with an `<update>` document that specifies modifications using *update operators* (page 594) and the `multi` option set to true.
`db.collection.updateOne()` (page 125)	Equivalent to `db.collection.update( <query>, <update> )` (page 117) method with an `<update>` document that specifies modifications using *update operators* (page 594).

WiredTiger and `fsyncLock`

Starting in MongoDB 3.2, the WiredTiger storage engine supports the `fsync` (page 451) command with the `lock` option or the `mongo` (page 803) shell method `db.fsyncLock()` (page 180). That is, for the WiredTiger storage engine, these operations can guarantee that the data files do not change, ensuring consistency for the purposes of creating backups.

Platform Support

Starting in 3.2, 32-bit binaries are deprecated and will be unavailable in future releases.

MongoDB 3.2 deprecates support for Red Hat Enterprise Linux 5.

`$type` Operator Accepts String Aliases

`$type` (page 540) operator accepts string aliases for the BSON types in addition to the numbers corresponding to the BSON types.

`explain()` Support for `distinct()` Operation

`db.collection.explain()` (page 48) adds support for `db.collection.distinct()` (page 44) method. For more information, see `db.collection.explain()` (page 48).

Deprecation of the HTTP Interface

Starting in 3.2, MongoDB deprecates its HTTP interface.

`keysExamined` Count Includes the Last Scanned Key

For `explain` operations run in `executionStats` or `allPlansExecution` mode, the *explain output* (page 946) contains the `keysExamined` statistic, representing the number of index keys examined during index scans.

Prior to 3.2, `keysExamined` count in some queries did not include the last scanned key. As of 3.2 this error has been corrected. For more information, see :data:‘ ~explain.executionStats.executionStages.inputStage.keysExamined‘.

The diagnostic logs and the system profiler report on this statistic.

Geospatial Optimization

MongoDB 3.2 introduces version 3 of `2dsphere indexes`, which index `GeoJSON geometries` at a finer gradation. The new version improves performance of `2dsphere index` queries over smaller regions. In addition, for both `2d indexes` and `2dsphere indexes`, the performance of geoNear queries has been improved for dense datasets.

See also:

2dsphere Index Version 3 Compatibility (page 1001)

Diagnostic Data Capture

To facilitate analysis of the MongoDB server behavior by MongoDB engineers, MongoDB 3.2 introduces a diagnostic data collection mechanism for logging server statistics to diagnostic files at periodic intervals. By default, the mechanism captures data at 1 second intervals. To modify the interval, see `diagnosticDataCollectionPeriodMillis` (page 937).

MongoDB creates a `diagnostic.data` directory under the `mongod` (page 770) instance's `--dbpath` or `storage.dbPath` (page 915). The diagnostic data is stored in files under this directory.

The maximum size of the diagnostic files is configurable with the `diagnosticDataCollectionFileSizeMB` (page 937), and the maximum size of the `diagnostic.data` directory is configurable with `diagnosticDataCollectionDirectorySizeMB` (page 936).

The default values for the capture interval and the maximum sizes are chosen to provide useful data to MongoDB engineers with minimal impact on performance and storage size. Typically, these values will only need modifications as requested by MongoDB engineers for specific diagnostic purposes.

Write Concern

- For replica sets using `protocolVersion: 1`, secondaries acknowlege write operations after the secondary members have written to their respective on-disk `journals`, regardless of the *j* option.

- For replica sets using `protocolVersion: 1`, `w: "majority"` implies *j: true*.
- With `j: true`, MongoDB returns only after the requested number of members, including the primary, have written to the journal. Previously `j: true` write concern in a replica set only requires the *primary* to write to the journal, regardless of the *w: <value>* write concern.

`journalCommitInterval` for WiredTiger

MongoDB 3.2 adds support for specifying the journal commit interval for the WiredTiger storage engine. See *journalCommitInterval* option. In previous versions, the option is applicable to MMAPv1 storage engine only.

For the corresponding configuration file setting, MongoDB 3.2 adds the `storage.journal.commitIntervalMs` (page 916) setting and deprecates `storage.mmapv1.journal.commitIntervalMs` (page 918). The deprecated `storage.mmapv1.journal.commitIntervalMs` (page 918) setting acts as an alias to the new `storage.journal.commitIntervalMs` (page 916) setting.

Changes Affecting Compatibility

Some MongoDB 3.2 changes can affect compatibility and may require user actions. For a detailed list of compatibility changes, see *Compatibility Changes in MongoDB 3.2* (page 1000).

Compatibility Changes in MongoDB 3.2

On this page

- Default Storage Engine Change (page 1000)
- Index Changes (page 1001)
- Aggregation Compatibility Changes (page 1001)
- SpiderMonkey Compatibility Changes (page 1001)
- Driver Compatibility Changes (page 1002)
- General Compatibility Changes (page 1003)
- Additional Information (page 1003)

The following 3.2 changes can affect the compatibility with older versions of MongoDB. See also *Release Notes for MongoDB 3.2* (page 979) for the list of the 3.2 changes.

Default Storage Engine Change Starting in 3.2, MongoDB uses the WiredTiger as the default storage engine. Previous versions used the MMAPv1 as the default storage engine.

For existing deployments, if you do not specify the `--storageEngine` or the `storage.engine` (page 917) setting, MongoDB automatically determines the storage engine used to create the data files in the `--dbpath` or `storage.dbPath` (page 915).

For new deployments, to use MMAPv1, you must explicitly specify the storage engine setting either:

- On the command line with the `--storageEngine` option:

  ```
  mongod --storageEngine mmapv1
  ```

- Or in a *configuration file* (page 895), using the `storage.engine` (page 917) setting:

```
storage:
   engine: mmapv1
```

Index Changes

Version 0 Indexes MongoDB 3.2 disallows the creation of version 0 indexes (i.e. `{v: 0}`). If version 0 indexes exist, MongoDB 3.2 outputs a warning log message, specifying the collection and the index.

Starting in MongoDB 2.0, MongoDB started automatically upgrading `v: 0` indexes during *initial sync*, `mongorestore` (page 824) or `reIndex` (page 460) operations.

If a version 0 index exists, you can use any of the aforementioned operations as well as drop and recreate the index to upgrade to the `v: 1` version.

For example, if upon startup, a warning message indicated that an index `index { v: 0, key: { x: 1.0 }, name: "x_1", ns: "test.legacyOrders" }` is a version 0 index, to upgrade to the appropriate version, you can drop and recreate the index:

1. Drop the index either by name:

   ```
   use test
   db.legacyOrders.dropIndex( "x_1" )
   ```

   or by key:

   ```
   use test
   db.legacyOrders.dropIndex( { x: 1 } )
   ```

2. Recreate the index without the version option `v`:

   ```
   db.legacyOrders.createIndex( { x: 1 } )
   ```

Text Index Version 3 Compatibility *Text index (version 3)* (page 995) is incompatible with earlier versions of MongoDB. Earlier versions of MongoDB will not start if `text index (version 3)` exists in the database.

`2dsphere` Index Version 3 Compatibility *2dsphere index (version 3)* (page 999) is incompatible with earlier versions of MongoDB. Earlier versions of MongoDB will not start if `2dsphere` index (version 3) exists in the database.

Aggregation Compatibility Changes

- `$avg` (page 732) accumulator returns null when run against a non-existent field. Previous versions returned `0`.
- `$substr` (page 697) errors when the result is an invalid UTF-8. Previous versions output the invalid UTF-8 result.
- Array elements are no longer treated as literals in the aggregation pipeline. Instead, each element of an array is now parsed as an expression. To treat the element as a literal instead of an expression, use the `$literal` (page 712) operator to create a literal value.

SpiderMonkey Compatibility Changes

JavaScript Changes in MongoDB 3.2

On this page

- Modernized JavaScript Implementation (ES6) (page 1002)
- Changes to the `mongo` Shell (page 1002)
- Removed Non-Standard V8 Features (page 1002)

In MongoDB 3.2, the javascript engine used for both the `mongo` (page 803) shell and for server-side javascript in `mongod` (page 770) changed from V8 to SpiderMonkey[192].

To confirm which JavaScript engine you are using, you can use either `interpreterVersion()` method in the `mongo` (page 803) shell and the `javascriptEngine` (page 472) field in the output of `db.serverBuildInfo()` (page 196)

In MongoDB 3.2, this will appear as `MozJS-38` and `mozjs`, respectively.

Modernized JavaScript Implementation (ES6) SpiderMonkey brings with it increased support for features defined in the 6th edition of ECMAScript[193], abbreviated as ES6. ES6 adds many new language features, including:

- arrow functions[194],
- destructuring assignment[195],
- for-of loops[196], and
- generators[197].

Changes to the `mongo` Shell MongoDB 3.2 will return JavaScript and BSON `undefined` (page 962) values intact if saved into a collection. Previously, the `mongo` (page 803) shell would convert `undefined` values into `null`.

MongoDB 3.2 also adds the `disableJavaScriptJIT` (page 933) parameter to `mongod` (page 770), which allows you to disable the JavaScript engine's JIT acceleration. The `mongo` (page 803) shell has a corresponding *`--disableJavaScriptJIT`* (page 805) flag.

Removed Non-Standard V8 Features SpiderMonkey does **not** allow the non-standard `Error.captureStackTrace()` function that prior versions of MongoDB supported. If you must record stack traces, you can capture the `Error().stack` property as a workaround.

MongoDB 3.2 changes the JavaScript engine from V8 to SpiderMonkey. The change allows the use of more modern JavaScript language features, and comes along with minor `mongo` (page 803) shell improvements and compatibility changes.

See *JavaScript Changes in MongoDB 3.2* (page 1002) for more information about this change.

Driver Compatibility Changes A driver upgrade is necessary to support `find` (page 334) and `getMore` (page 354).

[192] https://developer.mozilla.org/en-US/docs/SpiderMonkey
[193] http://www.ecma-international.org/ecma-262/6.0/index.html
[194] http://www.ecma-international.org/ecma-262/6.0/index.html#sec-arrow-function-definitions
[195] http://www.ecma-international.org/ecma-262/6.0/index.html#sec-destructuring-assignment
[196] http://www.ecma-international.org/ecma-262/6.0/index.html#sec-for-in-and-for-of-statements
[197] http://www.ecma-international.org/ecma-262/6.0/index.html#sec-generator-function-definitions

General Compatibility Changes

- In MongoDB 3.2, `cursor.showDiskLoc()` is deprecated in favor of `cursor.showRecordId()` (page 154), and both return a new document format.
- MongoDB 3.2 renamed the `serverStatus.repl.slaves` field to to `repl.replicationProgress` (page 503). See: the **db.serverStatus()** *repl* (page 502) reference for more information.
- The default changed from `--moveParanoia` (page 783) to `--noMoveParanoia` (page 783).

Additional Information See also *Release Notes for MongoDB 3.2* (page 979).

Upgrade Process

Upgrade MongoDB to 3.2

On this page

- Upgrade Recommendations and Checklists (page 1003)
- Upgrade Standalone `mongod` Instance to MongoDB 3.2 (page 1003)
- Upgrade a Replica Set to 3.2 (page 1004)
- Upgrade a Sharded Cluster to 3.2 (page 1005)
- Additional Resources (page 1007)

Before you attempt any upgrade, please familiarize yourself with the content of this document.

If you need guidance on upgrading to 3.2, MongoDB offers 3.2 upgrade services[198] to help ensure a smooth transition without interruption to your MongoDB application.

Upgrade Recommendations and Checklists When upgrading, consider the following:

Upgrade Requirements To upgrade an existing MongoDB deployment to 3.2, you must be running a 3.0-series release.

To upgrade from a 2.6-series release, you *must* upgrade to the latest 3.0-series release before upgrading to 3.2. For the procedure to upgrade from the 2.6-series to a 3.0-series release, see *Upgrade MongoDB to 3.0* (page 1054).

Preparedness Before beginning your upgrade, see the *Compatibility Changes in MongoDB 3.2* (page 1000) document to ensure that your applications and deployments are compatible with MongoDB 3.2. Resolve the incompatibilities in your deployment before starting the upgrade.

Before upgrading MongoDB, always test your application in a staging environment before deploying the upgrade to your production environment.

Upgrade Standalone `mongod` Instance to MongoDB 3.2 The following steps outline the procedure to upgrade a standalone `mongod` (page 770) from version 3.0 to 3.2. To upgrade from version 2.6 to 3.2, *upgrade to the latest 3.0-series release* (page 1054) *first*, and then use the following procedure to upgrade from 3.0 to 3.2.

[198] https://www.mongodb.com/contact/mongodb-3-2-upgrade-services?jmp=docs

Upgrade with Package Manager If you installed MongoDB from the MongoDB `apt`, `yum`, `dnf`, or `zypper` repositories, you should upgrade to 3.2 using your package manager. Follow the appropriate `installation instructions` for your Linux system. This will involve adding a repository for the new release, then performing the actual upgrade.

Manual Upgrade Otherwise, you can manually upgrade MongoDB:

Step 1: Download 3.2 binaries. Download binaries of the latest release in the 3.2 series from the MongoDB Download Page[199]. See `https://docs.mongodb.org/manual/installation` for more information.

Step 2: Replace with 3.2 binaries Shut down your `mongod` (page 770) instance. Replace the existing binary with the 3.2 `mongod` (page 770) binary and restart `mongod` (page 770).

Note: MongoDB 3.2 generates core dumps on some `mongod` (page 770) failures. For production environments, you may prefer to turn off core dumps for the operating system, if not already.

Upgrade a Replica Set to 3.2

Prerequisites All replica set members must be running version 3.0 before you can upgrade them to version 3.2. To upgrade a replica set from an earlier MongoDB version, *upgrade all members of the replica set to the latest 3.0-series release* (page 1054) *first*, and then follow the procedure to upgrade from MongoDB 3.0 to 3.2.

Upgrade Binaries You can upgrade from MongoDB 3.0 to 3.2 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 replica set one at a time:

- Shut down the `mongod` (page 770) instance and replace the 3.0 binary with the 3.2 binary.
- Restart the member and wait for the member to recover to `SECONDARY` state before upgrading the next secondary member. To check the member's state, issue `rs.status()` (page 262) in the `mongo` (page 803) shell.

Step 2: Step down the replica set primary. Connect a `mongo` (page 803) shell to the primary and use `rs.stepDown()` (page 263) to step down the primary and force an election of a new primary:

Step 3: Upgrade the primary. When `rs.status()` (page 262) shows that the primary has stepped down and another member has assumed `PRIMARY` state, upgrade the stepped-down primary:

- Shut down the stepped-down primary and replace the `mongod` (page 770) binary with the 3.2 binary.
- Restart.

[199] http://www.mongodb.org/downloads?jmp=docs

Step 4: Upgrade the replication protocol. Connect a `mongo` (page 803) shell to the current primary and upgrade the replication protocol

```
cfg = rs.conf();
cfg.protocolVersion=1;
rs.reconfig(cfg);
```

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.

Note: MongoDB 3.2 generates core dumps on some `mongod` (page 770) failures. For production environments, you may prefer to turn off core dumps for the operating system, if not already.

Upgrade a Sharded Cluster to 3.2

Prerequisites

- **Version 3.0 or Greater** To upgrade a sharded cluster to 3.2, **all** members of the cluster must be at least version 3.0. The upgrade process checks all components of the cluster and will produce warnings if any component is running version earlier than 3.0.
- **Stop Metadata Changes during the Upgrade** During the upgrade, ensure that clients do not make changes to the collection metadata. For example, during the upgrade, do **not** perform any of the following operations:
  - `sh.enableSharding()` (page 273)
  - `sh.shardCollection()` (page 277)
  - `sh.addShard()` (page 269)
  - `db.createCollection()` (page 167)
  - `db.collection.drop()` (page 45)
  - `db.dropDatabase()` (page 177)
  - any operation that creates a database
  - any other operation that modifies the cluster metadata in any way.

  See the `https://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Not all commands on the `https://docs.mongodb.org/manual/reference/sharding` page modify the cluster metadata.
- *Disable the balancer*
- **Back up the `config` Database** *Optional but Recommended.* As a precaution, take a backup of the `config` database *before* upgrading the sharded cluster.

Upgrade Binaries

Step 1: Disable the Balancer. Disable the balancer as described in *sharding-balancing-disable-temporarily*.

Step 2: Upgrade the shards. Upgrade the shards one at a time. If the shards are replica sets, for each shard:

1. Upgrade the *secondary* members of the replica set one at a time:
   - Shut down the `mongod` (page 770) instance and replace the 3.0 binary with the 3.2 binary.
   - Restart the member and wait for the member to recover to `SECONDARY` state before upgrading the next secondary member. To check the member's state, issue `rs.status()` (page 262) in the `mongo` (page 803) shell.
2. Step down the replica set primary.

   Connect a `mongo` (page 803) shell to the primary and use `rs.stepDown()` (page 263) to step down the primary and force an election of a new primary:

   ```
   rs.stepDown()
   ```
3. When `rs.status()` (page 262) shows that the primary has stepped down and another member has assumed `PRIMARY` state, upgrade the stepped-down primary:
   - Shut down the stepped-down primary and replace the `mongod` (page 770) binary with the 3.2 binary.
   - Restart.
4. Connect a `mongo` (page 803) shell to the current primary and upgrade the `replication protocol` for the shard:

   ```
   cfg = rs.conf();
   cfg.protocolVersion=1;
   rs.reconfig(cfg);
   ```

Step 3: Upgrade the config servers. Upgrade the config servers one at a time in reverse order of the `configDB` (page 926) or `--configdb` (page 795) setting for the `mongos` (page 792). That is, if the `mongos` (page 792) has the following `--configdb` (page 795) listing:

```
mongos --configdb confserver1:port1,confserver2:port2,confserver3:port2
```

Upgrade first `confserver3`, then `confserver2`, and lastly `confserver1`.

Starting with the last config server listed in the `configDB` (page 926) setting:

1. Stop the config server and replace with the 3.2 binary.
2. Start the 3.2 binary with both the `--configsvr` and `--port` options:

   ```
   mongod --configsvr --port <port> --dbpath <path>
   ```

   If using a *configuration file* (page 895), specify `sharding.clusterRole: configsvr` (page 922) and `net.port` (page 902) in the file:

   ```
   sharding:
      clusterRole: configsvr
   net:
      port: <port>
   storage:
      dbpath: <path>
   ```

Repeat for the config server listed *second* in the `configDB` (page 926) setting, and finally the config server listed *first* in the `configDB` (page 926) setting.

Step 4: Upgrade the `mongos` instances. Replace each `mongos` (page 792) instance with the 3.2 binary and restart.

```
mongos --configdb <cfgsvr1:port1>,<cfgsvr2:port2>,<cfgsvr3:port3>
```

Step 5: Re-enable the balancer. Re-enable the balancer as described in *sharding-balancing-enable*.

Note: MongoDB 3.2 generates core dumps on some `mongod` (page 770) failures. For production environments, you may prefer to turn off core dumps for the operating system, if not already.

Once the sharded cluster binaries have been upgraded to 3.2, existing config servers will continue to run as mirrored `mongod` (page 770) instances. Redeploying existing config servers as a replica set (*CSRS*) requires downtime. For instructions, see `https://docs.mongodb.org/manual/tutorial/upgrade-config-servers-to-replica-set`.

Additional Resources

- Getting ready for MongoDB 3.2? Get our help.[200]

Downgrade MongoDB from 3.2

On this page

- Downgrade Recommendations and Checklist (page 1007)
- Prerequisites (page 1008)
- Downgrade a Standalone `mongod` Instance (page 1008)
- Downgrade a 3.2 Replica Set (page 1009)
- Downgrade a 3.2 Sharded Cluster (page 1009)

Before you attempt any downgrade, familiarize yourself with the content of this document, particularly the *Downgrade Recommendations and Checklist* (page 1007) and the procedure for *downgrading sharded clusters* (page 1009).

Downgrade Recommendations and Checklist When downgrading, consider the following:

Downgrade Path To downgrade, use the latest version in the 3.0-series.

Preparedness

- *Remove or downgrade version 3 text indexes* (page 1008) before downgrading MongoDB 3.2 to 3.0.
- *Remove or downgrade version 3 2dsphere indexes* (page 1008) before downgrading MongoDB 3.2 to 3.0.

Procedures Follow the downgrade procedures:

- To downgrade sharded clusters, see *Downgrade a 3.2 Sharded Cluster* (page 1009).
- To downgrade replica sets, see *Downgrade a 3.2 Replica Set* (page 1009).
- To downgrade a standalone MongoDB instance, see *Downgrade a Standalone mongod Instance* (page 1008).

[200]https://www.mongodb.com/contact/mongodb-3-2-upgrade-services?jmp=docs

Prerequisites

Text Index Version Check If you have *version 3* text indexes (i.e. the default version for text indexes in MongoDB 3.2), drop the *version 3* 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 73) 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 3.

```
{
   "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" : 3
}
```

`2dsphere` Index Version Check If you have *version 3* `2dsphere` indexes (i.e. the default version for `2dsphere` indexes in MongoDB 3.2), drop the *version 3* `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 73) 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 3.

```
{
   "v" : 1,
   "key" : {
      "geo" : "2dsphere"
   },
   "name" : "geo_2dsphere",
   "ns" : "test.locations",
   "sparse" : true,
   "2dsphereIndexVersion" : 3
}
```

Downgrade a Standalone `mongod` Instance The following steps outline the procedure to downgrade a standalone `mongod` (page 770) from version 3.2 to 3.0.

Step 1: Download the latest 3.0 binaries. For the downgrade, use the latest release in the 3.0 series.

Step 2: Restart with the latest 3.0 `mongod` instance.

Important: If your `mongod` (page 770) instance is using the `WiredTiger` storage engine, you must include the

--storageEngine (page 775) option (or `storage.engine` (page 917) if using the configuration file) with the 3.0 binary.

Shut down your `mongod` (page 770) instance. Replace the existing binary with the downloaded `mongod` (page 770) binary and restart.

Downgrade a 3.2 Replica Set The following steps outline a "rolling" downgrade process for the replica set. The "rolling" downgrade process minimizes downtime by downgrading the members individually while the other members are available:

Step 1: Downgrade the protocolVersion. Connect a `mongo` (page 803) shell to the current primary and downgrade the replication protocol:

```
cfg = rs.conf();
cfg.protocolVersion=0;
rs.reconfig(cfg);
```

Step 2: 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 770). See *terminate-mongod-processes* for instructions on safely terminating `mongod` (page 770) processes.
2. Replace the 3.2 binary with the 3.0 binary and restart.

   Important: If your `mongod` (page 770) instance is using the `WiredTiger` storage engine, you must include the *--storageEngine* (page 775) option (or `storage.engine` (page 917) if using the configuration file) with the 3.0 binary.

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 262) method in the `mongo` (page 803) shell.

Step 3: Step down the primary. Use `rs.stepDown()` (page 263) in the `mongo` (page 803) shell to step down the *primary* and force the normal *failover* procedure.

```
rs.stepDown()
```

`rs.stepDown()` (page 263) expedites the failover procedure and is preferable to shutting down the primary directly.

Step 4: Replace and restart former primary mongod. When `rs.status()` (page 262) shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` (page 770) binary with the 3.0 binary and start the new instance.

Important: If your `mongod` (page 770) instance is using the `WiredTiger` storage engine, you must include the *--storageEngine* (page 775) option (or `storage.engine` (page 917) if using the configuration file) with the 3.0 binary.

Replica set failover is not instant but will render the set unavailable to writes and interrupt reads until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the downgrade during a predetermined maintenance window.

Downgrade a 3.2 Sharded Cluster

Requirements While the downgrade is in progress, you cannot make changes to the collection metadata. For example, during the downgrade, do **not** do any of the following:

- `sh.enableSharding()` (page 273)
- `sh.shardCollection()` (page 277)
- `sh.addShard()` (page 269)
- `db.createCollection()` (page 167)
- `db.collection.drop()` (page 45)
- `db.dropDatabase()` (page 177)
- any operation that creates a database
- any other operation that modifies the cluster meta-data in any way. See `https://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `https://docs.mongodb.org/manual/reference/sharding` page modifies the cluster meta-data.

Downgrade a Sharded Cluster with SCCC Config Servers

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 replica set shard:

1. Downgrade the protocolVersion.
2. Downgrade the `mongod` (page 770) secondaries *before* downgrading the primary.
3. To downgrade the primary, run `replSetStepDown` (page 405) and then downgrade.

For details on downgrading a replica set, see *Downgrade a 3.2 Replica Set* (page 1009).

Step 3: Downgrade the SCCC config servers. If the sharded cluster uses 3 mirrored `mongod` (page 770) instances for the config servers, downgrade all three instances in reverse order of their listing in the `--configdb` (page 795) option for `mongos` (page 792). For example, if `mongos` (page 792) has the following `--configdb` (page 795) listing:

```
--configdb confserver1,confserver2,confserver3
```

Downgrade first `confserver3`, then `confserver2`, and lastly, `confserver1`. If your `mongod` (page 770) instance is using the `WiredTiger` storage engine, you must include the `--storageEngine` (page 775) option (or `storage.engine` (page 917) if using the configuration file) with the 3.0 binary.

```
mongod --configsvr --dbpath <path> --port <port> --storageEngine <storageEngine>
```

Step 4: Downgrade the `mongos` instances. Downgrade the binaries and restart.

Step 5: Re-enable the balancer. Once the downgrade of sharded cluster components is complete, *re-enable the balancer*.

Downgrade a Sharded Cluster with CSRS Config Servers

Step 1: Disable the Balancer. Turn off the *balancer* in the sharded cluster, as described in *sharding-balancing-disable-temporarily*.

Step 2: Prepare CSRS Config Servers for downgrade. If the sharded cluster uses CSRS:

1. *Remove secondary members from the replica set* to have only a primary and two secondaries and only the primary can vote and be eligible to be primary; i.e. the other two members have 0 for votes and priority.

   Connect a mongo (page 803) shell to the primary and run:

   ```
   rs.reconfig(
      {
         "_id" : <name>,
         "configsvr" : true,
         "protocolVersion" : NumberLong(1),
         "members" : [
            {
               "_id" : 0,
               "host" : "<host1>:<port1>",
               "priority" : 1,
               "votes" : 1
            },
            {
               "_id" : 1,
               "host" : "<host2>:<port2>",
               "priority" : 0,
               "votes" : 0
            },
            {
               "_id" : 2,
               "host" : "<host3>:<port3>",
               "priority" : 0,
               "votes" : 0
            }
         ]
      }
   )
   ```

2. Step down the primary using replSetStepDown (page 405) against the admin database. Ensure enough time for the secondaries to catch up.

   Connect a mongo (page 803) shell to the primary and run:

   ```
   db.adminCommand( { replSetStepDown: 360, secondaryCatchUpPeriodSecs: 300, force: true })
   ```

3. Shut down all members of the config server replica set, the mongos (page 792) instances, and the shards.

4. Restart each config server as standalone 3.2 mongod (page 770); i.e. without the *--replSet* (page 780) or, if using a configuration file, replication.replSetName (page 922).

   ```
   mongod --configsvr --dbpath <path> --port <port> --storageEngine <storageEngine>
   ```

Step 3: Update the protocolVersion for each shard. Restart each replica set shard and update the protocolVersion.

Connect a mongo (page 803) shell to the current primary and downgrade the replication protocol:

```
cfg = rs.conf();
cfg.protocolVersion=0;
rs.reconfig(cfg);
```

Step 4: Downgrade the mongos instances.

Important: As the config servers changed from a replica set to three mirrored mongod (page 770) instances, update the the *--configsvr* (page 782) setting. All mongos (page 792) must use the same *--configsvr* (page 782) string.

Downgrade the binaries and restart.

Step 5: Downgrade Config Servers. Downgrade the binaries and restart. Downgrade in reverse order of their listing in the *--configdb* (page 795) option for mongos (page 792).

If your mongod (page 770) instance is using the WiredTiger storage engine, you must include the *--storageEngine* (page 775) option (or storage.engine (page 917) if using the configuration file) with the 3.0 binary.

```
mongod --configsvr --dbpath <path> --port <port> --storageEngine <storageEngine>
```

Step 6: Downgrade each shard, one at a time. For each replica set shard, downgrade the mongod (page 770) binaries and restart. If your mongod (page 770) instance is using the WiredTiger storage engine, you must include the *--storageEngine* (page 775) option (or storage.engine (page 917) if using the configuration file) with the 3.0 binary.

1. Downgrade the mongod (page 770) secondaries *before* downgrading the primary.
2. To downgrade the primary, run replSetStepDown (page 405) and then downgrade.

For details on downgrading a replica set, see *Downgrade a 3.2 Replica Set* (page 1009).

Step 7: Re-enable the balancer. Once the downgrade of sharded cluster components is complete, *re-enable the balancer*.

See *Upgrade MongoDB to 3.2* (page 1003) for full upgrade instructions.

Known Issues in 3.2.1

List of known issues in the 3.2.1 release:

- Clients may fail to discover new primaries when clock skew between nodes is greater than electionTimeout: SERVER-21744[201]
- fromMigrate flag never set for deletes in oplog: SERVER-21678[202]
- Running explain (page 467) with a read preference in a v3.2 mongo shell connected to a v3.0 mongos or in a v3.0 mongo shell connected to a v3.0 mongos but with v3.2 shards is incompatible: SERVER-21661[203]
- Results of the connPoolStats command are no longer correct: SERVER-21597[204]
- ApplyOps background index creation may deadlock: SERVER-21583[205]
- Performance regression for w:majority writes with replica set protocolVersion 1: SERVER-21581[206]
- Performance regression on unicode-aware text processing logic (text index v3): SERVER-19936[207]

[201] https://jira.mongodb.org/browse/SERVER-21744
[202] https://jira.mongodb.org/browse/SERVER-21678
[203] https://jira.mongodb.org/browse/SERVER-21661
[204] https://jira.mongodb.org/browse/SERVER-21597
[205] https://jira.mongodb.org/browse/SERVER-21583
[206] https://jira.mongodb.org/browse/SERVER-21581
[207] https://jira.mongodb.org/browse/SERVER-19936

Known Issues in 3.2.0

List of known issues in the 3.2.0 release:

- findAndModify operations not captured by the profiler: SERVER-21772[208]
- `getMore` (page 354) command does not set `"nreturned"` operation counter: SERVER-21750[209]
- Clients may fail to discover new primaries when clock skew between nodes is greater than `electionTimeout`: SERVER-21744[210]
- `fromMigrate` flag never set for deletes in oplog: SERVER-21678[211]
- Running `explain` (page 467) with a `read preference` in a v3.2 `mongo` shell connected to a v3.0 `mongos` or in a v3.0 `mongo` shell connected to a v3.0 `mongos` but with v3.2 shards is incompatible: SERVER-21661[212]
- Results of the connPoolStats command are no longer correct: SERVER-21597[213]
- ApplyOps background index creation may deadlock: SERVER-21583[214]
- Performance regression for `w:majority` writes with replica set protocolVersion 1: SERVER-21581[215]
- Performance regression on unicode-aware text processing logic (text index v3): SERVER-19936[216]
- Severe performance regression in insert workload under Windows with WiredTiger: SERVER-21792[217]

Download

To download MongoDB 3.2, go to the downloads page[218].

See also:

- All Third Party License Notices[219].
- All JIRA issues resolved in 3.2[220].

Additional Resources

- Getting ready for MongoDB 3.2? Get our help.[221]

7.2 Previous Stable Releases

7.2.1 Release Notes for MongoDB 3.0

[208] https://jira.mongodb.org/browse/SERVER-21772
[209] https://jira.mongodb.org/browse/SERVER-21750
[210] https://jira.mongodb.org/browse/SERVER-21744
[211] https://jira.mongodb.org/browse/SERVER-21678
[212] https://jira.mongodb.org/browse/SERVER-21661
[213] https://jira.mongodb.org/browse/SERVER-21597
[214] https://jira.mongodb.org/browse/SERVER-21583
[215] https://jira.mongodb.org/browse/SERVER-21581
[216] https://jira.mongodb.org/browse/SERVER-19936
[217] https://jira.mongodb.org/browse/SERVER-21792
[218] http://www.mongodb.org/downloads
[219] https://github.com/mongodb/mongo/blob/v3.2/distsrc/THIRD-PARTY-NOTICES
[220] http://bit.ly/1XXomL9
[221] https://www.mongodb.com/contact/mongodb-3-2-upgrade-services?jmp=docs

On this page

- Minor Releases (page 1014)
- Major Changes (page 1041)
- Replica Sets (page 1043)
- Sharded Clusters (page 1044)
- Security Improvements (page 1044)
- Improvements (page 1044)
- MongoDB Enterprise Features (page 1046)
- Additional Information (page 1046)
- Additional Resources (page 1068)

March 3, 2015

MongoDB 3.0 is now available. Key features include support for the WiredTiger storage engine, pluggable storage engine API, `SCRAM-SHA-1` authentication mechanism, and improved `explain` functionality.

MongoDB Ops Manager, which includes Automation, Backup, and Monitoring, is now also available. See the Ops Manager documentation[222] and the Ops Manager release notes[223] for more information.

Minor Releases

3.0 Changelog

On this page

- 3.0.9 Changelog (page 1014)
- 3.0.8 Changelog (page 1016)
- 3.0.7 Changelog (page 1018)
- 3.0.6 Changelog (page 1021)
- 3.0.5 Changelog (page 1023)
- 3.0.4 Changelog (page 1026)
- 3.0.3 Changelog (page 1028)
- 3.0.2 Changelog (page 1032)
- 3.0.1 Changelog (page 1035)

3.0.9 Changelog

Security SERVER-21724[224] Backup role can't read system.profile

Sharding

- SERVER-19266[225] An error document is returned with result set
- SERVER-21382[226] Sharding migration transfers all document deletions

[222] http://docs.opsmanager.mongodb.com/current/
[223] http://docs.opsmanager.mongodb.com/current/release-notes/application/
[224] https://jira.mongodb.org/browse/SERVER-21724
[225] https://jira.mongodb.org/browse/SERVER-19266
[226] https://jira.mongodb.org/browse/SERVER-21382

- SERVER-22114[227] Mongos can accumulate multiple copies of ChunkManager when a shard restarts

Replication

- SERVER-18219[228] “control reaches end of non-void function” errors in GCC with WCE retry loop
- SERVER-21583[229] ApplyOps background index creation may deadlock
- SERVER-22109[230] Invariant failure when running applyOps to create an index with a bad ns field

Query

- SERVER-19128[231] Fatal assertion during secondary index build
- SERVER-19996[232] Queries which specify sort and batch size can generate results out of order, if documents concurrently updated
- SERVER-20083[233] Add log statement at default log level for when an index filter is set or cleared successfully
- SERVER-21602[234] Reduce execution time of cursor_timeout.js
- SERVER-21776[235] Move per-operation log lines for queries out of the QUERY log component

Write Operations SERVER-21647[236] $rename changes field ordering

Aggregation SERVER-7656[237] Optimize aggregation on sharded setup if first stage is exact match on shard key

Storage

- SERVER-20858[238] Invariant failure in OplogStones for non-capped oplog creation
- SERVER-20866[239] Race condition in oplog insert transaction rollback
- SERVER-21545[240] collMod and invalid parameter triggers fassert on dropCollection on mmapv1
- SERVER-22014[241] index_bigkeys_nofail.js triggers spurious failures when run in parallel with other tests

WiredTiger

- SERVER-20961[242] Large amounts of create and drop collections can cause listDatabases to be slow under WiredTiger
- SERVER-22129[243] WiredTiger changes for MongoDB 3.0.9

[227] https://jira.mongodb.org/browse/SERVER-22114
[228] https://jira.mongodb.org/browse/SERVER-18219
[229] https://jira.mongodb.org/browse/SERVER-21583
[230] https://jira.mongodb.org/browse/SERVER-22109
[231] https://jira.mongodb.org/browse/SERVER-19128
[232] https://jira.mongodb.org/browse/SERVER-19996
[233] https://jira.mongodb.org/browse/SERVER-20083
[234] https://jira.mongodb.org/browse/SERVER-21602
[235] https://jira.mongodb.org/browse/SERVER-21776
[236] https://jira.mongodb.org/browse/SERVER-21647
[237] https://jira.mongodb.org/browse/SERVER-7656
[238] https://jira.mongodb.org/browse/SERVER-20858
[239] https://jira.mongodb.org/browse/SERVER-20866
[240] https://jira.mongodb.org/browse/SERVER-21545
[241] https://jira.mongodb.org/browse/SERVER-22014
[242] https://jira.mongodb.org/browse/SERVER-20961
[243] https://jira.mongodb.org/browse/SERVER-22129

Operations SERVER-20358[244] Usernames can contain NULL characters

Build and Packaging

- SERVER-17747[245] FreeBSD 11.0-CURRENT build issue
- SERVER-18162[246] Fail to start with non-existing /var/run/mongodb/
- SERVER-18953[247] Generate debug symbols on OS X

Internals

- SERVER-18373[248] MONGO_COMPILER_UNREACHABLE should terminate if violated
- SERVER-19110[249] Ignore failed operations in mixed_storage_version_replication.js
- SERVER-21934[250] Add extra information to OSX stack traces to facilitate addr2line translation
- SERVER-21960[251] Include symbol name in stacktrace json when available
- SERVER-22013[252] coll_mod_bad_spec.js tries to pass filter to getCollectionInfos on v3.0 branch
- SERVER-22054[253] Authentication failure reports incorrect IP address
- SERVER-22191[254] Race condition in CurOp constructor (<=3.0 only)
- TOOLS-1002[255] oplog_rollover test is flaky

3.0.8 Changelog

Security SERVER-21278[256] Remove executable bit from mongod.lock

Sharding

- SERVER-20407[257] findAndModify on mongoS upserts to the wrong shard
- SERVER-20839[258] trace_missing_docs_test.js compares Timestamp instances using < operator in mongo shell

Query

- SERVER-2454[259] Queries that are killed during a yield should return error to user instead of partial result set
- SERVER-21227[260] MultiPlanStage::invalidate() should not flag and drop invalidated WorkingSetMembers

[244] https://jira.mongodb.org/browse/SERVER-20358
[245] https://jira.mongodb.org/browse/SERVER-17747
[246] https://jira.mongodb.org/browse/SERVER-18162
[247] https://jira.mongodb.org/browse/SERVER-18953
[248] https://jira.mongodb.org/browse/SERVER-18373
[249] https://jira.mongodb.org/browse/SERVER-19110
[250] https://jira.mongodb.org/browse/SERVER-21934
[251] https://jira.mongodb.org/browse/SERVER-21960
[252] https://jira.mongodb.org/browse/SERVER-22013
[253] https://jira.mongodb.org/browse/SERVER-22054
[254] https://jira.mongodb.org/browse/SERVER-22191
[255] https://jira.mongodb.org/browse/TOOLS-1002
[256] https://jira.mongodb.org/browse/SERVER-21278
[257] https://jira.mongodb.org/browse/SERVER-20407
[258] https://jira.mongodb.org/browse/SERVER-20839
[259] https://jira.mongodb.org/browse/SERVER-2454
[260] https://jira.mongodb.org/browse/SERVER-21227

- SERVER-21275[261] Document not found due to WT commit visibility issue

Storage

- SERVER-20650[262] Backport MongoRocks changes to 3.0
- SERVER-21543[263] Lengthen delay before deleting old journal files

WiredTiger

- SERVER-20303[264] Negative scaling at low thread count under WiredTiger when inserting large documents
- SERVER-21063[265] MongoDB with WiredTiger can build very deep trees
- SERVER-21442[266] WiredTiger changes for MongoDB 3.0.8
- SERVER-21553[267] Oplog grows to 3x configured size

Build and Packaging

- SERVER-10512[268] Add scons flag to set -fno-omit-frame-pointer
- SERVER-19755[269] scons should require c++11 on 3.0
- SERVER-20699[270] Add build manifest to every build
- SERVER-20830[271] set push and docs_tickets tasks as not available for patch testing
- SERVER-20834[272] Perf tasks should only require compiling once before execution
- SERVER-21209[273] PIDFILEPATH computation in init scripts fails to handle comments after values
- SERVER-21477[274] 3.0.7 RPMs missing for yum RHEL server versions

Tools

- TOOLS-702[275] bsondump does not keep attribut order
- TOOLS-920[276] mongodump issue with temporary map/reduce collections
- TOOLS-939[277] Error restoring database "insertion error: EOF"

[261] https://jira.mongodb.org/browse/SERVER-21275
[262] https://jira.mongodb.org/browse/SERVER-20650
[263] https://jira.mongodb.org/browse/SERVER-21543
[264] https://jira.mongodb.org/browse/SERVER-20303
[265] https://jira.mongodb.org/browse/SERVER-21063
[266] https://jira.mongodb.org/browse/SERVER-21442
[267] https://jira.mongodb.org/browse/SERVER-21553
[268] https://jira.mongodb.org/browse/SERVER-10512
[269] https://jira.mongodb.org/browse/SERVER-19755
[270] https://jira.mongodb.org/browse/SERVER-20699
[271] https://jira.mongodb.org/browse/SERVER-20830
[272] https://jira.mongodb.org/browse/SERVER-20834
[273] https://jira.mongodb.org/browse/SERVER-21209
[274] https://jira.mongodb.org/browse/SERVER-21477
[275] https://jira.mongodb.org/browse/TOOLS-702
[276] https://jira.mongodb.org/browse/TOOLS-920
[277] https://jira.mongodb.org/browse/TOOLS-939

Internals

- SERVER-8728[278] jstests/profile1.js is a race and fails randomly
- SERVER-20521[279] Update Mongo-perf display names in Evergreen to sort better
- SERVER-20527[280] Delete resmoke.py from the 3.0 branch
- SERVER-20876[281] Hang in scenario with sharded ttl collection under WiredTiger
- SERVER-21027[282] Reduced performance of index lookups after removing documents from collection
- SERVER-21099[283] Improve logging in SecureRandom and PseudoRandom classes
- SERVER-21150[284] Basic startup logging should be done as early as possible in initAndListen
- SERVER-21208[285] "server up" check in perf.yml is in the wrong place
- SERVER-21305[286] Lock 'timeAcquiringMicros' value is much higher than the actual time spent
- SERVER-21433[287] Perf.yml project should kill unwanted processes before starting tests
- SERVER-21533[288] Lock manager is not fair in the presence of compatible requests which can be granted immediately

3.0.7 Changelog

Security

- SERVER-13647[289] `root` role does not contain sufficient privileges for a `mongorestore` (page 824) of a system with security enabled
- SERVER-15893[290] `root` role should be able to run validate on system collections
- SERVER-19131[291] `clusterManager` role does not have permission for adding tag ranges
- SERVER-19284[292] Should not be able to create role with same name as builtin role
- SERVER-20394[293] Remove non-integer test case from `iteration_count_control.js`
- SERVER-20401[294] Publicly expose `net.ssl.disabledProtocols` (page 909)

Sharding

- SERVER-17886[295] dbKillCursors op asserts on mongos when at log level 3

[278] https://jira.mongodb.org/browse/SERVER-8728
[279] https://jira.mongodb.org/browse/SERVER-20521
[280] https://jira.mongodb.org/browse/SERVER-20527
[281] https://jira.mongodb.org/browse/SERVER-20876
[282] https://jira.mongodb.org/browse/SERVER-21027
[283] https://jira.mongodb.org/browse/SERVER-21099
[284] https://jira.mongodb.org/browse/SERVER-21150
[285] https://jira.mongodb.org/browse/SERVER-21208
[286] https://jira.mongodb.org/browse/SERVER-21305
[287] https://jira.mongodb.org/browse/SERVER-21433
[288] https://jira.mongodb.org/browse/SERVER-21533
[289] https://jira.mongodb.org/browse/SERVER-13647
[290] https://jira.mongodb.org/browse/SERVER-15893
[291] https://jira.mongodb.org/browse/SERVER-19131
[292] https://jira.mongodb.org/browse/SERVER-19284
[293] https://jira.mongodb.org/browse/SERVER-20394
[294] https://jira.mongodb.org/browse/SERVER-20401
[295] https://jira.mongodb.org/browse/SERVER-17886

- SERVER-20191[296] multi-updates/remove can make successive queries skip shard version checking
- SERVER-20460[297] `listIndexes` (page 450) on 3.0 `mongos` (page 792) with 2.6 `mongod` (page 770) instances returns erroneous “not authorized”
- SERVER-20557[298] Active window setting is not being processed correctly

Replication

- SERVER-20262[299] Replica set nodes can get stuck in a state where they will not step themselves down
- SERVER-20473[300] calling setMaintenanceMode(true) while running for election crashes server

Query

- SERVER-17895[301] Server should not clear collection plan cache periodically when write operations are issued
- SERVER-19412[302] NULL PlanStage in getStageByType causes segfault during stageDebug command
- SERVER-19725[303] NULL pointer crash in `QueryPlanner::plan` with `$near` (page 565) operator
- SERVER-20139[304] Enable CachedPlanStage replanning by default in 3.0
- SERVER-20219[305] Add startup warning to 3.0 if have indexes with partialFilterExpression option
- SERVER-20347[306] Document is not found when searching on a field indexed by a hash index using a `$in` (page 532) clause with regular expression
- SERVER-20364[307] Cursor is not closed when querying `system.profile` collection with `clusterMonitor` role

Write Operations

- SERVER-11746[308] Improve shard version checking for versioned (single) updates after yield
- SERVER-19361[309] Insert of document with duplicate `_id` fields should be forbidden
- SERVER-20531[310] Mongodb server crash: Invariant failure res.existing

Storage

- SERVER-18624[311] `listCollections` (page 438) command should not be O(n^2) on MMAPv1
- SERVER-20617[312] `wt_nojournal_toggle.js` failing intermittently in noPassthrough_WT

[296] https://jira.mongodb.org/browse/SERVER-20191
[297] https://jira.mongodb.org/browse/SERVER-20460
[298] https://jira.mongodb.org/browse/SERVER-20557
[299] https://jira.mongodb.org/browse/SERVER-20262
[300] https://jira.mongodb.org/browse/SERVER-20473
[301] https://jira.mongodb.org/browse/SERVER-17895
[302] https://jira.mongodb.org/browse/SERVER-19412
[303] https://jira.mongodb.org/browse/SERVER-19725
[304] https://jira.mongodb.org/browse/SERVER-20139
[305] https://jira.mongodb.org/browse/SERVER-20219
[306] https://jira.mongodb.org/browse/SERVER-20347
[307] https://jira.mongodb.org/browse/SERVER-20364
[308] https://jira.mongodb.org/browse/SERVER-11746
[309] https://jira.mongodb.org/browse/SERVER-19361
[310] https://jira.mongodb.org/browse/SERVER-20531
[311] https://jira.mongodb.org/browse/SERVER-18624
[312] https://jira.mongodb.org/browse/SERVER-20617

- SERVER-20638[313] Reading the profiling level shouldn't create databases that don't exist

WiredTiger

- SERVER-18250[314] Once enabled journal cannot be disabled under WiredTiger
- SERVER-20008[315] Stress test deadlock in WiredTiger
- SERVER-20091[316] Poor query throughput and erratic behavior at high connection counts under WiredTiger
- SERVER-20159[317] Out of memory on index build during initial sync even with low cacheSize parameter
- SERVER-20176[318] Deletes with `j:true` slower on WT than MMAPv1
- SERVER-20204[319] Segmentation fault during index build on 3.0 secondary

Operations

- SERVER-14750[320] Convert RPM and DEB mongod.conf files to new YAML format
- SERVER-18506[321] Balancer section of printShardingStatus should respect passed-in configDB

Build and Packaging

- SERVER-18516[322] ubuntu/debian packaging : Release files report wrong Codename
- SERVER-18581[323] The Ubuntu package should start the mongod with group=mongodb
- SERVER-18749[324] Ubuntu startup files have an inconsistent directory for dbpath and logs
- SERVER-18793[325] Enterprise RPM build issues
- SERVER-19088[326] The –cache flag should force –build-fast-and-loose=off
- SERVER-19509[327] The nproc ulimits are different across packages
- SERVER-19661[328] Build fails: error: expected expression

Tools

- TOOLS-767[329] `mongorestore` (page 824): error parsing metadata: call of reflect.Value.Set on zero Value
- TOOLS-847[330] `mongorestore` (page 824) exits in response to SIGHUP, even when run under nohup

[313]https://jira.mongodb.org/browse/SERVER-20638
[314]https://jira.mongodb.org/browse/SERVER-18250
[315]https://jira.mongodb.org/browse/SERVER-20008
[316]https://jira.mongodb.org/browse/SERVER-20091
[317]https://jira.mongodb.org/browse/SERVER-20159
[318]https://jira.mongodb.org/browse/SERVER-20176
[319]https://jira.mongodb.org/browse/SERVER-20204
[320]https://jira.mongodb.org/browse/SERVER-14750
[321]https://jira.mongodb.org/browse/SERVER-18506
[322]https://jira.mongodb.org/browse/SERVER-18516
[323]https://jira.mongodb.org/browse/SERVER-18581
[324]https://jira.mongodb.org/browse/SERVER-18749
[325]https://jira.mongodb.org/browse/SERVER-18793
[326]https://jira.mongodb.org/browse/SERVER-19088
[327]https://jira.mongodb.org/browse/SERVER-19509
[328]https://jira.mongodb.org/browse/SERVER-19661
[329]https://jira.mongodb.org/browse/TOOLS-767
[330]https://jira.mongodb.org/browse/TOOLS-847

- TOOLS-874[331] `mongoimport` (page 841) $date close to epoch not working
- TOOLS-916[332] `mongoexport` (page 850) throws reflect.Value.Type errors

Internals

- SERVER-18178[333] Fix `mr_drop.js` test to not fail from nondeterministic collection drop timing
- SERVER-19819[334] Update perf.yml to use new mongo-perf release
- SERVER-19820[335] Update perf.yml to use mongo-perf check script
- SERVER-19899[336] Mongo-perf analysis script – Check for per thread level regressions
- SERVER-19901[337] Mongo-perf analysis script – Compare to tagged baseline
- SERVER-19902[338] Mongo-perf analysis script – Use noise data for regression comparison instead of fixed percentage
- SERVER-20035[339] Updated perf_regresison_check.py script to output report.json summarizing results
- SERVER-20121[340] XorShift PRNG should use unsigned arithmetic
- SERVER-20216[341] Extend optional Command properties to SASL
- SERVER-20316[342] Relax thread level comparisons on mongo-perf check script
- SERVER-20322[343] Wiredtiger develop can lose records following stop even with log enabled
- SERVER-20383[344] Cleanup old connections after every ChunkManagerTest
- SERVER-20429[345] Canceled lock attempts should unblock pending requests
- SERVER-20464[346] Add units of measurement to log output of perf regression analysis
- SERVER-20691[347] Improve SASL and SCRAM compatibility
- TOOLS-894[348] `mongoimport --upsert --type json` with _id being an object does not work most of the times
- TOOLS-898[349] Mongo tools attempt to connect as ipv6 rather than ipv4 by default, when built with go 1.5

3.0.6 Changelog

[331] https://jira.mongodb.org/browse/TOOLS-874
[332] https://jira.mongodb.org/browse/TOOLS-916
[333] https://jira.mongodb.org/browse/SERVER-18178
[334] https://jira.mongodb.org/browse/SERVER-19819
[335] https://jira.mongodb.org/browse/SERVER-19820
[336] https://jira.mongodb.org/browse/SERVER-19899
[337] https://jira.mongodb.org/browse/SERVER-19901
[338] https://jira.mongodb.org/browse/SERVER-19902
[339] https://jira.mongodb.org/browse/SERVER-20035
[340] https://jira.mongodb.org/browse/SERVER-20121
[341] https://jira.mongodb.org/browse/SERVER-20216
[342] https://jira.mongodb.org/browse/SERVER-20316
[343] https://jira.mongodb.org/browse/SERVER-20322
[344] https://jira.mongodb.org/browse/SERVER-20383
[345] https://jira.mongodb.org/browse/SERVER-20429
[346] https://jira.mongodb.org/browse/SERVER-20464
[347] https://jira.mongodb.org/browse/SERVER-20691
[348] https://jira.mongodb.org/browse/TOOLS-894
[349] https://jira.mongodb.org/browse/TOOLS-898

Security SERVER-19538[350] Segfault when calling dbexit in SSLManager with auditing enabled

Querying

- SERVER-19553[351] Mongod shouldn't use sayPiggyBack to send KillCursor messages

Replication

- SERVER-19719[352] Failure to rollback noPadding should not cause fatal error
- SERVER-19644[353] Seg Fault on cloneCollection (specifically gridfs)

WiredTiger

- SERVER-19673[354] Excessive memory allocated by WiredTiger journal
- SERVER-19987[355] Limit the size of the per-session cursor cache
- SERVER-19751[356] WiredTiger panic halt in eviction-server
- SERVER-19744[357] WiredTiger changes for MongoDB 3.0.6
- SERVER-19573[358] MongoDb crash due to segfault
- SERVER-19522[359] Capped collection insert rate declines over time under WiredTiger

MMAPv1 SERVER-19805[360] MMap memory mapped file address allocation code cannot handle addresses non-aligned to memory mapped granularity size

Networking

- SERVER-19389[361] Remove wire level endianness check

Aggregation Framework

- SERVER-19553[362] Mongod shouldn't use sayPiggyBack to send KillCursor messages
- SERVER-19464[363] $sort stage in aggregation doesn't call scoped connections done ()

[350]https://jira.mongodb.org/browse/SERVER-19538
[351]https://jira.mongodb.org/browse/SERVER-19553
[352]https://jira.mongodb.org/browse/SERVER-19719
[353]https://jira.mongodb.org/browse/SERVER-19644
[354]https://jira.mongodb.org/browse/SERVER-19673
[355]https://jira.mongodb.org/browse/SERVER-19987
[356]https://jira.mongodb.org/browse/SERVER-19751
[357]https://jira.mongodb.org/browse/SERVER-19744
[358]https://jira.mongodb.org/browse/SERVER-19573
[359]https://jira.mongodb.org/browse/SERVER-19522
[360]https://jira.mongodb.org/browse/SERVER-19805
[361]https://jira.mongodb.org/browse/SERVER-19389
[362]https://jira.mongodb.org/browse/SERVER-19553
[363]https://jira.mongodb.org/browse/SERVER-19464

Build and Testing

- SERVER-19650[364] update YML files to tag system/test command types
- SERVER-19236[365] clang-format the v3.0 branch
- SERVER-19540[366] Add perf.yml file to 3.0 branch for mongo-perf regressions

Internal Code

- SERVER-19856[367] Register for PRESHUTDOWN notifications on Windows Vista+

Tools

`mongoexport` and `bsondump`

- TOOLS-848[368] Can't handle some regexes

mongoimport

- TOOLS-874[369] `mongoimport` (page 841) `$date` close to epoch not working

mongotop

- TOOLS-864[370] `mongotop` (page 867) "i/o timeout error"

3.0.5 Changelog

Querying

- SERVER-19489[371] Assertion failure and segfault in WorkingSet::free in 3.0.5-rc0
- SERVER-18461[372] Range predicates comparing against a BinData value should be covered, but are not in 2.6
- SERVER-17815[373] Plan ranking tie breaker is computed incorrectly
- SERVER-17259[374] Coverity analysis defect 56350: Dereference null return value
- SERVER-18926[375] Full text search extremely slow and uses a lot of memory under WiredTiger

[364] https://jira.mongodb.org/browse/SERVER-19650
[365] https://jira.mongodb.org/browse/SERVER-19236
[366] https://jira.mongodb.org/browse/SERVER-19540
[367] https://jira.mongodb.org/browse/SERVER-19856
[368] https://jira.mongodb.org/browse/TOOLS-848
[369] https://jira.mongodb.org/browse/TOOLS-874
[370] https://jira.mongodb.org/browse/TOOLS-864
[371] https://jira.mongodb.org/browse/SERVER-19489
[372] https://jira.mongodb.org/browse/SERVER-18461
[373] https://jira.mongodb.org/browse/SERVER-17815
[374] https://jira.mongodb.org/browse/SERVER-17259
[375] https://jira.mongodb.org/browse/SERVER-18926

Replication

- SERVER-19375[376] choosing syncsource should compare against last fetched optime rather than last applied
- SERVER-19298[377] Use userCreateNS w/options consistently in cloner
- SERVER-18994[378] producer thread can continue producing after a node becomes primary
- SERVER-18455[379] master/slave keepalives are not silent on slaves
- SERVER-18280[380] ReplicaSetMonitor should use electionId to avoid talking to old primaries
- SERVER-17689[381] Server crash during initial replication sync

Sharding SERVER-18955[382] mongoS doesn't set batch size (and keeps the old one, 0) on getMore if performed on first _cursor->more()

Storage

- SERVER-19283[383] WiredTiger changes for MongoDB 3.0.5
- SERVER-18874[384] Backport changes to RocksDB from mongo-partners repo
- SERVER-18838[385] DB fails to recover creates and drops after system crash
- SERVER-17370[386] Clean up storage engine-specific index and collection options
- SERVER-15901[387] Cleanup unused locks on the lock manager

WiredTiger

- SERVER-19513[388] Truncating a capped collection may not unindex deleted documents in WiredTiger
- SERVER-19283[389] WiredTiger changes for MongoDB 3.0.5
- SERVER-19189[390] Improve performance under high number of threads with WT
- SERVER-19178[391] In WiredTiger capped collection truncates, avoid walking lists of deleted items
- SERVER-19052[392] Remove sizeStorer recalculations at startup with WiredTiger
- SERVER-18926[393] Full text search extremely slow and uses a lot of memory under WiredTiger
- SERVER-18902[394] Retrieval of large documents slower on WiredTiger than MMAPv1

[376] https://jira.mongodb.org/browse/SERVER-19375
[377] https://jira.mongodb.org/browse/SERVER-19298
[378] https://jira.mongodb.org/browse/SERVER-18994
[379] https://jira.mongodb.org/browse/SERVER-18455
[380] https://jira.mongodb.org/browse/SERVER-18280
[381] https://jira.mongodb.org/browse/SERVER-17689
[382] https://jira.mongodb.org/browse/SERVER-18955
[383] https://jira.mongodb.org/browse/SERVER-19283
[384] https://jira.mongodb.org/browse/SERVER-18874
[385] https://jira.mongodb.org/browse/SERVER-18838
[386] https://jira.mongodb.org/browse/SERVER-17370
[387] https://jira.mongodb.org/browse/SERVER-15901
[388] https://jira.mongodb.org/browse/SERVER-19513
[389] https://jira.mongodb.org/browse/SERVER-19283
[390] https://jira.mongodb.org/browse/SERVER-19189
[391] https://jira.mongodb.org/browse/SERVER-19178
[392] https://jira.mongodb.org/browse/SERVER-19052
[393] https://jira.mongodb.org/browse/SERVER-18926
[394] https://jira.mongodb.org/browse/SERVER-18902

- SERVER-18875[395] Oplog performance on WT degrades over time after accumulation of deleted items
- SERVER-18838[396] DB fails to recover creates and drops after system crash
- SERVER-18829[397] Cache usage exceeds configured maximum during index builds under WiredTiger
- SERVER-18321[398] Speed up background index build with WiredTiger LSM
- SERVER-17689[399] Server crash during initial replication sync
- SERVER-17386[400] Cursor cache causes excessive memory utilization in WiredTiger
- SERVER-17254[401] WT: drop collection while concurrent oplog tailing may greatly reduce throughput
- SERVER-17078[402] show databases taking extraordinarily long with wiredTiger

Networking

- SERVER-19255[403] Listener::waitUntilListening may return before listening has started

Build and Packaging

- SERVER-18911[404] Update source tarball push
- SERVER-18910[405] Path in distribution does not contain version
- SERVER-18371[406] Add SSL library config detection
- SERVER-17782[407] Generate source tarballs with pre-interpolated version metadata files from SCons
- SERVER-17568[408] Report most-vexing parse warnings as errors on MSVC
- SERVER-17329[409] Improve management of server version in build system
- SERVER-18977[410] Initscript does not stop a running mongod daemon
- SERVER-18911[411] Update source tarball push

Shell

- SERVER-18795[412] db.printSlaveReplicationInfo()/rs.printSlaveReplicationInfo() can not work with ARBITER role

[395]https://jira.mongodb.org/browse/SERVER-18875
[396]https://jira.mongodb.org/browse/SERVER-18838
[397]https://jira.mongodb.org/browse/SERVER-18829
[398]https://jira.mongodb.org/browse/SERVER-18321
[399]https://jira.mongodb.org/browse/SERVER-17689
[400]https://jira.mongodb.org/browse/SERVER-17386
[401]https://jira.mongodb.org/browse/SERVER-17254
[402]https://jira.mongodb.org/browse/SERVER-17078
[403]https://jira.mongodb.org/browse/SERVER-19255
[404]https://jira.mongodb.org/browse/SERVER-18911
[405]https://jira.mongodb.org/browse/SERVER-18910
[406]https://jira.mongodb.org/browse/SERVER-18371
[407]https://jira.mongodb.org/browse/SERVER-17782
[408]https://jira.mongodb.org/browse/SERVER-17568
[409]https://jira.mongodb.org/browse/SERVER-17329
[410]https://jira.mongodb.org/browse/SERVER-18977
[411]https://jira.mongodb.org/browse/SERVER-18911
[412]https://jira.mongodb.org/browse/SERVER-18795

Logging and Diagnostics

- SERVER-19054[413] Don't be too chatty about periodic tasks taking a few ms
- SERVER-18979[414] Duplicate uassert & fassert codes
- SERVER-19382[415] mongod enterprise crash running as snmp sub-agent

Internal Code and Testing

- SERVER-19353[416] Compilation failure with GCC 5.1
- SERVER-19298[417] Use userCreateNS w/options consistently in cloner
- SERVER-19255[418] Listener::waitUntilListening may return before listening has started
- SERVER-17728[419] typeid(glvalue) warns on clang 3.6
- SERVER-17567[420] Unconditional export of parseNumberFromStringWithBase
- SERVER-19540[421] Add perf.yml file to 3.0 branch for mongo-perf regressions
- SERVER-18068[422] Coverity analysis defect 72413: Resource leak
- SERVER-17259[423] Coverity analysis defect 56350: Dereference null return value
- SERVER-15017[424] benchRun might return incorrect stats values
- SERVER-19525[425] use of wrong type for size count of rolling back insert

3.0.4 Changelog

Security

- SERVER-18475[426] `authSchemaUpgrade` (page 372) fails when the `system.users` (page 893) contains non `MONGODB-CR` users
- SERVER-18312[427] Upgrade PCRE to latest

Querying

- SERVER-18364[428] Ensure non-negation predicates get chosen over negation predicates for multikey index bounds construction
- SERVER-16265[429] Add query details to getmore entry in profiler and `db.currentOp()` (page 171)

[413] https://jira.mongodb.org/browse/SERVER-19054
[414] https://jira.mongodb.org/browse/SERVER-18979
[415] https://jira.mongodb.org/browse/SERVER-19382
[416] https://jira.mongodb.org/browse/SERVER-19353
[417] https://jira.mongodb.org/browse/SERVER-19298
[418] https://jira.mongodb.org/browse/SERVER-19255
[419] https://jira.mongodb.org/browse/SERVER-17728
[420] https://jira.mongodb.org/browse/SERVER-17567
[421] https://jira.mongodb.org/browse/SERVER-19540
[422] https://jira.mongodb.org/browse/SERVER-18068
[423] https://jira.mongodb.org/browse/SERVER-17259
[424] https://jira.mongodb.org/browse/SERVER-15017
[425] https://jira.mongodb.org/browse/SERVER-19525
[426] https://jira.mongodb.org/browse/SERVER-18475
[427] https://jira.mongodb.org/browse/SERVER-18312
[428] https://jira.mongodb.org/browse/SERVER-18364
[429] https://jira.mongodb.org/browse/SERVER-16265

- SERVER-15225[430] `CachedPlanStage` should execute for trial period and re-plan if query performs poorly
- SERVER-13875[431] `ensureIndex()` (page 47) of `2dsphere` index breaks after upgrading to 2.6 (with the new `createIndex` command)

Replication

- SERVER-18566[432] Primary member can trip fatal assertion if stepping down while running findAndModify op resulting in an upsert
- SERVER-18511[433] Report upstream progress when initial sync completes
- SERVER-18409[434] Retry failed heartbeats before marking a node as DOWN
- SERVER-18326[435] Rollback attempted during initial sync is fatal
- SERVER-17923[436] Creating/dropping multiple background indexes on the same collection can cause fatal error on secondaries
- SERVER-17913[437] New primary should log voters at default log level
- SERVER-17807[438] drain ops before restarting initial sync
- SERVER-15252[439] Write unit tests of ScatterGatherRunner
- SERVER-15192[440] Make all logOp listeners rollback-safe
- SERVER-18190[441] Secondary reads block replication

Sharding

- SERVER-18822[442] Sharded clusters with WiredTiger primaries may lose writes during chunk migration
- SERVER-18246[443] getmore on secondary in recovery mode can crash `mongos` (page 792)

Storage SERVER-18442[444] better error message when attempting to change storage engine metadata options

WiredTiger

- SERVER-18647[445] WiredTiger changes for MongoDB 3.0.4
- SERVER-18646[446] Avoid WiredTiger checkpointing dead handles
- SERVER-18629[447] WiredTiger journal system syncs wrong directory

[430] https://jira.mongodb.org/browse/SERVER-15225
[431] https://jira.mongodb.org/browse/SERVER-13875
[432] https://jira.mongodb.org/browse/SERVER-18566
[433] https://jira.mongodb.org/browse/SERVER-18511
[434] https://jira.mongodb.org/browse/SERVER-18409
[435] https://jira.mongodb.org/browse/SERVER-18326
[436] https://jira.mongodb.org/browse/SERVER-17923
[437] https://jira.mongodb.org/browse/SERVER-17913
[438] https://jira.mongodb.org/browse/SERVER-17807
[439] https://jira.mongodb.org/browse/SERVER-15252
[440] https://jira.mongodb.org/browse/SERVER-15192
[441] https://jira.mongodb.org/browse/SERVER-18190
[442] https://jira.mongodb.org/browse/SERVER-18822
[443] https://jira.mongodb.org/browse/SERVER-18246
[444] https://jira.mongodb.org/browse/SERVER-18442
[445] https://jira.mongodb.org/browse/SERVER-18647
[446] https://jira.mongodb.org/browse/SERVER-18646
[447] https://jira.mongodb.org/browse/SERVER-18629

- SERVER-18460[448] Segfault during eviction under load
- SERVER-18316[449] Database with WT engine fails to recover after system crash
- SERVER-18315[450] Throughput drop during transaction pinned phase of checkpoints under WiredTiger
- SERVER-18213[451] Lots of `WriteConflict` during multi-upsert with WiredTiger storage engine
- SERVER-18079[452] Large performance drop with documents > 16k on Windows
- SERVER-17944[453] `WiredTigerRecordStore::truncate` spends a lot of time sleeping

HTTP Console SERVER-18117[454] Bring back the _replSet page in the html interface

Build and Packaging

- SERVER-18894[455] OSX SSL builds should use unique filename
- SERVER-18421[456] Create SSL Builder for OS X
- SERVER-18312[457] Upgrade PCRE to latest
- SERVER-13596[458] Support –prefix rpm installation

Internal Code SERVER-6826[459] Potential memory leak in `ConnectionString::connect`

Testing

- SERVER-18318[460] Disable `jsCore_small_oplog` suite in Windows
- SERVER-17336[461] fix `core/compact_keeps_indexes.js` in a master/slave test configuration
- SERVER-13237[462] `benchRun` should use a thread-safe random number generator
- SERVER-18097[463] Remove `mongosTest_auth` and `mongosTest_WT` tasks from evergreen.yml

3.0.3 Changelog

Security

- SERVER-18290[464] Adding a read role for a user doesn't seem to propagate to secondary until restart
- SERVER-18239[465] `dumpauth.js` uses ambiguous `--db`/`--collection` args

[448] https://jira.mongodb.org/browse/SERVER-18460
[449] https://jira.mongodb.org/browse/SERVER-18316
[450] https://jira.mongodb.org/browse/SERVER-18315
[451] https://jira.mongodb.org/browse/SERVER-18213
[452] https://jira.mongodb.org/browse/SERVER-18079
[453] https://jira.mongodb.org/browse/SERVER-17944
[454] https://jira.mongodb.org/browse/SERVER-18117
[455] https://jira.mongodb.org/browse/SERVER-18894
[456] https://jira.mongodb.org/browse/SERVER-18421
[457] https://jira.mongodb.org/browse/SERVER-18312
[458] https://jira.mongodb.org/browse/SERVER-13596
[459] https://jira.mongodb.org/browse/SERVER-6826
[460] https://jira.mongodb.org/browse/SERVER-18318
[461] https://jira.mongodb.org/browse/SERVER-17336
[462] https://jira.mongodb.org/browse/SERVER-13237
[463] https://jira.mongodb.org/browse/SERVER-18097
[464] https://jira.mongodb.org/browse/SERVER-18290
[465] https://jira.mongodb.org/browse/SERVER-18239

- SERVER-18169[466] Regression: Auth enabled arbiter cannot be shutdown using command
- SERVER-18140[467] Allow `getParameter` (page 461) to be executed locally against an arbiter in an authenticated replica set
- SERVER-18051[468] OpenSSL internal error when using SCRAM-SHA1 authentication in FIPS mode
- SERVER-18021[469] Allow `serverStatus` (page 492) to be executed locally against an arbiter in an authenticated replica set
- SERVER-17908[470] Allow `getCmdLineOpts` (page 483) to be executed locally against an arbiter in an authenticated replica set
- SERVER-17832[471] Memory leak when `mongod` (page 770) configured with SSL required and handle insecure connection
- SERVER-17812[472] LockPinger has audit-related GLE failure
- SERVER-17591[473] Add SSL flag to select supported protocols
- SERVER-16073[474] Allow disabling SSL Ciphers via hidden flag: `sslCipherConfig`
- SERVER-12235[475] Don't require a database read on every new localhost connection when auth is on

Querying

- SERVER-18304[476] duplicates on FindAndModify with remove option
- SERVER-17815[477] Plan ranking tie breaker is computed incorrectly

Replication

- SERVER-18211[478] MongoDB fails to correctly roll back collection creation
- SERVER-17273[479] Add support for `secondaryCatchupPeriodSecs` to `rs.stepdown()` shell helper

Sharding

- SERVER-17812[480] LockPinger has audit-related GLE failure
- SERVER-17749[481] `collMod` (page 457) `usePowerOf2Sizes` (page 458) fails on `mongos` (page 792)
- SERVER-16987[482] `sh.getRecentMigrations()` shows aborted migration as success

[466]https://jira.mongodb.org/browse/SERVER-18169
[467]https://jira.mongodb.org/browse/SERVER-18140
[468]https://jira.mongodb.org/browse/SERVER-18051
[469]https://jira.mongodb.org/browse/SERVER-18021
[470]https://jira.mongodb.org/browse/SERVER-17908
[471]https://jira.mongodb.org/browse/SERVER-17832
[472]https://jira.mongodb.org/browse/SERVER-17812
[473]https://jira.mongodb.org/browse/SERVER-17591
[474]https://jira.mongodb.org/browse/SERVER-16073
[475]https://jira.mongodb.org/browse/SERVER-12235
[476]https://jira.mongodb.org/browse/SERVER-18304
[477]https://jira.mongodb.org/browse/SERVER-17815
[478]https://jira.mongodb.org/browse/SERVER-18211
[479]https://jira.mongodb.org/browse/SERVER-17273
[480]https://jira.mongodb.org/browse/SERVER-17812
[481]https://jira.mongodb.org/browse/SERVER-17749
[482]https://jira.mongodb.org/browse/SERVER-16987

Storage

- SERVER-18211[483] MongoDB fails to correctly roll back collection creation
- SERVER-18111[484] mongod allows user inserts into system.profile collection
- SERVER-17939[485] Backport mongo-rocks updates to v3.0 branch
- SERVER-17745[486] Improve dirty page estimation in mmapv1 on Windows

WiredTiger

- SERVER-18205[487] WiredTiger changes for MongoDB 3.0.3
- SERVER-18192[488] Crash running WiredTiger with "cache_resident=true"
- SERVER-18014[489] Dropping a collection can block creating a new collection for an extended time under WiredTiger
- SERVER-17907[490] B-tree eviction blocks access to collection for extended period under WiredTiger
- SERVER-17892[491] Explicitly turn checksum on for all collections/indexes in WiredTiger by default

Indexing

- SERVER-18087[492] index_retry.js and index_no_retry.js not checking for presence of "progress" field in currentOp() result
- SERVER-17882[493] Update with key too large to index crashes WiredTiger/RockDB secondary

Write Ops

- SERVER-18111[494] mongod allows user inserts into system.profile collection

Networking

- SERVER-17832[495] Memory leak when MongoD configured with SSL required and handle insecure connection
- SERVER-17591[496] Add SSL flag to select supported protocols
- SERVER-16073[497] Allow disabling SSL Ciphers via hidden flag: sslCipherConfig

[483] https://jira.mongodb.org/browse/SERVER-18211
[484] https://jira.mongodb.org/browse/SERVER-18111
[485] https://jira.mongodb.org/browse/SERVER-17939
[486] https://jira.mongodb.org/browse/SERVER-17745
[487] https://jira.mongodb.org/browse/SERVER-18205
[488] https://jira.mongodb.org/browse/SERVER-18192
[489] https://jira.mongodb.org/browse/SERVER-18014
[490] https://jira.mongodb.org/browse/SERVER-17907
[491] https://jira.mongodb.org/browse/SERVER-17892
[492] https://jira.mongodb.org/browse/SERVER-18087
[493] https://jira.mongodb.org/browse/SERVER-17882
[494] https://jira.mongodb.org/browse/SERVER-18111
[495] https://jira.mongodb.org/browse/SERVER-17832
[496] https://jira.mongodb.org/browse/SERVER-17591
[497] https://jira.mongodb.org/browse/SERVER-16073

Concurrency

- SERVER-18304[498] duplicates on FindAndModify with remove option
- SERVER-16636[499] Deadlock detection should check cycles for stability or should be disabled

Geo

- SERVER-17835[500] Aggregation geoNear deprecated uniqueDocs warning
- SERVER-9220[501] allow more than two values in the coordinate-array when using 2dsphere index

Aggregation Framework

- SERVER-17835[502] Aggregation geoNear deprecated uniqueDocs warning

MapReduce

- SERVER-17889[503] Using eval command to run mapReduce with non-inline "out" option triggers fatal assertion failure

Admin

- SERVER-18290[504] Adding a read role for a user doesn't seem to propagate to secondary until restart
- SERVER-18169[505] Regression: Auth enabled arbiter cannot be shutdown using command
- SERVER-17820[506] Windows service stop can lead to mongod abrupt termination due to long shutdown time

Build and Packaging

- SERVER-18344[507] logs should be sent to updated logkeeper server
- SERVER-18299[508] Use ld wrapper for compiling Enterprise GO tools in RHEL 5
- SERVER-18082[509] Change `smoke.py` buildlogger command line options to environment variables
- SERVER-17730[510] Parsing of Variables on Windows doesn't respect windows conventions
- SERVER-17694[511] support `RPATH=value` in top-level `SConstruct`
- SERVER-17465[512] `--use-system-tcmalloc` does not support `tcmalloc setParameters` and extension
- SERVER-17961[513] *THIRD-PARTY-NOTICES.windows*' needs to be updated

[498] https://jira.mongodb.org/browse/SERVER-18304
[499] https://jira.mongodb.org/browse/SERVER-16636
[500] https://jira.mongodb.org/browse/SERVER-17835
[501] https://jira.mongodb.org/browse/SERVER-9220
[502] https://jira.mongodb.org/browse/SERVER-17835
[503] https://jira.mongodb.org/browse/SERVER-17889
[504] https://jira.mongodb.org/browse/SERVER-18290
[505] https://jira.mongodb.org/browse/SERVER-18169
[506] https://jira.mongodb.org/browse/SERVER-17820
[507] https://jira.mongodb.org/browse/SERVER-18344
[508] https://jira.mongodb.org/browse/SERVER-18299
[509] https://jira.mongodb.org/browse/SERVER-18082
[510] https://jira.mongodb.org/browse/SERVER-17730
[511] https://jira.mongodb.org/browse/SERVER-17694
[512] https://jira.mongodb.org/browse/SERVER-17465
[513] https://jira.mongodb.org/browse/SERVER-17961

- SERVER-17780[514] Init script sets process ulimit to different value compared to documentation

JavaScript

- SERVER-17453[515] warn that db.eval() / eval command is deprecated

Shell

- SERVER-17951[516] db.currentOp() fails with read preference set
- SERVER-17273[517] Add support for secondaryCatchupPeriodSecs to rs.stepdown shell helper
- SERVER-16987[518] sh.getRecentMigrations shows aborted migration as success

Testing

- SERVER-18302[519] remove test buildlogger instance
- SERVER-18262[520] setup_multiversion_mongodb should retry links download on timeouts
- SERVER-18239[521] dumpauth.js uses ambiguous –db/–collection args
- SERVER-18229[522] Smoke.py with PyMongo 3.0.1 fails to run certain tests
- SERVER-18073[523] Fix smoke.py to work with pymongo 3.0
- SERVER-17998[524] Ignore socket exceptions in initial_sync_unsupported_auth_schema.js test
- SERVER-18293[525] ASAN tests should run on larger instance size
- SERVER-17761[526] RestAdminAccess/NoAdminAccess objects leak at shutdown

3.0.2 Changelog

Security

- SERVER-17719[527] `mongo` (page 803) Shell crashes if -p is missing and user matches
- SERVER-17705[528] Fix credentials field inconsistency in HTTP interface
- SERVER-17671[529] Refuse to complete initial sync from nodes with 2.4-style auth data
- SERVER-17669[530] Remove auth prompt in webserver when auth is not enabled

[514] https://jira.mongodb.org/browse/SERVER-17780
[515] https://jira.mongodb.org/browse/SERVER-17453
[516] https://jira.mongodb.org/browse/SERVER-17951
[517] https://jira.mongodb.org/browse/SERVER-17273
[518] https://jira.mongodb.org/browse/SERVER-16987
[519] https://jira.mongodb.org/browse/SERVER-18302
[520] https://jira.mongodb.org/browse/SERVER-18262
[521] https://jira.mongodb.org/browse/SERVER-18239
[522] https://jira.mongodb.org/browse/SERVER-18229
[523] https://jira.mongodb.org/browse/SERVER-18073
[524] https://jira.mongodb.org/browse/SERVER-17998
[525] https://jira.mongodb.org/browse/SERVER-18293
[526] https://jira.mongodb.org/browse/SERVER-17761
[527] https://jira.mongodb.org/browse/SERVER-17719
[528] https://jira.mongodb.org/browse/SERVER-17705
[529] https://jira.mongodb.org/browse/SERVER-17671
[530] https://jira.mongodb.org/browse/SERVER-17669

- SERVER-17647[531] Compute BinData length in v8
- SERVER-17529[532] Can't list collections when `mongos` (page 792) is running 3.0 and config servers are running 2.6 and auth is on

Query and Indexing

- SERVER-8188[533] Configurable idle cursor timeout
- SERVER-17469[534] `2d` nearSphere queries scan entire collection
- SERVER-17642[535] `WriteConfictException` during background index create

Replication

- SERVER-17677[536] Replica Set member backtraces sometimes when removed from replica set
- SERVER-17672[537] `serverStatus` (page 492) command with `{oplog: 1}` option can trigger segmentation fault in `mongod` (page 770)
- SERVER-17822[538] `OpDebug::writeConflicts` should be a 64-bit type

Sharding SERVER-17805[539] `logOp` / `OperationObserver` should always check shardversion

Storage SERVER-17613[540] Unable to start `mongod` (page 770) after unclean shutdown

WiredTiger

- SERVER-17713[541] WiredTiger using zlib compression can create invalid compressed stream
- SERVER-17642[542] WriteConfictException during background index create
- SERVER-17587[543] Node crash scenario results in uncrecoverable error on subsequent startup under WiredTiger
- SERVER-17562[544] Invariant failure: `s->commit_transaction(s, NULL)` resulted in status `BadValue 22`
- SERVER-17551[545] mongod fatal assertion after "hazard pointer table full" message
- SERVER-17532[546] Duplicate key error message does not contain index name anymore
- SERVER-17471[547] WiredTiger Mutex on Windows can block the server

[531] https://jira.mongodb.org/browse/SERVER-17647
[532] https://jira.mongodb.org/browse/SERVER-17529
[533] https://jira.mongodb.org/browse/SERVER-8188
[534] https://jira.mongodb.org/browse/SERVER-17469
[535] https://jira.mongodb.org/browse/SERVER-17642
[536] https://jira.mongodb.org/browse/SERVER-17677
[537] https://jira.mongodb.org/browse/SERVER-17672
[538] https://jira.mongodb.org/browse/SERVER-17822
[539] https://jira.mongodb.org/browse/SERVER-17805
[540] https://jira.mongodb.org/browse/SERVER-17613
[541] https://jira.mongodb.org/browse/SERVER-17713
[542] https://jira.mongodb.org/browse/SERVER-17642
[543] https://jira.mongodb.org/browse/SERVER-17587
[544] https://jira.mongodb.org/browse/SERVER-17562
[545] https://jira.mongodb.org/browse/SERVER-17551
[546] https://jira.mongodb.org/browse/SERVER-17532
[547] https://jira.mongodb.org/browse/SERVER-17471

- SERVER-17382[548] rc10/wiredTiger multi collection/DB bulk insert slow than rc8 in initial insertion phase
- SERVER-16804[549] *mongod --repair* fails because `verify()` returns EBUSY under WiredTiger

MMAPv1

- SERVER-17616[550] Removing or inserting documents with large indexed arrays consumes excessive memory
- SERVER-17313[551] Segfault in `BtreeLogic::_insert` when inserting into previously-dropped namespace

RocksDB SERVER-17706[552] Sync new mongo+rocks changes to v3.0 branch

HTTP Console

- SERVER-17729[553] Cannot start `mongod` (page 770) `httpinterface`: sockets higher than 1023 not supported
- SERVER-17705[554] Fix credentials field inconsistency in HTTP interface
- SERVER-17669[555] Remove auth prompt in webserver when auth is not enabled

Admin

- SERVER-17570[556] MongoDB 3.0 NT Service shutdown race condition with `db.serverShutdown()`
- SERVER-17699[557] "locks" section empty in diagnostic log and profiler output for some operations
- SERVER-17337[558] RPM Init script breaks with quotes in `yaml` config file
- SERVER-16731[559] Remove unused DBPATH init script variable

Networking SERVER-17652[560] Cannot start mongod due to "sockets higher than 1023 not being supported"

Testing

- SERVER-17826[561] Ignore ismaster exceptions in `‘initial_sync_unsupported_auth_schema.js` test
- SERVER-17808[562] Ensure availability in `initial_sync_unsupported_auth_schema.js` test
- SERVER-17433[563] ASAN leak in small oplog suite `write_result.js`

[548] https://jira.mongodb.org/browse/SERVER-17382
[549] https://jira.mongodb.org/browse/SERVER-16804
[550] https://jira.mongodb.org/browse/SERVER-17616
[551] https://jira.mongodb.org/browse/SERVER-17313
[552] https://jira.mongodb.org/browse/SERVER-17706
[553] https://jira.mongodb.org/browse/SERVER-17729
[554] https://jira.mongodb.org/browse/SERVER-17705
[555] https://jira.mongodb.org/browse/SERVER-17669
[556] https://jira.mongodb.org/browse/SERVER-17570
[557] https://jira.mongodb.org/browse/SERVER-17699
[558] https://jira.mongodb.org/browse/SERVER-17337
[559] https://jira.mongodb.org/browse/SERVER-16731
[560] https://jira.mongodb.org/browse/SERVER-17652
[561] https://jira.mongodb.org/browse/SERVER-17826
[562] https://jira.mongodb.org/browse/SERVER-17808
[563] https://jira.mongodb.org/browse/SERVER-17433

3.0.1 Changelog

Security

- SERVER-17507[564] MongoDB3 enterprise AuditLog
- SERVER-17379[565] Change "or" to "and" in webserver localhost exception check
- SERVER-16944[566] dbAdminAnyDatabase should have full parity with dbAdmin for a given database
- SERVER-16849[567] On mongos we always invalidate the user cache once, even if no user definitions are changing
- SERVER-16452[568] Failed login attempts should log source IP address

Querying

- SERVER-17395[569] Add FSM tests to stress yielding
- SERVER-17387[570] invalid projection for findAndModify triggers fassert() failure
- SERVER-14723[571] Crash during query planning for geoNear with multiple 2dsphere indices
- SERVER-17486[572] Crash when parsing invalid polygon coordinates

Replication

- SERVER-17515[573] copyDatabase fails to replicate indexes to secondary
- SERVER-17499[574] Using eval command to run getMore on aggregation cursor trips fatal assertion
- SERVER-17487[575] cloner dropDups removes _id entries belonging to other records
- SERVER-17302[576] consider blacklist in shouldChangeSyncSource

Sharding

- SERVER-17398[577] Deadlock in MigrateStatus::startCommit
- SERVER-17300[578] Balancer tries to create config.tags index multiple times
- SERVER-16849[579] On mongos we always invalidate the user cache once, even if no user definitions are changing
- SERVER-5004[580] balancer should check for stopped between chunk moves in current round

[564] https://jira.mongodb.org/browse/SERVER-17507
[565] https://jira.mongodb.org/browse/SERVER-17379
[566] https://jira.mongodb.org/browse/SERVER-16944
[567] https://jira.mongodb.org/browse/SERVER-16849
[568] https://jira.mongodb.org/browse/SERVER-16452
[569] https://jira.mongodb.org/browse/SERVER-17395
[570] https://jira.mongodb.org/browse/SERVER-17387
[571] https://jira.mongodb.org/browse/SERVER-14723
[572] https://jira.mongodb.org/browse/SERVER-17486
[573] https://jira.mongodb.org/browse/SERVER-17515
[574] https://jira.mongodb.org/browse/SERVER-17499
[575] https://jira.mongodb.org/browse/SERVER-17487
[576] https://jira.mongodb.org/browse/SERVER-17302
[577] https://jira.mongodb.org/browse/SERVER-17398
[578] https://jira.mongodb.org/browse/SERVER-17300
[579] https://jira.mongodb.org/browse/SERVER-16849
[580] https://jira.mongodb.org/browse/SERVER-5004

Indexing

- SERVER-17521[581] improve createIndex validation of empty name
- SERVER-17436[582] MultiIndexBlock may access deleted collection after recovering from yield

Aggregation Framework SERVER-17224[583] Aggregation pipeline with 64MB document can terminate server

Write Ops

- SERVER-17489[584] in bulk ops, only mark last operation with commit=synchronous
- SERVER-17276[585] WriteConflictException retry loops needed for collection creation on upsert

Concurrency

- SERVER-17501[586] Increase journalling capacity limits
- SERVER-17416[587] Deadlock between MMAP V1 journal lock and oplog collection lock
- SERVER-17395[588] Add FSM tests to stress yielding

Storage

- SERVER-17515[589] copyDatabase fails to replicate indexes to secondary
- SERVER-17436[590] MultiIndexBlock may access deleted collection after recovering from yield
- SERVER-17416[591] Deadlock between MMAP V1 journal lock and oplog collection lock
- SERVER-17381[592] Rename rocksExperiment to RocksDB
- SERVER-17369[593] [Rocks] Fix the calculation of nextPrefix
- SERVER-17345[594] WiredTiger -> session.truncate: the start cursor position is after the stop cursor position
- SERVER-17331[595] RocksDB configuring and monitoring
- SERVER-17323[596] MMAPV1Journal lock counts are changing during WT run
- SERVER-17319[597] invariant at shutdown rc9, rc10, rc11 with wiredTiger
- SERVER-17293[598] Server crash setting wiredTigerEngineRuntimeConfig:"eviction=(threads_max=8)"

[581] https://jira.mongodb.org/browse/SERVER-17521
[582] https://jira.mongodb.org/browse/SERVER-17436
[583] https://jira.mongodb.org/browse/SERVER-17224
[584] https://jira.mongodb.org/browse/SERVER-17489
[585] https://jira.mongodb.org/browse/SERVER-17276
[586] https://jira.mongodb.org/browse/SERVER-17501
[587] https://jira.mongodb.org/browse/SERVER-17416
[588] https://jira.mongodb.org/browse/SERVER-17395
[589] https://jira.mongodb.org/browse/SERVER-17515
[590] https://jira.mongodb.org/browse/SERVER-17436
[591] https://jira.mongodb.org/browse/SERVER-17416
[592] https://jira.mongodb.org/browse/SERVER-17381
[593] https://jira.mongodb.org/browse/SERVER-17369
[594] https://jira.mongodb.org/browse/SERVER-17345
[595] https://jira.mongodb.org/browse/SERVER-17331
[596] https://jira.mongodb.org/browse/SERVER-17323
[597] https://jira.mongodb.org/browse/SERVER-17319
[598] https://jira.mongodb.org/browse/SERVER-17293

WiredTiger

- SERVER-17510[599] "Didn't find RecordId in WiredTigerRecordStore" on collections after an idle period
- SERVER-17506[600] Race between inserts and checkpoints can lose records under WiredTiger
- SERVER-17487[601] cloner dropDups removes _id entries belonging to other records
- SERVER-17481[602] WiredTigerRecordStore::validate should call WT_SESSION::verify
- SERVER-17451[603] WiredTiger unable to start if crash leaves 0-length journal file
- SERVER-17378[604] WiredTiger's compact code can return 'Operation timed out' error (invariant failure)
- SERVER-17345[605] WiredTiger -> session.truncate: the start cursor position is after the stop cursor position
- SERVER-17319[606] invariant at shutdown rc9, rc10, rc11 with wiredTiger

MMAPv1

- SERVER-17501[607] Increase journalling capacity limits
- SERVER-17416[608] Deadlock between MMAP V1 journal lock and oplog collection lock
- SERVER-17388[609] Invariant failure in MMAPv1 when disk full

RocksDB

- SERVER-17381[610] Rename rocksExperiment to RocksDB
- SERVER-17369[611] [Rocks] Fix the calculation of nextPrefix
- SERVER-17331[612] RocksDB configuring and monitoring

Shell and Administration

- SERVER-17226[613] 'top' command with 64MB result document can terminate server
- SERVER-17405[614] getLog command masserts when given number
- SERVER-17347[615] .explain() should be included in the shell's DBCollection help

[599] https://jira.mongodb.org/browse/SERVER-17510
[600] https://jira.mongodb.org/browse/SERVER-17506
[601] https://jira.mongodb.org/browse/SERVER-17487
[602] https://jira.mongodb.org/browse/SERVER-17481
[603] https://jira.mongodb.org/browse/SERVER-17451
[604] https://jira.mongodb.org/browse/SERVER-17378
[605] https://jira.mongodb.org/browse/SERVER-17345
[606] https://jira.mongodb.org/browse/SERVER-17319
[607] https://jira.mongodb.org/browse/SERVER-17501
[608] https://jira.mongodb.org/browse/SERVER-17416
[609] https://jira.mongodb.org/browse/SERVER-17388
[610] https://jira.mongodb.org/browse/SERVER-17381
[611] https://jira.mongodb.org/browse/SERVER-17369
[612] https://jira.mongodb.org/browse/SERVER-17331
[613] https://jira.mongodb.org/browse/SERVER-17226
[614] https://jira.mongodb.org/browse/SERVER-17405
[615] https://jira.mongodb.org/browse/SERVER-17347

Build and Packaging

- SERVER-17484[616] Migrate server MCI config into server repo
- SERVER-17463[617] Python error when specifying absolute path to scons cacheDir
- SERVER-17460[618] LIBDEPS_v8_SYSLIBDEP typo
- SERVER-14166[619] Semantics of the –osx-version-min flag should be improved
- SERVER-17517[620] mongodb-org rpm packages no longer "provide" mongo-10gen-server

Logging SERVER-16452[621] Failed login attempts should log source IP address

Platform

- SERVER-17252[622] Upgrade PCRE Version from 8.30 to Latest
- SERVER-14166[623] Semantics of the –osx-version-min flag should be improved

Internal Code SERVER-17338[624] NULL pointer crash when running copydb against stepped-down 2.6 primary

Testing

- SERVER-17443[625] get_replication_info_helper.js should assert.soon rather than assert for log messages
- SERVER-17442[626] increase tolerance for shutdown timeout in stepdown.js to fix windows build break
- SERVER-17395[627] Add FSM tests to stress yielding

3.0.9 – Jan 26, 2016

- Fixed issue where queries which specify sort and batch size can return results out of order if documents are concurrently updated. SERVER-19996[628]
- Fixed performance issue where large amounts of create and drop collections can cause `listDatabases` to be slow under WiredTiger. SERVER-20961[629]
- Modified the authentication failure message to include the client IP address. SERVER-22054[630]
- All issues closed in 3.0.9[631]

[616] https://jira.mongodb.org/browse/SERVER-17484
[617] https://jira.mongodb.org/browse/SERVER-17463
[618] https://jira.mongodb.org/browse/SERVER-17460
[619] https://jira.mongodb.org/browse/SERVER-14166
[620] https://jira.mongodb.org/browse/SERVER-17517
[621] https://jira.mongodb.org/browse/SERVER-16452
[622] https://jira.mongodb.org/browse/SERVER-17252
[623] https://jira.mongodb.org/browse/SERVER-14166
[624] https://jira.mongodb.org/browse/SERVER-17338
[625] https://jira.mongodb.org/browse/SERVER-17443
[626] https://jira.mongodb.org/browse/SERVER-17442
[627] https://jira.mongodb.org/browse/SERVER-17395
[628] https://jira.mongodb.org/browse/SERVER-19996
[629] https://jira.mongodb.org/browse/SERVER-20961
[630] https://jira.mongodb.org/browse/SERVER-22054
[631] https://jira.mongodb.org/issues/?jql=project%20in%20(SERVER%2C%20TOOLS)%20AND%20fixVersion%20%3D%203.0.9%20AND%20resolution%20%3D%2

3.0.8 – Dec 15, 2015

- Fixed issue where `findAndModify` (page 348) on `mongos` (page 792) can upsert to the wrong shard. SERVER-20407[632].
- Fixed WiredTiger commit visibility issue which caused document not found. SERVER-21275[633].
- Fixed issue where the oplog can grow to 3x configured size. SERVER-21553[634]
- All issues closed in 3.0.8[635]

3.0.7 – Oct 13, 2015

- Improvements to WiredTiger memory handling and performance: SERVER-20159[636], SERVER-20204[637], SERVER-20091[638], and SERVER-20176[639].
- Fixed issue whereby reconfig during a pending step down may prevent a primary from stepping down: SERVER-20262[640].
- Additional privileges for built-in roles: SERVER-19131[641], SERVER-15893[642], and SERVER-13647[643].
- All issues closed in 3.0.7[644]

3.0.6 – August 24, 2015

- Improvements to WiredTiger Stability SERVER-19751[645], SERVER-19673[646], and SERVER-19573[647].
- Fixed issue with the interaction between SSL and Auditing. SERVER-19538[648].
- Fixed issue with aggregation `$sort` (page 649) on sharded systems SERVER-19464[649].
- All issues closed in 3.0.6[650]

3.0.5 – July 28, 2015

- Improvements to WiredTiger for capped collections and replication (SERVER-19178[651], SERVER-18875[652] and SERVER-19513[653]).

632 https://jira.mongodb.org/browse/SERVER-20407
633 https://jira.mongodb.org/browse/SERVER-21275
634 https://jira.mongodb.org/browse/SERVER-21553
635 https://jira.mongodb.org/issues/?jql=project%20in%20(SERVER%2C%20TOOLS)%20AND%20fixVersion%20%3D%203.0.8%20AND%20resolution%20%3D%2
636 https://jira.mongodb.org/browse/SERVER-20159
637 https://jira.mongodb.org/browse/SERVER-20204
638 https://jira.mongodb.org/browse/SERVER-20091
639 https://jira.mongodb.org/browse/SERVER-20176
640 https://jira.mongodb.org/browse/SERVER-20262
641 https://jira.mongodb.org/browse/SERVER-19131
642 https://jira.mongodb.org/browse/SERVER-15893
643 https://jira.mongodb.org/browse/SERVER-13647
644 https://jira.mongodb.org/issues/?jql=project%20in%20(SERVER%2C%20TOOLS)%20AND%20fixVersion%20%3D%203.0.7%20AND%20resolution%20%3D%2
645 https://jira.mongodb.org/browse/SERVER-19751
646 https://jira.mongodb.org/browse/SERVER-19673
647 https://jira.mongodb.org/browse/SERVER-19573
648 https://jira.mongodb.org/browse/SERVER-19538
649 https://jira.mongodb.org/browse/SERVER-19464
650 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%223.0.6%22%20AND%20project%20%3D%20SERVER%20AND%20resolution%20%3D%20Fixed
651 https://jira.mongodb.org/browse/SERVER-19178
652 https://jira.mongodb.org/browse/SERVER-18875
653 https://jira.mongodb.org/browse/SERVER-19513

- Additional WiredTiger improvements for performance (SERVER-19189[654]) and improvements related to cache and session use (SERVER-18829[655] SERVER-17836[656]).
- Performance improvements for longer running queries, particularly `$text` and `$near` queries SERVER-18926[657].
- All issues closed in 3.0.5[658]

3.0.4 – June 16, 2015

- Fix missed writes with concurrent inserts during chunk migration from shards with WiredTiger primaries: SERVER-18822[659]
- Resolve write conflicts with multi-update updates with `upsert=true` with the Wired Tiger Storage engine: SERVER-18213[660]
- Fix case where secondary reads could block replication: SERVER-18190[661]
- Improve performance on Windows with WiredTiger and documents larger than 16kb: SERVER-18079[662]
- Fix issue where WiredTiger data files are not correctly recovered following unexpected system restarts: SERVER-18316[663]
- All issues closed in 3.0.4[664]

3.0.3 – May 12, 2015

- Deprecate `db.eval()` (page 178) and add warnings: SERVER-17453[665]
- Fix potential for abrupt termination with the Windows service stop operation: SERVER-17802[666]
- Fix crash caused by update with a *key too large to index* on WiredTiger and RocksDB storage engines: SERVER-17882[667]
- Remove inconsistent support for `mapReduce` (page 318) in `eval` (page 358) environment: SERVER-17889[668]
- All issues closed in 3.0.3[669]

3.0.2 – April 9, 2015

- Fix inefficient query plans for `2d` `$nearSphere` (page 567): SERVER-17469[670]

[654] https://jira.mongodb.org/browse/SERVER-19189
[655] https://jira.mongodb.org/browse/SERVER-18829
[656] https://jira.mongodb.org/browse/SERVER-17836
[657] https://jira.mongodb.org/browse/SERVER-18926
[658] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%223.0.5%22%20AND%20project%20%3D%20SERVER%20AND%20resolution%20%3D%20Fixed
[659] https://jira.mongodb.org/browse/SERVER-18822
[660] https://jira.mongodb.org/browse/SERVER-18213
[661] https://jira.mongodb.org/browse/SERVER-18190
[662] https://jira.mongodb.org/browse/SERVER-18079
[663] https://jira.mongodb.org/browse/SERVER-18316
[664] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%223.0.4%22%20AND%20project%20%3D%20SERVER%20AND%20resolution%20%3D%20Fixed
[665] https://jira.mongodb.org/browse/SERVER-17453
[666] https://jira.mongodb.org/browse/SERVER-17802
[667] https://jira.mongodb.org/browse/SERVER-17882
[668] https://jira.mongodb.org/browse/SERVER-17889
[669] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%223.0.3%22%20AND%20project%20%3D%20SERVER%20AND%20resolution%20%3D%20Fixed
[670] https://jira.mongodb.org/browse/SERVER-17469

- Fix problem starting `mongod` (page 770) during repair operations with WiredTiger: SERVER-17652[671] and SERVER-17729[672]
- Resolved invalid compression stream error with WiredTiger and `zlib` block compression: SERVER-17713[673]
- Fix memory use issue for inserts into large indexed arrays: SERVER-17616[674]
- All issues closed in 3.0.2[675]

3.0.1 – March 17, 2015

- Fixed race condition in WiredTiger between inserts and checkpoints that could result in lost records: SERVER-17506[676].
- Resolved issue in WiredTiger's capped collections implementation that caused a server crash: SERVER-17345[677].
- Fixed issue is initial sync with duplicate `_id` entries: SERVER-17487[678].
- Fixed deadlock condition in MMAPv1 between the journal lock and the oplog collection lock: SERVER-17416[679].
- All issues closed in 3.0.1[680]

Major Changes

Pluggable Storage Engine API

MongoDB 3.0 introduces a pluggable storage engine API that allows third parties to develop storage engines for MongoDB.

WiredTiger

MongoDB 3.0 introduces support for the WiredTiger[681] storage engine. With the support for WiredTiger, MongoDB now supports two storage engines:

- MMAPv1, the storage engine available in previous versions of MongoDB and the default storage engine for MongoDB 3.0, and
- WiredTiger[682], available only in the 64-bit versions of MongoDB 3.0.

[671] https://jira.mongodb.org/browse/SERVER-17652
[672] https://jira.mongodb.org/browse/SERVER-17729
[673] https://jira.mongodb.org/browse/SERVER-17713
[674] https://jira.mongodb.org/browse/SERVER-17616
[675] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%223.0.2%22%20AND%20project%20%3D%20SERVER%20AND%20resolution%20%3D%20Fixed
[676] https://jira.mongodb.org/browse/SERVER-17506
[677] https://jira.mongodb.org/browse/SERVER-17345
[678] https://jira.mongodb.org/browse/SERVER-17487
[679] https://jira.mongodb.org/browse/SERVER-17416
[680] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%223.0.1%22%20AND%20project%20%3D%20SERVER%20AND%20resolution%20%3D%20Fixed
[681] http://wiredtiger.com
[682] http://wiredtiger.com

WiredTiger Usage WiredTiger is an alternate to the default MMAPv1 storage engine. WiredTiger supports all MongoDB features, including operations that report on server, database, and collection statistics. Switching to WiredTiger, however, requires a change to the *on-disk storage format* (page 1047). For instructions on changing the storage engine to WiredTiger, see the appropriate sections in the *Upgrade MongoDB to 3.0* (page 1054) documentation.

MongoDB 3.0 replica sets and sharded clusters can have members with different storage engines; however, performance can vary according to workload. For details, see the appropriate sections in the *Upgrade MongoDB to 3.0* (page 1054) documentation.

The WiredTiger storage engine requires the latest official MongoDB drivers. For more information, see *WiredTiger and Driver Version Compatibility* (page 1047).

See also:

Support for touch Command (page 1048), `https://docs.mongodb.org/manual/core/wiredtiger` documentation

WiredTiger Configuration To configure the behavior and properties of the WiredTiger storage engine, see `storage.wiredTiger` configuration options. You can set *WiredTiger options on the command line* (page 779).

See also:

`https://docs.mongodb.org/manual/core/wiredtiger`

WiredTiger Concurrency and Compression The 3.0 WiredTiger storage engine provides document-level locking and compression.

By default, WiredTiger compresses collection data using the *snappy* compression library. WiredTiger uses *prefix compression* on all indexes by default.

See also:

prod-notes-wired-tiger-concurrency section in the `https://docs.mongodb.org/manual/administration/production`
the blog post New Compression Options in MongoDB 3.0[683]

MMAPv1 Improvements

MMAPv1 Concurrency Improvement In version 3.0, the MMAPv1 storage engine adds support for collection-level locking.

MMAPv1 Configuration Changes To support multiple storage engines, some configuration settings for MMAPv1 have changed. See *Configuration File Options Changes* (page 1047).

MMAPv1 Record Allocation Behavior Changes MongoDB 3.0 no longer implements dynamic record allocation and deprecates *paddingFactor*. The default allocation strategy for collections in instances that use MMAPv1 is *power of 2 allocation*, which has been improved to better handle large document sizes. In 3.0, the `usePowerOf2Sizes` flag is ignored, so the power of 2 strategy is used for all collections that do not have `noPadding` flag set.

For collections with workloads that consist only of inserts or in-place updates (such as incrementing counters), you can disable the power of 2 strategy. To disable the power of 2 strategy for a collection, use the `collMod` (page 457) command with the `noPadding` (page 458) flag or the `db.createCollection()` (page 167) method with the `noPadding` option.

[683] https://www.mongodb.com/blog/post/new-compression-options-mongodb-30?jmp=docs

Warning: Do not set `noPadding` if the workload includes removes or any updates that may cause documents to grow. For more information, see *exact-fit-allocation*.

When low on disk space, MongoDB 3.0 no longer errors on all writes but only when the required disk allocation fails. As such, MongoDB now allows in-place updates and removes when low on disk space.

See also:

Dynamic Record Allocation (page 1048)

Replica Sets

Increased Number of Replica Set Members

In MongoDB 3.0, replica sets can have up to 50 members. [684] The following drivers support the larger replica sets:

- C# (.NET) Driver 1.10
- Java Driver 2.13
- Python Driver (PyMongo) 3.0
- Ruby Driver 2.0
- Node.JS Driver 2.0

The C, C++, Perl, and PHP drivers, as well as the earlier versions of the Ruby, Python, and Node.JS drivers, discover and monitor replica set members serially, and thus are not suitable for use with large replica sets.

Replica Set Step Down Behavior Changes

The process that a *primary* member of a *replica set* uses to step down has the following changes:

- Before stepping down, `replSetStepDown` (page 405) will attempt to terminate long running user operations that would block the primary from stepping down, such as an index build, a write operation or a map-reduce job.
- To help prevent rollbacks, the `replSetStepDown` (page 405) will wait for an electable secondary to catch up to the state of the primary before stepping down. Previously, a primary would wait for a secondary to catch up to within 10 seconds of the primary (i.e. a secondary with a replication lag of 10 seconds or less) before stepping down.
- `replSetStepDown` (page 405) 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 `https://docs.mongodb.org/manual/reference/replica-configuration`. For details, see *Replica Set Configuration Validation* (page 1048).
- For pre-existing collections on secondary members, MongoDB 3.0 no longer automatically builds missing `_id` indexes.

[684] The maximum number of *voting* members remains at 7.

See also:

Replication Changes (page 1048) in *Compatibility Changes in MongoDB 3.0* (page 1046)

Sharded Clusters

MongoDB 3.0 provides the following enhancements to sharded clusters:

- Adds a new `sh.removeTagRange()` (page 271) helper to improve management of sharded collections with tags. The new `sh.removeTagRange()` (page 271) method acts as a complement to `sh.addTagRange()` (page 270).
- Provides a more predictable read preference behavior. `mongos` (page 792) instances no longer pin connections to members of replica sets when performing read operations. Instead, `mongos` (page 792) reevaluates `read preferences` for every operation to provide a more predictable read preference behavior when read preferences change.
- Provides a new `writeConcern` setting to configure the `write concern` of chunk migration operations. You can configure the `writeConcern` setting for the *balancer* as well as for `moveChunk` (page 427) and `cleanupOrphaned` (page 415) commands.
- Improves visibility of balancer operations. `sh.status()` (page 279) includes information about the state of the balancer. See `sh.status()` (page 279) for details.

See also:

Sharded Cluster Setting (page 1049) in *Compatibility Changes in MongoDB 3.0* (page 1046)

Security Improvements

MongoDB 3.0 includes the following security enhancements:

- MongoDB 3.0 adds a new *SCRAM-SHA-1* challenge-response user authentication mechanism. `SCRAM-SHA-1` requires a driver upgrade if your current driver version does not support `SCRAM-SHA-1`. For the driver versions that support `SCRAM-SHA-1`, see *Upgrade Drivers* (page 1059).
- Increases restrictions when using the *localhost-exception* to access MongoDB. For details, see *Localhost Exception Changed* (page 1050).

See also:

Security Changes (page 1049)

Improvements

New Query Introspection System

MongoDB 3.0 includes a new query introspection system that provides an improved output format and a finer-grained introspection into both query plan and query execution.

For details, see the new `db.collection.explain()` (page 48) method and the new `explain` (page 467) command as well as the updated `cursor.explain()` (page 140) method.

For information on the format of the new output, see *Explain Results* (page 946).

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 *Log Messages* (page 964).

MongoDB Tools Enhancements

All MongoDB tools except for `mongosniff` (page 873) and `mongoperf` (page 875) are now written in Go and maintained as a separate project.

- New options for parallelized `mongodump` (page 816) and `mongorestore` (page 824). You can control the number of collections that `mongorestore` (page 824) will restore at a time with the `--numParallelCollections` option.
- New options `-excludeCollection` and `--excludeCollectionsWithPrefix` for `mongodump` (page 816) to exclude collections.
- `mongorestore` (page 824) can now accept BSON data input from standard input in addition to reading BSON data from file.
- `mongostat` (page 858) and `mongotop` (page 867) can now return output in JSON format with the `--json` option.
- Added configurable *write concern* to `mongoimport` (page 841), `mongorestore` (page 824), and `mongofiles` (page 878). Use the `--writeConcern` option. The default writeConcern has been changed to ‘w:majority’.
- `mongofiles` (page 878) now allows you to configure the GridFS prefix with the `--prefix` option so that you can use custom namespaces and store multiple GridFS namespaces in a single database.

See also:

MongoDB Tools Changes (page 1049)

Indexes

- Background index builds will no longer automatically interrupt if `dropDatabase` (page 437), `drop` (page 439), `dropIndexes` (page 450) operations occur for the database or collection affected by the index builds. The `dropDatabase` (page 437), `drop` (page 439), and `dropIndexes` (page 450) commands will still fail with the error message `a background operation is currently running`, as in 2.6.
- If you specify multiple indexes to the `createIndexes` (page 446) command,
  - the command only scans the collection once, and
  - if at least one index is to be built in the foreground, the operation will build all the specified indexes in the foreground.
- For sharded collections, indexes can now *cover queries* that execute against the `mongos` (page 792) if the index includes the shard key.

See also:

Indexes (page 1051) in *Compatibility Changes in MongoDB 3.0* (page 1046)

Query Enhancements

MongoDB 3.0 includes the following query enhancements:

- For geospatial queries, adds support for "big" polygons for `$geoIntersects` (page 562) and `$geoWithin` (page 560) queries. "Big" polygons are single-ringed GeoJSON polygons with areas greater than that of a single hemisphere. See `$geometry` (page 569), `$geoIntersects` (page 562), and `$geoWithin` (page 560) for details.
- For `aggregate()` (page 20), adds a new `$dateToString` (page 724) operator to facilitate converting a date to a formatted string.
- Adds the `$eq` (page 527) query operator to query for equality conditions.

See also:

2d Indexes and Geospatial Near Queries (page 1051)

Distributions and Supported Versions

Most non-Enterprise MongoDB distributions now include support for TLS/SSL. Previously, only MongoDB Enterprise distributions came with TLS/SSL support included; for non-Enterprise distributions, you had to build MongoDB locally with the `--ssl` flag (i.e. `scons --ssl`).

32-bit MongoDB builds are available for testing, but are not for production use. 32-bit MongoDB builds do not include the WiredTiger storage engine.

MongoDB builds for Solaris do not support the WiredTiger storage engine.

MongoDB builds are available for Windows Server 2003 and Windows Vista (as "64-bit Legacy"), but the minimum officially supported Windows version is Windows Server 2008.

See also:

Platform Support (page 1053), *faq-32-bit-limitations*

Package Repositories

Non-Enterprise MongoDB Linux packages for 3.0 and later are in a new repository. Follow the appropriate `Linux installation instructions` to install the 3.0 packages from the new location.

MongoDB Enterprise Features

Auditing

`https://docs.mongodb.org/manual/core/auditing` in MongoDB Enterprise can filter on `any field in the audit message`, including the fields returned in the *param* document. This enhancement, along with the `auditAuthorizationSuccess` (page 939) parameter, enables auditing to filter on CRUD operations. However, enabling `auditAuthorizationSuccess` (page 939) to audit of all authorization successes degrades performance more than auditing only the authorization failures.

Additional Information

Changes Affecting Compatibility

Compatibility Changes in MongoDB 3.0

On this page

- Storage Engine (page 1047)
- Replication Changes (page 1048)
- MongoDB Tools Changes (page 1049)
- Sharded Cluster Setting (page 1049)
- Security Changes (page 1049)
- Indexes (page 1051)
- Driver Compatibility Changes (page 1051)
- General Compatibility Changes (page 1052)

The following 3.0 changes can affect the compatibility with older versions of MongoDB. See *Release Notes for MongoDB 3.0* (page 1013) for the full list of the 3.0 changes.

Storage Engine

Configuration File Options Changes With the introduction of additional storage engines in 3.0, some *configuration file options* (page 895) have changed:

Previous Setting	New Setting
`storage.journal.commitIntervalMs`	`storage.mmapv1.journal.commitIntervalMs` (page 918)
`storage.journal.debugFlags`	`storage.mmapv1.journal.debugFlags` (page 918)
`storage.nsSize`	`storage.mmapv1.nsSize` (page 917)
`storage.preallocDataFiles`	`storage.mmapv1.preallocDataFiles` (page 917)
`storage.quota.enforced`	`storage.mmapv1.quota.enforced` (page 918)
`storage.quota.maxFilesPerDB`	`storage.mmapv1.quota.maxFilesPerDB` (page 918)
`storage.smallFiles`	`storage.mmapv1.smallFiles` (page 918)

3.0 `mongod` (page 770) instances are backward compatible with existing configuration files, but will issue warnings when if you attempt to use the old settings.

Data Files Must Correspond to Configured Storage Engine The files in the `dbPath` (page 915) directory must correspond to the configured storage engine (i.e. `--storageEngine`). `mongod` (page 770) will not start if `dbPath` (page 915) contains data files created by a storage engine other than the one specified by `--storageEngine`.

See also:

Change Storage Engine to WiredTiger sections in *Upgrade MongoDB to 3.0* (page 1054)

WiredTiger and Driver Version Compatibility For MongoDB 3.0 deployments that use the WiredTiger storage engine, the following operations return no output when issued in previous versions of the `mongo` (page 803) shell or drivers:

- `db.getCollectionNames()` (page 185)
- `db.collection.getIndexes()` (page 73)
- `show collections`
- `show tables`

Use the 3.0 `mongo` (page 803) shell or the *3.0 compatible version* (page 1051) of the official drivers when connecting to 3.0 `mongod` (page 770) instances that use WiredTiger. The 2.6.8 `mongo` (page 803) shell is also compatible with 3.0 `mongod` (page 770) instances that use WiredTiger.

`db.fsyncLock()` is not Compatible with WiredTiger With WiredTiger the `db.fsyncLock()` (page 180) and `db.fsyncUnlock()` (page 181) operations *cannot* guarantee that the data files do not change. As a result, do not use these methods to ensure consistency for the purposes of creating backups.

Support for `touch` Command If a storage engine does not support the `touch` (page 464), then the `touch` (page 464) command will return an error.

- The MMAPv1 storage engine supports `touch` (page 464).
- The WiredTiger storage engine *does not* support `touch` (page 464).

Dynamic Record Allocation MongoDB 3.0 no longer supports dynamic record allocation and deprecates *paddingFactor*.

MongoDB 3.0 deprecates the `newCollectionsUsePowerOf2Sizes` (page 932) parameter such that you can no longer use the parameter to disable the power of 2 sizes allocation for a collection. Instead, use the `collMod` (page 457) command with the `noPadding` (page 458) flag or the `db.createCollection()` (page 167) method with the `noPadding` option. Only set `noPadding` for collections with workloads that consist only of inserts or in-place updates (such as incrementing counters).

Warning: Only set `noPadding` (page 458) to `true` for collections whose workloads have *no* update operations that cause documents to grow, such as for collections with workloads that are insert-only. For more information, see *exact-fit-allocation*.

For more information, see *MMAPv1 Record Allocation Behavior Changes* (page 1042).

Replication Changes

Replica Set Oplog Format Change MongoDB 3.0 is not compatible with oplog entries generated by versions of MongoDB before 2.2.1. If you upgrade from one of these versions, you must wait for new oplog entries to overwrite *all* old oplog entries generated by one of these versions before upgrading to 3.0.0 or earlier.

Secondaries may abort if they replay a pre-2.6 oplog with an index build operation that would fail on a 2.6 or later primary.

Replica Set Configuration Validation MongoDB 3.0 provides a stricter validation of `replica set configuration settings` and replica sets invalid replica set configurations.

Stricter validations include:

- Arbiters can only have `1` vote. Previously, arbiters could also have a value of `0` for `members[n].votes`. If an arbiter has any value other than `1` for `members[n].votes`, you must fix the setting.
- Non-arbiter members can **only** have value of `0` or `1` for `members[n].votes`. If a non-arbiter member has any other value for `members[n].votes`, you must fix the setting.
- `_id` in the `https://docs.mongodb.org/manual/reference/replica-configuration` must specify the same name as that specified by `--replSet` or `replication.replSetName` (page 922). Otherwise, you must fix the setting.
- Disallows `0` for `settings.getLastErrorDefaults` value. If `settings.getLastErrorDefaults` value is `0`, you must fix the setting.
- `settings` can only contain the recognized settings. Previously, MongoDB ignored unrecognized settings. If `settings` contains unrecognized settings, you must remove the unrecognized settings.

To fix the settings before upgrading to MongoDB 3.0, connect to the primary and `reconfigure` (page 260) your replica set to valid configuration settings.

If you have already upgraded to MongoDB 3.0, you must *downgrade to MongoDB 2.6* (page 1062) first and then fix the settings. Once you have `reconfigured` (page 260) the replica set, you can re-upgrade to MongoDB 3.0.

Change of `w: majority` Semantics A write concern with a *w: majority* value is satisfied when a majority of the *voting* members replicates a write operation. In previous versions, *majority* referred a majority of all voting and non-voting members of the set.

Remove `local.slaves` Collection MongoDB 3.0 removes the `local.slaves` collection that tracked the secondaries' replication progress. To track the replication progress, use the `rs.status()` (page 262) method.

Replica Set State Change The `FATAL` replica set state does not exist as of 3.0.0.

HTTP Interface The HTTP Interface (i.e. `net.http.enabled` (page 905)) no longer reports replication data.

MongoDB Tools Changes

Require a Running MongoDB Instance The 3.0 versions of MongoDB tools, `mongodump` (page 816), `mongorestore` (page 824), `mongoexport` (page 850), `mongoimport` (page 841), `mongofiles` (page 878), and `mongooplog` (page 835), must connect to running MongoDB instances and these tools *cannot* directly modify the data files with `--dbpath` as in previous versions. Ensure that you start your `mongod` (page 770) instance(s) before using these tools.

Removed Options

- Removed `--dbpath`, `--journal`, and `--filter` options for `mongodump` (page 816), `mongorestore` (page 824), `mongoimport` (page 841), `mongoexport` (page 850), and `bsondump` (page 833).
- Removed `--locks` option for `mongotop` (page 867).
- Removed `--noobjcheck` option for `bsondump` (page 833) and `mongorestore` (page 824).
- Removed `--csv` option for `mongoexport` (page 850). Use the new *`--type`* (page 846) option to specify the export format type (`csv` or `json`).

See also:

MongoDB Tools Enhancements (page 1045)

Sharded Cluster Setting

Remove `releaseConnectionsAfterResponse` Parameter MongoDB now always releases connections after response. `releaseConnectionsAfterResponse` parameter is no longer available.

Security Changes

MongoDB 2.4 User Model Removed MongoDB 3.0 completely removes support for the deprecated 2.4 user model. MongoDB 3.0 will exit with an error message if there is user data with the 2.4 schema, i.e. if `authSchema` version is less than `3`.

To verify the version of your existing 2.6 schema, query the `system.version` collection in the `admin` database:

Note: You must have privileges to query the collection.

```
use admin
db.system.version.find( { _id: "authSchema" })
```

If you are currently using `auth` and you have schema version 2 or 3, the query returns the `currentVersion` of the existing `authSchema`.

If you do not currently have any users *or* you are using `authSchema` version 1, the query will not return any result.

If your `authSchema` version is less than `3` or the query does not return any results, see *Upgrade User Authorization Data to 2.6 Format* (page 1115) to upgrade the `authSchema` version before upgrading to MongoDB 3.0.

After upgrading MongoDB to 3.0 from 2.6, to use the new `SCRAM-SHA-1` challenge-response mechanism if you have existing user data, you will need to upgrade the authentication schema a second time. This upgrades the `MONGODB-CR` user model to `SCRAM-SHA-1` user model. See *Upgrade to SCRAM-SHA-1* (page 1058) for details.

Localhost Exception Changed In 3.0, the localhost exception changed so that these connections *only* have access to create the first user on the `admin` database. In previous versions, connections that gained access using the localhost exception had unrestricted access to the MongoDB instance.

See *localhost-exception* for more information.

`db.addUser()` Removed 3.0 removes the legacy `db.addUser()` method. Use `db.createUser()` (page 230) and `db.updateUser()` (page 232) instead.

TLS/SSL Configuration Option Changes MongoDB 3.0 introduced new `net.ssl.allowConnectionsWithoutCertificates` (page 908) configuration file setting and `--sslAllowConnectionsWithoutCertificates` command line option for `mongod` (page 770) and `mongos` (page 792). These options replace previous `net.ssl.weakCertificateValidation` and `--sslWeakCertificateValidation` options, which became aliases. Update your configuration to ensure future compatibility.

TLS/SSL Certificates Validation By default, when running in SSL mode, MongoDB instances will *only* start if its certificate (i.e. `net.ssl.PemKeyFile`) is valid. You can disable this behavior with the `net.ssl.allowInvalidCertificates` (page 908) setting or the `--sslAllowInvalidCertificates` command line option.

To start the `mongo` (page 803) shell with `--ssl`, you must explicitly specify either the `--sslCAFile` or `--sslAllowInvalidCertificates` option at startup. See `https://docs.mongodb.org/manual/tutorial/configure-ssl-clients` for more information.

TLS/SSL Certificate Hostname Validation By default, MongoDB validates the hostnames of hosts attempting to connect using certificates against the hostnames listed in those certificates. In certain deployment situations this behavior may be undesirable. It is now possible to disable such hostname validation without disabling validation of the rest of the certificate information with the `net.ssl.allowInvalidHostnames` (page 909) setting or the `--sslAllowInvalidHostnames` command line option.

SSLv3 Ciphers Disabled In light of vulnerabilities in legacy SSL ciphers[685], these ciphers have been explicitly disabled in MongoDB. No configuration changes are necessary.

`mongo` Shell Version Compatibility Versions of the `mongo` (page 803) shell before 3.0 are *not* compatible with 3.0 deployments of MongoDB that enforce access control. If you have a 3.0 MongoDB deployment that requires access control, you must use 3.0 versions of the `mongo` (page 803) shell.

HTTP Status Interface and REST API Compatibility Neither the HTTP status interface nor the REST API support the *SCRAM-SHA-1* challenge-response user authentication mechanism introduced in version 3.0.

Indexes

Remove `dropDups` Option `dropDups` option is no longer available for `createIndex()` (page 36), `ensureIndex()` (page 47), and `createIndexes` (page 446).

Changes to Restart Behavior during Background Indexing For 3.0 `mongod` (page 770) instances, if a background index build is in progress when the `mongod` (page 770) 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 770) will exit with an error.

To start the `mongod` (page 770) after a failed index build, use the `storage.indexBuildRetry` (page 915) or *`--noIndexBuildRetry`* to skip the index build on start up.

`2d` Indexes and Geospatial Near Queries For `$near` (page 565) queries that use a `2d` index:

- MongoDB no longer uses a default limit of 100 documents.
- Specifying a `batchSize()` (page 135) is no longer analogous to specifying a `limit()` (page 144).

For `$nearSphere` (page 567) queries that use a `2d` index, MongoDB no longer uses a default limit of 100 documents.

Driver Compatibility Changes Each officially supported driver has release a version that includes support for all new features introduced in MongoDB 3.0. Upgrading to one of these version is strongly recommended as part of the upgrade process.

A driver upgrade is **necessary** in certain scenarios due to changes in functionality:

- Use of the `SCRAM-SHA-1` authentication method
- Use of functionality that calls `listIndexes` (page 450) or `listCollections` (page 438)

The minimum 3.0-compatible driver versions are:

[685] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566

Driver Language	Minimum 3.0-Compatible Version
C[686]	1.1.0[687]
C++[688]	1.0.0[689]
C#[690]	1.10[691]
Java[692]	2.13[693]
Node.js[694]	1.4.29[695]
Perl[696]	0.708.0.0[697]
PHP[698]	1.6[699]
Python[700]	2.8[701]
Motor[702]	0.4[703]
Ruby[704]	1.12[705]
Scala[706]	2.8.0[707]

General Compatibility Changes

`findAndModify` Return Document In MongoDB 3.0, when performing an update with `findAndModify` (page 348) that also specifies `upsert: true` *and* either the `new` option is not set or `new: false`, `findAndModify` (page 348) returns `null` in the `value` field if the `query` does not match any document, regardless of the `sort` specification.

In previous versions, `findAndModify` (page 348) returns an empty document `{}` in the `value` field if a `sort` is specified for the update, and `upsert: true`, and the `new` option is not set or `new: false`.

`upsert:true` with a Dotted `_id` Query When you execute an `update()` (page 117) with `upsert: true` and the query matches no existing document, MongoDB will refuse to insert a new document if the query specifies conditions on the `_id` field using *dot notation*.

This restriction ensures that the order of fields embedded in the `_id` document is well-defined and not bound to the order specified in the query

If you attempt to insert a document in this way, MongoDB will raise an error.

For example, consider the following update operation. Since the update operation specifies `upsert:true` and the query specifies conditions on the `_id` field using dot notation, then the update will result in an error when constructing the document to insert.

[686] https://docs.mongodb.org/ecosystem/drivers/c
[687] https://github.com/mongodb/mongo-c-driver/releases
[688] https://github.com/mongodb/mongo-cxx-driver
[689] https://github.com/mongodb/mongo-cxx-driver/releases
[690] https://docs.mongodb.org/ecosystem/drivers/csharp
[691] https://github.com/mongodb/mongo-csharp-driver/releases
[692] https://docs.mongodb.org/ecosystem/drivers/java
[693] https://github.com/mongodb/mongo-java-driver/releases
[694] https://docs.mongodb.org/ecosystem/drivers/node-js
[695] https://github.com/mongodb/node-mongodb-native/releases
[696] https://docs.mongodb.org/ecosystem/drivers/perl
[697] http://search.cpan.org/dist/MongoDB/
[698] https://docs.mongodb.org/ecosystem/drivers/php
[699] http://pecl.php.net/package/mongo
[700] https://docs.mongodb.org/ecosystem/drivers/python
[701] https://pypi.python.org/pypi/pymongo/
[702] https://docs.mongodb.org/ecosystem/drivers/python
[703] https://pypi.python.org/pypi/motor/
[704] https://docs.mongodb.org/ecosystem/drivers/ruby
[705] https://rubygems.org/gems/mongo
[706] https://docs.mongodb.org/ecosystem/drivers/scala
[707] https://github.com/mongodb/casbah/releases

```
db.collection.update( { "_id.name": "Robert Frost", "_id.uid": 0 },
   { "categories": ["poet", "playwright"] },
   { upsert: true } )
```

Deprecate Access to `system.indexes` and `system.namespaces` MongoDB 3.0 deprecates *direct* access to `system.indexes` and `system.namespaces` collections. Use the `createIndexes` (page 446) and `listIndexes` (page 450) commands instead. See also *WiredTiger and Driver Version Compatibility* (page 1047).

Collection Name Validation MongoDB 3.0 more consistently enforces the `collection naming restrictions` (page 946). Ensure your application does not create or depend on invalid collection names.

Platform Support Commercial support is no longer provided for MongoDB on 32-bit platforms (Linux and Windows). Linux RPM and DEB packages are also no longer available. However, binary archives are still available.

Linux Package Repositories Non-Enterprise MongoDB Linux packages for 3.0 and later are in a new repository. Follow the appropriate `Linux installation instructions` to install the 3.0 packages from the new location.

Removed/Deprecated Commands The following commands and methods are no longer available in MongoDB 3.0:

- `closeAllDatabases`
- `getoptime`
- `text`
- `indexStats`, `db.collection.getIndexStats()`, and `db.collection.indexStats()`

The following commands and methods are deprecated in MongoDB 3.0:

- `diagLogging` (page 483)
- `eval` (page 358), `db.eval()` (page 178)
- `db.collection.copyTo()` (page 35)

In addition, you cannot use the now deprecated `eval` (page 358) command or the `db.eval()` (page 178) method to invoke `mapReduce` (page 318) or `db.collection.mapReduce()` (page 90).

Date and Timestamp Comparison Order MongoDB 3.0 no longer treats the *Timestamp* and the *Date* data types as equivalent for comparison purposes. Instead, the *Timestamp* data type has a higher comparison/sort order (i.e. is "greater") than the *Date* data type. If your application relies on the equivalent comparison/sort order of Date and Timestamp objects, modify your application accordingly before upgrading.

Server Status Output Change The `serverStatus` (page 492) command and the `db.serverStatus()` (page 197) method no longer return `workingSet`, `indexCounters`, and `recordStats` sections in the output.

Unix Socket Permissions Change Unix domain socket file permission now defaults to `0700`. To change the permission, MongoDB provides the `net.unixDomainSocket.filePermissions` (page 904) setting as well as the `--filePermission` option.

cloneCollection The cloneCollection (page 443) command and the db.cloneCollection() (page 162) method will now return an error if the collection already exists, instead of inserting into it.

Some changes in 3.0 can affect *compatibility* (page 1046) and may require user actions. For a detailed list of compatibility changes, see *Compatibility Changes in MongoDB 3.0* (page 1046).

Upgrade Process

Upgrade MongoDB to 3.0

On this page

- Upgrade Recommendations and Checklists (page 1054)
- Upgrade MongoDB Processes (page 1054)
- Upgrade Existing MONGODB-CR Users to Use SCRAM-SHA-1 (page 1058)
- General Upgrade Procedure (page 1058)

In the general case, the upgrade from MongoDB 2.6 to 3.0 is a binary-compatible "drop-in" upgrade: shut down the mongod (page 770) instances and replace them with mongod (page 770) instances running 3.0. **However**, before you attempt any upgrade please familiarize yourself with the content of this document, particularly the procedure for *upgrading sharded clusters* (page 1056).

If you need guidance on upgrading to 3.0, MongoDB offers consulting[708] to help ensure a smooth transition without interruption to your MongoDB application.

Upgrade Recommendations and Checklists When upgrading, consider the following:

Upgrade Requirements To upgrade an existing MongoDB deployment to 3.0, you must be running 2.6. If you're running a version of MongoDB before 2.6, you *must* upgrade to 2.6 before upgrading to 3.0. See *Upgrade MongoDB to 2.6* (page 1110) for the procedure to upgrade from 2.4 to 2.6. Once upgraded to MongoDB 2.6, you **cannot** downgrade to any version earlier than MongoDB 2.4.

If your existing MongoDB deployment is already running with authentication and authorization, your user data model authSchema must be at least version 3. To verify the version of your existing authSchema, see *MongoDB 2.4 User Model Removed* (page 1050). To upgrade your authSchema version, see *Upgrade User Authorization Data to 2.6 Format* (page 1115) for details.

Preparedness Before upgrading MongoDB, always test your application in a staging environment before deploying the upgrade to your production environment.

Some changes in MongoDB 3.0 require manual checks and intervention. Before beginning your upgrade, see the *Compatibility Changes in MongoDB 3.0* (page 1046) document to ensure that your applications and deployments are compatible with MongoDB 3.0. Resolve the incompatibilities in your deployment before starting the upgrade.

Downgrade Limitations Once upgraded to MongoDB 3.0, you **cannot** downgrade to a version lower than **2.6.8**.

If you upgrade to 3.0 and have run authSchemaUpgrade (page 372), you **cannot** downgrade to 2.6 without disabling *--auth* (page 774) or restoring a pre-upgrade backup, as authSchemaUpgrade (page 372) discards the MONGODB-CR credentials used in 2.6. See *Upgrade Existing MONGODB-CR Users to Use SCRAM-SHA-1* (page 1058).

Upgrade MongoDB Processes

[708] https://www.mongodb.com/products/consulting?jmp=docs#major_version_upgrade

Upgrade Standalone `mongod` Instance to MongoDB 3.0 The following steps outline the procedure to upgrade a standalone `mongod` (page 770) from version 2.6 to 3.0. To upgrade from version 2.4 to 3.0, *upgrade to version 2.6* (page 1110) *first*, and then use the following procedure to upgrade from 2.6 to 3.0.

Upgrade Binaries If you installed MongoDB from the MongoDB `apt`, `yum`, or `zypper` repositories, you should upgrade to 3.0 using your package manager. Follow the appropriate `installation instructions` for your Linux system. This will involve adding a repository for the new release, then performing the actual upgrade.

Otherwise, you can manually upgrade MongoDB:

Step 1: Download 3.0 binaries. Download binaries of the latest release in the 3.0 series from the MongoDB Download Page[709]. See `https://docs.mongodb.org/manual/installation` for more information.

Step 2: Replace 2.6 binaries. Shut down your `mongod` (page 770) instance. Replace the existing binary with the 3.0 `mongod` (page 770) binary and restart `mongod` (page 770).

Change Storage Engine for Standalone to WiredTiger To change the storage engine for a standalone `mongod` (page 770) instance to WiredTiger, see `https://docs.mongodb.org/manual/tutorial/change-standalone-wiredtiger`.

Upgrade a Replica Set to 3.0

Prerequisites

- If the oplog contains entries generated by versions of MongoDB that precede version 2.2.1, you must wait for the entries to be overwritten by later versions *before* you can upgrade to MongoDB 3.0. For more information, see *Replica Set Oplog Format Change* (page 1048)
- *Stricter validation in MongoDB 3.0* (page 1048) of replica set configuration may invalidate previously-valid replica set configuration, preventing replica sets from starting in MongoDB 3.0. For more information, see *Replica Set Configuration Validation* (page 1048).
- All replica set members must be running version 2.6 before you can upgrade them to version 3.0. To upgrade a replica set from an earlier MongoDB version, *upgrade all members of the replica set to version 2.6* (page 1110) *first*, and then follow the procedure to upgrade from MongoDB 2.6 to 3.0.

Upgrade Binaries You can upgrade from MongoDB 2.6 to 3.0 using a "rolling" upgrade to minimize downtime by upgrading the members individually while the other members are available:

Step 1: Upgrade secondary members of the replica set. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` (page 770) and replacing the 2.6 binary with the 3.0 binary. After upgrading a `mongod` (page 770) 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 262) in the `mongo` (page 803) shell.

Step 2: Step down the replica set primary. Use `rs.stepDown()` (page 263) in the `mongo` (page 803) shell to step down the *primary* and force the set to *failover*. `rs.stepDown()` (page 263) expedites the failover procedure and is preferable to shutting down the primary directly.

[709] http://www.mongodb.org/downloads?jmp=docs

Step 3: Upgrade the primary. When `rs.status()` (page 262) shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` (page 770) binary with the 3.0 binary and start the new instance.

Replica set failover is not instant and will render the set unavailable to accept writes until the failover process completes. This may take 30 seconds or more: schedule the upgrade procedure during a scheduled maintenance window.

Change Replica Set Storage Engine to WiredTiger To change the storage engine for a replica set to WiredTiger, see `https://docs.mongodb.org/manual/tutorial/change-replica-set-wiredtiger`.

Upgrade a Sharded Cluster to 3.0 Only upgrade sharded clusters to 3.0 if **all** members of the cluster are currently running instances of 2.6. The only supported upgrade path for sharded clusters running 2.4 is via 2.6. The upgrade process checks all components of the cluster and will produce warnings if any component is running version 2.4.

Considerations The upgrade process does not require any downtime. However, while you upgrade the sharded cluster, ensure that clients do not make changes to the collection meta-data. For example, during the upgrade, do **not** do any of the following:

- `sh.enableSharding()` (page 273)
- `sh.shardCollection()` (page 277)
- `sh.addShard()` (page 269)
- `db.createCollection()` (page 167)
- `db.collection.drop()` (page 45)
- `db.dropDatabase()` (page 177)
- any operation that creates a database
- any other operation that modifies the cluster metadata in any way. See `https://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `https://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 3.0 `mongos` (page 792) instance with the `configDB` (page 926) pointing to the cluster's config servers and with the `--upgrade` option.

To run a `mongos` (page 792) with the `--upgrade` option, you can upgrade an existing `mongos` (page 792) instance to 3.0, or if you need to avoid reconfiguring a production `mongos` (page 792) instance, you can use a new 3.0 `mongos` (page 792) 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 792) instances in your cluster, such as *--sslOnNormalPorts* or *--sslPEMKeyFile*.

The 3.0 mongos (page 792) will output informational log messages.

```
<timestamp> I SHARDING [mongosMain] MongoS version 3.0.0 starting: ...
...
<timestamp> I SHARDING [mongosMain] starting upgrade of config server from v5 to v6
<timestamp> I SHARDING [mongosMain] starting next upgrade step from v5 to v6
<timestamp> I SHARDING [mongosMain] about to log new metadata event: ...
<timestamp> I SHARDING [mongosMain] checking that version of host ... is compatible with 2.6
...
<timestamp> I SHARDING [mongosMain] upgrade of config server to v6 successful
...
<timestamp> I SHARDING [mongosMain] distributed lock 'configUpgrade/...' unlocked.
<timestamp> I -        [mongosMain] Config database is at version v6
```

The mongos (page 792) 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 792) will exit upon completion of the meta data upgrade process. If successful, the process will log the following messages:

```
<timestamp> I SHARDING [mongosMain] upgrade of config server to v6 successful
...
<timestamp> I -        [mongosMain] Config database is at version v6
```

After a successful upgrade, restart the mongos (page 792) instance. If mongos (page 792) fails to start, check the log for more information.

If the mongos (page 792) instance loses its connection to the config servers during the upgrade or if the upgrade is otherwise unsuccessful, you may always safely retry the upgrade.

Step 4: Upgrade the remaining `mongos` instances to 3.0. Upgrade and restart **without** the *--upgrade* (page 796) option the other mongos (page 792) instances in the sharded cluster.

After you have successfully upgraded *all* mongos (page 792) instances, you can proceed to upgrade the other components in your sharded cluster.

Warning: Do not upgrade the mongod (page 770) instances until after you have upgraded *all* the mongos (page 792) instances.

Step 5: Upgrade the config servers. After you have successfully upgraded *all* mongos (page 792) instances, upgrade all 3 mongod (page 770) config server instances, leaving the *first* config server listed in the *mongos --configdb* (page 795) argument to upgrade *last*.

Step 6: Upgrade the shards. Upgrade each shard, one at a time, upgrading the mongod (page 770) secondaries before running replSetStepDown (page 405) 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*.

Change Sharded Cluster Storage Engine to WiredTiger For a sharded cluster in MongoDB 3.0, you can choose to update the shards to use WiredTiger storage engine and have the config servers use MMAPv1. If you update the config servers to use WiredTiger, you must update all three config servers to use WiredTiger.

To change a sharded cluster to use WiredTiger, see `https://docs.mongodb.org/manual/tutorial/change-sharded-c`

Upgrade Existing `MONGODB-CR` Users to Use `SCRAM-SHA-1` After upgrading the binaries, see *Upgrade to SCRAM-SHA-1* (page 1058) for details on `SCRAM-SHA-1` upgrade scenarios.

General Upgrade Procedure **Except** as described on this page, moving between 2.6 and 3.0 is a drop-in replacement:

Step 1: Stop the existing `mongod` instance. For example, on Linux, run 2.6 `mongod` (page 770) with the *`--shutdown`* (page 775) option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915). See also the *terminate-mongod-processes* for alternate methods of stopping a `mongod` (page 770) instance.

Step 2: Start the new `mongod` instance. Ensure you start the 3.0 `mongod` (page 770) with the same `dbPath` (page 915):

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915).

Upgrade to `SCRAM-SHA-1`

On this page

- Overview (page 1058)
- Considerations (page 1059)
- Upgrade 2.6 `MONGODB-CR` Users to `SCRAM-SHA-1` (page 1061)
- Result (page 1061)
- Additional Resources (page 1062)

Overview MongoDB 3.0 includes support for the *SCRAM-SHA-1* challenge-response user authentication mechanism, which changes how MongoDB uses and stores user credentials.

For deployments that already contain user authentication data, to use the `SCRAM-SHA-1` mechanism, you must upgrade the authentication schema in addition to upgrading the MongoDB processes.

You may, alternatively, opt to continue to use the `MONGODB-CR` challenge-response mechanism and skip this upgrade.

See *Upgrade Scenarios* (page 1058) for details.

Upgrade Scenarios The following scenarios are possible when upgrading from 2.6 to 3.0:

Continue to Use `MONGODB-CR` If you are upgrading from a 2.6 database with **existing** user authentication data, to continue to use `MONGODB-CR` for existing challenge-response users, **no upgrade to the existing user data is required**. However, new challenge-response users created in 3.0 will use the following authentication mechanism:

- If you populated MongoDB 3.0 user data by importing the 2.6 user authentication data, including user data, new challenge-response users created in MongoDB 3.0 will use `SCRAM-SHA1`.
- If you run MongoDB 3.0 binary against the 2.6 data files, including the user authentication data files, new challenge-response users created in MongoDB 3.0 will continue to use the `MONGODB-CR`.

You can execute the upgrade to `SCRAM-SHA-1` at any point in the future.

Use `SCRAM-SHA-1`

- If you are starting with a new 3.0 installation without any users or upgrading from a 2.6 database that has no users, to use `SCRAM-SHA-1`, **no user data upgrade is required**. All newly created users will have the correct format for `SCRAM-SHA-1`.
- If you are upgrading from a 2.6 database with **existing** user data, to use `SCRAM-SHA-1`, follow the steps in *Upgrade 2.6 MONGODB-CR Users to SCRAM-SHA-1* (page 1061).

Important: Before you attempt any upgrade, familiarize yourself with the *Considerations* (page 1059) as the upgrade to `SCRAM-SHA-1` is **irreversible** short of restoring from backups.

Recommendation `SCRAM-SHA-1` represents a significant improvement in security over `MONGODB-CR`, the previous default authentication mechanism: you are strongly urged to upgrade. For advantages of using `SCRAM-SHA-1`, see *SCRAM-SHA-1*.

Considerations

Backwards Incompatibility The procedure to upgrade to `SCRAM-SHA-1` **discards** the `MONGODB-CR` credentials used by 2.6. As such, the procedure is **irreversible**, short of restoring from backups.

The procedure also disables `MONGODB-CR` as an authentication mechanism.

Upgrade Binaries Before upgrading the authentication model, you should first upgrade MongoDB binaries to 3.0. For sharded clusters, ensure that **all** cluster components are 3.0.

Upgrade Drivers You must upgrade all drivers used by applications that will connect to upgraded database instances to version that support `SCRAM-SHA-1`. The minimum driver versions that support `SCRAM-SHA-1` are:

Driver Language	Version
C[710]	1.1.0[711]
C++[712]	1.0.0[713]
C#[714]	1.10[715]
Java[716]	2.13[717]
Node.js[718]	1.4.29[719]
Perl[720]	0.708.0.0[721]
PHP[722]	1.6[723]
Python[724]	2.8[725]
Motor[726]	0.4[727]
Ruby[728]	1.12[729]
Scala[730]	2.8.0[731]

See the MongoDB Drivers Page[732] for links to download upgraded drivers.

Requirements To upgrade the authentication model, you must have a user in the `admin` database with the role `userAdminAnyDatabase`.

Timing Because downgrades are more difficult after you upgrade the user authentication model, once you upgrade the MongoDB binaries to version 3.0, allow your MongoDB deployment to run for a day or two before following this procedure.

This allows 3.0 some time to "burn in" and decreases the likelihood of downgrades occurring after the user privilege model upgrade. The user authentication and access control will continue to work as it did in 2.6.

If you decide to upgrade the user authentication model immediately instead of waiting the recommended "burn in" period, then for sharded clusters, you must wait at least 10 seconds after upgrading the sharded clusters to run the authentication upgrade command.

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.

[710] https://docs.mongodb.org/ecosystem/drivers/c
[711] https://github.com/mongodb/mongo-c-driver/releases
[712] https://github.com/mongodb/mongo-cxx-driver
[713] https://github.com/mongodb/mongo-cxx-driver/releases
[714] https://docs.mongodb.org/ecosystem/drivers/csharp
[715] https://github.com/mongodb/mongo-csharp-driver/releases
[716] https://docs.mongodb.org/ecosystem/drivers/java
[717] https://github.com/mongodb/mongo-java-driver/releases
[718] https://docs.mongodb.org/ecosystem/drivers/node-js
[719] https://github.com/mongodb/node-mongodb-native/releases
[720] https://docs.mongodb.org/ecosystem/drivers/perl
[721] http://search.cpan.org/dist/MongoDB/
[722] https://docs.mongodb.org/ecosystem/drivers/php
[723] http://pecl.php.net/package/mongo
[724] https://docs.mongodb.org/ecosystem/drivers/python
[725] https://pypi.python.org/pypi/pymongo/
[726] https://docs.mongodb.org/ecosystem/drivers/python
[727] https://pypi.python.org/pypi/motor/
[728] https://docs.mongodb.org/ecosystem/drivers/ruby
[729] https://rubygems.org/gems/mongo
[730] https://docs.mongodb.org/ecosystem/drivers/scala
[731] https://github.com/mongodb/casbah/releases
[732] https://docs.mongodb.org/ecosystem/drivers

Sharded Clusters For a sharded cluster, connect to one `mongos` (page 792) instance and run the upgrade procedure to upgrade the cluster's authentication data. By default, the procedure will upgrade the authentication data of the shards as well.

To override this behavior, run `authSchemaUpgrade` (page 372) with the `upgradeShards: false` option. If you choose to override, you must run the upgrade procedure on the `mongos` (page 792) 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 792) instance to interact with the config database.

Upgrade 2.6 `MONGODB-CR` Users to `SCRAM-SHA-1`

Warning: The procedure to upgrade to `SCRAM-SHA-1` **discards** the `MONG`
such, the procedure is **irreversible**, short of restoring from backups.
The procedure also disables `MONGODB-CR` as an authentication mechanism.

Important: To use the `SCRAM-SHA-1` authentication mechanism, a driver upgrade is **necessary** if your current driver version does not support `SCRAM-SHA-1`. See *required driver versions* (page 1059) for details.

Step 1: Connect to the MongoDB instance. Connect and authenticate to the `mongod` (page 770) instance for a single deployment, the primary `mongod` for a replica set, or a `mongos` (page 792) for a sharded cluster as an `admin` database user with the role `userAdminAnyDatabase`.

Step 2: Upgrade authentication schema. Use the `authSchemaUpgrade` (page 372) command in the `admin` database to update the user data using the `mongo` (page 803) shell.

Run `authSchemaUpgrade` command.

```
db.adminCommand({authSchemaUpgrade: 1});
```

In case of error, you may safely rerun the `authSchemaUpgrade` (page 372) command.

Sharded cluster `authSchemaUpgrade` consideration. For a sharded cluster *without shard local users*, `authSchemaUpgrade` (page 372) will, by default, upgrade the authorization data of the shards as well, completing the upgrade.

You can, however, override this behavior by including `upgradeShards: false` in the command, as in the following example:

```
db.adminCommand(
   {authSchemaUpgrade: 1, upgradeShards: false }
);
```

If you override the default behavior or your cluster has shard local users, after running `authSchemaUpgrade` (page 372) on a `mongos` (page 792) instance, you will need to connect to the primary for each shard and repeat the upgrade process after upgrading on the `mongos` (page 792).

Result After this procedure is complete, all users in the database will have `SCRAM-SHA-1`-style credentials, and any subsequently-created users will also have this type of credentials.

Additional Resources

- Blog Post: Improved Password-Based Authentication in MongoDB 3.0: SCRAM Explained (Part 1)[733]
- Blog Post: Improved Password-Based Authentication in MongoDB 3.0: SCRAM Explained (Part 2)[734]

Downgrade MongoDB from 3.0

On this page

- Downgrade Recommendations and Checklist (page 1062)
- Downgrade MongoDB Processes (page 1062)
- General Downgrade Procedure (page 1067)

Before you attempt any downgrade, familiarize yourself with the content of this document, particularly the *Downgrade Recommendations and Checklist* (page 1062) and the procedure for *downgrading sharded clusters* (page 1064).

Downgrade Recommendations and Checklist When downgrading, consider the following:

Downgrade Path Once upgraded to MongoDB 3.0, you **cannot** downgrade to a version lower than **2.6.8**.

Important: If you upgrade to MongoDB 3.0 and have run `authSchemaUpgrade` (page 372), you **cannot** downgrade to the 2.6 series without disabling `--auth` (page 774).

Procedures Follow the downgrade procedures:

- To downgrade sharded clusters, see *Downgrade a 3.0 Sharded Cluster* (page 1064).
- To downgrade replica sets, see *Downgrade a 3.0 Replica Set* (page 1063).
- To downgrade a standalone MongoDB instance, see *Downgrade a Standalone mongod Instance* (page 1062).

Note: *Optional*. Consider `compacting` (page 454) collections after downgrading. Otherwise, older versions will not be able to reuse free space regions larger than 2MB created while running 3.0. This can result in wasted space but no data loss following the downgrade.

Downgrade MongoDB Processes

Downgrade a Standalone `mongod` Instance If you have changed the storage engine to `WiredTiger`, change the storage engine to MMAPv1 before downgrading to 2.6.

Change Storage Engine to MMAPv1 To change storage engine to MMAPv1 for a standalone `mongod` (page 770) instance, you will need to manually export and upload the data using `mongodump` (page 816) and `mongorestore` (page 824).

Step 1: Ensure 3.0 `mongod` is running with WiredTiger.

[733] https://www.mongodb.com/blog/post/improved-password-based-authentication-mongodb-30-scram-explained-part-1?jmp=docs
[734] https://www.mongodb.com/blog/post/improved-password-based-authentication-mongodb-30-scram-explained-part-2?jmp=docs

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 816) 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 770) has read and write permissions for the new directory.

Step 4: Restart the mongod with MMAPv1. Restart the 3.0 mongod (page 770), specifying the newly created data directory for MMAPv1 as the *--dbpath* (page 776). You do not have to specify *--storageEngine* (page 775) 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 824) for available options.

Downgrade Binaries The following steps outline the procedure to downgrade a standalone mongod (page 770) from version 3.0 to 2.6.

Once upgraded to MongoDB 3.0, you **cannot** downgrade to a version lower than **2.6.8**.

Step 1: Download 2.6 binaries. Download binaries of the latest release in the 2.6 series from the MongoDB Download Page[735]. See https://docs.mongodb.org/manual/installation for more information.

Step 2: Replace with 2.6 binaries. Shut down your mongod (page 770) instance. Replace the existing binary with the 2.6 mongod (page 770) binary and restart mongod (page 770).

Downgrade a 3.0 Replica Set If you have changed the storage engine to WiredTiger, change the storage engine to MMAPv1 before downgrading to 2.6.

Change Storage Engine to MMAPv1 You can update members to use the MMAPv1 storage engine in a rolling manner.

Note: When running a replica set with mixed storage engines, performance can vary according to workload.

To change the storage engine to MMAPv1 for an existing secondary replica set member, remove the member's data and perform an `initial sync`:

Step 1: Shutdown the secondary member. Stop the mongod (page 770) instance for the secondary member.

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

Step 2: Prepare data directory for MMAPv1. Prepare *--dbpath* (page 776) 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 770) has read and write permissions for the new directory.

Step 3: Restart the secondary member with MMAPv1. Restart the 3.0 `mongod` (page 770), specifying the MMAPv1 data directory as the *--dbpath* (page 776). Specify additional options as appropriate for the member. You do not have to specify *--storageEngine* (page 775) since MMAPv1 is the default.

```
mongod --dbpath <preparedMMAPv1DBPath>
```

Since no data exists in the `--dbpath`, the `mongod` (page 770) will perform an initial sync. The length of the initial sync process depends on the size of the database and network connection between members of the replica set.

Repeat for the remaining the secondary members. Once all the secondary members have switched to MMAPv1, step down the primary, and update the stepped-down member.

Downgrade Binaries Once upgraded to MongoDB 3.0, you **cannot** downgrade to a version lower than **2.6.8**.

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 770). See *terminate-mongod-processes* for instructions on safely terminating `mongod` (page 770) processes.
2. Replace the 3.0 binary with the 2.6 binary and restart.
3. Wait for the member to recover to `SECONDARY` state before downgrading the next secondary. To check the member's state, use the `rs.status()` (page 262) method in the `mongo` (page 803) shell.

Step 2: Step down the primary. Use `rs.stepDown()` (page 263) in the `mongo` (page 803) shell to step down the *primary* and force the normal *failover* procedure.

```
rs.stepDown()
```

`rs.stepDown()` (page 263) 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 262) shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` (page 770) binary with the 2.6 binary and start the new instance.

Replica set failover is not instant but will render the set unavailable to writes and interrupt reads until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the downgrade during a predetermined maintenance window.

Downgrade a 3.0 Sharded Cluster

Requirements While the downgrade is in progress, you cannot make changes to the collection meta-data. For example, during the downgrade, do **not** do any of the following:

- `sh.enableSharding()` (page 273)
- `sh.shardCollection()` (page 277)
- `sh.addShard()` (page 269)
- `db.createCollection()` (page 167)
- `db.collection.drop()` (page 45)
- `db.dropDatabase()` (page 177)
- any operation that creates a database
- any other operation that modifies the cluster meta-data in any way. See `https://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `https://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.

Change Shards to Use MMAPv1 To change the storage engine to MMAPv1, refer to the procedure in *Change Storage Engine to MMAPv1 for replica set members* (page 1063) and *Change Storage Engine to MMAPv1 for standalone mongod* (page 1062) as appropriate for your shards.

Change Config Servers to Use MMAPv1

Note: During this process, only two config servers will be running at any given time to ensure that the sharded cluster's metadata is *read only*.

Step 1: Disable the Balancer. Turn off the *balancer* in the sharded cluster, as described in *sharding-balancing-disable-temporarily*.

Step 2: Stop the last config server listed in the `mongos`‘ `configDB` setting.

Step 3: Export data of the second config server listed in the `mongos`‘ `configDB` setting.

```
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See `mongodump` (page 816) for available options.

Step 4: For the second config server, create a new data directory for MMAPv1. Ensure that the user account running `mongod` (page 770) has read and write permissions for the new directory.

Step 5: Restart the second config server with MMAPv1. Specify the newly created MMAPv1 data directory as the *`--dbpath`* (page 776) as well as any additional options as appropriate.

```
mongod --dbpath <newMMAPv1DBPath> --configsvr
```

Step 6: Upload the exported data using `mongorestore` to the second config server.

```
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` (page 824) for available options.

Step 7: Shut down the second config server.

Step 8: Restart the third config server.

Step 9: Export data of the third config server.

```
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See `mongodump` (page 816) for available options.

Step 10: For the third config server, create a new data directory for MMAPv1. Ensure that the user account running `mongod` (page 770) has read and write permissions for the new directory.

Step 11: Restart the third config server with MMAPv1. Specify the newly created MMAPv1 data directory as the *`--dbpath`* (page 776) as well as any additional options as appropriate.

```
mongod --dbpath <newMMAPv1DBPath> --configsvr
```

Step 12: Upload the exported data using `mongorestore` to the third config server.

```
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` (page 824) for available options

Step 13: Export data of the first config server listed in the `mongos`‘ `configDB` setting.

```
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See `mongodump` (page 816) for available options.

Step 14: For the first config server, create data directory for MMAPv1. Ensure that the user account running `mongod` (page 770) has read and write permissions for the new directory.

Step 15: Restart the first config server with MMAPv1. Specify the newly created MMAPv1 data directory as the *`--dbpath`* (page 776) as well as any additional options as appropriate.

```
mongod --dbpath <newMMAPv1DBPath> --configsvr
```

Step 16: Upload the exported data using `mongorestore` to the first config server.

```
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` (page 824) for available options

Step 17: Enable writes to the sharded cluster's metadata. Restart the **second** config server, specifying the newly created MMAPv1 data directory as the *--dbpath* (page 776). Specify additional options as appropriate.

```
mongod --dbpath <newMMAPv1DBPath> --configsvr
```

Once all three config servers are up, the sharded cluster's metadata is available for writes.

Step 18: Re-enable the balancer. Once all three config servers are up and running with WiredTiger, *Re-enable the balancer*.

Downgrade Binaries Once upgraded to MongoDB 3.0, you **cannot** downgrade to a version lower than **2.6.8**.

The downgrade procedure for a sharded cluster reverses the order of the upgrade procedure. The version `v6` config database is backwards compatible with MongoDB 2.6.

Step 1: Disable the Balancer. Turn off the *balancer* 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 770) secondaries *before* downgrading the primary.
2. To downgrade the primary, run `replSetStepDown` (page 405) and downgrade.

Step 3: Downgrade the config servers. Downgrade all 3 `mongod` (page 770) 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 792), 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 3.0 is a drop-in replacement:

Step 1: Stop the existing `mongod` instance. For example, on Linux, run 3.0 `mongod` (page 770) with the *--shutdown* (page 775) option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915). See also the *terminate-mongod-processes* for alternate methods of stopping a `mongod` (page 770) instance.

Step 2: Start the new `mongod` instance. Ensure you start the 2.6 `mongod` (page 770) with the same `dbPath` (page 915):

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915).

See *Upgrade MongoDB to 3.0* (page 1054) for full upgrade instructions.

Download

To download MongoDB 3.0, go to the downloads page[736].

See also:

- All Third Party License Notices[737].
- All JIRA issues resolved in 3.0[738].

Additional Resources

- Blog Post: Announcing MongoDB 3.0[739]
- Whitepaper: What's New in MongoDB 3.0[740]
- Webinar: What's New in MongoDB 3.0[741]

7.2.2 Release Notes for MongoDB 2.6

On this page

- Minor Releases (page 1068)
- Major Changes (page 1094)
- Security Improvements (page 1096)
- Query Engine Improvements (page 1096)
- Improvements (page 1096)
- Operational Changes (page 1098)
- MongoDB Enterprise Features (page 1099)
- Additional Information (page 1099)

April 8, 2014

MongoDB 2.6 is now available. Key features include aggregation enhancements, text-search integration, query-engine improvements, a new write-operation protocol, and security enhancements.

Minor Releases

2.6 Changelog

[736] http://www.mongodb.org/downloads
[737] https://github.com/mongodb/mongo/blob/v3.0/distsrc/THIRD-PARTY-NOTICES
[738] http://bit.ly/1CpOu6t
[739] http://www.mongodb.com/blog/post/announcing-mongodb-30?jmp=docs
[740] https://www.mongodb.com/lp/white-paper/mongodb-3.0?jmp=docs
[741] https://www.mongodb.com/webinar/Whats-New-in-MongoDB-3-0?jmp=docs

On this page

- 2.6.11 – Changes (page 1069)
- 2.6.10 – Changes (page 1070)
- 2.6.9 – Changes (page 1072)
- 2.6.8 – Changes (page 1073)
- 2.6.7 – Changes (page 1075)
- 2.6.6 – Changes (page 1075)
- 2.6.5 – Changes (page 1078)
- 2.6.4 – Changes (page 1081)
- 2.6.3 – Changes (page 1085)
- 2.6.2 – Changes (page 1085)
- 2.6.1 – Changes (page 1089)

2.6.11 – Changes

Querying

- SERVER-19553[742] `mongod` (page 770) shouldn't use `sayPiggyBack` to send `killCursor` messages
- SERVER-18620[743] Reduce frequency of "`staticYield can't unlock`" log message
- SERVER-18461[744] Range predicates comparing against a BinData value should be covered, but are not in 2.6
- SERVER-17815[745] Plan ranking tie breaker is computed incorrectly
- SERVER-16265[746] Add query details to getmore entry in profiler and `db.currentOp()` (page 171)
- SERVER-15217[747] v2.6 query plan ranking test "`NonCoveredIxisectFetchesLess`" relies on order of deleted record list
- SERVER-14070[748] Compound index not providing sort if equality predicate given on sort field

Replication

- SERVER-18280[749] `ReplicaSetMonitor` should use `electionId` to avoid talking to old primaries
- SERVER-18795[750] `db.printSlaveReplicationInfo()` (page 195)/`rs.printSlaveReplicationInfo()` (page 260) can not work with `ARBITER` role

Sharding

- SERVER-19464[751] `$sort` (page 649) stage in aggregation doesn't call scoped connections done ()
- SERVER-18955[752] `mongos` (page 792) doesn't set batch size (and keeps the old one, 0) on getMore if performed on first `_cursor->more()`

[742]https://jira.mongodb.org/browse/SERVER-19553
[743]https://jira.mongodb.org/browse/SERVER-18620
[744]https://jira.mongodb.org/browse/SERVER-18461
[745]https://jira.mongodb.org/browse/SERVER-17815
[746]https://jira.mongodb.org/browse/SERVER-16265
[747]https://jira.mongodb.org/browse/SERVER-15217
[748]https://jira.mongodb.org/browse/SERVER-14070
[749]https://jira.mongodb.org/browse/SERVER-18280
[750]https://jira.mongodb.org/browse/SERVER-18795
[751]https://jira.mongodb.org/browse/SERVER-19464
[752]https://jira.mongodb.org/browse/SERVER-18955

Indexing

- SERVER-19559[753] Document growth of "key too large" document makes it disappear from the index
- SERVER-16348[754] `Assertion failure n >= 0 && n < static_cast<int>(_files.size()) src/mongo/db/storage/extent_manager.cpp 109`
- SERVER-13875[755] `ensureIndex()` (page 47) of `2dsphere` index breaks after upgrading to 2.6 (with the new `createIndex` command)

Networking SERVER-19389[756] Remove wire level endianness check

Build and Testing

- SERVER-18097[757] Remove `mongosTest_auth` and `mongosTest_WT` tasks from `evergreen.yml`
- SERVER-18068[758] Coverity analysis defect 72413: Resource leak
- SERVER-18371[759] Add SSL library config detection

2.6.10 – Changes

Security

- SERVER-18312[760] Upgrade PCRE to latest
- SERVER-17812[761] LockPinger has audit-related GLE failure
- SERVER-17647[762] Compute BinData length in v8
- SERVER-17591[763] Add SSL flag to select supported protocols
- SERVER-16849[764] On mongos we always invalidate the user cache once, even if no user definitions are changing
- SERVER-11980[765] Improve user cache invalidation enforcement on mongos

Querying

- SERVER-18364[766] Ensure non-negation predicates get chosen over negation predicates for multikey index bounds construction
- SERVER-17815[767] Plan ranking tie breaker is computed incorrectly
- SERVER-16256[768] $all clause with elemMatch uses wider bounds than needed

[753] https://jira.mongodb.org/browse/SERVER-19559
[754] https://jira.mongodb.org/browse/SERVER-16348
[755] https://jira.mongodb.org/browse/SERVER-13875
[756] https://jira.mongodb.org/browse/SERVER-19389
[757] https://jira.mongodb.org/browse/SERVER-18097
[758] https://jira.mongodb.org/browse/SERVER-18068
[759] https://jira.mongodb.org/browse/SERVER-18371
[760] https://jira.mongodb.org/browse/SERVER-18312
[761] https://jira.mongodb.org/browse/SERVER-17812
[762] https://jira.mongodb.org/browse/SERVER-17647
[763] https://jira.mongodb.org/browse/SERVER-17591
[764] https://jira.mongodb.org/browse/SERVER-16849
[765] https://jira.mongodb.org/browse/SERVER-11980
[766] https://jira.mongodb.org/browse/SERVER-18364
[767] https://jira.mongodb.org/browse/SERVER-17815
[768] https://jira.mongodb.org/browse/SERVER-16256

Replication

- SERVER-18211[769] MongoDB fails to correctly roll back collection creation
- SERVER-17771[770] Reconfiguring a replica set to remove a node causes a segmentation fault on 2.6.8
- SERVER-13542[771] Expose electionId on primary in isMaster

Sharding

- SERVER-17812[772] LockPinger has audit-related GLE failure
- SERVER-17805[773] logOp / OperationObserver should always check shardversion
- SERVER-17749[774] `collMod` (page 457) `usePowerOf2Sizes` (page 458) fails on `mongos` (page 792)
- SERVER-11980[775] Improve user cache invalidation enforcement on `mongos` (page 792)

Storage

- SERVER-18211[776] MongoDB fails to correctly roll back collection creation
- SERVER-17653[777] ERROR: socket XXX is higher than 1023; not supported on 2.6.*

Indexing SERVER-17018[778] Assertion failure false `src/mongo/db/structure/btree/key.cpp` Line 433 on remove operation

Write Ops

- SERVER-18111[779] `mongod` (page 770) allows user inserts into `system.profile` collection
- SERVER-13542[780] Expose `electionId` on primary in `isMaster` (page 409)

Networking

- SERVER-18096[781] Shard primary incorrectly reuses closed sockets after relinquish and re-election
- SERVER-17591[782] Add SSL flag to select supported protocols

Build and Packaging

- SERVER-18344[783] logs should be sent to updated logkeeper server
- SERVER-18082[784] Change smoke.py buildlogger command line options to environment variables

[769] https://jira.mongodb.org/browse/SERVER-18211
[770] https://jira.mongodb.org/browse/SERVER-17771
[771] https://jira.mongodb.org/browse/SERVER-13542
[772] https://jira.mongodb.org/browse/SERVER-17812
[773] https://jira.mongodb.org/browse/SERVER-17805
[774] https://jira.mongodb.org/browse/SERVER-17749
[775] https://jira.mongodb.org/browse/SERVER-11980
[776] https://jira.mongodb.org/browse/SERVER-18211
[777] https://jira.mongodb.org/browse/SERVER-17653
[778] https://jira.mongodb.org/browse/SERVER-17018
[779] https://jira.mongodb.org/browse/SERVER-18111
[780] https://jira.mongodb.org/browse/SERVER-13542
[781] https://jira.mongodb.org/browse/SERVER-18096
[782] https://jira.mongodb.org/browse/SERVER-17591
[783] https://jira.mongodb.org/browse/SERVER-18344
[784] https://jira.mongodb.org/browse/SERVER-18082

- SERVER-18312[785] Upgrade PCRE to latest
- SERVER-17780[786] Init script sets process ulimit to different value compared to documentation
- SERVER-16563[787] Debian repo component mismatch - mongodb/10gen

Shell SERVER-17951[788] db.currentOp() fails with read preference set

Testing

- SERVER-18262[789] setup_multiversion_mongodb should retry links download on timeouts
- SERVER-18229[790] `smoke.py` with PyMongo 3.0.1 fails to run certain tests
- SERVER-18073[791] Fix `smoke.py` to work with PyMongo 3.0

2.6.9 – Changes

Security SERVER-16073[792] Create hidden `net.ssl.sslCipherConfig` flag

Querying

- SERVER-14723[793] Crash during query planning for `geoNear` (page 327) with multiple `2dsphere` indexes
- SERVER-14071[794] For queries with `sort()` (page 156), bad non-blocking plan can be cached if there are zero results
- SERVER-8188[795] Configurable idle cursor timeout

Replication and Sharding

- SERVER-17429[796] the message logged when changing sync target due to stale data should format OpTimes in a consistent way
- SERVER-17441[797] `mongos` (page 792) crash right after "not master" error

Storage SERVER-15907[798] Use `ftruncate` rather than `fallocate` when running on `tmpfs`

[785] https://jira.mongodb.org/browse/SERVER-18312
[786] https://jira.mongodb.org/browse/SERVER-17780
[787] https://jira.mongodb.org/browse/SERVER-16563
[788] https://jira.mongodb.org/browse/SERVER-17951
[789] https://jira.mongodb.org/browse/SERVER-18262
[790] https://jira.mongodb.org/browse/SERVER-18229
[791] https://jira.mongodb.org/browse/SERVER-18073
[792] https://jira.mongodb.org/browse/SERVER-16073
[793] https://jira.mongodb.org/browse/SERVER-14723
[794] https://jira.mongodb.org/browse/SERVER-14071
[795] https://jira.mongodb.org/browse/SERVER-8188
[796] https://jira.mongodb.org/browse/SERVER-17429
[797] https://jira.mongodb.org/browse/SERVER-17441
[798] https://jira.mongodb.org/browse/SERVER-15907

Aggregation Framework

- SERVER-17426[799] Aggregation framework query by `_id` returns duplicates in sharded cluster (orphan documents)
- SERVER-17224[800] Aggregation pipeline with 64MB document can terminate server

Build and Platform

- SERVER-17484[801] Migrate server MCI config into server repo
- SERVER-17252[802] Upgrade PCRE Version from 8.30 to Latest

Diagnostics and Internal Code

- SERVER-17226[803] `top` (page 488) command with 64MB result document can terminate server
- SERVER-17338[804] NULL pointer crash when running `copydb` (page 433) against stepped-down 2.6 primary
- SERVER-14992[805] Query for Windows 7 File Allocation Fix, and other hotfixes

2.6.8 – Changes

Security and Networking

- SERVER-17278[806] BSON BinData validation enforcement
- SERVER-17022[807] No SSL Session Caching may not be respected
- SERVER-17264[808] improve bson validation

Query and Aggregation

- SERVER-16655[809] Geo predicate is unable to use compound 2dsphere index if it is root of `$or` (page 534) clause
- SERVER-16527[810] `2dsphere` explain reports “works” for `nscanned` & `nscannedObjects`
- SERVER-15802[811] Query optimizer should always use equality predicate over unique index when possible
- SERVER-14044[812] Incorrect `{$meta: 'text'}` reference in aggregation `$sort` (page 649) error message

[799] https://jira.mongodb.org/browse/SERVER-17426
[800] https://jira.mongodb.org/browse/SERVER-17224
[801] https://jira.mongodb.org/browse/SERVER-17484
[802] https://jira.mongodb.org/browse/SERVER-17252
[803] https://jira.mongodb.org/browse/SERVER-17226
[804] https://jira.mongodb.org/browse/SERVER-17338
[805] https://jira.mongodb.org/browse/SERVER-14992
[806] https://jira.mongodb.org/browse/SERVER-17278
[807] https://jira.mongodb.org/browse/SERVER-17022
[808] https://jira.mongodb.org/browse/SERVER-17264
[809] https://jira.mongodb.org/browse/SERVER-16655
[810] https://jira.mongodb.org/browse/SERVER-16527
[811] https://jira.mongodb.org/browse/SERVER-15802
[812] https://jira.mongodb.org/browse/SERVER-14044

Replication

- SERVER-16599[813] `copydb` (page 433) and `clone` (page 442) commands can crash the server if a primary steps down
- SERVER-16315[814] Replica set nodes should not threaten to veto nodes whose config version is higher than their own
- SERVER-16274[815] secondary `fasserts` trying to replicate an index
- SERVER-15471[816] Better error message when replica is not found in `GhostSync::associateSlave`

Sharding

- SERVER-17191[817] Spurious warning during upgrade of sharded cluster
- SERVER-17163[818] Fatal error "logOp but not primary" in `MigrateStatus::go`
- SERVER-16984[819] `UpdateLifecycleImpl` can return empty `collectionMetadata` even if `ns` is sharded
- SERVER-10904[820] Possible for _`master` and _`slaveConn` to be pointing to different connections even with primary read pref

Storage

- SERVER-17087[821] Add listCollections command functionality to 2.6 shell & client
- SERVER-14572[822] Increase C runtime stdio file limit

Tools

- SERVER-17216[823] 2.6 `mongostat` (page 858) cannot be used with 3.0 `mongod` (page 770)
- SERVER-14190[824] `mongorestore` (page 824) `parseMetadataFile` passes non-null terminated string to ‘`fromjson`‘

Build and Packaging

- SERVER-14803[825] Support static libstdc++ builds for non-Linux builds
- SERVER-15400[826] Create Windows Enterprise Zip File with vcredist and dependent dlls

Usability SERVER-14756[827] The YAML `storage.quota.enforced` option is not found

[813] https://jira.mongodb.org/browse/SERVER-16599
[814] https://jira.mongodb.org/browse/SERVER-16315
[815] https://jira.mongodb.org/browse/SERVER-16274
[816] https://jira.mongodb.org/browse/SERVER-15471
[817] https://jira.mongodb.org/browse/SERVER-17191
[818] https://jira.mongodb.org/browse/SERVER-17163
[819] https://jira.mongodb.org/browse/SERVER-16984
[820] https://jira.mongodb.org/browse/SERVER-10904
[821] https://jira.mongodb.org/browse/SERVER-17087
[822] https://jira.mongodb.org/browse/SERVER-14572
[823] https://jira.mongodb.org/browse/SERVER-17216
[824] https://jira.mongodb.org/browse/SERVER-14190
[825] https://jira.mongodb.org/browse/SERVER-14803
[826] https://jira.mongodb.org/browse/SERVER-15400
[827] https://jira.mongodb.org/browse/SERVER-14756

Testing SERVER-16421[828] `sharding_rs2.js` should clean up data on all replicas

2.6.7 – Changes

Stability

- SERVER-16237[829] Don't check the shard version if the primary server is down

Querying

- SERVER-16408[830] `max_time_ms.js` should not run in parallel suite.

Replication

- SERVER-16732[831] `SyncSourceFeedback::replHandshake()` may perform an illegal erase from a `std::map` in some circumstances

Sharding

- SERVER-16683[832] Decrease mongos memory footprint when shards have several tags
- SERVER-15766[833] prefix_shard_key.js depends on primary allocation to particular shards
- SERVER-14306[834] `mongos` (page 792) can cause shards to hit the in-memory sort limit by requesting more results than needed.

Packaging

- SERVER-16081[835] `/etc/init.d/mongod` startup script fails, with dirname message

2.6.6 – Changes

Security

- SERVER-15673[836] Disable SSLv3 ciphers
- SERVER-15515[837] New test for mixed version replSet, 2.4 primary, user updates
- SERVER-15500[838] New test for system.user operations

[828] https://jira.mongodb.org/browse/SERVER-16421
[829] https://jira.mongodb.org/browse/SERVER-16237
[830] https://jira.mongodb.org/browse/SERVER-16408
[831] https://jira.mongodb.org/browse/SERVER-16732
[832] https://jira.mongodb.org/browse/SERVER-16683
[833] https://jira.mongodb.org/browse/SERVER-15766
[834] https://jira.mongodb.org/browse/SERVER-14306
[835] https://jira.mongodb.org/browse/SERVER-16081
[836] https://jira.mongodb.org/browse/SERVER-15673
[837] https://jira.mongodb.org/browse/SERVER-15515
[838] https://jira.mongodb.org/browse/SERVER-15500

Stability

- SERVER-12061[839] Do not silently ignore read errors when syncing a replica set node
- SERVER-12058[840] Primary should abort if encountered problems writing to the oplog

Querying

- SERVER-16291[841] Cannot set/list/clear index filters on the secondary
- SERVER-15958[842] The "isMultiKey" value is not correct in the output of aggregation explain plan
- SERVER-15899[843] Querying against path in document containing long array of subdocuments with nested arrays causes stack overflow
- SERVER-15696[844] `$regex` (page 546), `$in` (page 532) and `$sort` with index returns too many results
- SERVER-15639[845] Text queries can return incorrect results and leak memory when multiple predicates given on same text index prefix field
- SERVER-15580[846] Evaluating candidate query plans with concurrent writes on same collection may crash `mongod` (page 770)
- SERVER-15528[847] Distinct queries can scan many index keys without yielding read lock
- SERVER-15485[848] CanonicalQuery::canonicalize can leak a LiteParsedQuery
- SERVER-15403[849] `$min` and `$max` equal errors in 2.6 but not in 2.4
- SERVER-15233[850] Cannot run `planCacheListQueryShapes` on a Secondary
- SERVER-14799[851] `count` (page 307) with hint doesn't work when hint is a document

Replication

- SERVER-16107[852] 2.6 `mongod` crashes with segfault when added to a 2.8 replica set with >= 12 nodes.
- SERVER-15994[853] `listIndexes` and `listCollections` can be run on secondaries without slaveOk bit
- SERVER-15849[854] do not forward replication progress for nodes that are no longer part of a replica set
- SERVER-15491[855] `SyncSourceFeedback` can crash due to a `SocketException` in `authenticateInternalUser`

[839] https://jira.mongodb.org/browse/SERVER-12061
[840] https://jira.mongodb.org/browse/SERVER-12058
[841] https://jira.mongodb.org/browse/SERVER-16291
[842] https://jira.mongodb.org/browse/SERVER-15958
[843] https://jira.mongodb.org/browse/SERVER-15899
[844] https://jira.mongodb.org/browse/SERVER-15696
[845] https://jira.mongodb.org/browse/SERVER-15639
[846] https://jira.mongodb.org/browse/SERVER-15580
[847] https://jira.mongodb.org/browse/SERVER-15528
[848] https://jira.mongodb.org/browse/SERVER-15485
[849] https://jira.mongodb.org/browse/SERVER-15403
[850] https://jira.mongodb.org/browse/SERVER-15233
[851] https://jira.mongodb.org/browse/SERVER-14799
[852] https://jira.mongodb.org/browse/SERVER-16107
[853] https://jira.mongodb.org/browse/SERVER-15994
[854] https://jira.mongodb.org/browse/SERVER-15849
[855] https://jira.mongodb.org/browse/SERVER-15491

Sharding

- SERVER-15318[856] `copydb` (page 433) should not use exhaust flag when used against `mongos` (page 792)
- SERVER-14728[857] Shard depends on string comparison of replica set connection string
- SERVER-14506[858] special top chunk logic can move max chunk to a shard with incompatible tag
- SERVER-14299[859] For sharded limit=N queries with sort, mongos can request >N results from shard
- SERVER-14080[860] Have migration result reported in the changelog correctly
- SERVER-12472[861] Fail MoveChunk if an index is needed on TO shard and data exists

Storage

- SERVER-16283[862] Can't start new wiredtiger node with log file or config file in data directory - false detection of old `mmapv1` files
- SERVER-15986[863] Starting with different storage engines in the same dbpath should error/warn
- SERVER-14057[864] Changing TTL expiration time with collMod does not correctly update index definition

Indexing and write Operations

- SERVER-14287[865] ensureIndex can abort reIndex and lose indexes
- SERVER-14886[866] Updates against paths composed with array index notation and positional operator fail with error

Data Aggregation SERVER-15552[867] Errors writing to temporary collections during `mapReduce` (page 318) command execution should be operation-fatal

Build and Packaging

- SERVER-14184[868] Unused preprocessor macros from s2 conflict on OS X Yosemite
- SERVER-14015[869] S2 Compilation on GCC 4.9/Solaris fails
- SERVER-16017[870] Suse11 enterprise packages fail due to unmet dependencies
- SERVER-15598[871] Ubuntu 14.04 Enterprise packages depend on unavailable libsnmp15 package
- SERVER-13595[872] Red Hat init.d script error: YAML config file parsing

[856] https://jira.mongodb.org/browse/SERVER-15318
[857] https://jira.mongodb.org/browse/SERVER-14728
[858] https://jira.mongodb.org/browse/SERVER-14506
[859] https://jira.mongodb.org/browse/SERVER-14299
[860] https://jira.mongodb.org/browse/SERVER-14080
[861] https://jira.mongodb.org/browse/SERVER-12472
[862] https://jira.mongodb.org/browse/SERVER-16283
[863] https://jira.mongodb.org/browse/SERVER-15986
[864] https://jira.mongodb.org/browse/SERVER-14057
[865] https://jira.mongodb.org/browse/SERVER-14287
[866] https://jira.mongodb.org/browse/SERVER-14886
[867] https://jira.mongodb.org/browse/SERVER-15552
[868] https://jira.mongodb.org/browse/SERVER-14184
[869] https://jira.mongodb.org/browse/SERVER-14015
[870] https://jira.mongodb.org/browse/SERVER-16017
[871] https://jira.mongodb.org/browse/SERVER-15598
[872] https://jira.mongodb.org/browse/SERVER-13595

Logging and Diagnostics

- SERVER-13471[873] Increase log level of "did reduceInMemory" message in map/reduce
- SERVER-16324[874] Command execution log line displays "`query not recording (too large)`" instead of abbreviated command object
- SERVER-10069[875] Improve errorcodes.py so it captures multiline messages

Testing and Internals

- SERVER-15632[876] `MultiHostQueryOp::PendingQueryContext::doBlockingQuery` can leak a cursor object
- SERVER-15629[877] `GeoParser::parseMulti{Line|Polygon}` does not clear objects owned by out parameter
- SERVER-16316[878] Remove unsupported behavior in shard3.js
- SERVER-14763[879] Update jstests/sharding/split_large_key.js
- SERVER-14249[880] Add tests for querying oplog via mongodump using –dbpath
- SERVER-13726[881] indexbg_drop.js

2.6.5 – Changes

Security

- SERVER-15465[882] OpenSSL crashes on stepdown
- SERVER-15360[883] User document changes made on a 2.4 primary and replicated to a 2.6 secondary don't make the 2.6 secondary invalidate its user cache
- SERVER-14887[884] Allow user document changes made on a 2.4 primary to replicate to a 2.6 secondary
- SERVER-14727[885] Details of SASL failures aren't logged
- SERVER-12551[886] Audit DML/CRUD operations

Stability SERVER-9032[887] mongod fails when launched with misconfigured locale

[873] https://jira.mongodb.org/browse/SERVER-13471
[874] https://jira.mongodb.org/browse/SERVER-16324
[875] https://jira.mongodb.org/browse/SERVER-10069
[876] https://jira.mongodb.org/browse/SERVER-15632
[877] https://jira.mongodb.org/browse/SERVER-15629
[878] https://jira.mongodb.org/browse/SERVER-16316
[879] https://jira.mongodb.org/browse/SERVER-14763
[880] https://jira.mongodb.org/browse/SERVER-14249
[881] https://jira.mongodb.org/browse/SERVER-13726
[882] https://jira.mongodb.org/browse/SERVER-15465
[883] https://jira.mongodb.org/browse/SERVER-15360
[884] https://jira.mongodb.org/browse/SERVER-14887
[885] https://jira.mongodb.org/browse/SERVER-14727
[886] https://jira.mongodb.org/browse/SERVER-12551
[887] https://jira.mongodb.org/browse/SERVER-9032

Querying

- SERVER-15287[888] Query planner sort analysis incorrectly allows index key pattern plugin fields to provide sort
- SERVER-15286[889] Assertion in date indexes when opposite-direction-sorted and double "or" filtered
- SERVER-15279[890] Disable hash-based index intersection (AND_HASH) by default
- SERVER-15152[891] When evaluating plans, some index candidates cause complete index scan
- SERVER-15015[892] Assertion failure when combining `$max` and `$min` and reverse index scan
- SERVER-15012[893] Server crashes on indexed rooted $or queries using a 2d index
- SERVER-14969[894] Dropping index during active aggregation operation can crash server
- SERVER-14961[895] Plan ranker favors intersection plans if predicate generates empty range index scan
- SERVER-14892[896] Invalid `{$elemMatch: {$where}}` query causes memory leak
- SERVER-14706[897] Queries that use negated $type predicate over a field may return incomplete results when an index is present on that field
- SERVER-13104[898] Plan enumerator doesn't enumerate all possibilities for a nested `$or` (page 534)
- SERVER-14984[899] Server aborts when running `$centerSphere` (page 572) query with `NaN` radius
- SERVER-14981[900] Server aborts when querying against `2dsphere` index with `coarsestIndexedLevel:0`
- SERVER-14831[901] Text search trips assertion when default language only supported in `textIndexVersion=1` used

Replication

- SERVER-15038[902] Multiple background index builds may not interrupt cleanly for commands, on secondaries
- SERVER-14887[903] Allow user document changes made on a 2.4 primary to replicate to a 2.6 secondary
- SERVER-14805[904] Use multithreaded oplog replay during initial sync

Sharding

- SERVER-15056[905] Sharded connection cleanup on setup error can crash mongos
- SERVER-13702[906] Commands without optional query may target to wrong shards on mongos

[888] https://jira.mongodb.org/browse/SERVER-15287
[889] https://jira.mongodb.org/browse/SERVER-15286
[890] https://jira.mongodb.org/browse/SERVER-15279
[891] https://jira.mongodb.org/browse/SERVER-15152
[892] https://jira.mongodb.org/browse/SERVER-15015
[893] https://jira.mongodb.org/browse/SERVER-15012
[894] https://jira.mongodb.org/browse/SERVER-14969
[895] https://jira.mongodb.org/browse/SERVER-14961
[896] https://jira.mongodb.org/browse/SERVER-14892
[897] https://jira.mongodb.org/browse/SERVER-14706
[898] https://jira.mongodb.org/browse/SERVER-13104
[899] https://jira.mongodb.org/browse/SERVER-14984
[900] https://jira.mongodb.org/browse/SERVER-14981
[901] https://jira.mongodb.org/browse/SERVER-14831
[902] https://jira.mongodb.org/browse/SERVER-15038
[903] https://jira.mongodb.org/browse/SERVER-14887
[904] https://jira.mongodb.org/browse/SERVER-14805
[905] https://jira.mongodb.org/browse/SERVER-15056
[906] https://jira.mongodb.org/browse/SERVER-13702

- SERVER-15156[907] MongoDB upgrade 2.4 to 2.6 check returns error in `config.changelog collection`

Storage

- SERVER-15369[908] explicitly zero .ns files on creation
- SERVER-15319[909] Verify 2.8 freelist is upgrade-downgrade safe with 2.6
- SERVER-15111[910] partially written journal last section causes recovery to fail

Indexing

- SERVER-14848[911] Port `index_id_desc.js` to v2.6 and master branches
- SERVER-14205[912] ensureIndex failure reports `ok: 1` on some failures

Write Operations

- SERVER-15106[913] Incorrect nscanned and nscannedObjects for idhack updates in 2.6.4 profiler or slow query log
- SERVER-15029[914] The `$rename` (page 598) modifier uses incorrect dotted source path
- SERVER-14829[915] `UpdateIndexData::clear()` should reset all member variables

Data Aggregation

- SERVER-15087[916] Server crashes when running concurrent mapReduce and dropDatabase commands
- SERVER-14969[917] Dropping index during active aggregation operation can crash server
- SERVER-14168[918] Warning logged when incremental MR collections are unsuccessfully dropped on secondaries

Packaging

- SERVER-14679[919] (CentOS 7/RHEL 7) `init.d` script should create directory for `pid` file if it is missing
- SERVER-14023[920] Support for RHEL 7 Enterprise `.rpm` packages
- SERVER-13243[921] Support for Ubuntu 14 “Trusty” Enterprise `.deb` packages
- SERVER-11077[922] Support for Debian 7 Enterprise `.deb` packages
- SERVER-10642[923] Generate Community and Enterprise packages for SUSE 11

[907] https://jira.mongodb.org/browse/SERVER-15156
[908] https://jira.mongodb.org/browse/SERVER-15369
[909] https://jira.mongodb.org/browse/SERVER-15319
[910] https://jira.mongodb.org/browse/SERVER-15111
[911] https://jira.mongodb.org/browse/SERVER-14848
[912] https://jira.mongodb.org/browse/SERVER-14205
[913] https://jira.mongodb.org/browse/SERVER-15106
[914] https://jira.mongodb.org/browse/SERVER-15029
[915] https://jira.mongodb.org/browse/SERVER-14829
[916] https://jira.mongodb.org/browse/SERVER-15087
[917] https://jira.mongodb.org/browse/SERVER-14969
[918] https://jira.mongodb.org/browse/SERVER-14168
[919] https://jira.mongodb.org/browse/SERVER-14679
[920] https://jira.mongodb.org/browse/SERVER-14023
[921] https://jira.mongodb.org/browse/SERVER-13243
[922] https://jira.mongodb.org/browse/SERVER-11077
[923] https://jira.mongodb.org/browse/SERVER-10642

Logging and Diagnostics

- SERVER-14964[924] nscanned not written to the logs at `logLevel` 1 unless `slowms` exceeded or profiling enabled
- SERVER-12551[925] Audit DML/CRUD operations
- SERVER-14904[926] Adjust dates in `tool/exportimport_date.js` to account for different timezones

Internal Code and Testing

- SERVER-13770[927] `Helpers::removeRange` should check all runner states
- SERVER-14284[928] jstests should not leave profiler enabled at test run end
- SERVER-14076[929] remove test `replset_remove_node.js`
- SERVER-14778[930] Hide function and data pointers for natively-injected v8 functions

2.6.4 – Changes

Security

- SERVER-14701[931] The "backup" auth role should allow running the "collstats" command for all resources
- SERVER-14518[932] Allow disabling hostname validation for SSL
- SERVER-14268[933] Potential information leak
- SERVER-14170[934] Cannot read from secondary if both audit and auth are enabled in a sharded cluster
- SERVER-13833[935] userAdminAnyDatabase role should be able to create indexes on admin.system.users and admin.system.roles
- SERVER-12512[936] Add role-based, selective audit logging.
- SERVER-9482[937] Add build flag for sslFIPSMode

Querying

- SERVER-14625[938] Query planner can construct incorrect bounds for negations inside $elemMatch
- SERVER-14607[939] hash intersection of fetched and non-fetched data can discard data from a result
- SERVER-14532[940] Improve logging in the case of plan ranker ties

[924] https://jira.mongodb.org/browse/SERVER-14964
[925] https://jira.mongodb.org/browse/SERVER-12551
[926] https://jira.mongodb.org/browse/SERVER-14904
[927] https://jira.mongodb.org/browse/SERVER-13770
[928] https://jira.mongodb.org/browse/SERVER-14284
[929] https://jira.mongodb.org/browse/SERVER-14076
[930] https://jira.mongodb.org/browse/SERVER-14778
[931] https://jira.mongodb.org/browse/SERVER-14701
[932] https://jira.mongodb.org/browse/SERVER-14518
[933] https://jira.mongodb.org/browse/SERVER-14268
[934] https://jira.mongodb.org/browse/SERVER-14170
[935] https://jira.mongodb.org/browse/SERVER-13833
[936] https://jira.mongodb.org/browse/SERVER-12512
[937] https://jira.mongodb.org/browse/SERVER-9482
[938] https://jira.mongodb.org/browse/SERVER-14625
[939] https://jira.mongodb.org/browse/SERVER-14607
[940] https://jira.mongodb.org/browse/SERVER-14532

- SERVER-14350[941] Server crash when $centerSphere has non-positive radius
- SERVER-14317[942] Dead code in IDHackRunner::applyProjection
- SERVER-14311[943] skipping of index keys is not accounted for in plan ranking by the index scan stage
- SERVER-14123[944] some operations can create BSON object larger than the 16MB limit
- SERVER-14034[945] Sorted $in query with large number of elements can't use merge sort
- SERVER-13994[946] do not aggressively pre-fetch data for parallelCollectionScan

Replication

- SERVER-14665[947] Build failure for v2.6 in closeall.js caused by access violation reading _me
- SERVER-14505[948] cannot dropAllIndexes when index builds in progress assertion failure
- SERVER-14494[949] Dropping collection during active background index build on secondary triggers segfault
- SERVER-13822[950] Running resync before replset config is loaded can crash `mongod` (page 770)
- SERVER-11776[951] Replication 'isself' check should allow mapped ports

Sharding

- SERVER-14551[952] Runner yield during migration cleanup (removeRange) results in fassert
- SERVER-14431[953] Invalid chunk data after splitting on a key that's too large
- SERVER-14261[954] stepdown during migration range delete can abort `mongod` (page 770)
- SERVER-14032[955] v2.6 `mongos` (page 792) doesn't verify _id is present for config server upserts
- SERVER-13648[956] better stats from migration cleanup
- SERVER-12750[957] `mongos` (page 792) shouldn't accept initial query with "exhaust" flag set
- SERVER-9788[958] `mongos` (page 792) does not re-evaluate read preference once a valid replica set member is chosen
- SERVER-9526[959] Log messages regarding chunks not very informative when the shard key is of type BinData

[941] https://jira.mongodb.org/browse/SERVER-14350
[942] https://jira.mongodb.org/browse/SERVER-14317
[943] https://jira.mongodb.org/browse/SERVER-14311
[944] https://jira.mongodb.org/browse/SERVER-14123
[945] https://jira.mongodb.org/browse/SERVER-14034
[946] https://jira.mongodb.org/browse/SERVER-13994
[947] https://jira.mongodb.org/browse/SERVER-14665
[948] https://jira.mongodb.org/browse/SERVER-14505
[949] https://jira.mongodb.org/browse/SERVER-14494
[950] https://jira.mongodb.org/browse/SERVER-13822
[951] https://jira.mongodb.org/browse/SERVER-11776
[952] https://jira.mongodb.org/browse/SERVER-14551
[953] https://jira.mongodb.org/browse/SERVER-14431
[954] https://jira.mongodb.org/browse/SERVER-14261
[955] https://jira.mongodb.org/browse/SERVER-14032
[956] https://jira.mongodb.org/browse/SERVER-13648
[957] https://jira.mongodb.org/browse/SERVER-12750
[958] https://jira.mongodb.org/browse/SERVER-9788
[959] https://jira.mongodb.org/browse/SERVER-9526

Storage

- SERVER-14198[960] Std::set<pointer> and Windows Heap Allocation Reuse produces non-deterministic results
- SERVER-13975[961] Creating index on collection named "system" can cause server to abort
- SERVER-13729[962] Reads & Writes are blocked during data file allocation on Windows
- SERVER-13681[963] `mongod` (page 770) B stalls during background flush on Windows

Indexing SERVER-14494[964] Dropping collection during active background index build on secondary triggers segfault

Write Ops

- SERVER-14257[965] "remove" command can cause process termination by throwing unhandled exception if profiling is enabled
- SERVER-14024[966] Update fails when query contains part of a DBRef and results in an insert (upsert:true)
- SERVER-13764[967] debug mechanisms report incorrect nscanned / nscannedObjects for updates

Networking SERVER-13734[968] Remove catch (...) from handleIncomingMsg

Geo

- SERVER-14039[969] $nearSphere query with 2d index, skip, and limit returns incomplete results
- SERVER-13701[970] Query using 2d index throws exception when using explain()

Text Search

- SERVER-14738[971] Updates to documents with text-indexed fields may lead to incorrect entries
- SERVER-14027[972] Renaming collection within same database fails if wildcard text index present

Tools

- SERVER-14212[973] `mongorestore` (page 824) may drop system users and roles
- SERVER-14048[974] `mongodump` (page 816) against `mongos` (page 792) can't send dump to standard output

[960] https://jira.mongodb.org/browse/SERVER-14198
[961] https://jira.mongodb.org/browse/SERVER-13975
[962] https://jira.mongodb.org/browse/SERVER-13729
[963] https://jira.mongodb.org/browse/SERVER-13681
[964] https://jira.mongodb.org/browse/SERVER-14494
[965] https://jira.mongodb.org/browse/SERVER-14257
[966] https://jira.mongodb.org/browse/SERVER-14024
[967] https://jira.mongodb.org/browse/SERVER-13764
[968] https://jira.mongodb.org/browse/SERVER-13734
[969] https://jira.mongodb.org/browse/SERVER-14039
[970] https://jira.mongodb.org/browse/SERVER-13701
[971] https://jira.mongodb.org/browse/SERVER-14738
[972] https://jira.mongodb.org/browse/SERVER-14027
[973] https://jira.mongodb.org/browse/SERVER-14212
[974] https://jira.mongodb.org/browse/SERVER-14048

Admin

- SERVER-14556[975] Default dbpath for `mongod` (page 770) `--configsvr` (page 782) changes in 2.6
- SERVER-14355[976] Allow dbAdmin role to manually create system.profile collections

Packaging SERVER-14283[977] Parameters in installed config file are out of date

JavaScript

- SERVER-14254[978] Do not store native function pointer as a property in function prototype
- SERVER-13798[979] v8 garbage collection can cause crash due to independent lifetime of DBClient and Cursor objects
- SERVER-13707[980] mongo shell may crash when converting invalid regular expression

Shell

- SERVER-14341[981] negative opcounter values in serverStatus
- SERVER-14107[982] Querying for a document containing a value of either type Javascript or JavascriptWithScope crashes the shell

Usability SERVER-13833[983] userAdminAnyDatabase role should be able to create indexes on admin.system.users and admin.system.roles

Logging and Diagnostics

- SERVER-12512[984] Add role-based, selective audit logging.
- SERVER-14341[985] negative opcounter values in serverStatus

Testing

- SERVER-14731[986] plan_cache_ties.js sometimes fails
- SERVER-14147[987] make index_multi.js retry on connection failure
- SERVER-13615[988] sharding_rs2.js intermittent failure due to reliance on opcounters

[975] https://jira.mongodb.org/browse/SERVER-14556
[976] https://jira.mongodb.org/browse/SERVER-14355
[977] https://jira.mongodb.org/browse/SERVER-14283
[978] https://jira.mongodb.org/browse/SERVER-14254
[979] https://jira.mongodb.org/browse/SERVER-13798
[980] https://jira.mongodb.org/browse/SERVER-13707
[981] https://jira.mongodb.org/browse/SERVER-14341
[982] https://jira.mongodb.org/browse/SERVER-14107
[983] https://jira.mongodb.org/browse/SERVER-13833
[984] https://jira.mongodb.org/browse/SERVER-12512
[985] https://jira.mongodb.org/browse/SERVER-14341
[986] https://jira.mongodb.org/browse/SERVER-14731
[987] https://jira.mongodb.org/browse/SERVER-14147
[988] https://jira.mongodb.org/browse/SERVER-13615

2.6.3 – Changes

- SERVER-14302[989] Fixed: "Equality queries on `_id` with projection may return no results on sharded collections"
- SERVER-14304[990] Fixed: "Equality queries on `_id` with projection on `_id` may return orphan documents on sharded collections"

2.6.2 – Changes

Security

- SERVER-13727[991] The `backup` authorization role now includes privileges to run the `collStats` (page 473) command.
- SERVER-13804[992] The built-in role `restore` now has privileges on `system.roles` collection.
- SERVER-13612[993] Fixed: "SSL-enabled server appears not to be sending the list of supported certificate issuers to the client"
- SERVER-13753[994] Fixed: "`mongod` (page 770) may terminate if x.509 authentication certificate is invalid"
- SERVER-13945[995] For *replica set/sharded cluster member authentication*, now matches x.509 cluster certificates by attributes instead of by substring comparison.
- SERVER-13868[996] Now marks V1 users as probed on databases that do not have surrogate user documents.
- SERVER-13850[997] 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 792).
- SERVER-13588[998] Fixed: "Shell prints startup warning when auth enabled"

Querying

- SERVER-13731[999] Fixed: "Stack overflow when parsing deeply nested `$not` (page 536) query"
- SERVER-13890[1000] Fixed: "Index bounds builder constructs invalid bounds for multiple negations joined by an `$or` (page 534)"
- SERVER-13752[1001] Verified assertion on empty `$in` (page 532) clause and sort on second field in a compound index.
- SERVER-13337[1002] Re-enabled `idhack` for queries with projection.
- SERVER-13715[1003] Fixed: "Aggregation pipeline execution can fail with $or and blocking sorts"
- SERVER-13714[1004] Fixed: "non-top-level indexable `$not` (page 536) triggers query planning bug"

[989] https://jira.mongodb.org/browse/SERVER-14302
[990] https://jira.mongodb.org/browse/SERVER-14304
[991] https://jira.mongodb.org/browse/SERVER-13727
[992] https://jira.mongodb.org/browse/SERVER-13804
[993] https://jira.mongodb.org/browse/SERVER-13612
[994] https://jira.mongodb.org/browse/SERVER-13753
[995] https://jira.mongodb.org/browse/SERVER-13945
[996] https://jira.mongodb.org/browse/SERVER-13868
[997] https://jira.mongodb.org/browse/SERVER-13850
[998] https://jira.mongodb.org/browse/SERVER-13588
[999] https://jira.mongodb.org/browse/SERVER-13731
[1000] https://jira.mongodb.org/browse/SERVER-13890
[1001] https://jira.mongodb.org/browse/SERVER-13752
[1002] https://jira.mongodb.org/browse/SERVER-13337
[1003] https://jira.mongodb.org/browse/SERVER-13715
[1004] https://jira.mongodb.org/browse/SERVER-13714

- SERVER-13769[1005] Fixed: "`distinct` (page 310) command on indexed field with geo predicate fails to execute"
- SERVER-13675[1006] Fixed "Plans with differing performance can tie during plan ranking"
- SERVER-13899[1007] Fixed: "'Whole index scan' query solutions can use incompatible indexes, return incorrect results"
- SERVER-13852[1008] Fixed "IndexBounds::endKeyInclusive not initialized by constructor"
- SERVER-14073[1009] planSummary no longer truncated at 255 characters
- SERVER-14174[1010] Fixed: "If ntoreturn is a limit (rather than batch size) extra data gets buffered during plan ranking"
- SERVER-13789[1011] Some nested queries no longer trigger an assertion error
- SERVER-14064[1012] Added planSummary information for `count` (page 307) command log message.
- SERVER-13960[1013] Queries containing `$or` (page 534) no longer miss results if multiple clauses use the same index.
- SERVER-14180[1014] Fixed: "Crash with 'and' clause, `$elemMatch` (page 579), and nested `$mod` (page 544) or regex"
- SERVER-14176[1015] Natural order sort specification no longer ignored if query is specified.
- SERVER-13754[1016] Bounds no longer combined for `$or` (page 534) queries that can use merge sort.

Geospatial SERVER-13687[1017] Results of `$near` (page 565) query on compound multi-key 2dsphere index are now sorted by distance.

Write Operations SERVER-13802[1018] Insert field validation no longer stops at first `Timestamp()` field.

Replication

- SERVER-13993[1019] Fixed: "log a message when `shouldChangeSyncTarget()` believes a node should change sync targets"
- SERVER-13976[1020] Fixed: "Cloner needs to detect failure to create collection"

[1005] https://jira.mongodb.org/browse/SERVER-13769
[1006] https://jira.mongodb.org/browse/SERVER-13675
[1007] https://jira.mongodb.org/browse/SERVER-13899
[1008] https://jira.mongodb.org/browse/SERVER-13852
[1009] https://jira.mongodb.org/browse/SERVER-14073
[1010] https://jira.mongodb.org/browse/SERVER-14174
[1011] https://jira.mongodb.org/browse/SERVER-13789
[1012] https://jira.mongodb.org/browse/SERVER-14064
[1013] https://jira.mongodb.org/browse/SERVER-13960
[1014] https://jira.mongodb.org/browse/SERVER-14180
[1015] https://jira.mongodb.org/browse/SERVER-14176
[1016] https://jira.mongodb.org/browse/SERVER-13754
[1017] https://jira.mongodb.org/browse/SERVER-13687
[1018] https://jira.mongodb.org/browse/SERVER-13802
[1019] https://jira.mongodb.org/browse/SERVER-13993
[1020] https://jira.mongodb.org/browse/SERVER-13976

Sharding

- SERVER-13616[1021] Resolved: "'type 7' (OID) error when acquiring distributed lock for first time"
- SERVER-13812[1022] Now catches exception thrown by `getShardsForQuery` for geo query.
- SERVER-14138[1023] `mongos` (page 792) will now correctly target multiple shards for nested field shard key predicates.
- SERVER-11332[1024] Fixed: "Authentication requests delayed if first config server is unresponsive"

Map/Reduce

- SERVER-14186[1025] Resolved: "`rs.stepDown` (page 263) during mapReduce causes fassert in logOp"
- SERVER-13981[1026] Temporary map/reduce collections are now correctly replicated to secondaries.

Storage

- SERVER-13750[1027] `convertToCapped` (page 444) on empty collection no longer aborts after `invariant()` failure.
- SERVER-14056[1028] Moving large collection across databases with renameCollection no longer triggers fatal assertion.
- SERVER-14082[1029] Fixed: "Excessive freelist scanning for MaxBucket"
- SERVER-13737[1030] CollectionOptions parser now skips non-numeric for "size"/"max" elements if values non-numeric.

Build and Packaging

- SERVER-13950[1031] MongoDB Enterprise now includes required dependency list.
- SERVER-13862[1032] Support for mongodb-org-server installation 2.6.1-1 on RHEL5 via RPM.
- SERVER-13724[1033] Added SCons flag to override treating all warnings as errors.

Diagnostics

- SERVER-13587[1034] Resolved: "`ndeleted` in `system.profile` documents reports 1 too few documents removed"
- SERVER-13368[1035] Improved exposure of timing information in `currentOp`.

[1021] https://jira.mongodb.org/browse/SERVER-13616
[1022] https://jira.mongodb.org/browse/SERVER-13812
[1023] https://jira.mongodb.org/browse/SERVER-14138
[1024] https://jira.mongodb.org/browse/SERVER-11332
[1025] https://jira.mongodb.org/browse/SERVER-14186
[1026] https://jira.mongodb.org/browse/SERVER-13981
[1027] https://jira.mongodb.org/browse/SERVER-13750
[1028] https://jira.mongodb.org/browse/SERVER-14056
[1029] https://jira.mongodb.org/browse/SERVER-14082
[1030] https://jira.mongodb.org/browse/SERVER-13737
[1031] https://jira.mongodb.org/browse/SERVER-13950
[1032] https://jira.mongodb.org/browse/SERVER-13862
[1033] https://jira.mongodb.org/browse/SERVER-13724
[1034] https://jira.mongodb.org/browse/SERVER-13587
[1035] https://jira.mongodb.org/browse/SERVER-13368

Administration SERVER-13954[1036] `security.javascriptEnabled` (page 911) option is now available in the YAML configuration file.

Tools

- SERVER-10464[1037] `mongodump` (page 816) can now query `oplog.$main` and `oplog.rs` when using *`--dbpath`*.
- SERVER-13760[1038] `mongoexport` (page 850) can now handle large timestamps on Windows.

Shell

- SERVER-13865[1039] Shell now returns correct `WriteResult` for compatibility-mode upsert with non-OID equality predicate on `_id` field.
- SERVER-13037[1040] Fixed typo in error message for "compatibility mode".

Internal Code

- SERVER-13794[1041] Fixed: "Unused snapshot history consuming significant heap space"
- SERVER-13446[1042] Removed Solaris builds dependency on ILLUMOS libc.
- SERVER-14092[1043] MongoDB upgrade 2.4 to 2.6 check no longer returns an error in internal collections.
- SERVER-14000[1044] Added new lsb file location for Debian 7.1

Testing

- SERVER-13723[1045] Stabilized `tags.js` after a change in its timeout when it was ported to use write commands.
- SERVER-13494[1046] Fixed: "`setup_multiversion_mongodb.py` doesn't download 2.4.10 because of non-numeric version sorting"
- SERVER-13603[1047] Fixed: "Test suites with options tests fail when run with *`--nopreallocj`*"
- SERVER-13948[1048] Fixed: "`awaitReplication()` failures related to getting a config version from master causing test failures"
- SERVER-13839[1049] Fixed `sync2.js` failure.
- SERVER-13972[1050] Fixed `connections_opened.js` failure.
- SERVER-13712[1051] Reduced peak disk usage of test suites.

[1036] https://jira.mongodb.org/browse/SERVER-13954
[1037] https://jira.mongodb.org/browse/SERVER-10464
[1038] https://jira.mongodb.org/browse/SERVER-13760
[1039] https://jira.mongodb.org/browse/SERVER-13865
[1040] https://jira.mongodb.org/browse/SERVER-13037
[1041] https://jira.mongodb.org/browse/SERVER-13794
[1042] https://jira.mongodb.org/browse/SERVER-13446
[1043] https://jira.mongodb.org/browse/SERVER-14092
[1044] https://jira.mongodb.org/browse/SERVER-14000
[1045] https://jira.mongodb.org/browse/SERVER-13723
[1046] https://jira.mongodb.org/browse/SERVER-13494
[1047] https://jira.mongodb.org/browse/SERVER-13603
[1048] https://jira.mongodb.org/browse/SERVER-13948
[1049] https://jira.mongodb.org/browse/SERVER-13839
[1050] https://jira.mongodb.org/browse/SERVER-13972
[1051] https://jira.mongodb.org/browse/SERVER-13712

- SERVER-14249[1052] Added tests for querying oplog via `mongodump` (page 816) using *--dbpath*
- SERVER-10462[1053] Fixed: "Windows file locking related buildbot failures"

2.6.1 – Changes

Stability SERVER-13739[1054] Repair database failure can delete database files

Build and Packaging

- SERVER-13287[1055] Addition of debug symbols has doubled compile time
- SERVER-13563[1056] Upgrading from 2.4.x to 2.6.0 via `yum` clobbers configuration file
- SERVER-13691[1057] yum and apt "stable" repositories contain release candidate 2.6.1-rc0 packages
- SERVER-13515[1058] Cannot install MongoDB as a service on Windows

Querying

- SERVER-13066[1059] Negations over multikey fields do not use index
- SERVER-13495[1060] Concurrent `GETMORE` and `KILLCURSORS` operations can cause race condition and server crash
- SERVER-13503[1061] The `$where` (page 558) operator should not be allowed under `$elemMatch` (page 579)
- SERVER-13537[1062] Large skip and and limit values can cause crash in blocking sort stage
- SERVER-13557[1063] Incorrect negation of $elemMatch value in 2.6
- SERVER-13562[1064] Queries that use tailable cursors do not stream results if skip() is applied
- SERVER-13566[1065] Using the OplogReplay flag with extra predicates can yield incorrect results
- SERVER-13611[1066] Missing sort order for compound index leads to unnecessary in-memory sort
- SERVER-13618[1067] Optimization for sorted $in queries not applied to reverse sort order
- SERVER-13661[1068] Increase the maximum allowed depth of query objects
- SERVER-13664[1069] Query with `$elemMatch` (page 579) using a compound multikey index can generate incorrect results

[1052] https://jira.mongodb.org/browse/SERVER-14249
[1053] https://jira.mongodb.org/browse/SERVER-10462
[1054] https://jira.mongodb.org/browse/SERVER-13739
[1055] https://jira.mongodb.org/browse/SERVER-13287
[1056] https://jira.mongodb.org/browse/SERVER-13563
[1057] https://jira.mongodb.org/browse/SERVER-13691
[1058] https://jira.mongodb.org/browse/SERVER-13515
[1059] https://jira.mongodb.org/browse/SERVER-13066
[1060] https://jira.mongodb.org/browse/SERVER-13495
[1061] https://jira.mongodb.org/browse/SERVER-13503
[1062] https://jira.mongodb.org/browse/SERVER-13537
[1063] https://jira.mongodb.org/browse/SERVER-13557
[1064] https://jira.mongodb.org/browse/SERVER-13562
[1065] https://jira.mongodb.org/browse/SERVER-13566
[1066] https://jira.mongodb.org/browse/SERVER-13611
[1067] https://jira.mongodb.org/browse/SERVER-13618
[1068] https://jira.mongodb.org/browse/SERVER-13661
[1069] https://jira.mongodb.org/browse/SERVER-13664

- SERVER-13677[1070] Query planner should traverse through $all while handling $elemMatch object predicates
- SERVER-13766[1071] Dropping index or collection while $or query is yielding triggers fatal assertion

Geospatial

- SERVER-13666[1072] `$near` (page 565) queries with out-of-bounds points in legacy format can lead to crashes
- SERVER-13540[1073] The `geoNear` (page 327) command no longer returns distance in radians for legacy point
- SERVER-13486[1074]: The `geoNear` (page 327) command can create too large BSON objects for aggregation.

Replication

- SERVER-13500[1075] Changing replica set configuration can crash running members
- SERVER-13589[1076] Background index builds from a 2.6.0 primary fail to complete on 2.4.x secondaries
- SERVER-13620[1077] Replicated data definition commands will fail on secondaries during background index build
- SERVER-13496[1078] Creating index with same name but different spec in mixed version replicaset can abort replication

Sharding

- SERVER-12638[1079] Initial sharding with hashed shard key can result in duplicate split points
- SERVER-13518[1080] The `_id` field is no longer automatically generated by `mongos` (page 792) when missing
- SERVER-13777[1081] Migrated ranges waiting for deletion do not report cursors still open

Security

- SERVER-9358[1082] Log rotation can overwrite previous log files
- SERVER-13644[1083] Sensitive credentials in startup options are not redacted and may be exposed
- SERVER-13441[1084] Inconsistent error handling in user management shell helpers

Write Operations

- SERVER-13466[1085] Error message in collection creation failure contains incorrect namespace
- SERVER-13499[1086] Yield policy for batch-inserts should be the same as for batch-updates/deletes

[1070] https://jira.mongodb.org/browse/SERVER-13677
[1071] https://jira.mongodb.org/browse/SERVER-13766
[1072] https://jira.mongodb.org/browse/SERVER-13666
[1073] https://jira.mongodb.org/browse/SERVER-13540
[1074] https://jira.mongodb.org/browse/SERVER-13486
[1075] https://jira.mongodb.org/browse/SERVER-13500
[1076] https://jira.mongodb.org/browse/SERVER-13589
[1077] https://jira.mongodb.org/browse/SERVER-13620
[1078] https://jira.mongodb.org/browse/SERVER-13496
[1079] https://jira.mongodb.org/browse/SERVER-12638
[1080] https://jira.mongodb.org/browse/SERVER-13518
[1081] https://jira.mongodb.org/browse/SERVER-13777
[1082] https://jira.mongodb.org/browse/SERVER-9358
[1083] https://jira.mongodb.org/browse/SERVER-13644
[1084] https://jira.mongodb.org/browse/SERVER-13441
[1085] https://jira.mongodb.org/browse/SERVER-13466
[1086] https://jira.mongodb.org/browse/SERVER-13499

- SERVER-13516[1087] Array updates on documents with more than 128 BSON elements may crash `mongod` (page 770)

2.6.11 – Aug 12, 2015

- Improvements to query plan ranking SERVER-17815[1088]
- Improved ability for `mongos` (page 792) to detect replica set failover and correctly route read operations to the new primary SERVER-18280[1089]
- Improved reporting of queries in `getMore` operation in `db.currentOp()` (page 171) and the database profiler SERVER-16265[1090]
- All issues closed in 2.6.11[1091]

2.6.10 – May 19, 2015

- Improve user cache invalidation enforcement on `mongos` (page 792) SERVER-11980[1092]
- Provide correct rollbacks for collection creation SERVER-18211[1093]
- Allow user inserts into the `system.profile` collection SERVER-18211[1094]
- Fix to query system to ensure non-negation predicates get chosen over negation predicates for multikey index bounds construction SERVER-18364[1095]
- All issues closed in 2.6.10[1096]

2.6.9 – March 24, 2015

- Resolve connection handling related crash with `mongos` (page 792) instances SERVER-17441[1097]
- Add server parameter to configure idle cursor timeout SERVER-8188[1098]
- Remove duplicated (orphan) documents from aggregation pipelines with `_id` queries in sharded clusters SERVER-17426[1099]
- Fixed crash in `geoNear` (page 327) queries with multiple `2dsphere` indexes SERVER-14723[1100]
- All issues closed in 2.6.9[1101]

[1087] https://jira.mongodb.org/browse/SERVER-13516
[1088] https://jira.mongodb.org/browse/SERVER-17815
[1089] https://jira.mongodb.org/browse/SERVER-18280
[1090] https://jira.mongodb.org/browse/SERVER-16265
[1091] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.11%22%20AND%20project%20%3D%20SERVER
[1092] https://jira.mongodb.org/browse/SERVER-11980
[1093] https://jira.mongodb.org/browse/SERVER-18211
[1094] https://jira.mongodb.org/browse/SERVER-18211
[1095] https://jira.mongodb.org/browse/SERVER-18364
[1096] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.10%22%20AND%20project%20%3D%20SERVER
[1097] https://jira.mongodb.org/browse/SERVER-17441
[1098] https://jira.mongodb.org/browse/SERVER-8188
[1099] https://jira.mongodb.org/browse/SERVER-17426
[1100] https://jira.mongodb.org/browse/SERVER-14723
[1101] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.9%22%20AND%20project%20%3D%20SERVER

2.6.8 – February 25, 2015

- Add `listCollections` (page 438) command functionality to 2.6 shell and client SERVER-17087[1102]
- `copydb` (page 433)/`clone` (page 442) commands can crash the server if a primary steps down SERVER-16599[1103]
- Secondary fasserts trying to replicate an index SERVER-16274[1104]
- Query optimizer should always use equality predicate over unique index when possible SERVER-15802[1105]
- All issues closed in 2.6.8[1106]

2.6.7 – January 13, 2015

- Decreased `mongos` (page 792) memory footprint when shards have several tags SERVER-16683[1107]
- Removed check for shard version if the primary server is down SERVER-16237[1108]
- Fixed: `/etc/init.d/mongod` startup script failure with dirname message SERVER-16081[1109]
- Fixed: `mongos` (page 792) can cause shards to hit the in-memory sort limit by requesting more results than needed SERVER-14306[1110]
- All issues closed in 2.6.7[1111]

2.6.6 – December 09, 2014

- Fixed: Evaluating candidate query plans with concurrent writes on same collection may crash `mongod` (page 770) SERVER-15580[1112]
- Fixed: 2.6 `mongod` (page 770) crashes with segfault when added to a 2.8 replica set with 12 or more members SERVER-16107[1113]
- Fixed: `$regex` (page 546), `$in` (page 532) and `$sort` with index returns too many results SERVER-15696[1114]
- Change: `moveChunk` (page 427) will fail if there is data on the target shard and a required index does not exist. SERVER-12472[1115]
- Primary should abort if encountered problems writing to the oplog SERVER-12058[1116]
- All issues closed in 2.6.6[1117]

[1102] https://jira.mongodb.org/browse/SERVER-17087
[1103] https://jira.mongodb.org/browse/SERVER-16599
[1104] https://jira.mongodb.org/browse/SERVER-16274
[1105] https://jira.mongodb.org/browse/SERVER-15802
[1106] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.8%22%20AND%20project%20%3D%20SERVER
[1107] https://jira.mongodb.org/browse/SERVER-16683
[1108] https://jira.mongodb.org/browse/SERVER-16237
[1109] https://jira.mongodb.org/browse/SERVER-16081
[1110] https://jira.mongodb.org/browse/SERVER-14306
[1111] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.7%22%20AND%20project%20%3D%20SERVER
[1112] https://jira.mongodb.org/browse/SERVER-15580
[1113] https://jira.mongodb.org/browse/SERVER-16107
[1114] https://jira.mongodb.org/browse/SERVER-15696
[1115] https://jira.mongodb.org/browse/SERVER-12472
[1116] https://jira.mongodb.org/browse/SERVER-12058
[1117] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.6%22%20AND%20project%20%3D%20SERVER

2.6.5 – October 07, 2014

- `$rename` now uses correct dotted source paths SERVER-15029[1118]
- Partially written journal last section does not affect recovery SERVER-15111[1119]
- Explicitly zero `.ns` files on creation SERVER-15369[1120]
- Plan ranker will no longer favor intersection plans if predicate generates empty range index scan SERVER-14961[1121]
- Generate Community and Enterprise packages for SUSE 11 SERVER-10642[1122]
- All issues closed in 2.6.5[1123]

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[1124]
- Check the size of the split point before performing a manual split chunk operation SERVER-14431[1125]
- 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[1126]
- Allow read from secondaries when both audit and authorization are enabled in a sharded cluster SERVER-14170[1127]
- All issues closed in 2.6.4[1128]

2.6.3 – June 19, 2014

- Equality queries on `_id` with projection may return no results on sharded collections SERVER-14302[1129].
- Equality queries on `_id` with projection on `_id` may return orphan documents on sharded collections SERVER-14304[1130].
- All issues closed in 2.6.3[1131].

2.6.2 – June 16, 2014

- Query plans with differing performance can tie during plan ranking SERVER-13675[1132].

[1118] https://jira.mongodb.org/browse/SERVER-15029
[1119] https://jira.mongodb.org/browse/SERVER-15111
[1120] https://jira.mongodb.org/browse/SERVER-15369
[1121] https://jira.mongodb.org/browse/SERVER-14961
[1122] https://jira.mongodb.org/browse/SERVER-10642
[1123] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.5%22%20AND%20project%20%3D%20SERVER
[1124] https://jira.mongodb.org/browse/SERVER-14738
[1125] https://jira.mongodb.org/browse/SERVER-14431
[1126] https://jira.mongodb.org/browse/SERVER-9788
[1127] https://jira.mongodb.org/browse/SERVER-14170
[1128] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.4%22%20AND%20project%20%3D%20SERVER
[1129] https://jira.mongodb.org/browse/SERVER-14302
[1130] https://jira.mongodb.org/browse/SERVER-14304
[1131] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.3%22%20AND%20project%20%3D%20SERVER
[1132] https://jira.mongodb.org/browse/SERVER-13675

- `mongod` (page 770) may terminate if x.509 authentication certificate is invalid SERVER-13753[1133].
- Temporary map/reduce collections are incorrectly replicated to secondaries SERVER-13981[1134].
- `mongos` (page 792) incorrectly targets multiple shards for nested field shard key predicates SERVER-14138[1135].
- `rs.stepDown()` (page 263) during mapReduce causes `fassert` when writing to op log SERVER-14186[1136].
- All issues closed in 2.6.2[1137].

2.6.1 – May 5, 2014

- Fix to install MongoDB service on Windows with the `--install` option SERVER-13515[1138].
- Allow direct upgrade from 2.4.x to 2.6.0 via `yum` SERVER-13563[1139].
- Fix issues with background index builds on secondaries: SERVER-13589[1140] and SERVER-13620[1141].
- Redact credential information passed as startup options SERVER-13644[1142].
- *2.6.1 Changelog* (page 1089).
- All issues closed in 2.6.1[1143].

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 20) 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:
  - `$out` (page 656) stage to output to a collection.
  - `$redact` (page 637) stage to allow additional control to accessing the data.
- New or modified operators:

[1133] https://jira.mongodb.org/browse/SERVER-13753
[1134] https://jira.mongodb.org/browse/SERVER-13981
[1135] https://jira.mongodb.org/browse/SERVER-14138
[1136] https://jira.mongodb.org/browse/SERVER-14186
[1137] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.2%22%20AND%20project%20%3D%20SERVER
[1138] https://jira.mongodb.org/browse/SERVER-13515
[1139] https://jira.mongodb.org/browse/SERVER-13563
[1140] https://jira.mongodb.org/browse/SERVER-13589
[1141] https://jira.mongodb.org/browse/SERVER-13620
[1142] https://jira.mongodb.org/browse/SERVER-13644
[1143] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.1%22%20AND%20project%20%3D%20SERVER

  - *set expression operators* (page 663).
  - `$let` (page 710) and `$map` (page 709) operators to allow for the use of variables.
  - `$literal` (page 712) operator and `$size` (page 707) operator.
  - `$cond` (page 726) 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 `$match` (page 635) stage, includes the `$text` (page 549) operator, which resolves text-search queries.

MongoDB 2.6 includes an updated `text index` format and deprecates the `text` command.

Insert and Update Improvements

Improvements to the update and insert systems include additional operations and improvements that increase consistency of modified data.

- MongoDB preserves the order of the document fields following write operations *except* for the following cases:
  - The `_id` field is always the first field in the document.
  - Updates that include `renaming` (page 598) of field names may result in the reordering of fields in the document.
- New or enhanced update operators:
  - `$bit` (page 627) operator supports bitwise `xor` operation.
  - `$min` (page 602) and `$max` (page 604) operators that perform conditional update depending on the relative size of the specified value and the current value of a field.
  - `$push` (page 616) operator has enhanced support for the `$sort` (page 622), `$slice` (page 619), and `$each` (page 619) modifiers and supports a new `$position` (page 625) modifier.
  - `$currentDate` (page 605) operator to set the value of a field to the current date.
- The `$mul` (page 596) operator for multiplicative increments for insert and update operations.

See also:

Update Operator Syntax Validation (page 1105)

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 337), `update` (page 340), and `delete` (page 345), which provide the basis for the improved bulk insert. All officially supported MongoDB drivers support the new write commands.

The `mongo` (page 803) shell now includes methods to perform bulk-write operations. See `Bulk()` (page 210) for more information.

See also:

Write Method Acknowledgements (page 1102)

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 770) and `mongos` (page 792) instances using SSL connections.
  - *Prompt for passphrase* by `mongod` (page 770) or `mongos` (page 792) 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` (page 905).

See also:

New Authorization Model (page 1103), *SSL Certificate Hostname Validation* (page 1104), and `https://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 204) methods to view and clear the `query plans` cached by the query optimizer.
- MongoDB can now use `count()` (page 137) with `hint()` (page 142). See `count()` (page 137) 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 534) expressions.

See also:

2dsphere Index Version 2 (page 1104), *$maxDistance Changes* (page 1107), *Deprecated $uniqueDocs* (page 1107), *Stronger Validation of Geospatial Queries* (page 1107)

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 770) 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 770) instance will now restart the interrupted index build when the instance restarts.

  To disable this behavior, use the `--noIndexBuildRetry` command-line option.
- `ensureIndex()` (page 47) now wraps a new `createIndex` command.
- The `dropDups` option to `ensureIndex()` (page 47) and `createIndex` is deprecated.

See also:

Enforce Index Key Length Limit (page 1100)

Enhanced Sharding and Replication Administration

- New `cleanupOrphaned` (page 415) command to remove *orphaned documents* from a shard.
- New `mergeChunks` (page 421) command to combine contiguous chunks located on a single shard. See `mergeChunks` (page 421) and `https://docs.mongodb.org/manual/tutorial/merge-chunks-in-sharded-cluster`.
- New `rs.printReplicationInfo()` (page 259) and `rs.printSlaveReplicationInfo()` (page 260) 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* (page 895) for more information.

Operational Changes

Storage

usePowerOf2Sizes (page 458) 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 457) with usePowerOf2Sizes (page 458) set to false.
- For all new collections on an entire mongod (page 770) instance, set newCollectionsUsePowerOf2Sizes (page 932) to false.

  New collections include those: created during *initial sync*, as well as those created by the mongorestore (page 824) and mongoimport (page 841) tools, by running mongod (page 770) with the *--repair* option, as well as the restoreDatabase command.

See /core/storage[1144] for more information about MongoDB's storage system.

Networking

- Removed upward limit for the maxIncomingConnections (page 903) for mongod (page 770) and mongos (page 792). Previous versions capped the maximum possible maxIncomingConnections (page 903) setting at 20,000 connections.
- Connection pools for a mongos (page 792) 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 409) command instead of replSetGetStatus (page 400). This allows the C++ driver to support systems that require authentication.
- New cursor.maxTimeMS() (page 146) and corresponding maxTimeMS option for commands to specify a time limit.

Tool Improvements

- mongo (page 803) shell supports a global */etc/mongorc.js* (page 808).
- All MongoDB *executable files* (page 769) now support the --quiet option to suppress all logging output except for error messages.
- mongoimport (page 841) uses the input filename, without the file extension if any, as the collection name if run without the -c or --collection specification.
- mongoexport (page 850) can now constrain export data using *--skip* (page 856) and *--limit* (page 856), as well as order the documents in an export using the *--sort* (page 856) option.
- mongostat (page 858) can support the use of *--rowcount* (page 871) (*-n* (page **??**)) with the *--discover* (page 862) option to produce the specified number of output lines.
- Add strict mode representation for data_numberlong (page 963) for use by mongoexport (page 850) and mongoimport (page 841).

[1144] https://docs.mongodb.org/v2.6/core/storage

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 770) and `mongos` (page 792) 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 `https://docs.mongodb.org/manual/tutorial/configure-ldap-sasl-openldap` and `https://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 1110) 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 197).

See also:

SNMP Changes (page 1105)

Additional Information

Changes Affecting Compatibility

Compatibility Changes in MongoDB 2.6

On this page

- Index Changes (page 1100)
- Write Method Acknowledgements (page 1102)
- `db.collection.aggregate()` Change (page 1103)
- Write Concern Validation (page 1103)
- Security Changes (page 1103)
- `2dsphere` Index Version 2 (page 1104)
- Log Messages (page 1104)
- Package Configuration Changes (page 1105)
- Remove Method Signature Change (page 1105)
- Update Operator Syntax Validation (page 1105)
- Updates Enforce Field Name Restrictions (page 1106)
- Query and Sort Changes (page 1106)
- Replica Set/Sharded Cluster Validation (page 1109)
- Time Format Changes (page 1110)
- Other Resources (page 1110)

The following 2.6 changes can affect the compatibility with older versions of MongoDB. See *Release Notes for MongoDB 2.6* (page 1068) 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 47), `db.collection.reIndex()` (page 97), `compact` (page 454), and `repairDatabase` (page 462) 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 97), `compact` (page 454), and `repairDatabase` (page 462) 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 462) command, from continuing with the remainder of the process.

Inserts will error:

- `db.collection.insert()` (page 79) and other operations that perform inserts (e.g. `db.collection.save()` (page 105) and `db.collection.update()` (page 117) 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 824) and `mongoimport` (page 841) 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 117) and `db.collection.save()` (page 105) 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 202) 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 803) 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` (page 932) 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 47) will not create a new index with an invalid or empty key name.
- `db.collection.reIndex()` (page 97), `compact` (page 454), and `repairDatabase` (page 462) 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 202) 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 803) shell to your MongoDB 2.4 database and run the method.

`ensureIndex` and Existing Indexes

Description `db.collection.ensureIndex()` (page 47) 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 47) 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 47) 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 803) shell write methods `db.collection.insert()` (page 79), `db.collection.update()` (page 117), `db.collection.save()` (page 105) and `db.collection.remove()` (page 101) now integrate the `write concern` directly into the method rather than with a separate `getLastError` (page 355) command to provide *acknowledgement of writes* whether run interactively in the `mongo` (page 803) shell or non-interactively in a script. In previous versions, these methods exhibited a "fire-and-forget" behavior. [1145]

- Existing scripts for the `mongo` (page 803) shell that used these methods will now wait for acknowledgement, which take **longer** than the previous "fire-and-forget" behavior.
- The write methods now return a `WriteResult` (page 289) object that contains the results of the operation, including any write errors and write concern errors, and obviates the need to call `getLastError` (page 355) command to get the status of the results. See `db.collection.insert()` (page 79), `db.collection.update()` (page 117), `db.collection.save()` (page 105) and `db.collection.remove()` (page 101) for details.
- In sharded environments, `mongos` (page 792) no longer supports "fire-and-forget" behavior. This limits throughput when writing data to sharded clusters.

Solution Scripts that used these `mongo` (page 803) shell methods for bulk write operations with "fire-and-forget" behavior should use the `Bulk()` (page 210) methods.

In sharded environments, applications using any driver or `mongo` (page 803) shell should use `Bulk()` (page 210) 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 210) operation:

```
var bulk = db.test.initializeUnorderedBulkOp();

for (var i = 1; i <= 1000000; i++) {
    bulk.insert( { x : i} );
}

bulk.execute( { w: 1 } );
```

Bulk method returns a `BulkWriteResult` (page 291) object that contains the result of the operation.

See also:

New Write Operation Protocol (page 1095), `Bulk()` (page 210), `Bulk.execute()` (page 223), `db.collection.initializeUnorderedBulkOp()` (page 213), `db.collection.initializeOrderedBulkOp()` (page 212)

[1145] In previous versions, when using the `mongo` (page 803) shell interactively, the `mongo` (page 803) shell automatically called the `getLastError` (page 355) command after a write method to provide acknowlegement of the write. Scripts, however, would observe "fire-and-forget" behavior in previous versions unless the scripts included an **explicit** call to the `getLastError` (page 355) command after a write method.

`db.collection.aggregate()` Change

Description The `db.collection.aggregate()` (page 20) method in the `mongo` (page 803) 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 940) 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 20) to return a document with a `results` array to handle cursors instead.

See also:

Aggregation Enhancements (page 1094), `db.collection.aggregate()` (page 20),

Write Concern Validation

Description Specifying a write concern that includes `j: true` to a `mongod` (page 770) or `mongos` (page 792) instance running with `--nojournal` (page 778) 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 770) or `mongos` (page 792) instance with `--nojournal` (page 778) or run `mongod` (page 770) or `mongos` (page 792) 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 1110).

Important: Before upgrading the authorization model, you should first upgrade MongoDB binaries to 2.6. For sharded clusters, ensure that **all** cluster components are 2.6. If there are users in any database, be sure you have at least one user in the `admin` database with the role `userAdminAnyDatabase` **before** upgrading the MongoDB binaries.

See also:

Security Improvements (page 1096)

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` (page 908) setting. The setting is available for:

- `mongod` (page 770) and `mongos` (page 792) to bypass the validation of SSL certificates on other servers in the cluster.
- `mongo` (page 803) shell, *MongoDB tools that support SSL*, and the C++ driver to bypass the validation of server certificates.

When using the `allowInvalidCertificates` (page 908) setting, MongoDB logs as a warning the use of the invalid certificates.

Warning: The `allowInvalidCertificates` (page 908) 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 1110). Previous versions used the `ctime` format.

Solution MongoDB adds a new option `--timeStampFormat` (page 794) which supports timestamp format in `ctime` (page 794), `iso8601-utc` (page 794), and `iso8601-local` (page 794) (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` (page 903) value attaches MongoDB components to the localhost interface *only*. These packages set this default in the default configuration file (i.e. `/etc/mongod.conf`.)

Solution If you use one of these packages and have *not* modified the default `/etc/mongod.conf` file, you will need to set `bindIp` (page 903) before or during the upgrade.

There is no default `bindIp` (page 903) 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 101) 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 101) 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 594) must specify a non-empty operand expression. For example, the following expression is now invalid:

  ```
  { $set: { } }
  ```

- *Update operators (e.g $set)* (page 594) 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 $set)* (page 594) to target fields with empty field names (i.e. `""`).
- Updates no longer support saving field names that contain a dot (`.`) or a field name that starts with a dollar sign (`$`).

Solution

- For existing documents that have fields with empty names `""`, replace the whole document. See `db.collection.update()` (page 117) and `db.collection.save()` (page 105) 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 602) the field. To find fields whose names contain a dot, run `db.upgradeCheckAllDBs()` (page 202).
- For existing documents that have fields with names that start with a dollar sign (`$`), `unset` (page 602) or `rename` (page 598) those fields. To find fields whose names start with a dollar sign, run `db.upgradeCheckAllDBs()` (page 202).

See *New Write Operation Protocol* (page 1095) for the changes to the write operation protocol, and *Insert and Update Improvements* (page 1095) for the changes to the insert and update operations. Also consider the documentation of the `Restrictions on Field Names` (page 946).

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 602) or `rename` (page 598) existing fields whose names start with a dollar sign (`$`). Run `db.upgradeCheckAllDBs()` (page 202) 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 142) 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 142).

See *sparse-index-incomplete-results* for an example that details the new behavior.

`sort()` Specification Values

Description The `sort()` (page 156) 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 592) 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 155).

Previous versions ignored `skip()` (page 155) when performing an equality match on the `_id` field.

`explain()` Retains Query Plan Cache

Description `explain()` (page 140) no longer clears the `query plans` cached for that *query shape*.

In previous versions, `explain()` (page 140) would have the side effect of clearing the query plan cache for that query shape.

See also:

The `PlanCache()` (page 204) reference.

Geospatial Changes

`$maxDistance` Changes

Description

- For `$near` (page 565) queries on GeoJSON data, if the queries specify a `$maxDistance` (page 571), `$maxDistance` (page 571) must be inside of the `$near` (page 565) document.

  In previous version, `$maxDistance` (page 571) could be either inside or outside the `$near` (page 565) document.

- `$maxDistance` (page 571) must be a positive value.

Solution

- Update any existing `$near` (page 565) queries on GeoJSON data that currently have the `$maxDistance` (page 571) outside the `$near` (page 565) document
- Update any existing queries where `$maxDistance` (page 571) is a negative value.

Deprecated `$uniqueDocs`

Description MongoDB 2.6 deprecates `$uniqueDocs` (page 575), 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 536) 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 536) expressions.

`null` Comparison Queries

Description

- `$lt` (page 530) and `$gt` (page 529) 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 `$all` (page 575) operator is now equivalent to an `$and` (page 535) 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 575) 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 `$all` (page 575) operator returns no match if the array field contains nested arrays (e.g. `field: [ "a", ["b"] ]`) and `$all` (page 575) 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 544) 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 545) and *Too Many Elements Error* (page 546) for details.

In previous versions, if passed an array with one element, the `$mod` (page 544) operator uses `0` as the second element, and if passed an array with more than two elements, the `$mod` (page 544) 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 544) 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 `$where` (page 558) expressions can now only be at top level and cannot be nested within another expression, such as `$elemMatch` (page 579).

Solution Update existing queries that nest `$where` (page 558).

`$exists` and `notablescan` If the MongoDB server has disabled collection scans, i.e. `notablescan` (page 932), then `$exists` (page 538) 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 579) 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 579) 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 549) 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 1110) 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 1110).

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* (page 960) in *Strict mode*, such as `mongoexport` (page 850) and the REST and HTTP Interfaces[1146].

Other Resources

- All backwards incompatible changes (JIRA)[1147].
- *Release Notes for MongoDB 2.6* (page 1068).
- *Upgrade MongoDB to 2.6* (page 1110) for the upgrade process.

Some changes in 2.6 can affect *compatibility* (page 1099) and may require user actions. The 2.6 `mongo` (page 803) shell provides a `db.upgradeCheckAllDBs()` (page 202) method to perform a check for upgrade preparedness for some of these changes.

See *Compatibility Changes in MongoDB 2.6* (page 1099) for a detailed list of compatibility changes.

See also:

All Backwards incompatible changes (JIRA)[1148].

Upgrade Process

[1146] https://docs.mongodb.org/ecosystem/tools/http-interfaces
[1147] 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
[1148] 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

Upgrade MongoDB to 2.6

On this page

- Upgrade Recommendations and Checklists (page 1111)
- Upgrade MongoDB Processes (page 1112)
- Upgrade Procedure (page 1114)

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 770) instances and replace them with `mongod` (page 770) 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 1111), the procedure for *upgrading sharded clusters* (page 1113), and the considerations for *reverting to 2.4 after running 2.6* (page 1116).

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 1138) for the procedure to upgrade from 2.2 to 2.4.

If you use MongoDB Cloud Manager[1149] 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 803) shell to your MongoDB 2.4 `mongos` (page 792) or `mongod` (page 770) and run the `db.upgradeCheckAllDBs()` (page 202) to check your data set for compatibility. This is a preliminary automated check. Assess and resolve all issues identified by `db.upgradeCheckAllDBs()` (page 202).

Some changes in MongoDB 2.6 require manual checks and intervention. See *Compatibility Changes in MongoDB 2.6* (page 1099) for an explanation of these changes. Resolve all incompatibilities in your deployment before continuing.

For a deployment that uses authentication and authorization, be sure you have at least one user in the `admin` database with the role `userAdminAnyDatabase` **before** upgrading the MongoDB binaries. For deployments currently using authentication and authorization, see the *consideration for deployments that use authentication and authorization* (page 1111).

Authentication MongoDB 2.6 includes significant changes to the authorization model, which requires changes to the way that MongoDB stores users' credentials. As a result, in addition to upgrading MongoDB processes, if your deployment uses authentication and authorization, after upgrading all MongoDB process to 2.6 you **must** also upgrade the authorization model.

Before beginning the upgrade process for a deployment that uses authentication and authorization:

- Ensure that at least one user exists in the `admin` database with the role `userAdminAnyDatabase`.
- 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 770) or `mongos` (page 792) instances.
- You must fully complete the upgrade procedure for *all* MongoDB processes before upgrading the authorization model.

[1149] https://cloud.mongodb.com/?jmp=docs

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 1115).

See *Upgrade User Authorization Data to 2.6 Format* (page 1115) 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 770) from version 2.4 to 2.6. To upgrade from version 2.2 to 2.6, *upgrade to version 2.4* (page 1138) *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[1150]. See `https://docs.mongodb.org/manual/installation` for more information.
2. Shut down your `mongod` (page 770) instance. Replace the existing binary with the 2.6 `mongod` (page 770) binary and restart `mongod` (page 770).

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 1138) *first*, and then follow the procedure to upgrade from MongoDB 2.4 to 2.6.

You can upgrade from MongoDB 2.4 to 2.6 using a "rolling" upgrade to minimize downtime by upgrading the members individually while the other members are available:

Step 1: Upgrade secondary members of the replica set. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` (page 770) and replacing the 2.4 binary with the 2.6 binary. After upgrading a `mongod` (page 770) 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 262) in the `mongo` (page 803) shell.

[1150]http://www.mongodb.org/downloads

Step 2: Step down the replica set primary. Use `rs.stepDown()` (page 263) in the `mongo` (page 803) shell to step down the *primary* and force the set to *failover*. `rs.stepDown()` (page 263) expedites the failover procedure and is preferable to shutting down the primary directly.

Step 3: Upgrade the primary. When `rs.status()` (page 262) shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` (page 770) 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 273)
- `sh.shardCollection()` (page 277)
- `sh.addShard()` (page 269)
- `db.createCollection()` (page 167)
- `db.collection.drop()` (page 45)
- `db.dropDatabase()` (page 177)
- any operation that creates a database
- any other operation that modifies the cluster metadata in any way. See `https://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `https://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 792) instance with the `configDB` (page 926) pointing to the cluster's config servers and with the `--upgrade` option.

To run a `mongos` (page 792) with the `--upgrade` option, you can upgrade an existing `mongos` (page 792) instance to 2.6, or if you need to avoid reconfiguring a production `mongos` (page 792) instance, you can use a new 2.6 `mongos` (page 792) 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 792) instances in your cluster, such as *--sslOnNormalPorts* or *--sslPEMKeyFile*.

The `mongos` (page 792) 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 792) 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 792) instance. If `mongos` (page 792) fails to start, check the log for more information.

If the `mongos` (page 792) 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 796) option the other `mongos` (page 792) instances in the sharded cluster. After upgrading all the `mongos` (page 792), see *Complete Sharded Cluster Upgrade* (page 1114) for information on upgrading the other cluster components.

Complete Sharded Cluster Upgrade After you have successfully upgraded *all* `mongos` (page 792) instances, you can upgrade the other instances in your MongoDB deployment.

> **Warning:** Do not upgrade `mongod` (page 770) instances until after you have upgraded *all* `mongos` (page 792) instances.

While the balancer is still disabled, upgrade the components of your sharded cluster in the following order:

- Upgrade all 3 `mongod` (page 770) 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 770) secondaries before running `replSetStepDown` (page 405) 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 770) with the *--shutdown* (page 775) option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915). See also the *terminate-mongod-processes* for alternate methods of stopping a `mongod` (page 770) instance.

Step 2: Start the new `mongod` instance. Ensure you start the 2.6 `mongod` (page 770) with the same `dbPath` (page 915):

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915).

Upgrade User Authorization Data to 2.6 Format	**On this page**
	• Considerations (page 1115) • Requirements (page 1116) • Procedure (page 1116) • Result (page 1116)

MongoDB 2.6 includes significant changes to the authorization model, which requires changes to the way that MongoDB stores users' credentials. As a result, in addition to upgrading MongoDB processes, if your deployment uses authentication and authorization, after upgrading all MongoDB process to 2.6 you **must** also upgrade the authorization model.

Considerations

Complete all other Upgrade Requirements Before upgrading the authorization model, you should first upgrade MongoDB binaries to 2.6. For sharded clusters, ensure that **all** cluster components are 2.6. If there are users in any database, be sure you have at least one user in the `admin` database with the role `userAdminAnyDatabase` **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 792) 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 792) 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 792) 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 770) instance for a single deployment or a `mongos` (page 792) for a sharded cluster as an `admin` database user with the role `userAdminAnyDatabase`.

Step 2: Upgrade authorization schema. Use the `authSchemaUpgrade` (page 372) command in the `admin` database to update the user data using the `mongo` (page 803) shell.

Run `authSchemaUpgrade` command.

```
db.getSiblingDB("admin").runCommand({authSchemaUpgrade: 1 });
```

In case of error, you may safely rerun the `authSchemaUpgrade` (page 372) command.

Sharded cluster `authSchemaUpgrade` consideration. For a sharded cluster, `authSchemaUpgrade` (page 372) 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 372) on a `mongos` (page 792) instance, you will need to connect to the primary for each shard and repeat the upgrade process after upgrading on the `mongos` (page 792).

Result All users in a 2.6 system are stored in the `admin.system.users` (page 893) collection. To manipulate these users, use the *user management methods* (page 228).

The upgrade procedure copies the version 2.4 `admin.system.users` collection to `admin.system.backup_users`.

The upgrade procedure leaves the version 2.4 `<database>.system.users` collection(s) intact.

Downgrade MongoDB from 2.6

On this page

- Downgrade Recommendations and Checklist (page 1116)
- Downgrade 2.6 User Authorization Model (page 1117)
- Downgrade Updated Indexes (page 1119)
- Downgrade MongoDB Processes (page 1120)
- Downgrade Procedure (page 1122)

Before you attempt any downgrade, familiarize yourself with the content of this document, particularly the *Downgrade Recommendations and Checklist* (page 1116) and the procedure for *downgrading sharded clusters* (page 1121).

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 1119) before downgrading MongoDB 2.6 to 2.4.
- *Remove or downgrade version 2 2dsphere indexes* (page 1120) before downgrading MongoDB 2.6 to 2.4.
- *Downgrade 2.6 User Authorization Model* (page 1117). 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 1121).
- To downgrade replica sets, see *Downgrade a 2.6 Replica Set* (page 1121).
- To downgrade a standalone MongoDB instance, see *Downgrade 2.6 Standalone mongod Instance* (page 1120).

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 241) 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 237) 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 770) instance for a single deployment or a `mongos` (page 792) for a sharded cluster with the appropriate privileges. See *Access Control Prerequisites* (page 1117) for details.

Step 2: Create backup of 2.6 `admin.system.users` collection. Copy all documents in the `admin.system.users` (page 893) 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 289) object with the status of the operation. Upon successful update, the `WriteResult` (page 289) 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 289) 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 792) instances will fail to detect the authorization model downgrade until the user cache is refreshed. You can run `invalidateUserCache` (page 398) on each `mongos` (page 792) instance to refresh immediately, or you can wait until the cache is refreshed automatically at the end of the `user cache invalidation interval` (page 930). To run `invalidateUserCache` (page 398), 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 73) 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 73) 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 770) from version 2.6 to 2.4.

1. Download binaries of the latest release in the 2.4 series from the MongoDB Download Page[1151]. See `https://docs.mongodb.org/manual/installation` for more information.
2. Shut down your `mongod` (page 770) instance. Replace the existing binary with the 2.4 `mongod` (page 770) binary and restart `mongod` (page 770).

[1151] http://www.mongodb.org/downloads

Downgrade a 2.6 Replica Set The following steps outline a "rolling" downgrade process for the replica set. The "rolling" downgrade process minimizes downtime by downgrading the members individually while the other members are available:

Step 1: Downgrade each secondary member, one at a time. For each *secondary* in a replica set:

Replace and restart secondary `mongod` instances. First, shut down the `mongod` (page 770), then replace these binaries with the 2.4 binary and restart `mongod` (page 770). See *terminate-mongod-processes* for instructions on safely terminating `mongod` (page 770) 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 262) method in the `mongo` (page 803) shell.

Step 2: Step down the primary. Use `rs.stepDown()` (page 263) in the `mongo` (page 803) shell to step down the *primary* and force the normal *failover* procedure.

```
rs.stepDown()
```

`rs.stepDown()` (page 263) 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 262) shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` (page 770) 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 273)
- `sh.shardCollection()` (page 277)
- `sh.addShard()` (page 269)
- `db.createCollection()` (page 167)
- `db.collection.drop()` (page 45)
- `db.dropDatabase()` (page 177)
- any operation that creates a database
- any other operation that modifies the cluster meta-data in any way. See `https://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `https://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 770) secondaries *before* downgrading the primary.
   (b) To downgrade the primary, run `replSetStepDown` (page 405) and downgrade.
3. Downgrade all 3 `mongod` (page 770) config server instances, leaving the *first* system in the *`mongos --configdb`* argument to downgrade *last*.
4. Downgrade and restart each `mongos` (page 792), 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 770) with the *`--shutdown`* (page 775) option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915). See also the *terminate-mongod-processes* for alternate methods of stopping a `mongod` (page 770) instance.

Step 2: Start the new `mongod` instance. Ensure you start the 2.4 `mongod` (page 770) with the same `dbPath` (page 915):

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915).

See *Upgrade MongoDB to 2.6* (page 1110) for full upgrade instructions.

Download

To download MongoDB 2.6, go to the downloads page[1152].

Other Resources

- All JIRA issues resolved in 2.6[1153].
- All Third Party License Notices[1154].

[1152]http://www.mongodb.org/downloads
[1153]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%2
rc1%22%2C+%222.6.0-rc2%22%2C+%222.6.0-rc3%22%29
[1154]https://github.com/mongodb/mongo/blob/v2.6/distsrc/THIRD-PARTY-NOTICES

7.2.3 Release Notes for MongoDB 2.4

March 19, 2013

On this page

- Minor Releases (page 1123)
- Major New Features (page 1130)
- Security Enhancements (page 1131)
- Performance Improvements (page 1131)
- Enterprise (page 1137)
- Additional Information (page 1138)

MongoDB 2.4 includes enhanced geospatial support, switch to V8 JavaScript engine, security enhancements, and text search (beta) and hashed index.

Minor Releases

2.4 Changelog

On this page

- 2.4.14 (page 1123)
- 2.4.13 - Changes (page 1123)
- 2.4.12 - Changes (page 1124)
- 2.4.11 - Changes (page 1124)
- 2.4.10 - Changes (page 1124)
- Previous Releases (page 1125)

2.4.14

- Packaging: Init script sets process ulimit to different value compared to documentation (SERVER-17780[1155])
- Security: Compute BinData length in v8 (SERVER-17647[1156])
- Build: Upgrade PCRE Version from 8.30 to Latest (SERVER-17252[1157])

2.4.13 - Changes

- Security: Enforce BSON BinData length validation (SERVER-17278[1158])
- Security: Disable SSLv3 ciphers (SERVER-15673[1159])
- Networking: Improve BSON validation (SERVER-17264[1160])

[1155] https://jira.mongodb.org/browse/SERVER-17780
[1156] https://jira.mongodb.org/browse/SERVER-17647
[1157] https://jira.mongodb.org/browse/SERVER-17252
[1158] https://jira.mongodb.org/browse/SERVER-17278
[1159] https://jira.mongodb.org/browse/SERVER-15673
[1160] https://jira.mongodb.org/browse/SERVER-17264

2.4.12 - Changes

- Sharding: Sharded connection cleanup on setup error can crash mongos (SERVER-15056[1161])
- Sharding: "type 7" (OID) error when acquiring distributed lock for first time (SERVER-13616[1162])
- Storage: explicitly zero .ns files on creation (SERVER-15369[1163])
- Storage: partially written journal last section causes recovery to fail (SERVER-15111[1164])

2.4.11 - Changes

- Security: Potential information leak (SERVER-14268[1165])
- Replication: `_id` with `$prefix` field causes replication failure due to unvalidated insert (SERVER-12209[1166])
- Sharding: Invalid access: seg fault in `SplitChunkCommand::run` (SERVER-14342[1167])
- Indexing: Creating descending index on `_id` can corrupt namespace (SERVER-14833[1168])
- Text Search: Updates to documents with text-indexed fields may lead to incorrect entries (SERVER-14738[1169])
- Build: Add SCons flag to override treating all warnings as errors (SERVER-13724[1170])
- Packaging: Fix mongodb enterprise 2.4 init script to allow multiple processes per host (SERVER-14336[1171])
- JavaScript: Do not store native function pointer as a property in function prototype (SERVER-14254[1172])

2.4.10 - Changes

- Indexes: Fixed issue that can cause index corruption when building indexes concurrently (SERVER-12990[1173])
- Indexes: Fixed issue that can cause index corruption when shutting down secondary node during index build (SERVER-12956[1174])
- Indexes: Mongod now recognizes incompatible "future" text and geo index versions and exits gracefully (SERVER-12914[1175])
- Indexes: Fixed issue that can cause secondaries to fail replication when building the same index multiple times concurrently (SERVER-12662[1176])
- Indexes: Fixed issue that can cause index corruption on the tenth index in a collection if the index build fails (SERVER-12481[1177])
- Indexes: Introduced versioning for text and geo indexes to ensure backwards compatibility (SERVER-12175[1178])

[1161] https://jira.mongodb.org/browse/SERVER-15056
[1162] https://jira.mongodb.org/browse/SERVER-13616
[1163] https://jira.mongodb.org/browse/SERVER-15369
[1164] https://jira.mongodb.org/browse/SERVER-15111
[1165] https://jira.mongodb.org/browse/SERVER-14268
[1166] https://jira.mongodb.org/browse/SERVER-12209
[1167] https://jira.mongodb.org/browse/SERVER-14342
[1168] https://jira.mongodb.org/browse/SERVER-14833
[1169] https://jira.mongodb.org/browse/SERVER-14738
[1170] https://jira.mongodb.org/browse/SERVER-13724
[1171] https://jira.mongodb.org/browse/SERVER-14336
[1172] https://jira.mongodb.org/browse/SERVER-14254
[1173] https://jira.mongodb.org/browse/SERVER-12990
[1174] https://jira.mongodb.org/browse/SERVER-12956
[1175] https://jira.mongodb.org/browse/SERVER-12914
[1176] https://jira.mongodb.org/browse/SERVER-12662
[1177] https://jira.mongodb.org/browse/SERVER-12481
[1178] https://jira.mongodb.org/browse/SERVER-12175

- Indexes: Disallowed building indexes on the system.indexes collection, which can lead to initial sync failure on secondaries (SERVER-10231[1179])
- Sharding: Avoid frequent immediate balancer retries when config servers are out of sync (SERVER-12908[1180])
- Sharding: Add indexes to locks collection on config servers to avoid long queries in case of large numbers of collections (SERVER-12548[1181])
- Sharding: Fixed issue that can corrupt the config metadata cache when sharding collections concurrently (SERVER-12515[1182])
- Sharding: Don't move chunks created on collections with a hashed shard key if the collection already contains data (SERVER-9259[1183])
- Replication: Fixed issue where node appears to be down in a replica set during a compact operation (SERVER-12264[1184])
- Replication: Fixed issue that could cause delays in elections when a node is not vetoing an election (SERVER-12170[1185])
- Replication: Step down all primaries if multiple primaries are detected in replica set to ensure correct election result (SERVER-10793[1186])
- Replication: Upon clock skew detection, secondaries will switch to sync directly from the primary to avoid sync cycles (SERVER-8375[1187])
- Runtime: The SIGXCPU signal is now caught and mongod writes a log message and exits gracefully (SERVER-12034[1188])
- Runtime: Fixed issue where mongod fails to start on Linux when /sys/dev/block directory is not readable (SERVER-9248[1189])
- Windows: No longer zero-fill newly allocated files on systems other than Windows 7 or Windows Server 2008 R2 (SERVER-8480[1190])
- GridFS: Chunk size is decreased to 255 kB (from 256 kB) to avoid overhead with usePowerOf2Sizes option (SERVER-13331[1191])
- SNMP: Fixed MIB file validation under smilint (SERVER-12487[1192])
- Shell: Fixed issue in V8 memory allocation that could cause long-running shell commands to crash (SERVER-11871[1193])
- Shell: Fixed memory leak in the md5sumFile shell utility method (SERVER-11560[1194])

Previous Releases

[1179]https://jira.mongodb.org/browse/SERVER-10231
[1180]https://jira.mongodb.org/browse/SERVER-12908
[1181]https://jira.mongodb.org/browse/SERVER-12548
[1182]https://jira.mongodb.org/browse/SERVER-12515
[1183]https://jira.mongodb.org/browse/SERVER-9259
[1184]https://jira.mongodb.org/browse/SERVER-12264
[1185]https://jira.mongodb.org/browse/SERVER-12170
[1186]https://jira.mongodb.org/browse/SERVER-10793
[1187]https://jira.mongodb.org/browse/SERVER-8375
[1188]https://jira.mongodb.org/browse/SERVER-12034
[1189]https://jira.mongodb.org/browse/SERVER-9248
[1190]https://jira.mongodb.org/browse/SERVER-8480
[1191]https://jira.mongodb.org/browse/SERVER-13331
[1192]https://jira.mongodb.org/browse/SERVER-12487
[1193]https://jira.mongodb.org/browse/SERVER-11871
[1194]https://jira.mongodb.org/browse/SERVER-11560

- All 2.4.9 improvements[1195].
- All 2.4.8 improvements[1196].
- All 2.4.7 improvements[1197].
- All 2.4.6 improvements[1198].
- All 2.4.5 improvements[1199].
- All 2.4.4 improvements[1200].
- All 2.4.3 improvements[1201].
- All 2.4.2 improvements[1202]
- All 2.4.1 improvements[1203].

2.4.14 – April 28, 2015

- Init script sets process ulimit to different value compared to documentation SERVER-17780[1204]
- Compute BinData length in v8 SERVER-17647[1205]
- Upgrade PCRE Version from 8.30 to Latest SERVER-17252[1206]
- *2.4.14 Changelog* (page 1123).
- All 2.4.14 improvements[1207].

2.4.13 – February 25, 2015

- Enforce BSON BinData length validation SERVER-17278[1208]
- Disable SSLv3 ciphers SERVER-15673[1209]
- Improve BSON validation SERVER-17264[1210]
- *2.4.13 Changelog* (page 1123).
- All 2.4.13 improvements[1211].

1195 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.9%22%20AND%20project%20%3D%20SERVER
1196 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.8%22%20AND%20project%20%3D%20SERVER
1197 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.7%22%20AND%20project%20%3D%20SERVER
1198 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.6%22%20AND%20project%20%3D%20SERVER
1199 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.5%22%20AND%20project%20%3D%20SERVER
1200 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.4%22%20AND%20project%20%3D%20SERVER
1201 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.3%22%20AND%20project%20%3D%20SERVER
1202 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.2%22%20AND%20project%20%3D%20SERVER
1203 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.1%22%20AND%20project%20%3D%20SERVER
1204 https://jira.mongodb.org/browse/SERVER-17780
1205 https://jira.mongodb.org/browse/SERVER-17647
1206 https://jira.mongodb.org/browse/SERVER-17252
1207 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.14%22%20AND%20project%20%3D%20SERVER
1208 https://jira.mongodb.org/browse/SERVER-17278
1209 https://jira.mongodb.org/browse/SERVER-15673
1210 https://jira.mongodb.org/browse/SERVER-17264
1211 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.13%22%20AND%20project%20%3D%20SERVER

2.4.12 – October 16, 2014

- Partially written journal last section causes recovery to fail SERVER-15111[1212].
- Explicitly zero `.ns` files on creation SERVER-15369[1213].
- *2.4.12 Changelog* (page 1124).
- All 2.4.12 improvements[1214].

2.4.11 – August 18, 2014

- Fixed potential information leak: SERVER-14268[1215].
- Resolved issue were an `_id` with a `$prefix` field caused replication failure due to unvalidated insert SERVER-12209[1216].
- Addressed issue where updates to documents with text-indexed fields could lead to incorrect entries SERVER-14738[1217].
- Resolved issue where creating descending index on `_id` could corrupt namespace SERVER-14833[1218].
- *2.4.11 Changelog* (page 1124).
- All 2.4.11 improvements[1219].

2.4.10 – April 4, 2014

- Performs fast file allocation on Windows when available SERVER-8480[1220].
- Start elections if more than one primary is detected SERVER-10793[1221].
- Changes to allow safe downgrading from v2.6 to v2.4 SERVER-12914[1222], SERVER-12175[1223].
- Fixes for edge cases in index creation SERVER-12481[1224], SERVER-12956[1225].
- *2.4.10 Changelog* (page 1124).
- All 2.4.10 improvements[1226].

2.4.9 – January 10, 2014

- Fix for instances where `mongos` (page 792) incorrectly reports a successful write SERVER-12146[1227].

[1212] https://jira.mongodb.org/browse/SERVER-15111
[1213] https://jira.mongodb.org/browse/SERVER-15369
[1214] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.12%22%20AND%20project%20%3D%20SERVER
[1215] https://jira.mongodb.org/browse/SERVER-14268
[1216] https://jira.mongodb.org/browse/SERVER-12209
[1217] https://jira.mongodb.org/browse/SERVER-14738
[1218] https://jira.mongodb.org/browse/SERVER-14833
[1219] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.11%22%20AND%20project%20%3D%20SERVER
[1220] https://jira.mongodb.org/browse/SERVER-8480
[1221] https://jira.mongodb.org/browse/SERVER-10793
[1222] https://jira.mongodb.org/browse/SERVER-12914
[1223] https://jira.mongodb.org/browse/SERVER-12175
[1224] https://jira.mongodb.org/browse/SERVER-12481
[1225] https://jira.mongodb.org/browse/SERVER-12956
[1226] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.10%22%20AND%20project%20%3D%20SERVER
[1227] https://jira.mongodb.org/browse/SERVER-12146

- Make non-primary read preferences consistent with `slaveOK` versioning logic SERVER-11971[1228].
- Allow new sharded cluster connections to read from secondaries when primary is down SERVER-7246[1229].
- All 2.4.9 improvements[1230].

2.4.8 – November 1, 2013

- Increase future compatibility for 2.6 authorization features SERVER-11478[1231].
- Fix `dbhash` cache issue for config servers SERVER-11421[1232].
- All 2.4.8 improvements[1233].

2.4.7 – October 21, 2013

- Fixed over-aggressive caching of V8 Isolates SERVER-10596[1234].
- Removed extraneous initial count during mapReduce SERVER-9907[1235].
- Cache results of dbhash command SERVER-11021[1236].
- Fixed memory leak in aggregation SERVER-10554[1237].
- All 2.4.7 improvements[1238].

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[1239].
- Fix for C++ client shutdown issues SERVER-8891[1240].
- Improved replication robustness in presence of high network latency SERVER-10085[1241].
- Improved Solaris support SERVER-9832[1242], SERVER-9786[1243], and SERVER-7080[1244].
- All 2.4.6 improvements[1245].

[1228] https://jira.mongodb.org/browse/SERVER-11971
[1229] https://jira.mongodb.org/browse/SERVER-7246
[1230] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.9%22%20AND%20project%20%3D%20SERVER
[1231] https://jira.mongodb.org/browse/SERVER-11478
[1232] https://jira.mongodb.org/browse/SERVER-11421
[1233] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.8%22%20AND%20project%20%3D%20SERVER
[1234] https://jira.mongodb.org/browse/SERVER-10596
[1235] https://jira.mongodb.org/browse/SERVER-9907
[1236] https://jira.mongodb.org/browse/SERVER-11021
[1237] https://jira.mongodb.org/browse/SERVER-10554
[1238] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.7%22%20AND%20project%20%3D%20SERVER
[1239] https://jira.mongodb.org/browse/SERVER-10478
[1240] https://jira.mongodb.org/browse/SERVER-8891
[1241] https://jira.mongodb.org/browse/SERVER-10085
[1242] https://jira.mongodb.org/browse/SERVER-9832
[1243] https://jira.mongodb.org/browse/SERVER-9786
[1244] https://jira.mongodb.org/browse/SERVER-7080
[1245] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.6%22%20AND%20project%20%3D%20SERVER

2.4.5 – July 3, 2013

- Fix for CVE-2013-4650 Improperly grant user system privileges on databases other than local SERVER-9983[1246].
- Fix for CVE-2013-3969 Remotely triggered segmentation fault in Javascript engine SERVER-9878[1247].
- Fix to prevent identical background indexes from being built SERVER-9856[1248].
- Config server performance improvements SERVER-9864[1249] and SERVER-5442[1250].
- Improved initial sync resilience to network failure SERVER-9853[1251].
- All 2.4.5 improvements[1252].

2.4.4 – June 4, 2013

- Performance fix for Windows version SERVER-9721[1253]
- Fix for config upgrade failure SERVER-9661[1254].
- Migration to Cyrus SASL library for MongoDB Enterprise SERVER-8813[1255].
- All 2.4.4 improvements[1256].

2.4.3 – April 23, 2013

- Fix for `mongo` shell ignoring modified object's `_id` field SERVER-9385[1257].
- Fix for race condition in log rotation SERVER-4739[1258].
- Fix for `copydb` command with authorization in a sharded cluster SERVER-9093[1259].
- All 2.4.3 improvements[1260].

2.4.2 – April 17, 2013

- Several V8 memory leak and performance fixes SERVER-9267[1261] and SERVER-9230[1262].
- Fix for upgrading sharded clusters SERVER-9125[1263].

[1246] https://jira.mongodb.org/browse/SERVER-9983
[1247] https://jira.mongodb.org/browse/SERVER-9878
[1248] https://jira.mongodb.org/browse/SERVER-9856
[1249] https://jira.mongodb.org/browse/SERVER-9864
[1250] https://jira.mongodb.org/browse/SERVER-5442
[1251] https://jira.mongodb.org/browse/SERVER-9853
[1252] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.5%22%20AND%20project%20%3D%20SERVER
[1253] https://jira.mongodb.org/browse/SERVER-9721
[1254] https://jira.mongodb.org/browse/SERVER-9661
[1255] https://jira.mongodb.org/browse/SERVER-8813
[1256] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.4%22%20AND%20project%20%3D%20SERVER
[1257] https://jira.mongodb.org/browse/SERVER-9385
[1258] https://jira.mongodb.org/browse/SERVER-4739
[1259] https://jira.mongodb.org/browse/SERVER-9093
[1260] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.3%22%20AND%20project%20%3D%20SERVER
[1261] https://jira.mongodb.org/browse/SERVER-9267
[1262] https://jira.mongodb.org/browse/SERVER-9230
[1263] https://jira.mongodb.org/browse/SERVER-9125

- Fix for high volume connection crash SERVER-9014[1264].
- All 2.4.2 improvements[1265]

2.4.1 – April 17, 2013

- Fix for losing index changes during initial sync SERVER-9087[1266]
- All 2.4.1 improvements[1267].

Major New Features

The following changes in MongoDB affect both standard and Enterprise editions:

Text Search

Add support for text search of content in MongoDB databases as a *beta* feature. See `https://docs.mongodb.org/manual/core/index-text` for more information.

Geospatial Support Enhancements

- Add new `2dsphere index`. The new index supports GeoJSON[1268] objects `Point`, `LineString`, and `Polygon`. See `https://docs.mongodb.org/manual/core/2dsphere` and `https://docs.mongodb.org/manual/applications/geospatial-indexes`.
- Introduce operators `$geometry` (page 569), `$geoWithin` (page 560) and `$geoIntersects` (page 562) 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 `$geoWithin` (page 560) operator and the `$geoNear` (page 651) pipeline stage.
- Improve sort efficiency when the `$sort` (page 649) stage immediately precedes a `$limit` (page 640) in the pipeline.
- Add new operators `$millisecond` (page 723) and `$concat` (page 696) and modify how `$min` (page 738) operator processes `null` values.

[1264] https://jira.mongodb.org/browse/SERVER-9014
[1265] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.2%22%20AND%20project%20%3D%20SERVER
[1266] https://jira.mongodb.org/browse/SERVER-9087
[1267] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.1%22%20AND%20project%20%3D%20SERVER
[1268] http://geojson.org/geojson-spec.html

Changes to Update Operators

- Add new `$setOnInsert` (page 599) operator for use with `upsert` (page 117) .
- Enhance functionality of the `$push` (page 616) operator, supporting its use with the `$each` (page 619), the `$sort` (page 622), and the `$slice` (page 619) modifiers.

Additional Limitations for Map-Reduce and `$where` Operations

The `mapReduce` (page 318) command, `group` (page 313) command, and the `$where` (page 558) operator expressions cannot access certain global functions or properties, such as `db`, that are available in the `mongo` (page 803) shell. See the individual command or operator for details.

Improvements to `serverStatus` Command

Provide additional metrics and customization for the `serverStatus` (page 492) command. See `db.serverStatus()` (page 197) and `serverStatus` (page 492) for more information.

Security Enhancements

- Introduce a role-based access control system User Privileges[1269] 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 `https://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

On this page

- Improved Concurrency (page 1132)
- Modernized JavaScript Implementation (ES5) (page 1132)
- Removed Non-Standard SpiderMonkey Features (page 1132)

Consider the following impacts of *V8 JavaScript Engine* (page 1131) in MongoDB 2.4:

Tip

Use the new `interpreterVersion()` method in the `mongo` (page 803) shell and the `javascriptEngine` (page 472) field in the output of `db.serverBuildInfo()` (page 196) to determine which JavaScript engine a MongoDB binary uses.

[1269] https://docs.mongodb.org/v2.4/reference/user-privileges

Improved Concurrency Previously, MongoDB operations that required the JavaScript interpreter had to acquire a lock, and a single `mongod` (page 770) 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[1270], abbreviated as ES5, adds many new language features, including:

- standardized JSON[1271],
- strict mode[1272],
- function.bind()[1273],
- array extensions[1274], and
- getters and setters.

With V8, MongoDB supports the ES5 implementation of Javascript with the following exceptions.

Note: The following features do not work as expected on documents **returned from MongoDB queries**:

- `Object.seal()` throws an exception on documents returned from MongoDB queries.
- `Object.freeze()` throws an exception on documents returned from MongoDB queries.
- `Object.preventExtensions()` incorrectly allows the addition of new properties on documents returned from MongoDB queries.
- `enumerable` properties, when added to documents returned from MongoDB queries, are not saved during write operations.

See SERVER-8216[1275], SERVER-8223[1276], SERVER-8215[1277], and SERVER-8214[1278] 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[1279] JavaScript extensions, previously supported by MongoDB's use of SpiderMonkey as its JavaScript engine.

E4X Extensions V8 does not support the *non-standard* E4X[1280] extensions. E4X provides a native XML[1281] 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()`

[1270] http://www.ecma-international.org/publications/standards/Ecma-262.htm
[1271] http://www.ecma-international.org/ecma-262/5.1/#sec-15.12.1
[1272] http://www.ecma-international.org/ecma-262/5.1/#sec-4.2.2
[1273] http://www.ecma-international.org/ecma-262/5.1/#sec-15.3.4.5
[1274] http://www.ecma-international.org/ecma-262/5.1/#sec-15.4.4.16
[1275] https://jira.mongodb.org/browse/SERVER-8216
[1276] https://jira.mongodb.org/browse/SERVER-8223
[1277] https://jira.mongodb.org/browse/SERVER-8215
[1278] https://jira.mongodb.org/browse/SERVER-8214
[1279] https://developer.mozilla.org/en-US/docs/SpiderMonkey
[1280] https://developer.mozilla.org/en-US/docs/E4X
[1281] https://developer.mozilla.org/en-US/docs/E4X/Processing_XML_with_E4X

- `XMLList()`
- `isXMLName()`

Destructuring Assignment V8 does not support the non-standard destructuring assignments. Destructuring assignment "extract[s] data from arrays or objects using a syntax that mirrors the construction of array and object literals." - Mozilla docs[1282]

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[1283].

`InternalError()` V8 does not support `InternalError()`. Use `Error()` instead.

`for each...in` Construct V8 does not support the use of for each...in[1284] 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);
});
```

[1282] https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7#Destructuring_assignment_(Merge_into_own_page.2Fsection)
[1283] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Iterators_and_Generators
[1284] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/for_each...in

Array Comprehension V8 does not support Array comprehensions[1285].

Use other methods such as the `Array` instance methods `map()`, `filter()`, or `forEach()`.

Example

With V8, the following array comprehension is **invalid**:

```
var a = { w: 1, x: 2, y: 3, z: 4 }

var arr = [i * i for each (i in a) if (i > 2)]
printjson(arr)
```

Instead, you can implement using the `Array` *instance* method `forEach()` and the ES5 method `Object.keys()` :

```
var a = { w: 1, x: 2, y: 3, z: 4 }

var arr = [];
Object.keys(a).forEach(function (key) {
  var val = a[key];
  if (val > 2) arr.push(val * val);
})
printjson(arr)
```

Note: The new logic uses the `Array` *instance* method `forEach()` and not the *generic* method `Array.forEach()`; V8 does **not** support `Array` *generic* methods. See *Array Generic Methods* (page 1136) 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[1286].

Example

The following conditional function definition produces different outcomes in SpiderMonkey versus V8:

```
function test () {
   if (false) {
```

[1285] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Predefined_Core_Objects#Array_comprehensions
[1286] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Functions

```
      function go () {};
   }
   print(typeof go)
}
```

With SpiderMonkey, the conditional function outputs `undefined`, whereas with V8, the conditional function outputs `function`.

If your code defines functions this way, it is highly recommended that you refactor the code. The following example refactors the conditional function definition to work in both SpiderMonkey and V8.

```
function test () {
  var go;
  if (false) {
    go = function () {}
  }
  print(typeof go)
}
```

The refactored code outputs `undefined` in both SpiderMonkey and V8.

Note: ECMAscript prohibits conditional function definitions. To force V8 to throw an `Error`, enable strict mode[1287].

```
function test () {
  'use strict';

  if (false) {
    function go () {}
  }
}
```

The JavaScript code throws the following syntax error:

```
SyntaxError: In strict mode code, functions can only be declared at top level or immediately within a
```

String Generic Methods V8 does not support String generics[1288]. 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);
```

1287 http://www.nczonline.net/blog/2012/03/13/its-time-to-start-using-javascript-strict-mode/
1288 https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String#String_generic_methods

With V8, use the `String` *instance* methods instead of following *generic* methods:

`String.charAt()`	`String.quote()`	`String.toLocaleLowerCase()`
`String.charCodeAt()`	`String.replace()`	`String.toLocaleUpperCase()`
`String.concat()`	`String.search()`	`String.toLowerCase()`
`String.endsWith()`	`String.slice()`	`String.toUpperCase()`
`String.indexOf()`	`String.split()`	`String.trim()`
`String.lastIndexOf()`	`String.startsWith()`	`String.trimLeft()`
`String.localeCompare()`	`String.substr()`	`String.trimRight()`
`String.match()`	`String.substring()`	

Array Generic Methods V8 does not support Array generic methods[1289]. Array generics are a set of methods on the `Array` class that mirror instance methods.

Example

The following use of the generic method `Array.every()` is **invalid** with V8:

```
var arr = [4, 8, 15, 16, 23, 42];

function isEven (val) {
   return 0 === val % 2;
}

var allEven = Array.every(arr, isEven);
print(allEven);
```

With V8, use the `Array` instance method `every()` available through an *instance* of the `Array` class instead:

```
var allEven = arr.every(isEven);
print(allEven);
```

With V8, use the `Array` *instance* methods instead of the following *generic* methods:

`Array.concat()`	`Array.lastIndexOf()`	`Array.slice()`
`Array.every()`	`Array.map()`	`Array.some()`
`Array.filter()`	`Array.pop()`	`Array.sort()`
`Array.forEach()`	`Array.push()`	`Array.splice()`
`Array.indexOf()`	`Array.reverse()`	`Array.unshift()`
`Array.join()`	`Array.shift()`	

Array Instance Method `toSource()` V8 does not support the `Array` instance method toSource()[1290]. Use the `Array` instance method `toString()` instead.

`uneval()` V8 does not support the non-standard method `uneval()`. Use the standardized JSON.stringify()[1291] 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 318), `group` (page 313), and `eval` (page 358) and the query operator `$where` (page 558).

[1289] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array#Array_generic_methods
[1290] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/toSource
[1291] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/JSON/stringify

See *JavaScript Changes in MongoDB 2.4* (page 1131) for more information about all changes .

BSON Document Validation Enabled by Default for `mongod` and `mongorestore`

Enable basic *BSON* object validation for `mongod` (page 770) and `mongorestore` (page 824) when writing to MongoDB data files. See `wireObjectCheck` (page 903) for details.

Index Build Enhancements

- Add support for multiple concurrent index builds in the background by a single `mongod` (page 770) instance. See *building indexes in the background* for more information on background index builds.
- Allow the `db.killOp()` (page 192) method to terminate a foreground index build.
- Improve index validation during index creation. See *Compatibility and Index Type Changes in MongoDB 2.4* (page 1145) for more information.

Set Parameters as Command Line Options

Provide `--setParameter` as a command line option for `mongos` (page 792) and `mongod` (page 770). See `mongod` (page 770) and `mongos` (page 792) for list of available options for `setParameter` (page 914).

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 `https://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 `https://docs.mongodb.org/manual/tutorial/control-access-to-mongodb-with-kerberos-authentica` 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[1292] for more information on platform support.

Upgrade Process

Upgrade MongoDB to 2.4

On this page

- Upgrade Recommendations and Checklist (page 1138)
- Upgrade Standalone `mongod` Instance to MongoDB 2.4 (page 1139)
- Upgrade a Replica Set from MongoDB 2.2 to MongoDB 2.4 (page 1139)
- Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4 (page 1139)
- Rolling Upgrade Limitation for 2.2.0 Deployments Running with `auth` Enabled (page 1143)
- Upgrade from 2.3 to 2.4 (page 1143)
- Downgrade MongoDB from 2.4 to Previous Versions (page 1143)

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 770) instances and replace them with `mongod` (page 770) 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 1139) and the considerations for *reverting to 2.2 after running 2.4* (page 1143).

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 770) instance or instances.
- To upgrade to 2.4 sharded clusters *must* upgrade following the *meta-data upgrade procedure* (page 1139).
- If you're using 2.2.0 and running with `authorization` (page 910) 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 1143).
- If you have `system.users` documents (i.e. for `authorization` (page 910)) 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 1131) for more information.

[1292] http://www.mongodb.org/downloads/

Upgrade Standalone `mongod` Instance to MongoDB 2.4

1. Download binaries of the latest release in the 2.4 series from the MongoDB Download Page[1293]. See `https://docs.mongodb.org/manual/installation` for more information.
2. Shutdown your `mongod` (page 770) instance. Replace the existing binary with the 2.4 `mongod` (page 770) binary and restart `mongod` (page 770).

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 770) and replacing the 2.2 binary with the 2.4 binary. After upgrading a `mongod` (page 770) 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 262) in the `mongo` (page 803) shell.
2. Use the `mongo` (page 803) shell method `rs.stepDown()` (page 263) to step down the *primary* to allow the normal *failover* procedure. `rs.stepDown()` (page 263) 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 262), shut down the previous primary and replace `mongod` (page 770) 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 792) with the *`--upgrade`* option, described in this procedure. The upgrade process does not require downtime.

The upgrade to MongoDB 2.4 adds epochs to the meta-data for all collections and chunks in the existing cluster. MongoDB 2.2 processes are capable of handling epochs, even though 2.2 did not require them. This procedure applies only to upgrades from version 2.2. Earlier versions of MongoDB do not correctly handle epochs. See *Cluster Meta-data Upgrade* (page 1139) 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 792) instances in the cluster.
- Upgrade all 3 `mongod` (page 770) config server instances.
- Upgrade the `mongod` (page 770) instances for each shard, one at a time.

See *Upgrade Sharded Cluster Components* (page 1143) for more information.

Cluster Meta-data Upgrade

[1293] http://www.mongodb.org/downloads

Considerations Beware of the following properties of the cluster upgrade process:

- Before you start the upgrade, ensure that the amount of free space on the filesystem for the *config database* (page 885) is at least 4 to 5 times the amount of space currently used by the *config database* (page 885) data files.

  Additionally, ensure that all indexes in the *config database* (page 885) 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 273),
  - `sh.shardCollection()` (page 277),
  - `sh.addShard()` (page 269),
  - `db.createCollection()` (page 167),
  - `db.collection.drop()` (page 45),
  - `db.dropDatabase()` (page 177),
  - any operation that creates a database, or
  - any other operation that modifies the cluster meta-data in any way. See `https://docs.mongodb.org/manual/reference/sharding` for a complete list of sharding commands. Note, however, that not all commands on the `https://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 770) and `mongos` (page 792) 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 887) and `config.collections` (page 887) collections. As always, if storage requirements increase, the `mongod` (page 770) 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 885), 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 1139) 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 885) is at least 4 to 5 times the amount of space currently used by the *config database* (page 885) data files.

   Additionally, ensure that all indexes in the *config database* (page 885) 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 73).

If any index **on the config database** is `{v:0}`, you should rebuild those indexes by connecting to the `mongos` (page 792) and either: rebuild all indexes using the `db.collection.reIndex()` (page 97) method, or drop and rebuild specific indexes using `db.collection.dropIndex()` (page 46) and then `db.collection.ensureIndex()` (page 47). If you need to upgrade the `_id` index to `{v:1}` use `db.collection.reIndex()` (page 97).

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 816) or other backup tools.

3. Ensure there are no version 2.0 `mongod` (page 770) or `mongos` (page 792) 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 792) processes to confirm that none are still active.

4. Start a single 2.4 `mongos` (page 792) process with `configDB` (page 926) 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 792) 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 792).

   Start the `mongos` (page 792) with a command that resembles the following:

```
mongos --configdb <config servers> --upgrade
```

   Without the *`--upgrade`* option 2.4 `mongos` (page 792) 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 792) process starts successfully, the upgrade is complete. If the `mongos` (page 792) process fails to start, check the log for more information.

   If the `mongos` (page 792) 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 792) 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 1142) procedure.

Optional

If the `mongos` (page 792) 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 792) 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 792) 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 792) 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 792) may need to wait for specific collection locks. Do not force the specific collection locks.

6. Upgrade and restart other mongos (page 792) processes in the sharded cluster, *without* the *--upgrade* option.

   See *Upgrade Sharded Cluster Components* (page 1143) 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 792) 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 1140).

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 1139)), you have already disabled the balancer.
2. Shut down two of the three config servers, preferably the last two listed in the configDB (page 926) string. For example, if your configDB (page 926) 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 792) instances will have uninterrupted access to cluster meta-data.
3. mongodump (page 816) 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 824) 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 792) 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 792) instances (one-by-one, to avoid downtime) and use the rs.stepDown() (page 263) 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 792). Begin by connecting to the 2.2 mongos (page 792) with the mongo (page 803) 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 1139).

Upgrade Sharded Cluster Components After you have successfully completed the meta-data upgrade process described in *Meta-data Upgrade Procedure* (page 1140), and the 2.4 mongos (page 792) 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 792) instances in the cluster, in any order.
- Upgrade all 3 mongod (page 770) config server instances, upgrading the *first* system in the *mongos --configdb* argument *last*.
- Upgrade each shard, one at a time, upgrading the mongod (page 770) secondaries before running replSetStepDown (page 405) 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 770) 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 770) 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 770) 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 770) instances.

- `text` indexes are incompatible with 2.2 and earlier `mongod` (page 770) instances.
- using a `hashed` index as a shard key are incompatible with 2.2 and earlier `mongos` (page 792) instances.
- `hashed` indexes are incompatible with 2.0 and earlier `mongod` (page 770) instances.

Important: Collections sharded using hashed shard keys, should **not** use 2.2 `mongod` (page 770) instances, which cannot correctly support cluster operations for these collections.

If you completed the *meta-data upgrade for a sharded cluster* (page 1139), 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 1139), you cannot use 2.0 `mongod` (page 770) or `mongos` (page 792) 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 792) instances before `mongod` (page 770) 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 792) 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 792), 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 770) for a given data set; however, after you create the first `2dsphere` or `text` index with a 2.4 `mongod` (page 770) you will need to run a 2.2 `mongod` (page 770) 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 770), using the *`--shutdown`* option as follows:

  ```
  mongod --dbpath /var/mongod/data --shutdown
  ```

  Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915).
- start the new `mongod` (page 770) processes with the same `dbPath` (page 915) setting, for example:

  ```
  mongod --dbpath /var/mongod/data
  ```

  Replace `/var/mongod/data` with your MongoDB `dbPath` (page 915).

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 770) instance, you can downgrade at any time, by starting the `2.2` `mongod` (page 770) 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 46), for example:

```
db.records.dropIndex( { loc: "2dsphere" } )
db.records.dropIndex( "records_text" )
```

> **Warning:** *--upgrade* will run `repairDatabase` (page 462) 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 770) and you have created a `2dsphere` or `text` index, `mongod` (page 770) 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 [1294] 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

On this page

- New Index Types (page 1145)
- Index Type Validation (page 1145)

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 1143).

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 1143) to downgrade these databases and run 2.2 if needed, however this will run a full database repair (as with `repairDatabase` (page 462)) for all affected databases.

Index Type Validation In MongoDB 2.2 and earlier you could specify invalid index types that did not exist. In these situations, MongoDB would create an ascending (e.g. `1`) index. Invalid indexes include index types specified by strings that do not refer to an existing index type, and all numbers other than `1` and `-1`. [1295]

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:

[1294] The data file version (i.e. `pdfile version`) is independent and unrelated to the release version of MongoDB.

[1295] In 2.4, indexes that specify a type of `"1"` or `"-1"` (the strings `"1"` and `"-1"`) will continue to exist, despite a warning on start-up. **However**, a *secondary* in a replica set cannot complete an initial sync from a primary that has a `"1"` or `"-1"` index. Avoid all indexes with invalid types.

```
db.coll.ensureIndex( { field: "1" } )
```

MongoDB will return the following error document:

```
{
  "err" : "Unknown index plugin '1' in index { field: \"1\" }"
  "code": 16734,
  "n": <number>,
  "connectionId": <number>,
  "ok": 1
}
```

See *Upgrade MongoDB to 2.4* (page 1138) for full upgrade instructions.

Other Resources

- MongoDB Downloads[1296].
- All JIRA issues resolved in 2.4[1297].
- All Backwards incompatible changes[1298].
- All Third Party License Notices[1299].

7.2.4 Release Notes for MongoDB 2.2

On this page

- Upgrading (page 1146)
- Changes (page 1148)
- Licensing Changes (page 1155)
- Resources (page 1155)

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 770), 2.2 is a drop-in replacement for 2.0 and 1.8.

[1296]http://mongodb.org/downloads

[1297]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

[1298]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%22
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%

[1299]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 770) 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 792) instances before upgrading any `mongod` (page 770) 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 770) and `mongos` (page 792) 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[1300].

2. Shutdown your `mongod` (page 770) instance. Replace the existing binary with the 2.2 `mongod` (page 770) 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 770) and replacing the 2.0 binary with the 2.2 binary. After upgrading a `mongod` (page 770) 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 262) in the `mongo` (page 803) shell.

2. Use the `mongo` (page 803) shell method `rs.stepDown()` (page 263) to step down the *primary* to allow the normal *failover* procedure. `rs.stepDown()` (page 263) 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 262), shut down the previous primary and replace `mongod` (page 770) 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*.

[1300] http://downloads.mongodb.org/

- Upgrade all `mongos` (page 792) instances *first*, in any order.
- Upgrade all of the `mongod` (page 770) config server instances using the *stand alone* (page 1147) 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 1147) 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[1301] 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 303) command exposes the aggregation framework, and the `aggregate()` (page 20) helper in the `mongo` (page 803) shell provides an interface to these operations. Consider the following resources for background on the aggregation framework and its use:

- Documentation: `https://docs.mongodb.org/manual/aggregation`
- Reference: *Aggregation Reference* (page 746)

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 `https://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[1302]
2. Improved Yielding on Page Faults[1303]
3. Improved Page Fault Detection on Windows[1304]

To reflect these changes, MongoDB now provides changed and improved reporting for concurrency and use. See *locks* (page 498), recordStats[1305], `db.currentOp()` (page 171), *mongotop* (page 866), and *mongostat* (page 857).

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.

[1301] https://jira.mongodb.org/browse/SERVER-6902
[1302] https://jira.mongodb.org/browse/SERVER-4328
[1303] https://jira.mongodb.org/browse/SERVER-3357
[1304] https://jira.mongodb.org/browse/SERVER-4538
[1305] https://docs.mongodb.org/v2.2/reference/server-status

Shard tagging controls data location, and is complementary but separate from replica set tagging, which controls `read preference` 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 770) instances (e.g. "`production`" or "`reporting`") the application uses to service requests.

See the documentation for the following helpers in the `mongo` (page 803) shell that support tagged sharding configuration:

- `sh.addShardTag()` (page 270)
- `sh.addTagRange()` (page 270)
- `sh.removeShardTag()` (page 275)

Also, see `https://docs.mongodb.org/manual/core/tag-aware-sharding` and `https://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 792) 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 803) shell using the `readPref()` (page 152) cursor method.

Compatibility Changes

Authentication Changes MongoDB 2.2 provides more reliable and robust support for authentication clients, including drivers and `mongos` (page 792) 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 1147).

`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 348) commands will now return the following output:

```
{ 'ok': 1.0, 'value': null }
```

In the `mongo` (page 803) shell, upsert `findAndModify` (page 348) 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[1306] for more information.

`mongodump` 2.2 Output Incompatible with Pre-2.2 `mongorestore` If you use the `mongodump` (page 816) 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 824) to restore that dump.

See: SERVER-6961[1307] for more information.

[1306] https://jira.mongodb.org/browse/SERVER-6226
[1307] https://jira.mongodb.org/browse/SERVER-6961

`ObjectId().toString()` Returns String Literal `ObjectId("...")` In version 2.2, the `toString()` (page 288) method returns the string representation of the *ObjectId()* object and has the format `ObjectId("...")`.

Consider the following example that calls the `toString()` (page 288) 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 289) method returns the value of the *ObjectId()* object as a lowercase hexadecimal string.

Consider the following example that calls the `valueOf()` (page 289) 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.

Collections names may have any other valid UTF-8 string.

See the SERVER-4442[1308] and the *faq-restrictions-on-collection-names* FAQ item.

[1308] https://jira.mongodb.org/browse/SERVER-4442

Restrictions on Database Names for Windows Database names running on Windows can no longer contain the following characters:

```
/\. "*<>:|?
```

The names of the data files include the database name. If you attempt to upgrade a database instance with one or more of these characters, `mongod` (page 770) will refuse to start.

Change the name of these databases before upgrading. See SERVER-4584[1309] and SERVER-6729[1310] 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[1311] for more information.

New `$elemMatch` Projection Operator The `$elemMatch` (page 591) 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 591) reference and the SERVER-2238[1312] and SERVER-828[1313] 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[1314] for more information.

Service Support for `mongos.exe` You may now run `mongos.exe` (page 813) instances as a Windows Service. See the `mongos.exe` (page 813) reference and *manually-create-windows-service* and SERVER-1589[1315] for more information.

Log Rotate Command Support MongoDB for Windows now supports log rotation by way of the `logRotate` (page 465) database command. See SERVER-2612[1316] for more information.

New Build Using SlimReadWrite Locks for Windows Concurrency Labeled "2008+" on the Downloads Page[1317], 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[1318] for more information.

Tool Improvements

Index Definitions Handled by `mongodump` and `mongorestore` When you specify the `--collection` option to `mongodump` (page 816), `mongodump` (page 816) will now backup the definitions for all indexes that exist on the

[1309] https://jira.mongodb.org/browse/SERVER-4584
[1310] https://jira.mongodb.org/browse/SERVER-6729
[1311] https://jira.mongodb.org/browse/SERVER-5516
[1312] https://jira.mongodb.org/browse/SERVER-2238
[1313] https://jira.mongodb.org/browse/SERVER-828
[1314] https://jira.mongodb.org/browse/SERVER-5648
[1315] https://jira.mongodb.org/browse/SERVER-1589
[1316] https://jira.mongodb.org/browse/SERVER-2612
[1317] http://www.mongodb.org/downloads
[1318] https://jira.mongodb.org/browse/SERVER-3844

source database. When you attempt to restore this backup with mongorestore (page 824), the target mongod (page 770) will rebuild all indexes. See SERVER-808[1319] for more information.

mongorestore (page 824) now includes the *--noIndexRestore* option to provide the preceding behavior. Use *--noIndexRestore* to prevent mongorestore (page 824) from building previous indexes.

mongooplog for Replaying Oplogs The mongooplog (page 835) tool makes it possible to pull *oplog* entries from mongod (page 770) instance and apply them to another mongod (page 770) instance. You can use mongooplog (page 835) to achieve point-in-time backup of a MongoDB data set. See the SERVER-3873[1320] case and the mongooplog (page 835) reference.

Authentication Support for mongotop and mongostat mongotop (page 867) and mongostat (page 858) now contain support for username/password authentication. See SERVER-3875[1321] and SERVER-3871[1322] 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 841) 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[1323] for more information.

In mongorestore (page 824), the *--w* option provides support for configurable write concern.

mongodump Support for Reading from Secondaries You can now run mongodump (page 816) when connected to a *secondary* member of a *replica set*. See SERVER-3854[1324] for more information.

mongoimport Support for full 16MB Documents Previously, mongoimport (page 841) would only import documents that were less than 4 megabytes in size. This issue is now corrected, and you may use mongoimport (page 841) to import documents that are at least 16 megabytes ins size. See SERVER-4593[1325] 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 835) and mongodump (page 816) to query for specific timestamps. Consider the following mongodump (page 816) operation:

```
mongodump --db local --collection oplog.rs --query '{"ts":{"$gt":{"$timestamp" : {"t": 1344969612000,
```

See SERVER-3483[1326] for more information.

[1319] https://jira.mongodb.org/browse/SERVER-808
[1320] https://jira.mongodb.org/browse/SERVER-3873
[1321] https://jira.mongodb.org/browse/SERVER-3875
[1322] https://jira.mongodb.org/browse/SERVER-3871
[1323] https://jira.mongodb.org/browse/SERVER-3937
[1324] https://jira.mongodb.org/browse/SERVER-3854
[1325] https://jira.mongodb.org/browse/SERVER-4593
[1326] https://jira.mongodb.org/browse/SERVER-3483

Shell Improvements

Improved Shell User Interface 2.2 includes a number of changes that improve the overall quality and consistency of the user interface for the `mongo` (page 803) shell:

- Full Unicode support.
- Bash-like line editing features. See SERVER-4312[1327] for more information.
- Multi-line command support in shell history. See SERVER-3470[1328] for more information.
- Windows support for the `edit` command. See SERVER-3998[1329] for more information.

Helper to load Server-Side Functions The `db.loadServerScripts()` (page 192) loads the contents of the current database's `system.js` collection into the current `mongo` (page 803) shell session. See SERVER-1651[1330] for more information.

Support for Bulk Inserts If you pass an array of *documents* to the `insert()` (page 79) method, the `mongo` (page 803) shell will now perform a bulk insert operation. See SERVER-3819[1331] and SERVER-2395[1332] for more information.

Note: For bulk inserts on sharded clusters, the `getLastError` (page 355) 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[1333] case and the documentation of the `syslogFacility` (page 897) run-time option or the *`mongod --syslog`* and *`mongos --syslog`* command line-options.

`touch` Command Added the `touch` (page 464) command to read the data and/or indexes from a collection into memory. See: SERVER-2023[1334] and `touch` (page 464) 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[1335] and `indexCounters` for more information.

Padding Specifiable on `compact` Command See the documentation of the `compact` (page 454) and the SERVER-4018[1336] issue for more information.

[1327] https://jira.mongodb.org/browse/SERVER-4312
[1328] https://jira.mongodb.org/browse/SERVER-3470
[1329] https://jira.mongodb.org/browse/SERVER-3998
[1330] https://jira.mongodb.org/browse/SERVER-1651
[1331] https://jira.mongodb.org/browse/SERVER-3819
[1332] https://jira.mongodb.org/browse/SERVER-2395
[1333] https://jira.mongodb.org/browse/SERVER-2957
[1334] https://jira.mongodb.org/browse/SERVER-2023
[1335] https://jira.mongodb.org/browse/SERVER-5784
[1336] https://jira.mongodb.org/browse/SERVER-4018

Added Build Flag to Use System Libraries The Boost library, version 1.49, is now embedded in the MongoDB code base.

If you want to build MongoDB binaries using system Boost libraries, you can pass `scons` using the `--use-system-boost` flag, as follows:

```
scons --use-system-boost
```

When building MongoDB, you can also pass `scons` a flag to compile MongoDB using only system libraries rather than the included versions of the libraries. For example:

```
scons --use-system-all
```

See the SERVER-3829[1337] and SERVER-5172[1338] 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[1339] and SERVER-4683[1340]. For more information about TCMalloc, see the documentation of TCMalloc[1341] itself.

Replication

Improved Logging for Replica Set Lag When *secondary* members of a replica set fall behind in replication, `mongod` (page 770) 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[1342] for more information.

Replica Set Members can Sync from Specific Members The new `replSetSyncFrom` (page 407) command and new `rs.syncFrom()` (page 264) helper in the `mongo` (page 803) 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 407) 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[1343] 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` (page 922) setting and `--replIndexPrefetch` option allow administrators to disable this feature or allow the `mongod` (page 770) to pre-fetch only the index on the `_id` field. See SERVER-6718[1344] for more information.

[1337] https://jira.mongodb.org/browse/SERVER-3829
[1338] https://jira.mongodb.org/browse/SERVER-5172
[1339] https://jira.mongodb.org/browse/SERVER-188
[1340] https://jira.mongodb.org/browse/SERVER-4683
[1341] http://goog-perftools.sourceforge.net/doc/tcmalloc.html
[1342] https://jira.mongodb.org/browse/SERVER-3575
[1343] https://jira.mongodb.org/browse/SERVER-4160
[1344] https://jira.mongodb.org/browse/SERVER-6718

Map Reduce Improvements

In 2.2 Map Reduce received the following improvements:

- Improved support for sharded MapReduce[1345], and
- MapReduce will retry jobs following a config error[1346].

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[1347] 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[1348] and the SERVER-4683[1349] for more information.

Resources

- MongoDB Downloads[1350].
- All JIRA issues resolved in 2.2[1351].
- All backwards incompatible changes[1352].
- All third party license notices[1353].
- What's New in MongoDB 2.2 Online Conference[1354].

7.2.5 Release Notes for MongoDB 2.0

[1345] https://jira.mongodb.org/browse/SERVER-4521
[1346] https://jira.mongodb.org/browse/SERVER-4158
[1347] https://jira.mongodb.org/browse/SERVER-1506
[1348] https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES#L231
[1349] https://jira.mongodb.org/browse/SERVER-4683
[1350] http://mongodb.org/downloads
[1351] 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
[1352] https://jira.mongodb.org/issues/?filter=11225&jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(10483%2C%2010893%2C%2010894%2C%2
[1353] https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES
[1354] http://www.mongodb.com/events/webinar/mongodb-online-conference-sept

On this page

- Upgrading (page 1156)
- Changes (page 1157)
- Resources (page 1161)

Upgrading

Although the major version number has changed, MongoDB 2.0 is a standard, incremental production release and works as a drop-in replacement for MongoDB 1.8.

Preparation

Read through all release notes before upgrading, and ensure that no changes will affect your deployment.

If you create new indexes in 2.0, then downgrading to 1.8 is possible but you must reindex the new collections.

`mongoimport` (page 841) and `mongoexport` (page 850) 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[1355].

`Journaling` is **enabled by default** in 2.0 for 64-bit builds. If you still prefer to run without journaling, start `mongod` (page 770) with the `--nojournal` run-time option. Otherwise, MongoDB creates journal files during startup. The first time you start `mongod` (page 770) with journaling, you will see a delay as `mongod` (page 770) creates new files. In addition, you may see reduced write throughput.

2.0 `mongod` (page 770) instances are interoperable with 1.8 `mongod` (page 770) 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[1356].
2. Shutdown your `mongod` (page 770) instance. Replace the existing binary with the 2.0.x `mongod` (page 770) 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 770) and replacing the 1.8 binary with the 2.0.x binary from the MongoDB Download Page[1357].
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 263) to step down the primary to allow the normal *failover* procedure.

   `rs.stepDown()` (page 263) and `replSetStepDown` (page 405) provide for shorter and more consistent failover procedures than simply shutting down the primary directly.

[1355] https://jira.mongodb.org/browse/SERVER-1097
[1356] http://downloads.mongodb.org/
[1357] http://downloads.mongodb.org/

When the primary has stepped down, shut down its instance and upgrade by replacing the `mongod` (page 770) 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 792) routers in any order.

Changes

Compact Command

A `compact` (page 454) 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[1358] 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[1359].

[1358] https://jira.mongodb.org/browse/SERVER-2563
[1359] https://jira.mongodb.org/browse/SERVER-3866

- The repairDatabase (page 462) command converts indexes to a 2.0 indexes.

To convert all indexes for a given collection to the *2.0 type* (page 1157), invoke the compact (page 454) command.

Once you create new indexes, downgrading to 1.8.x will require a re-index of any indexes created using 2.0. See https://docs.mongodb.org/manual/tutorial/roll-back-to-v1.8-index.

Sharding Authentication

Applications can now use authentication with *sharded clusters*.

Replica Sets

Hidden Nodes in Sharded Clusters In 2.0, mongos (page 792) instances can now determine when a member of a replica set becomes "hidden" without requiring a restart. In 1.8, mongos (page 792) if you reconfigured a member as hidden, you *had* to restart mongos (page 792) 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[1360] 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 https://docs.mongodb.org/manual/reference/write-concern.

[1360] https://docs.mongodb.org/ecosystem/drivers

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 `https://docs.mongodb.org/manual/tutorial/reconfigure-replica-set-with-una`

Primary Checks for a Caught up Secondary before Stepping Down To minimize time without a *primary*, the `rs.stepDown()` (page 263) 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 `https://docs.mongodb.org/manual/tutorial/force-member-to-be-primary`.

Extended Shutdown on the Primary to Minimize Interruption When you call the `shutdown` (page 465) 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 465) 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 454) runs on a *secondary*, the secondary will automatically drop into "recovering" mode until the operation finishes. This prevents clients from trying to read from it while it's busy.

Geospatial Features

Multi-Location Documents Indexing is now supported on documents which have multiple location objects, embedded either inline or in embedded documents. Additional command options are also supported, allowing results to return with not only distance but the location used to generate the distance.

For more information, see *geospatial-indexes-multi-location*.

Polygon searches Polygonal `$within` (page 562) queries are also now supported for simple polygon shapes. For details, see the `$within` (page 562) 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 355) 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 355) 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 355) results.

Note: For bulk inserts on sharded clusters, the `getLastError` (page 355) command alone is insufficient to verify success. Applications should must verify the success of bulk inserts in application logic.

Map Reduce

Output to a Sharded Collection Using the new `sharded` flag, it is possible to send the result of a map/reduce to a sharded collection. Combined with the `reduce` or `merge` flags, it is possible to keep adding data to very large collections from map/reduce jobs.

For more information, see `https://docs.mongodb.org/manual/core/map-reduce/` and the `mapReduce` (page 318) 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 318) 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 546).

$and A special boolean `$and` (page 535) query operator is now available.

Command Output Changes

The output of the `validate` (page 485) 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 803) 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 803) reference.

Most Commands Require Authentication

In 2.0, when running with authentication (e.g. `authorization` (page 910)) *all* database commands require authentication, *except* the following commands.

- `isMaster` (page 409)
- `authenticate` (page 372)
- `getnonce` (page 372)
- `buildInfo` (page 471)
- `ping` (page 484)
- `isdbgrid` (page 430)

Resources

- MongoDB Downloads[1361]
- All JIRA Issues resolved in 2.0[1362]
- All Backward Incompatible Changes[1363]

7.2.6 Release Notes for MongoDB 1.8

On this page

- Upgrading (page 1161)
- Changes (page 1164)
- Resources (page 1167)

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 1162).
- The `mapReduce` (page 318) command has changed in 1.8, causing incompatibility with previous releases. `mapReduce` (page 318) 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 318) 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.

[1361] http://mongodb.org/downloads
[1362] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=11002
[1363] https://jira.mongodb.org/issues/?filter=11023&jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(10889%2C%2010886%2C%2010784%2C%2

Upgrading a Standalone mongod

1. Download the v1.8.x binaries from the MongoDB Download Page[1364].
2. Shutdown your mongod (page 770) instance.
3. Replace the existing binary with the 1.8.x mongod (page 770) binary.
4. Restart MongoDB.

Upgrading a Replica Set

1.8.x *secondaries* **can** replicate from 1.6.x *primaries*.

1.6.x secondaries **cannot** replicate from 1.8.x primaries.

Thus, to upgrade a *replica set* you must replace all of your secondaries first, then the primary.

For example, suppose you have a replica set with a primary, an *arbiter* and several secondaries. To upgrade the set, do the following:

1. For the arbiter:
   (a) Shut down the arbiter.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page[1365].
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 257) 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"
              },
```

[1364] http://downloads.mongodb.org/
[1365] http://downloads.mongodb.org/

```
                {
                        "_id" : 4,
                        "host" : "ubuntu:27021"
                },
        ]
}
config.version++
3
rs.isMaster()
{
        "setName" : "foo",
        "ismaster" : false,
        "secondary" : true,
        "hosts" : [
                "ubuntu:27017",
                "ubuntu:27018"
        ],
        "arbiters" : [
                "ubuntu:27019"
        ],
        "primary" : "ubuntu:27018",
        "ok" : 1
}
// for each secondary
config.members[0].priority = 0
config.members[3].priority = 0
config.members[4].priority = 0
rs.reconfig(config)
```

3. For each secondary:

   (a) Shut down the secondary.

   (b) Restart it with the 1.8.x binary from the MongoDB Download Page[1366].

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[1367].

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*:

[1366]http://downloads.mongodb.org/
[1367]http://downloads.mongodb.org/

- If the shard is a *replica set*, follow the directions above for *Upgrading a Replica Set* (page 1162).
- If the shard is a single `mongod` (page 770) process, shut it down and then restart it with the 1.8.x binary from the MongoDB Download Page[1368].

3. For each `mongos` (page 792):
   (a) Shut down the `mongos` (page 792) process.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page[1369].
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[1370].
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 `https://docs.mongodb.org/manual/core/journaling` to facilitate fast crash recovery and durability in the storage engine. With journaling enabled, a `mongod` (page 770) 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*.

[1368] http://downloads.mongodb.org/
[1369] http://downloads.mongodb.org/
[1370] http://downloads.mongodb.org/

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 318) 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 318) 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 792).
- `getLastError` (page 355) improvements for *sharding*.

1.7.4

- `mongos` (page 792) routes `slaveOk` queries to *secondaries* in *replica sets*.
- New `mapReduce` (page 318) output options.
- *index-type-sparse*.

1.7.3

- Initial *covered index* support.
- Distinct can use data from indexes when possible.
- `mapReduce` (page 318) can merge or reduce results into an existing collection.
- `mongod` (page 770) tracks and `mongostat` (page 858) displays network usage. See *mongostat* (page 857).
- Sharding stability improvements.

1.7.2

- `$rename` (page 598) operator allows renaming of fields in a document.
- `db.eval()` (page 178) 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 `$elemMatch` (page 591) on primitives in embedded arrays.
- Query optimizer enhancements on range queries.
- Window service enhancements.
- Replica set setup improvements.
- `$pull` (page 613) 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[1371], 1.8.0[1372]
- 1.7.6[1373], 1.7.5[1374], 1.7.4[1375], 1.7.3[1376], 1.7.2[1377], 1.7.1[1378], 1.7.0[1379]

Resources

- MongoDB Downloads[1380]
- All JIRA Issues resolved in 1.8[1381]

7.2.7 Release Notes for MongoDB 1.6

On this page

- Upgrading (page 1167)
- Sharding (page 1167)
- Replica Sets (page 1168)
- Other Improvements (page 1168)
- Installation (page 1168)
- 1.6.x Release Notes (page 1168)
- 1.5.x Release Notes (page 1168)

Upgrading

MongoDB 1.6 is a drop-in replacement for 1.4. To upgrade, simply shutdown `mongod` (page 770) 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

`https://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 770) can now be upgraded to a distributed cluster with zero downtime when the need arises.

- `https://docs.mongodb.org/manual/sharding`
- `https://docs.mongodb.org/manual/tutorial/deploy-shard-cluster`
- `https://docs.mongodb.org/manual/tutorial/convert-replica-set-to-replicated-shard-clust`

[1371] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/v09MbhEm62Y
[1372] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/JeHQOnam6Qk
[1373] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/3t6GNZ1qGYc
[1374] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/S5R0Tx9wkEg
[1375] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/9Om3Vuw-y9c
[1376] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/DfNUrdbmflI
[1377] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/df7mwK6Xixo
[1378] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/HUR9zYtTpA8
[1379] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/TUnJCg9161A
[1380] http://mongodb.org/downloads
[1381] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10172

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.

- `https://docs.mongodb.org/manual/replication`
- `https://docs.mongodb.org/manual/tutorial/deploy-replica-set`
- `https://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)
- *$or queries* (page 534)
- Improved concurrency
- *$slice* (page 594) operator for returning subsets of arrays
- 64 indexes per collection (formerly 40 indexes per collection)
- 64-bit integers can now be represented in the shell using NumberLong
- The `findAndModify` (page 348) 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[1382]

1.5.x Release Notes

- 1.5.8[1383]
- 1.5.7[1384]
- 1.5.6[1385]
- 1.5.5[1386]

[1382] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/06_QCC05Fpk
[1383] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/uJfF1QN6Thk
[1384] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/OYvz40RWs90
[1385] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/4l0N2U_H0cQ
[1386] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/oO749nvTARY

- 1.5.4[1387]
- 1.5.3[1388]
- 1.5.2[1389]
- 1.5.1[1390]
- 1.5.0[1391]

You can see a full list of all changes on JIRA[1392].

Thank you everyone for your support and suggestions!

7.2.8 Release Notes for MongoDB 1.4

On this page

- Upgrading (page 1169)
- Core Server Enhancements (page 1169)
- Replication and Sharding (page 1169)
- Deployment and Production (page 1170)
- Query Language Improvements (page 1170)
- Geo (page 1170)

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 770), then restart with the new binaries. (Users upgrading from release 1.0 should review the *1.2 release notes* (page 1170), in particular the instructions for upgrading the DB format.)

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`)

[1387] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/380V_Ec_q1c
[1388] https://groups.google.com/forum/?hl=en&fromgroups=#!topic/mongodb-user/hsUQL9CxTQw
[1389] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/94EE3HVidAA
[1390] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/7SBPQ2RSfdM
[1391] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/VAhJcjDGTy0
[1392] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10107

- replication handles clock skew on master
- *$inc* (page 595) 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 451) 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 465)
- enhancements to *serverStatus* (page 492) command (db.serverStatus()) - counters and *replication lag* stats
- new *mongostat* (page 857) tool

Query Language Improvements

- *$all* (page 575) with regex
- *$not* (page 536)
- partial matching of array elements *$elemMatch* (page 590)
- $ operator for updating arrays
- *$addToSet* (page 609)
- *$unset* (page 602)
- *$pull* (page 613) supports object matching
- *$set* (page 600) with array indexes

Geo

- `2d geospatial search`
- geo *$center* (page 572) and *$box* (page 573) searches

7.2.9 Release Notes for MongoDB 1.2.x

On this page

- New Features (page 1171)
- DB Upgrade Required (page 1171)
- Replication Changes (page 1171)
- mongoimport (page 1171)
- field filter changing (page 1171)

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 770) process
  - run `./mongod --upgrade`
  - start `mongod` (page 770) again
- use a slave
  - start a slave on a different port and data directory
  - when its synced, shut down the master, and start the new slave on the regular port.

Ask in the forums or IRC for more help.

Replication Changes

- There have been minor changes in replication. If you are upgrading a master/slave setup from <= 1.1.2 you have to update the slave first.

mongoimport

- `mongoimportjson` has been removed and is replaced with *mongoimport* (page 840) 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 538)