MongoDB Reference Manual

Release 2.2.2

MongoDB Documentation Project

December 20, 2012

CONTENTS

This document contains all of the reference material from the `MongoDB Manual`, reflecting the 2.2.2 release. See the full manual, for complete documentation of MongoDB, it's operation, and use.

Part I

About MongoDB Documentation

CHAPTER

ONE

ABOUT MONGODB

MongoDB is a *document*-oriented database management system designed for performance, horizontal scalability, high availability, and advanced queryability. See the following wiki pages for more information about MongoDB:

- Introduction
- Philosophy
- About

If you want to download MongoDB, see the downloads page.

If you'd like to learn how to use MongoDB with your programming language of choice, see the introduction to the `drivers`.

CHAPTER

TWO

ABOUT THE DOCUMENTATION PROJECT

2.1 This Manual

The MongoDB documentation project provides a complete manual for the MongoDB database. This resource is replacing eventually replace MongoDB's original documentation.

2.1.1 Licensing

2.1.2 Editions

In addition to the <http://docs.mongodb.org/manual/> site, you can also access this content in the following editions provided for your convenience:

- ePub Format
- Single HTML Page
- PDF Format

PDF files that provide access to subsets of the MongoDB Manual:

- MongoDB Reference Manual
- MongoDB Use Case Guide
- MongoDB CRUD Operation Introduction

For Emacs users Info/Texinfo users, the following experimental Texinfo manuals are available for offline use:

- MongoDB Manual Texinfo (tar.gz)
- MongoDB Reference Manual (tar.gz)
- MongoDB CURD Operation Introduction (tar.gz)

Important: The `texinfo` manuals are experimental. If you find an issue with one of these editions, please file an issue in the DOCS Jira project.

2.1.3 Version and Revisions

This version of the manual reflects version 2.2.2 of MongoDB.

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

This edition reflects "`master`" branch of the documentation as of the "`312d6123917eff61bb399f14104636314934389d`" revision. This branch is explicitly accessible via "http://docs.mongodb.org/master" and you can always reference the commit of the current manual in the release.txt file.

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

2.2 Contributing to the Documentation

The entire source of the documentation is available in the docs repository along with all of the other MongoDB project repositories on GitHub. You can clone the repository by issuing the following command at your system shell:

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

If you have a GitHub account and want to fork this repository, you may issue pull requests, and someone on the documentation team will merge in your contributions promptly. In order to accept your changes to the Manual, you have to complete the MongoDB/10gen Contributor Agreement.

This project tracks issues at MongoDB's DOCS project. If you see a problem with the documentation, please report it there.

2.3 Writing Documentation

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

You can view the documentation style guide and the build instructions in reStructured Text files in the top-level of the documentation repository. If you have any questions, please feel free to open a Jira Case.

Part II

`mongo` Shell and Database Command Reference

CHAPTER

THREE

COMMAND REFERENCE

This document contains a reference to all *database commands*.

The MongoDB command interface provides access to all *non CRUD* database operations. Fetching server stats, initializing a replica set, and running a map-reduce job are all accomplished by running a command.

You specify a command first by constructing a standard *BSON* document whose first key is the name of the command. For example, specify the `isMaster` command using the following *BSON* document:

```
{ isMaster: 1 }
```

The `mongo` shell provides a helper method for running commands called `db.runCommand()`. The following operation in `mongo` runs the above command:

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

Many `drivers` provide an equivalent for the `db.runCommand()` method. Internally, running commands with `db.runCommand()` is equivalent to a special query against the *$cmd* collection.

Many common commands have their own shell helpers or wrappers in the `mongo` shell and drivers, such as the `db.isMaster()` method in the `mongo` JavaScript shell. You must run some commands on the *admin database*. Normally, these operations resemble the followings:

```
use admin
db.runCommand( {buildInfo: 1} )
```

However, there's also a command helper that automatically runs the command in the context of the `admin` database:

```
db._adminCommand( {buildInfo: 1} )
```

All commands return, at minimum, a document with an `ok` field indicating whether the command has succeeded:

```
{ 'ok': 1 }
```

Failed commands return the `ok` field with a value of `0`.

All of the following command descriptions, provide a document template or prototype for each command. Some command documentation also includes the relevant `mongo` shell helpers. See `http://docs.mongodb.org/manual/reference/command` for a list of all commands.

3.1 User Commands

3.1.1 Sharding Commands

See Also:

`http://docs.mongodb.org/manual/sharding` for more information about MongoDB's sharding functionality.

addShard

Parameters

- **hostname** (*string*) – a hostname or replica-set/hostname string.
- **name** (*string*) – Optional. Unless specified, a name will be automatically provided to uniquely identify the shard.
- **maxSize** (*integer*) – Optional. Unless specified, shards will consume the total amount of available space on their machines if necessary. Use the `maxSize` value to limit the amount of space the database can use. Specify this value in megabytes.

Use the `addShard` (page 12) command to add a database instance or replica set to a *sharded cluster*. You must run this command when connected a `mongos` (page 256) instance.

The command takes the following form:

```
{ addShard: "<hostname>:<port>" }
```

Example

```
db.runCommand({addShard: "mongodb0.example.net:27027"})
```

Replace `<hostname>:<port>` with the hostname and port of the database instance you want to add as a shard.

Warning: Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.

The optimal configuration is to deploy shards across *replica sets*. To add a shard on a replica set you must specify the name of the replica set and the hostname of at least one member of the replica set. You must specify at least one member of the set, but can specify all members in the set or another subset if desired. `addShard` (page 12) takes the following form:

```
{ addShard: "replica-set/hostname:port" }
```

Example

```
db.runCommand( { addShard: "repl0/mongodb3.example.net:27327"} )
```

If you specify additional hostnames, all must be members of the same replica set.

Send this command to only one `mongos` (page 256) instance, it will store shard configuration information in the *config database*.

Note: Specify a `maxSize` when you have machines with different disk capacities, or if you want to limit the amount of data on some shards.

The `maxSize` constraint prevents the *balancer* from migrating chunks to the shard when the value of `mem.mapped` (page 172) exceeds the value of `maxSize`.

See Also:

- `sh.addShard()`
- `http://docs.mongodb.org/manual/administration/sharding`
- `http://docs.mongodb.org/manual/tutorial/add-shards-to-shard-cluster`
- `http://docs.mongodb.org/manual/tutorial/remove-shards-from-cluster`

listShards

Use the `listShards` (page 13) command to return a list of configured shards. The command takes the following form:

```
{ listShards: 1 }
```

enableSharding

The `enableSharding` (page 13) command enables sharding on a per-database level. Use the following command form:

```
{ enableSharding: 1 }
```

Once you've enabled sharding in a database, you can use the `shardCollection` (page 13) command to begin the process of distributing data among the shards.

shardCollection

The `shardCollection` (page 13) command marks a collection for sharding and will allow data to begin distributing among shards. You must run `enableSharding` (page 13) on a database before running the `shardCollection` (page 13) command.

```
{ shardCollection: "<db>.<collection>", key: <shardkey> }
```

This enables sharding for the collection specified by `<collection>` in the database named `<db>`, using the key `<shardkey>` to distribute documents among the shard. `<shardkey>` is a document, and takes the same form as an *index specification document*.

Choosing the right shard key to effectively distribute load among your shards requires some planning.

See Also:

`http://docs.mongodb.org/manual/sharding` for more information related to sharding. Also consider the section on *sharding-shard-key* for documentation regarding shard keys.

> **Warning:** There's no easy way to disable sharding after running `shardCollection` (page 13). In addition, you cannot change shard keys once set. If you must convert a sharded cluster to a *standalone* node or *replica set*, you must make a single backup of the entire cluster and then restore the backup to the standalone `mongod` or the replica set..

shardingState

The `shardingState` (page 13) command returns `true` if the `mongod` instance is a member of a sharded cluster. Run the command using the following syntax:

```
{ shardingState: 1 }
```

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed; however, the operation is typically short lived.

removeShard

Starts the process of removing a shard from a *cluster*. This is a multi-stage process. Begin by issuing the following command:

```
{ removeShard : "[shardName]" }
```

The balancer will then migrating chunks from the shard specified by `[shardName]`. This process happens slowly to avoid placing undue load on the overall cluster.

The command returns immediately, with the following message:

```
{ msg : "draining started successfully" , state: "started" , shard: "shardName" , ok : 1 }
```

If you run the command again, you'll see the following progress output:

```
{ msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: 23 , dbs: 1 }, ok: 1 }
```

The `remaining` *document* specifies how many chunks and databases remain on the shard. Use `printShardingStatus` (page 14) to list the databases that you must move from the shard.

Each database in a sharded cluster has a primary shard. If the shard you want to remove is also the primary of one the cluster's databases, then you must manually move the database to a new shard. This can be only after the shard is empty. See the `movePrimary` (page 55) command for details.

After removing all chunks and databases from the shard, you may issue the command again, to return:

```
{ msg: "remove shard completed successfully , stage: "completed", host: "shardName", ok : 1 }
```

printShardingStatus

Returns data regarding the status of a *sharded cluster* and includes information regarding the distribution of *chunks*. `printShardingStatus` (page 14) is only available when connected to a *sharded cluster* via a `mongos` (page 256). Typically, you will use the `sh.status()` `mongo` shell wrapper to access this data.

3.1.2 Aggregation Commands

group

The `group` (page 14) command 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 14) command takes the following prototype form:

```
{ group: { ns: <namespace>,
           key: <key>,
           $reduce: <reduce function>,
           $keyf: <key function>,
           cond: <query>,
           finalize: <finalize function> } }
```

The command fields are as follows:

Fields

- **ns** – Specifies the collection from which to perform the group by operation.
- **key** – Specifies one or more document fields to group. Returns a "key object" for use as the grouping key.
- **$reduce** – Specifies an aggregation function that operates on the documents during the grouping operation, such as compute a sum or a count. The aggregation function takes two arguments: the current document and an aggregation result document for that group.
- **initial** – Initializes the aggregation result document.

- **$keyf** – Optional. Alternative to the `key` field. Specifies a function that creates a "key object" for use as the grouping key. Use the `keyf` instead of `key` to group by calculated fields rather than existing document fields.
- **cond** – Optional. Specifies the selection criteria to determine which documents in the collection to process. If the `cond` field is omitted, the `db.collection.group()` processes all the documents in the collection for the group operation.
- **finalize** – Optional. Specifies a function that runs each item in the result set before `db.collection.group()` returns the final value. This function can either modify the result document or replace the result document as a whole.

  Note: Unlike the `$keyf` and the `$reduce` fields that specify a function, the field name is `finalize` and not `$finalize`.

Warning:
- The `group` (page 14) command does not work with *sharded clusters*. Use the *aggregation framework* or *map-reduce* in *sharded environments*.
- The `group` (page 14) command takes a read lock and does not allow any other threads to execute JavaScript while it is running.

Note: The result set must fit within the *maximum BSON document size* (page 265).

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

For the shell, MongoDB provides a wrapper method `db.collection.group()`; however, the `db.collection.group()` method takes the `keyf` field and the `reduce` field whereas the `group` (page 14) command takes the `$keyf` field and the `$reduce` field.

Consider the following examples of the `db.collection.group()` 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 }
}
```

- 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 documents 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
```

- 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
```

•The following example groups by the calculated `day_of_week` field, those documents that have `ord_dt` greater than `01/01/2012` and calculates the sum, count, and average of the `qty` field for each grouping:

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

                                  result.day_of_week = weekdays[result.day_of_week];
                                  result.avg = Math.round(result.total / result.count);
                     }

                   }
               } )
```

The `retval` field of the returned document is an array of documents that contain the group by fields and the calculated aggregation field:

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

      ],
  "count" : 13,
  "keys" : 3,
  "ok" : 1 }
```

See Also:

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

count

The count (page 17) command counts the number of documents in a collection. The command returns a document that contains the count as well as the command status. The count (page 17) command takes the following prototype form:

```
{ count: <collection>, query: <query>, limit: <limit>, skip: <skip> }
```

The command fields are as follows:

Fields

- **count** (*String*) – The name of the collection to count.
- **query** (*document*) – Optional. Specifies the selection query to determine which documents in the collection to count.
- **limit** (*integer*) – Optional. Specifies the limit for the documents matching the selection `query`.
- **skip** (*integer*) – Optional. Specifies the number of matching documents to skip.

Consider the following examples of the count (page 17) command:

- Count 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 the number of the documents in the `orders` collection with the field `ord_dt` greater than `new 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 }
```

- Count the number of the documents in the `orders` collection with the field `ord_dt` greater than `new Date('01/01/2012')` skipping the first `10` matching records:

```
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 }
```

Note: MongoDB also provides the `cursor.count()` method and the shell wrapper `db.collection.count()` method.

mapReduce

The mapReduce (page 18) command allows you to run *map-reduce* aggregation operations over a collection. The mapReduce (page 18) command has the following prototype form:

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

Pass the name of the collection to the `mapReduce` command (i.e. `<collection>`) to use as the source documents to perform the map reduce operation. The command also accepts the following parameters:

Parameters

- **map** – A JavaScript function that associates or "maps" a value with a key.

  The `map` function has the following prototype:

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

  The `map` function process every input document for the map-reduce operation. All the `key` and `value` pairs emitted by the map function. In map-reduce operations, the operation groups the output from the map phase by the `key` value and passes these groupings to the `reduce` function.

  Note:

  – 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.)

  – The `emit(key,value)` function associates the `key` with a `value`.

    * A single emit can only hold half of MongoDB's *maximum BSON document size* (page 265).

    * There is no limit to the number of times you may call the `emit` function per document.

  – The `map` function can access the variables defined in the `scope` parameter.

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

  The `reduce` function has the following prototype:

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

The `reduce` function accepts `key` and `values` arguments. The elements of the `values` array are the individual `value` objects emitted by the `<map>` function, grouped by the `key`.

Note:

– The `reduce` function should *not* access the database, even to perform read operations.

– The `reduce` function should *not* affect the outside system.

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

1. the *type* of the return object must be **identical** to the type of the `value` emitted by the `<map>` function to ensure that the following operations is true:

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

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

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

3. 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 ] )
```

– MongoDB will **not** call the `reduce` function for a key that has only a single value.

– The `reduce` function can access the variables defined in the `scope` parameter.

- **out** – Specifies the location of the result of the map-reduce operation. 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.

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

  – **Output to a collection**.

  ```
  { out: <collectionName> }
  ```

  – **Output to a collection and specify one of the following actions**. This option is only available when passing `out` a collection that already exists. This option is not available on secondary members of replica sets.

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

  ∗ `<action>`: Specify one of the following actions:

    · `replace`

    ```
    { out: { replace: <collectionName> } }
    ```

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

· `merge`

```
{ out: { merge: <collectionName> } }
```

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`

```
{ out: { reduce: <collectionName> } }
```

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 2.1. Optional. Specify output operation as non-atomic and is valid *only* for `merge` and `reduce` output modes which may take minutes to execute.

If `nonAtomic` is `true`, the post-processing step will prevent MongoDB from locking the database; however, other clients will be able to read intermediate states of the output collection. Otherwise the map reduce operation must lock the database during post-processing.

– **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 265).

- **query** – Optional. Specifies the selection criteria using *query operators* (page 111) for determining the documents input to the `map` function.
- **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.
- **limit** – Optional. Specifies a maximum number of documents to return from the collection.
- **finalize** – Optional. A JavaScript function that follows the `<reduce>` method and modifies the output and 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.

Note:

- 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.

- **scope** (*document*) – Optional. Specifies global variables that are accessible in the `map` , `reduce` and the `finalize` functions.
- **jsMode** (*Boolean*) – Optional. Specifies whether to convert intermediate data into BSON format between the execution of the `map` and `reduce` functions.

  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.
- **verbose** (*Boolean*) – Optional. Specifies whether to include the `timing` information in the result information. The `verbose` defaults to `true` to include the `timing` information.

Consider the following prototype map:dbcommand:*mapReduce* operation:

```
var mapFunction = function() { ... };
var reduceFunction = function(key, values) { ... };

db.runCommand(
               {
                 mapReduce: 'orders',
                 map: mapFunction,
                 reduce: reduceFunction,
                 out: { merge: 'map_reduce_results' },
                 query: { ord_date: { $gt: new Date('01/01/2012') } }
               }
             )
```

In the mongo, the db.collection.mapReduce() method is a wrapper around the mapReduce (page 18) command. The following examples use the db.collection.mapReduce():

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: 250,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

- Perform map-reduce operation on the orders collection to group by the cust_id, and for each cust_id, calculate the sum of the price for each cust_id:

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

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

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

  2. Define the corresponding reduce function with two arguments keyCustId and valuesPrices:

     - The valuesPrices is an array whose elements are the price values emitted by the map function and grouped by keyCustId.
     - The function reduces the valuesPrice array to the sum of its elements.

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

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

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

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

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

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

   – In the function, `this` refers to the document that the map-reduce operation is processing.

   – For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

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

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

   – `valuesCountObjects` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.

   – The function reduces the `valuesCountObjects` array to a single object `reducedValue` that also contains the `count` and the `qty` fields.

   – In `reducedValue`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

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

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

                          return reducedValue;
                      };
```

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

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

                            reducedValue.average = reducedValue.qty/reducedValue.count;

                            return reducedValue;
                        };
```

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

```
db.orders.mapReduce( mapFunction2,
                     reduceFunction2,
```

```
                    {
                      out: { merge: "map_reduce_example" },
                      query: { ord_date: { $gt: new Date('01/01/2012') } },
                      finalize: finalizeFunction2
                    }
                  )
```

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

For more information and examples, see the `Map-Reduce` page.

See Also:

- *map-reduce* and `db.collection.mapReduce()`
- `http://docs.mongodb.org/manual/applications/aggregation`

findAndModify

The `findAndModify` (page 25) command atomically 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>, <options> }
```

The `findAndModify` (page 25) command takes the following are sub-document `options`:

Fields

- **query** (*document*) – Optional. Specifies the selection criteria for the modification. The `query` field employs the same *query selectors* (page 111) as used in the `db.collection.find()` method. Although the query may match multiple documents, `findAndModify` (page 25) will only select one document to modify.

  The `query` field has the following syntax:

  ```
  query: { <query expression> }
  ```

- **sort** (*document*) – Optional. Determines which document the operation will modify if the query selects multiple documents. `findAndModify` (page 25) will modify the first document in the sort order specified by this argument.

  The `sort` field has the following syntax:

  ```
  sort: { field1: value1, field2: value2, ... }
  ```

- **remove** (*boolean*) – Optional if `update` field exists. When `true`, removes the selected document. The default is `false`.

  The `remove` field has the following syntax:

  ```
  remove: <boolean>
  ```

- **update** (*document*) – Optional if `remove` field exists. Performs an update of the selected document. The `update` field employs the same *update operators* (page 127) or `field: value` specifications to modify the selected document.

  ```
  update: { <update expression> }
  ```

- **new** (*boolean*) – Optional. When `true`, returns the modified document rather than the original. The `findAndModify` (page 25) method ignores the `new` option for `remove` operations. The default is `false`.

  ```
  new: <boolean>
  ```

- **fields** (*document*) – Optional. A subset of fields to return.

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

- **upsert** (*boolean*) – Optional. Used in conjunction with the `update` field. When `true`, the `findAndModify` (page 25) command creates a new document if the `query` returns no documents. The default is `false`. In version 2.2, the `findAndModify` (page 25) command returns `null` when `upsert` is `true`.

  ```
  upsert: <boolean>
  ```

  Changed in version 2.2: Previously, `upsert` operations returned a an empty document (e.g. `{ },`) see *the 2.2 release notes* (page 288) for more information.

Consider the following example:

```
{ 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` (page 112) 10.
2. The `sort` orders the results of the query in ascending order.
3. The update `increments` (page 127) the value of the `score` field by 1.
4. The command returns the original unmodified document selected for this update.

The shell and many *drivers* provide a `findAndModify()` (page 25) helper method. Using the shell helper, this same operation can take the following form:

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

> **Warning:** When using `findAndModify` (page 25) in a *sharded* environment, the `query` must contain the *shard key* for all operations against the shard cluster. `findAndModify` (page 25) operations issued against `mongos` (page 256) instances for non-sharded collections function normally.

Note: This command obtains a write lock on the affected database and will block other operations until it has completed; however, typically the write lock is short lived and equivalent to other similar `update()` operations.

`distinct`

The `distinct` (page 26) command finds the distinct values for a specified field across a single collection. The command returns a document that contains an array of the distinct values as well as the query plan and status. The command takes the following prototype form:

```
{ distinct: collection, key: <field>, query: <query> }
```

The command fields are as follows:

Fields

- **collection** (*String*) – The name of the collection to query for distinct values.
- **field** (*string*) – Specifies the field for which to return the distinct values.
- **query** (*document*) – Optional. Specifies the selection `query` to determine the subset of documents from which to retrieve the distinct values.

Consider the following examples of the `distinct` (page 26) command:

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

```
db.runCommand ( { distinct: 'orders', key: 'ord_dt' } )
```

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

```
db.runCommand ( { distinct: 'orders', key: 'item.sku' } )
```

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

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

Note:

- MongoDB also provides the shell wrapper method `db.collection.distinct()` for the `distinct` (page 26) command. Additionally, many MongoDB *drivers* also provide a wrapper method. Refer to the specific driver documentation.
- When possible, the `distinct` (page 26) command will use an index to find the documents in the query as well as to return the data.

---

eval

The `eval` (page 27) command evaluates JavaScript functions on the database server and has the following form:

```
{
  eval: <function>,
  args: [ <arg1>, <arg2> ... ],
  nolock: <boolean>
}
```

The command contains the following fields:

Parameters

- **function** (*JavaScript*) – A JavaScript function.

  The function may accept no arguments, as in the following example:

  ```
  function () {
    // ...
  }
  ```

  The function can also accept arguments, as in the following example:

```
function (arg1, arg2) {
   // ...
}
```

- **arguments** – A list of arguments to pass to the JavaScript `function` if the function accepts arguments. Omit if the `function` does not take arguments.

Fields

- **args** (*Array*) – An array of corresponding arguments to the `function`. Omit `args` if the `function` does not take arguments.
- **nolock** (*Boolean*) – Optional. Specifies whether to disable the write lock. By default, `eval` (page 27) takes a write lock. There are circumstances where the `eval` (page 27) executes a strictly read-only operation that does not need to block other operations. If `nolock` is `true`, the `eval` (page 27) does not take a write lock.

> **Warning:** Do not disable the write lock if the function may modify the contents of the database in any way.

Consider the following example which uses `eval` (page 27) 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 shell also provides a helper method `db.eval()`, 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 );
```

You cannot pass the `nolock` flag to the `db.eval()` in the `mongo` shell.

If you want to use the server's interpreter, you must run `eval` (page 27). Otherwise, the `mongo` shell's JavaScript interpreter evaluates functions entered directly into the shell.

If an error occurs, `eval` (page 27) throws an exception. Consider the following invalid function that 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:

```
{
   "errno" : -3,
   "errmsg" : "invoke failed: JS Error: ReferenceError: x is not defined nofile_b:1",
   "ok" : 0
}
```

> **Warning:**
> - The `db.eval()` operation takes a write lock by default. This means that `eval` (page 27) blocks all other read and write operations to the database while the `eval` (page 27) operation runs. You can, however, disable the lock by setting the `nolock` flag to `true` if the `eval` (page 27) performs a strictly read-only operation.
> - `eval` (page 27) also takes a JavaScript lock.
> - Do not use `eval` (page 27) for long running operations as `eval` (page 27) blocks all other operations. Consider using `other server side code execution options`.
> - You can not use `eval` (page 27) with *sharded* data. In general, you should avoid using `eval` (page 27) in *sharded cluster*; nevertheless, it is possible to use `eval` (page 27) with non-sharded collections and databases stored in a *sharded cluster*.

See Also:

`http://docs.mongodb.org/manual/applications/server-side-javascript`

`aggregate`

New in version 2.1.0. `aggregate` (page 29) implements the *aggregation framework*. Consider the following prototype form:

```
{ aggregate: "[collection]", pipeline: [pipeline] }
```

Where `[collection]` specifies the name of the collection that contains the data that you wish to aggregate. The `pipeline` argument holds an array that contains the specification for the aggregation operation. Consider the following example from the `aggregation documentation`.

```
db.runCommand(
{ aggregate : "article", pipeline : [
  { $project : {
     author : 1,
     tags : 1,
  } },
  { $unwind : "$tags" },
```

```
  { $group : {
     _id : "$tags",
     authors : { $addToSet : "$author" }
  } }
 ] }
);
```

More typically this operation would use the aggregate (page 29) helper in the mongo shell, and would resemble the following:

```
db.article.aggregate(
  { $project : {
     author : 1,
     tags : 1,
  } },
  { $unwind : "$tags" },
  { $group : {
     _id : "$tags",
     authors : { $addToSet : "$author" }
  } }
);
```

For more aggregation documentation, please see:

- http://docs.mongodb.org/manual/applications/aggregation
- *Aggregation Framework Reference* (page 141)
- http://docs.mongodb.org/manual/tutorial/aggregation-examples

3.1.3 Replication Commands

See Also:

"http://docs.mongodb.org/manual/core/replication" for more information regarding replication.

resync

The resync command forces an out-of-date slave mongod instance to re-synchronize itself. Note that this command is relevant to master-slave replication only. It does no apply to replica sets.

Warning: This command obtains a global write lock and will block other operations until it has completed.

replSetFreeze

The replSetFreeze command prevents a replica set member from seeking election for the specified number of seconds. Use this command in conjunction with the replSetStepDown command to make a different node in the replica set a primary.

The replSetFreeze command uses the following syntax:

```
{ replSetFreeze: <seconds> }
```

If you want to unfreeze a replica set member before the specified number of seconds has elapsed, you can issue the command with a seconds value of 0:

```
{ replSetFreeze: 0 }
```

Restarting the `mongod` process also unfreezes a replica set member.

`replSetFreeze` is an administrative command, and you must issue the it against the *admin database*.

replSetGetStatus

The `replSetGetStatus` command returns the status of the replica set from the point of view of the current server. You must run the command against the *admin database*. The command has the following prototype format:

```
{ replSetGetStatus: 1 }
```

However, you can also run this command from the shell like so:

```
rs.status()
```

See Also:

"*Replica Set Status Reference* (page 193)" and "`http://docs.mongodb.org/manual/core/replication`"

replSetInitiate

The `replSetInitiate` 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()` helper:

```
  config = {
      _id : "my_replica_set",
       members : [
           {_id : 0, host : "rs1.example.net:27017"},
           {_id : 1, host : "rs2.example.net:27017"},
           {_id : 2, host : "rs3.example.net", arbiterOnly: true},
       ]
  }

  rs.initiate(config)

Notice that omitting the port cause the host to use the default port
of 27017. Notice also that you can specify other options in the config
documents such as the ``arbiterOnly`` setting in this example.
```

See Also:

"`http://docs.mongodb.org/manual/reference/replica-configuration`," "`http://docs.mongodb.org/manual/administration/replica-sets`," and "*Replica Set Reconfiguration*."

`replSetReconfig`

The `replSetReconfig` command modifies the configuration of an existing replica set. You can use this command to add and remove members, and to alter the options set on existing members. Use the following syntax:

```
{ replSetReconfig: <new_config_document>, force: false }
```

You may also run the command using the shell's `rs.reconfig()` method.

Be aware of the following `replSetReconfig` 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` command can lead to a *rollback* situation. Use with caution.

  Use the force option to restore a replica set to new servers with different hostnames. This works even if the set members already have a copy of the data.
- A majority of the set's members must be operational for the changes to propagate properly.
- This command can cause downtime as the set renegotiates primary-status. Typically this is 10-20 seconds, but could be as long as a minute or more. Therefore, you should attempt to reconfigure only during scheduled maintenance periods.
- In some cases, `replSetReconfig` forces the current primary to step down, initiating an election for primary among the members of the replica set. When this happens, the set will drop all current connections.

Note: `replSetReconfig` obtains a special mutually exclusive lock to prevent more than one :dbcommand'replSetReconfig' operation from occurring at the same time.

`replSetSyncFrom`

New in version 2.2.

Options

- **host** – Specifies the name and port number of the set member that you want *this* member to sync from. Use the `[hostname]:[port]` form.

`replSetSyncFrom` allows you to explicitly configure which host the current `mongod` will poll *oplog* entries from. This operation may be useful for testing different patterns and in situations where a set member is not syncing from the host you want. The member to sync from must be a valid source for data in the set; a member of a replica set cannot sync from:

- itself.
- an arbiter, because arbiters do not hold data.
- a member that does not build indexes.
- an unreachable member.
- a `mongod` instance that is not a member of the same replica set.

If you attempt to sync from a member that is more than 10 seconds behind the current member, `mongod` will return and log a warning, but *will* sync from such members.

The command has the following prototype form:

```
{ replSetSyncFrom: "[hostname]:[port]" }
```

To run the command in the mongo shell, use the following invocation:

```
db.adminCommand( { replSetSyncFrom: "[hostname]:[port]" } )
```

You may also use the `rs.syncFrom()` helper in the mongo shell, in an operation with the following form:

```
rs.syncFrom("[hostname]:[port]")
```

Note: `replSetSyncFrom` provides a temporary override of default behavior. When you restart the `mongod` instance, if the connection that the `mongod` uses to sync, the `mongod` will revert to the default logic for selecting a sync source.

3.1.4 Geospatial Commands

geoNear

The `geoNear` (page 33) command provides an alternative to the `$near` (page 123) operator. In addition to the functionality of `$near` (page 123), `geoNear` (page 33) returns the distance of each item from the specified point along with additional diagnostic information. For example:

```
{ geoNear : "places" , near : [50,50], num : 10 }
```

Here, `geoNear` (page 33) returns the 10 items nearest to the coordinates `[50,50]` in the collection named `places`. *geoNear* provides the following options (specify all distances in the same units as the document coordinate system:)

Fields

- **near** – Takes the coordinates (e.g. `[ x, y ]`) to use as the center of a geospatial query.
- **num** – Optional. Specifies the maximum number of documents to return. The default value is 100.
- **maxDistance** – Optional. Limits the results to those falling within a given distance of the center coordinate.
- **query** – Optional. Further narrows the results using any standard MongoDB query operator or selection. See `db.collection.find()` and "*Operator Reference* (page 111)" for more information.
- **spherical** – Optional. Default: `false`. When `true` MongoDB will return the query as if the coordinate system references points on a spherical plane rather than a plane.
- **distanceMultiplier** – Optional. Specifies a factor to multiply all distances returned by `geoNear` (page 33). For example, use `distanceMultiplier` to convert from spherical queries returned in radians to linear units (i.e. miles or kilometers) by multiplying by the radius of the Earth.
- **includeLocs** – Optional. Default: `false`. When specified `true`, the query will return the location of the matching documents in the result.

- **uniqueDocs** – Optional. Default `true`. The default settings will only return a matching document once, even if more than one of its location fields match the query. When `false` the query will return documents with multiple matching location fields more than once. See `$uniqueDocs` (page 125) for more information on this option

geoSearch

The `geoSearch` (page 34) command provides an interface to MongoDB's *haystack index* functionality. These indexes are useful for returning results based on location coordinates *after* collecting results based on some other query (i.e. a "haystack.") Consider the following example:

```
{ geoSearch : "places", near : [33, 33], maxDistance : 6, search : { type : "restaurant" }, limi
```

The above command returns all documents with a `type` of `restaurant` having a maximum distance of 6 units from the coordinates `[30,33]` in the collection `places` up to a maximum of 30 results.

Unless specified otherwise, the `geoSearch` (page 34) command limits results to 50 documents.

3.1.5 Collection Commands

drop

The `drop` (page 34) command removes an entire collection from a database. The command has following syntax:

```
{ drop: <collection_name> }
```

The `mongo` shell provides the equivalent helper method:

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

Note that 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.

cloneCollection

The `cloneCollection` (page 34) command copies a collection from a remote server to the server where you run the command.

Parameters

- **from** – Specify a resolvable hostname, and optional port number of the remote server where the specified collection resides.
- **query** – Optional. A query document, in the form of a *document*, that filters the documents in the remote collection that `cloneCollection` (page 34) will copy to the current database. See `db.collection.find()`.
- **copyIndexes** (*Boolean*) – Optional. `true` by default. When set to `false` the indexes on the originating server are *not* copied with the documents in the collection.

Consider the following example:

```
{ cloneCollection: "users", from: "db.example.net:27017", query: { active: true }, copyIndexes:
```

This operation copies the "users" collection from the current database on the server at `db.example.net`. The operation only copies documents that satisfy the query `{ active: true }` and does not copy indexes. `cloneCollection` (page 34) copies indexes by default, but you can disable this behavior by setting `{ copyIndexes: false }`. The `query` and `copyIndexes` arguments are optional.

`cloneCollection` (page 34) creates a collection on the current database with the same name as the origin collection. If, in the above example, the `users` collection already exists, then MongoDB appends documents in the remote collection to the destination collection.

`create`

The `create` command explicitly creates a collection. The command uses the following syntax:

```
{ create: <collection_name> }
```

To create a *capped collection* limited to 40 KB, issue command in the following form:

```
{ create: "collection", capped: true, size: 40 * 1024 }
```

The options for creating capped collections are:

Options

- **capped** – Specify `true` to create a *capped collection*.
- **autoIndexId** – Specify `false` to disable the automatic index created on the `_id` field. Before 2.2, the default value for `autoIndexId` was `false`. See *_id Fields and Indexes on Capped Collections* (page 290) for more information.
- **size** – The maximum size for the capped collection. Once a capped collection reaches its max size, MongoDB will drop old documents from the database to make way for the new documents. You must specify a `size` argument for all capped collections.
- **max** – The maximum number of documents to preserve in the capped collection. This limit is subject to the overall size of the capped collection. If a capped collection reaches its maximum size before it contains the maximum number of documents, the database will remove old documents. Thus, if you use this option, ensure that the total size for the capped collection is sufficient to contain the max.

The `db.createCollection()` provides a wrapper function that provides access to this functionality.

Note: This 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.

`convertToCapped`

The `convertToCapped` (page 35) 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 35) takes an existing collection (`<collection>`) and transforms it into a capped collection with a maximum size in bytes, specified to the `size` argument (`<capped size>`).

During the conversion process, the `convertToCapped` (page 35) command exhibit the following behavior:

- MongoDB transverses 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 36) command creates the capped collection and imports the data.
  - MongoDB drops the original collection.
  - `renameCollection` (page 37) renames the new capped collection to the name of the original collection.

Note: MongoDB does not support the `convertToCapped` (page 35) command in a sharded cluster.

> **Warning:** The `convertToCapped` (page 35) will not recreate indexes from the original collection on the new collection. If you need indexes on this collection you will need to create these indexes after the conversion is complete.

See Also:

`create` (page 35)

> **Warning:** This command obtains a global write lock and will block other operations until it has completed.

cloneCollectionAsCapped

The `cloneCollectionAsCapped` (page 36) 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 35) command.

During the cloning, the `cloneCollectionAsCapped` (page 36) command exhibit the following behavior:

- MongoDB will transverse the documents in the original collection in *natural order* as they're loaded.
- If the `capped size` specified for the new collection is smaller than the size of the original uncapped collection, then MongoDB will begin overwriting earlier documents in insertion order, which is *first in, first out* (e.g "FIFO").

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.

collStats

The collStats (page 36) command returns a variety of storage statistics for a given collection. Use the following syntax:

```
{ collStats: "database.collection" , scale : 1024 }
```

Specify a namespace `database.collection` and use the `scale` argument to scale the output. The above example will display values in kilobytes.

Examine the following example output, which uses the `db.collection.stats()` helper in the `mongo` shell.

```
> db.users.stats()
{
        "ns" : "app.users",                // namespace
        "count" : 9,                       // number of documents
        "size" : 432,                      // collection size in bytes
        "avgObjSize" : 48,                 // average object size in bytes
        "storageSize" : 3840,              // (pre)allocated space for the collection
        "numExtents" : 1,                  // number of extents (contiguously allocated chunks of c
        "nindexes" : 2,                    // number of indexes
        "lastExtentSize" : 3840,           // size of the most recently created extent
        "paddingFactor" : 1,               // padding can speed up updates if documents grow
        "flags" : 1,
        "totalIndexSize" : 16384,          // total index size in bytes
        "indexSizes" : {                   // size of specific indexes in bytes
                "_id_" : 8192,
                "username" : 8192
        },
        "ok" : 1
}
```

Note: The scale factor rounds values to whole numbers. This can produce unpredictable and unexpected results in some situations.

See Also:

"*Collection Statistics Reference* (page 185)."

renameCollection

The renameCollection (page 37) command is an administrative command that changes the name of an existing collection. You specify collections to renameCollection (page 37) in the form of a complete *namespace*, which includes the database name. To rename a collection, issue the renameCollection (page 37) command against the *admin database* in the form:

```
{ renameCollection: <source-namespace>, to: <target-namespace>[, dropTarget: <boolean> ] }
```

The `dropTarget` argument is optional.

If you specify a collection to the `to` argument in a different database, the renameCollection (page 37) command will copy the collection to the new database and then drop the source collection.

Parameters

- **source-namespace** – Specifies the complete namespace of the collection to rename.
- **to** (*string*) – Specifies the new namespace of the collection.

- **dropTarget** (*boolean*) – Optional. If `true`, `mongod` will drop the `target` of `renameCollection` (page 37) prior to renaming the collection.

Exception

- **10026** – Raised if the `source` namespace does not exist.
- **10027** – Raised if the `target` namespace exists and `dropTarget` is either `false` or unspecified.
- **15967** – Raised if the `target` namespace is an invalid collection name.

You can use `renameCollection` (page 37) in production environments; however:

- `renameCollection` (page 37) will block all database activity for the duration of the operation.
- `renameCollection` (page 37) is incompatible with sharded collections.

> **Warning:** `renameCollection` (page 37) will fail if *target* is the name of an existing collection and you do not specify `dropTarget: true`.
> If the `renameCollection` (page 37) operation does not complete the `target` collection and indexes will not be usable and will require manual intervention to clean up.

The shell helper `db.collection.renameCollection()` provides a simpler interface to using this command within a database. The following is equivalent to the previous example:

```
db.source-namespace.renameCollection( "target" )
```

> **Warning:** You cannot use `renameCollection` (page 37) with sharded collections.

> **Warning:** This command obtains a global write lock and will block other operations until it has completed.

collMod

New in version 2.2. `collMod` (page 38) makes it possible to add flags to a collection to modify the behavior of MongoDB. In the current release the only available flag is `usePowerOf2Sizes` (page 38). 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.

usePowerOf2Sizes

The `usePowerOf2Sizes` (page 38) flag changes the method that MongoDB uses to allocate space on disk for documents in this collection. By setting `usePowerOf2Sizes` (page 38), you ensure that MongoDB will allocate space for documents in sizes that are powers of 2 (e.g. 4, 8, 16, 32, 64, 128, 256, 512...8388608). With this option MongoDB will be able to more effectively reuse space.

`usePowerOf2Sizes` (page 38) is useful for collections where you will be inserting and deleting large numbers of documents to ensure that MongoDB will effectively use space on disk.

Example

To enable `usePowerOf2Sizes` (page 38) on the collection named `sensor_readings`, use the following operation:

```
db.runCommand({collMod: "sensor_readings", usePowerOf2Sizes:true })
```

To disable `usePowerOf2Sizes` (page 38) on the collection `products`, use the following operation:

```
db.runCommand( { collMod: "products", "usePowerOf2Sizes": false })
```

Warning: Changed in version 2.2.1: `usePowerOf2Sizes` (page 38) now supports documents larger than 8 megabytes. If you enable `usePowerOf2Sizes` (page 38) you **must** use at least version 2.2.1. `usePowerOf2Sizes` (page 38) only affects subsequent allocations cased by document insertion or record relocation as a result of document growth, and *does not* affect existing allocations.

3.1.6 Administration Commands

touch

New in version 2.2. The `touch` (page 39) command loads data from the data storage layer into memory. `touch` (page 39) 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` will ideally be able to perform subsequent operations more efficiently. The `touch` (page 39) command has the following prototypical form:

```
{ touch: [collection], data: [boolean], index: [boolean] }
```

By default, `data` and `index` are false, and `touch` (page 39) 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` shell:

```
db.runCommand({ touch: "records", data: true, index: true })
```

`touch` (page 39) will not block read and write operations on a `mongod`, and can run on *secondary* members of replica sets.

Note: Using `touch` (page 39) to control or tweak what a `mongod` stores in memory may displace other records data in memory and hinder performance. Use with caution in production systems.

---

fsync

The `fsync` (page 39) command forces the `mongod` process to flush all pending writes to the storage layer. `mongod` is always writing data to the storage layer as applications write more data to the database. MongoDB guarantees that it will write all data to disk within the `syncdelay` interval, which is 60 seconds by default.

```
{ fsync: 1 }
```

The `fsync` (page 39) operation is synchronous by default, to run `fsync` (page 39) asynchronously, use the following form:

```
{ fsync: 1, async: true }
```

The connection will return immediately. You can check the output of `db.currentOp()` for the status of the `fsync` (page 39) operation.

The primary use of `fsync` (page 39) is to lock the database during backup operations. This will flush all data to the data storage layer and block all write operations until you unlock the database. Consider the following command form:

```
{ fsync: 1, lock: true }
```

Note: You may continue to perform read operations on a database that has a `fsync` (page 39) lock. However, following the first write operation all subsequent read operations wait until you unlock the database.

To check on the current state of the fsync lock, use `db.currentOp()`. Use the following JavaScript function in the shell to test if the database is currently locked:

```
serverIsLocked = function () {
                     var co = db.currentOp();
                     if (co && co.fsyncLock) {
                         return true;
                     }
                     return false;
                 }
```

After loading this function into your `mongo` shell session you can call it as follows:

```
serverIsLocked()
```

This function will return `true` if the database is currently locked and `false` if the database is not locked. To unlock the database, make a request for an unlock using the following command:

```
db.getSiblingDB("admin").$cmd.sys.unlock.findOne();
```

New in version 1.9.0: The `db.fsyncLock()` and `db.fsyncUnlock()` helpers in the shell. In the `mongo` shell, you may use the `db.fsyncLock()` and `db.fsyncUnlock()` wrappers for the `fsync` (page 39) lock and unlock process:

```
db.fsyncLock();
db.fsyncUnlock();
```

Note: `fsync` (page 39) lock is only possible on individual shards of a sharded cluster, not on the entire sharded cluster. To backup an entire sharded cluster, please read *considerations for backing up sharded clusters*.

If your `mongod` has *journaling* enabled, consider using *another method* to back up your database.

Note: The database cannot be locked with `db.fsyncLock()` while profiling is enabled. You must disable profiling before locking the database with `db.fsyncLock()`. Disable profiling using `db.setProfilingLevel()` as follows in the `mongo` shell:

```
db.setProfilingLevel(0)
```

dropDatabase

The `dropDatabase` (page 40) command drops a database, deleting the associated data files. `dropDatabase` (page 40) operates on the current database.

In the shell issue the `use <database>` command, replacing `<database>` with the name of the database you wish to delete. Then use the following command form:

```
{ dropDatabase: 1 }
```

The mongo shell also provides the following equivalent helper method:

```
db.dropDatabase();
```

> **Warning:** This command obtains a global write lock and will block other operations until it has completed.

dropIndexes

The dropIndexes (page 41) 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.

clone

The clone (page 41) command clone a database from a remote MongoDB instance to the current host. clone (page 41) copies the database on the remote instance with the same name as the current database. The command takes the following form:

```
{ clone: "db1.example.net:27017" }
```

Replace db1.example.net:27017 above with the resolvable hostname for the MongoDB instance you wish to copy from. Note the following behaviors:

- clone (page 41) can run against a *slave* or a non-*primary* member of a *replica set*.
- clone (page 41) 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 41) on the **destination server**.
- The destination server is not locked for the duration of the clone (page 41) operation. This means that clone (page 41) will occasionally yield to allow other operations to complete.

See copydb (page 45) for similar functionality.

> **Warning:** This command obtains an intermittent write-lock on the destination server, that can block other operations until it completes.

compact

New in version 2.0. The compact (page 41) command rewrites and defragments a single collection. Additionally, the command drops all indexes at the beginning of compaction and rebuilds the indexes at the end.

`compact` (page 41) is conceptually similar to `repairDatabase` (page 43), but works on a single collection rather than an entire database.

The command has the following syntax:

```
{ compact: <collection name> }
```

You may also specify the following options:

Parameters

- **force** – Changed in version 2.2: `compact` (page 41) blocks activities only for the database it is compacting. The `force` specifies whether the `compact` (page 41) command can run on the primary node in a *replica set*. Set to `true` to run the `compact` (page 41) command on the primary node in a *replica set*. Otherwise, the `compact` (page 41) command returns an error when invoked on a *replica set* primary because the command blocks all other activity.
- **paddingFactor** – New in version v2.2. *Default:* `1.0`

  Minimum: `1.0` (no padding.)

  Maximum: `4.0`

  The `paddingFactor` describes the *record size* allocated for each document as a factor of the document size. 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 41) 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 } )
  ```

- **paddingBytes** – New in version 2.2. The `paddingBytes` sets the padding as an absolute number of bytes. Specifying `paddingBytes` can be useful if your documents start small but then increase in size significantly. For example,if your documents are initially 40 bytes long and you grow them by 1KB, using `paddingBytes: 1024` might be reasonable since using `paddingFactor: 4.0` would specify a record size of 160 bytes (`4.0` times the initial document size), which would only provide a padding of 120 bytes (i.e. record size of 160 bytes minus the document size).

  With the following command, you can use the `paddingBytes` option of the `compact` (page 41) command to set the padding size to `100` bytes on the collection named by `<collection>`:

  ```
  db.runCommand ( { compact: '<collection>', paddingBytes: 100 } )
  ```

> **Warning:** Always have an up-to-date backup before performing server maintenance such as the `compact` (page 41) operation.

Note the following behaviors:

- `compact` (page 41) blocks all other activity. In MongoDB 2.2, `compact` (page 41) blocks activities only for its database. You may view the intermediate progress either by viewing the `mongod` log file, or by running the `db.currentOp()` in another shell instance.
- `compact` (page 41) removes any *padding factor* in the collection when issued without either the `paddingFactor` option or the `paddingBytes` option. This may impact performance if the documents grow regularly. However, `compact` (page 41) retains existing paddingFactor statistics for the collection that MongoDB will use to calculate the padding factor for future inserts.
- `compact` (page 41) generally uses less disk space than `repairDatabase` (page 43) and is faster. However,the `compact` (page 41) command is still slow and does block other database use. Only use `compact` (page 41) during scheduled maintenance periods.
- If you terminate the operation with the `db.killOp()` method or restart the server before it has finished:
  - If you have journaling enabled, the data remains consistent and usable, regardless of the state of the `compact` (page 41) operation. You may have to manually rebuild the indexes.
  - If you do not have journaling enabled and the `mongod` or `compact` (page 41) terminates during the operation, it's impossible to guarantee that the data is in a consistent 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.
- `compact` (page 41) may increase the total size and number of our 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.
- `compact` (page 41) requires a small amount of additional disk space while running but unlike `repairDatabase` (page 43) it does *not* free space on the file system.
- You may also wish to run the `collStats` (page 36) command before and after compaction to see how the storage space changes for the collection.
- `compact` (page 41) commands do not replicate to secondaries in a *replica set*:
  - Compact each member separately.
  - Ideally, compaction runs on a secondary. See option `force:true` above for information regarding compacting the primary.
  - If you run `compact` (page 41) on a secondary, the secondary will enter a "recovering" state to prevent clients from sending read operations during compaction. Once the compaction finishes the secondary will automatically return to secondary state.

  You may refer to the "partial script for automating step down and compaction") for an example.
- `compact` (page 41) is a command issued to a `mongod`. In a sharded environment, run `compact` (page 41) on each shard separately as a maintenance operation.
- It is not possible to compact *capped collections* because they don't have padding, and documents cannot grow in these collections. However, the documents of a *capped collections* are not subject to fragmentation.

See Also:

`repairDatabase` (page 43)

`repairDatabase`

> **Warning:** In general, if you have an intact copy of your data, such as would exist on a very recent backup or an intact member of a *replica set*, **do not** use `repairDatabase` (page 43) or related options like `db.repairDatabase()` in the `mongo` shell or *`mongod --repair`* (page 211). Restore from an intact copy of your data.

Note: When using *journaling*, there is almost never any need to run `repairDatabase` (page 43). In the event of an unclean shutdown, the server will be able restore the data files to a pristine state automatically.

The `repairDatabase` (page 43) command checks and repairs errors and inconsistencies with the data storage. The command is analogous to a `fsck` command for file systems.

If your `mongod` instance is not running with journaling the system experiences an unexpected system restart or crash, and you have *no* other intact replica set members with this data, you should run the `repairDatabase` (page 43) command to ensure that there are no errors in the data storage.

As a side effect, the `repairDatabase` (page 43) command will compact the database, as the `compact` (page 41) command, and also reduces the total size of the data files on disk. The `repairDatabase` (page 43) command will also recreate all indexes in the database.

Use the following syntax:

```
{ repairDatabase: 1 }
```

Be aware that this command can take a long time to run if your database is large. In addition, it requires a quantity of free disk space equal to the size of your database. If you lack sufficient free space on the same volume, you can mount a separate volume and use that for the repair. In this case, you must run the command line and use the *--repairpath* (page 211) switch to specify the folder in which to store the temporary repair files.

Warning: This command obtains a global write lock and will block other operations until it has completed.

This command is accessible via a number of different avenues. You may:

- Use the shell to run the above command, as above.
- Use the `db.repairDatabase()` in the `mongo` shell.
- Run `mongod` directly from your system's shell. Make sure that `mongod` isn't already running, and that you issue this command 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
  ```

  Note: This command 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. If you must repair a secondary or slave node, first restart the node as a standalone mongod by omitting the *--replSet* (page 212) or *--slave* (page 213) options, as necessary.

shutdown

The `shutdown` (page 44) command cleans up all database resources and then terminates the process. You must issue the `shutdown` (page 44) command against the *admin database* in the form:

```
{ shutdown: 1 }
```

Note: Run the `shutdown` (page 44) against the *admin database*. When using `shutdown` (page 44), 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 44) 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` shell helper syntax looks like this:

```
db.shutdownServer({timeoutSecs: 60});
```

copydb

The copydb (page 45) command copies a database from a remote host to the current host. The command has the following syntax:

```
{ copydb: 1:
  fromhost: <hostname>,
  fromdb: <db>,
  todb: <db>,
  slaveOk: <bool>,
  username: <username>,
  password: <password>,
  nonce: <nonce>,
  key: <key> }
```

All of the following arguments are optional:

- slaveOk
- username
- password
- nonce
- key

You can omit the `fromhost` argument, to copy one database to another database within a single MongoDB instance.

You must run this command on the destination, or the `todb` server.

Be aware of the following behaviors:

- copydb (page 45) can run against a *slave* or a non-*primary* member of a *replica set*. In this case, you must set the `slaveOk` option to `true`.
- copydb (page 45) does not snapshot the database. If the state of the database changes at any point during the operation, the resulting database may be inconsistent.
- You must run copydb (page 45) on the **destination server**.
- The destination server is not locked for the duration of the copydb (page 45) operation. This means that copydb (page 45) will occasionally yield to allow other operations to complete.
- If the remote server has authentication enabled, then you must include a username and password. You must also include a nonce and a key. The nonce is a one-time password that you request from the remote server using the copydbgetnonce (page 57) command. The `key` is a hash generated as follows:

  ```
  hex_md5(nonce + username + hex_md5(username + ":mongo:" + pass))
  ```

  If you need to copy a database and authenticate, it's easiest to use the shell helper:

  ```
  db.copyDatabase(<remote_db_name>, <local_db_name>, <from_host_name>, <username>, <password>
  ```

logout

The logout (page 45) command terminates the current authenticated session:

```
{ logout: 1 }
```

Note: If you're not logged in and using authentication, this command will have no effect.

logRotate

The logRotate (page 46) 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 46) command against the *admin database* in the form:

{ logRotate: 1 }

Note: Your mongod instance needs to be running with the *--logpath [file]* (page 208) option.

You may also rotate the logs by sending a SIGUSR1 signal to the mongod process. If your mongod has a process ID of 2200, here's how to send the signal on Linux:

```
kill -SIGUSR1 2200
```

logRotate (page 46) 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 46) creates a new log file with the same name as originally specified by the logpath setting to mongod or mongos (page 256).

Note: New in version 2.0.3: The logRotate (page 46) command is available to mongod instances running on Windows systems with MongoDB release 2.0.3 and higher.

setParameter

setParameter (page 46) is an administrative command for modifying options normally set on the command line. You must issue the setParameter (page 46) command against the *admin database* in the form:

```
{ setParameter: 1, <option>: <value> }
```

Replace the <option> with one of the following options supported by this command:

Options

- **journalCommitInterval** (*integer*) – Specify an integer between 1 and 500 signifying the number of milliseconds (ms) between journal commits.

  Consider the following example which sets the journalCommitInterval to 200 ms:

  ```
  use admin
  db.runCommand( { setParameter: 1, journalCommitInterval: 200 } )
  ```

  See Also:

  journalCommitInterval.

- **logLevel** (*integer*) – 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` to `2`:

  ```
  use admin
  db.runCommand( { setParameter: 1, logLevel: 2 } )
  ```

  See Also:

  `verbose`.

- **notablescan** (*boolean*) – Specify whether queries must use indexes. If `true`, queries that perform a table scan instead of using an index will fail.

  Consider the following example which sets the `notablescan` to true:

  ```
  use admin
  db.runCommand( { setParameter: 1, notablescan: true } )
  ```

  See Also:

  `notablescan`.

- **traceExceptions** (*boolean*) – New in version 2.1. Configures `mongod` log full stack traces on assertions or errors. If `true`, `mongod` will log full stack traces on assertions or errors.

  Consider the following example which sets the `traceExceptions` to true:

  ```
  use admin
  db.runCommand( { setParameter: 1, traceExceptions: true } )
  ```

  See Also:

  `traceExceptions`.

- **quiet** (*boolean*) – Sets quiet logging mode. If `true`, `mongod` will go into a quiet logging mode which will not log the following events/activities:

  – connection events;

  – the `drop` (page 34) command, the `dropIndexes` (page 41) command, the `diagLogging` (page 56) command, the `validate` (page 49) command, and the `clean` (page 57) command; and

  – replication synchronization activities.

  Consider the following example which sets the `quiet` to true:

  ```
  use admin
  db.runCommand( { setParameter: 1, quiet: true } )
  ```

  See Also:

  `quiet`.

- **syncdelay** (*integer*) – Specify the interval in seconds between *fsyncs* (i.e., flushes of memory to disk). By default, `mongod` will flush memory to disk every 60 seconds. Do not change this value unless you see a background flush average greater than 60 seconds.

  Consider the following example which sets the `syncdelay` to `60` seconds:

```
use admin
db.runCommand( { setParameter: 1, syncdelay: 60 } )
```

See Also:

`syncdelay`.

getParameter

`getParameter` (page 48) 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 46) for more about these parameters.

3.1.7 Diagnostic Commands

buildInfo

The `buildInfo` (page 48) command is an administrative command which returns a build summary for the current `mongod`.

```
{ buildInfo: 1 }
```

```
The information provided includes the following:
```

- The version of MongoDB currently running.
- The information about the system that built the "`mongod`" binary, including a timestamp for the build.
- The architecture of the binary (i.e. 64 or 32 bits.)
- The maximum allowable *BSON* object size in bytes (in the field `maxBsonObjectSize`.)

dbStats

The `dbStats` (page 48) command returns storage statistics for a given database. The command takes the following syntax:

```
{ dbStats: 1, scale: 1 }
```

The value of the argument (e.g. `1` above) to `dbStats` does 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.

The time required to run the command depends on the total size of the database. Because the command has to touch all data files, the command may take several seconds to run.

In the `mongo` shell, the `db.stats()` function provides a wrapper around this functionality. See the "*Database Statistics Reference* (page 183)" document for an overview of this output.

connPoolStats

> **Note:** connPoolStats (page 48) only returns meaningful results for mongos (page 256) instances and for mongod instances in sharded clusters.

The command connPoolStats (page 48) 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. The command takes the following form:

```
{ connPoolStats: 1 }
```

The value of the argument (i.e. `1`) does not affect the output of the command. See *Connection Pool Statistics Reference* (page 189) for full documentation of the connPoolStats (page 48) output.

getCmdLineOpts

The getCmdLineOpts (page 49) command returns a document containing command line options used to start the given mongod:

```
{ 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. 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 49):

```
{
        "argv" : [
               "/usr/bin/mongod",
               "--config",
               "/etc/mongodb.conf",
               "--fork"
        ],
        "parsed" : {
               "bind_ip" : "127.0.0.1",
               "config" : "/etc/mongodb/mongodb.conf",
               "dbpath" : "/srv/mongodb",
               "fork" : true,
               "logappend" : "true",
               "logpath" : "/var/log/mongodb/mongod.log",
               "quiet" : "true"
        },
        "ok" : 1
}
```

http://docs.mongodb.org/manual/administration/import-export/

validate

The `validate` command checks the contents of a namespace by scanning a collection's data and indexes for correctness. The command can be slow, particularly on larger data sets:

```
{ validate: "users" }
```

This command will validate the contents of the collection named `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` shell also provides a wrapper:

```
db.collection.validate();
```

Use one of the following forms to perform the full collection validation:

```
db.collection.validate(true)
db.runCommand( { validate: "collection", full: true } )
```

Warning: This command is resource intensive and may have an impact on the performance of your MongoDB instance.

`top`

The `top` (page 50) command is an administrative command which returns raw usage of each database, and 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

You must issue the `top` (page 50) command against the *admin database* in the form:

```
{ top: 1 }
```

`getLastError`

The `getLastError` (page 50) command returns the error status of the last operation on the *current connection*. By default MongoDB does not provide a response to confirm the success or failure of a write operation, clients typically use `getLastError` (page 50) in combination with write operations to ensure that the write succeeds.

Consider the following prototype form.

```
{ getLastError: 1 }
```

The following options are available:

Parameters

- **j** (*boolean*) – If `true`, wait for the next journal commit before returning, rather than a full disk flush. If `mongod` does not have journaling enabled, this option has no effect.
- **w** – When running with replication, this is the number of servers to replicate to before returning. A `w` value of 1 indicates the primary only. A `w` value of 2 includes the primary and at least one secondary, etc. In place of a number, you may also set `w` to `majority` to indicate that the command should wait until the latest write propagates to a majority of replica set members. If using `w`, you should also use `wtimeout`. Specifying a value

for `w` without also providing a `wtimeout` may cause `getLastError` (page 50) to block indefinitely.

- **fsync** (*boolean*) – If `true`, wait for `mongod` to write this data to disk before returning. Defaults to false. In most cases, use the `j` option to ensure durability and consistency of the data set.
- **wtimeout** (*integer*) – 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 50) command will return with an error status.

See Also:

Write Concern, *Replica Set Write Concern*, and `db.getLastError()`.

getLog

The `getLog` (page 51) command returns a document with a `log` array that contains recent messages from the `mongod` process log. The `getLog` (page 51) 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` is part of a *replica set*, `getLog` (page 51) 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` 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` shell connected to a replica set:

```
db.adminCommand({getLog: "*" })
{ "names" : [ "global", "rs", "startupWarnings" ], "ok" : 1 }
```

`getLog` (page 51) returns events from a RAM cache of the `mongod` events and *does not* read log data from the log :file.

listDatabases

The `listDatabases` (page 51) command provides a list of existing databases along with basic statistics about them:

```
{ listDatabases: 1 }
```

The value (e.g. `1`) does not effect the output of the command. `listDatabases` (page 51) returns a document for each database Each document contains a `name` field with the database name, a `sizeOnDisk` field with the total size of the database file on disk in bytes, and an `empty` field specifying whether the database has any data.

cursorInfo

The `cursorInfo` (page 51) command returns information about current cursor allotment and use. Use the following form:

```
{ cursorInfo: 1 }
```

The value (e.g. `1` above,) does not effect the output of the command.

`cursorInfo` (page 51) returns the total number of open cursors (`totalOpen`,) the size of client cursors in current use (`clientCursors_size`,) and the number of timed out cursors since the last server restart (`timedOut`.)

isMaster

The `isMaster` command provides a basic overview of the current replication configuration. MongoDB *drivers* and *clients* use this command to determine what kind of member they're connected to and to discover additional members of a *replica set*. The `db.isMaster()` method provides a wrapper around this database command.

The command takes the following form:

```
{ isMaster: 1 }
```

This command returns a *document* containing the following fields:

`isMaster.`**setname**

The name of the current replica set, if applicable.

`isMaster.`**ismaster**

A boolean value that reports when this node is writable. If `true`, then the current node is either a *primary* node in a *replica set*, a *master* node in a master-slave configuration, of a standalone `mongod`.

`isMaster.`**secondary**

A boolean value that, when `true`, indicates that the current node is a *secondary* member of a *replica set*.

`isMaster.`**hosts**

An array of strings in the format of "`[hostname]:[port]`" listing all nodes in the *replica set* that are not "*hidden*".

`isMaster.`**primary**

The `[hostname]:[port]` for the current *replica set primary*, if applicable.

`isMaster.`**me**

The `[hostname]:[port]` of the node responding to this command.

`isMaster.`**maxBsonObjectSize**

The maximum permitted size of a *BSON* object in bytes for this `mongod` process. If not provided, clients should assume a max size of "4 * 1024 * 1024."

`isMaster.`**localTime**

New in version 2.1.1. Returns the local server time in UTC. This value is a *ISOdate*. You can use the `toString()` JavaScript method to convert this value to a local date string, as in the following example:

```
db.isMaster().localTime.toString();
```

ping

The `ping` (page 52) 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.

serverStatus

The `serverStatus` (page 52) command returns a document that provides an overview of the database process's state. Most monitoring applications run this command at a regular interval to collection statistics about the instance:

```
{ serverStatus: 1 }
```

The value (i.e. `1` above), does not affect the operation of the command.

See Also:

`db.serverStatus()` and "*Server Status Reference* (page 167)"

resetError

The `resetError` (page 53) command resets the last error status.

See Also:

`db.resetError()`

getPrevError

The `getPrevError` (page 53) command returns the errors since the last `resetError` (page 53) command.

See Also:

`db.getPrevError()`

forceerror

The `forceerror` (page 53) command is for testing purposes only. Use `forceerror` (page 53) to force a user assertion exception. This command always returns an `ok` value of 0.

profile

Use the `profile` 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, which might information security implications for your deployment. Consider the following prototype syntax:

```
{ profile: <level> }
```

The following profiling levels are available:

Level	Setting
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` writes the output of the database profiler to the `system.profile` collection.

`mongod` records a record of queries that take longer than the `slowms` to the log even when the database profiler is not active.

See Also:

Additional documentation regarding database profiling *Database Profiling*.

See Also:

"`db.getProfilingStatus()`" and "`db.setProfilingLevel()`" provide wrappers around this functionality in the `mongo` shell.

Note: The database cannot be locked with `db.fsyncLock()` while profiling is enabled. You must disable profiling before locking the database with `db.fsyncLock()`. Disable profiling using `db.setProfilingLevel()` as follows in the `mongo` shell:

```
db.setProfilingLevel(0)
```

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 in place while the enabling and disabling the profiler, which is typically a short operation.

listCommands

The listCommands (page 54) command generates a list of all database commands implemented for the current mongod instance.

3.1.8 Other Commands

reIndex

The reIndex (page 54) command rebuilds all indexes for a specified collection. Use the following syntax:

```
{ reIndex: "collection" }
```

Normally, MongoDB compacts indexes during routine updates. For most users, the reIndex (page 54) 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.

Note that the reIndex (page 54) command will block the server against writes and may take a long time for large collections.

Call reIndex (page 54) using the following form:

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

Warning: This command obtains a write lock on the affected database and will block other operations until it has completed.

filemd5

The filemd5 (page 54) command returns the *md5* hashes for a single files 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" }
```

3.2 mongos Commands

flushRouterConfig

flushRouterConfig (page 54) clears the current cluster information cached by a mongos (page 256) instance and reloads all *sharded cluster* metadata from the *config database*.

This forces an update when the configuration database holds data that is newer that the data cached in the mongos (page 256) process.

> **Warning:** Do not modify the config data, except as explicitly documented. A config database cannot typically tolerate manual manipulation.

`flushRouterConfig` (page 54) is an administrative command that is only available for `mongos` (page 256) instances. New in version 1.8.2.

isdbgrid

This command verifies that a process is a `mongos` (page 256).

If you issue the `isdbgrid` (page 55) command when connected to a `mongos` (page 256), 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 55) command when connected to a `mongod`, MongoDB returns an error document. The `isdbgrid` (page 55) command is not available to `mongod`. The error document, however, also includes a line that reads `"isdbgrid" : 1`, just as in the document returned for a `mongos` (page 256). 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` command to determine connection to a `mongos` (page 256). When connected to a `mongos` (page 256), the `isMaster` command returns a document that contains the string `isdbgrid` in the `msg` field.

movePrimary

In a *sharded cluster*, this command reassigns the database's *primary shard*, which holds all un-sharded collections in the database. `movePrimary` (page 55) is an administrative command that is only available for `mongos` (page 256) instances. Only use `movePrimary` (page 55) when removing a shard from a sharded cluster.

> **Important:** Only use `movePrimary` (page 55) when:
> - the database does not contain any collections with data, *or*
> - you have drained all sharded collections using the `removeShard` (page 13) command.
>
> See `http://docs.mongodb.org/manual/tutorial/remove-shards-from-cluster` for a complete procedure.

`movePrimary` (page 55) changes the primary shard for this database in the cluster metadata, and 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 13) command.

split

The `split` (page 55) command creates new *chunks* in a *sharded* environment. While splitting is typically managed automatically by the `mongos` (page 256) instances, this command makes it possible for administrators to manually create splits.

In normal operation there is no need to manually split chunks

The *balancer* and other sharding infrastructure will automatically create chunks in the course of normal operations. See `http://docs.mongodb.org/manual/core/sharding-internals` for more information.

Consider the following example:

```
db.runCommand( { split : "test.people" , find : { _id : 99 } } )
```

This command inserts a new split in the collection named `people` in the `test` database. This will split the chunk that contains the document that matches the query `{ _id : 99 }` in half. If the document specified by the query does not (yet) exist, the `split` (page 55) will divide the chunk where that document *would* exist.

The split divides the chunk in half, and does *not* split the chunk using the identified document as the middle. To define an arbitrary split point, use the following form:

```
db.runCommand( { split : "test.people" , middle : { _id : 99 } } )
```

This form is typically used when *pre-splitting* data in a collection.

`split` (page 55) is an administrative command that is only available for `mongos` (page 256) instances.

3.3 Internal Use Commands

availableQueryOptions
`availableQueryOptions` (page 56) is an internal command that is only available on `mongos` (page 256) instances.

closeAllDatabases
`closeAllDatabases` (page 56) is an internal command that invalidates all cursors and closes the open database files. The next operation that uses the database will reopen the file.

Warning: This command obtains a global write lock and will block other operations until it has completed.

netstat
`netstat` (page 56) is an internal command that is only available on `mongos` (page 256) instances.

setShardVersion
`setShardVersion` (page 56) is an internal command that supports sharding functionality.

getShardVersion
`getShardVersion` is an internal command that supports sharding functionality.

unsetSharding
`unsetSharding` (page 56) is an internal command that supports sharding functionality.

whatsmyuri
`whatsmyuri` (page 56) is an internal command.

features
`features` (page 56) is an internal command that returns the build-level feature settings.

driverOIDTest
`driverOIDTest` (page 56) is an internal command.

diagLogging

diagLogging (page 56) is an internal command.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

copydbgetnonce

Client libraries use copydbgetnonce (page 57) to get a one-time password for use with the copydb (page 45) 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.

dbHash

dbHash (page 57) is an internal command.

medianKey

medianKey (page 57) is an internal command.

geoWalk

geoWalk (page 57) is an internal command.

sleep

sleep (page 57) is an internal command for testing purposes. The sleep (page 57) command forces the database to block all operations. It takes the following options:

Parameters

- **w** (*boolean*) – If true, obtain a global write lock. Otherwise obtains a read lock.
- **secs** (*integer*) – Specifies the number of seconds to sleep.

```
{ sleep: { w: true, secs: <seconds> } }
```

The above command places the mongod instance in a "write-lock" state for a specified (i.e. <seconds>) number of seconds. Without arguments, sleep (page 57), causes a "read lock" for 100 seconds.

> **Warning:** sleep (page 57) claims the lock specified in the w argument and blocks *all* operations on the mongod instance for the specified amount of time.

getnonce

Client libraries use getnonce (page 57) to generate a one-time password for authentication.

getoptime

getoptime (page 57) is an internal command.

godinsert

godinsert (page 57) 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.

clean

clean (page 57) is an internal command.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

applyOps

Parameters

- **operations** (*array*) – an array of operations to perform.
- **preCondition** (*array*) – Optional. Defines one or more conditions that the destination must meet applying the entries from the `<operations>` array. Use `ns` to specify a *namespace*, `q` to specify a *query* and `res` to specify the result that the query should match. You may specify zero, one, or many `preCondition` documents.

`applyOps` (page 58) provides a way to apply entries from an *oplog* created by *replica set* members and *master* instances in a master/*slave* deployment. `applyOps` (page 58) is primarily an internal command to support sharding functionality, and has the following prototype form:

```
db.runCommand( { applyOps: [ <operations> ], preCondition: [ { ns: <namespace>, q: <query>, res:
```

`applyOps` (page 58) applies oplog entries from the `<operations>` array, to the `mongod` instance. The `preCondition` array provides the ability to specify conditions that must be true in order to apply the oplog entry.

You can specify as many `preCondition` sets as needed. If you specify the `ns` option, `applyOps` (page 58) will only apply oplog entries for the *collection* described by that namespace. You may also specify a query in the `q` field with a corresponding expected result in the `res` field that must match in order to apply the oplog entry.

> **Warning:** This command obtains a global write lock and will block other operations until it has completed.

replSetElect

`replSetElect` (page 58) is an internal command that support replica set functionality.

replSetGetRBID

`replSetGetRBID` (page 58) is an internal command that supports replica set functionality.

replSetHeartbeat

`replSetHeartbeat` (page 58) is an internal command that supports replica set functionality.

replSetFresh

`replSetFresh` (page 58) is an internal command that supports replica set functionality.

replSetMaintenance

The `replSetMaintenance` admin command enables or disables the maintenance mode for a *secondary* member of a *replica set*.

The command has the following prototype form:

```
{ replSetMaintenance: <boolean> }
```

Consider the following behavior when running the `replSetMaintenance` command:

- You cannot run the command on the Primary.
- You must run the command against the `admin` database.
- When enabled `replSetMaintenance: 1`, the member enters the `RECOVERING` state. While the secondary is `RECOVERING`:
  - The member is not accessible for read operations.

–The member continues to sync its *oplog* from the Primary.

replSetTest

replSetTest (page 59) is internal diagnostic command used for regression tests that supports replica set functionality.

writeBacksQueued

writeBacksQueued (page 179) is an internal command that returns a document reporting there are operations in the write back queue for the given mongos (page 256) and information about the queues.

hasOpsQueued

Boolean.

hasOpsQueued (page 59) is `true` if there are `write Back` operations queued.

totalOpsQueued

Integer.

totalOpsQueued (page 59) reflects the number of operations queued.

queues

Document.

queues (page 59) holds a sub-document where the fields are all write back queues. These field hold a document with two fields that reports on the state of the queue. The fields in these documents are:

queues.**n**

queues.n (page 59) reflects the size, by number of items, in the queues.

queues.**minutesSinceLastCall**

The number of minutes since the last time the mongos (page 256) touched this queue.

The command document has the following prototype form:

```
{writeBacksQueued: 1}
```

To call writeBacksQueued (page 179) from the `mongo` shell, use the following `db.runCommand()` 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
}
```

connPoolSync

connPoolSync (page 59) is an internal command.

checkShardingIndex

checkShardingIndex (page 59) is an internal command that supports the sharding functionality.

getShardMap

getShardMap (page 60) is an internal command that supports the sharding functionality.

splitChunk

splitChunk (page 60) is an internal command. Use the sh.splitFind() and sh.splitAt() functions in the mongo shell to access this functionality.

moveChunk

moveChunk (page 60) is an internal administrative command that moves *chunks* between *shards*. You must issue the moveChunk (page 60) command against the *admin database* in the form:

```
db.runCommand( { moveChunk : <namespace> ,
                 find : <query> ,
                 to : <destination>,
                 <options> } )
```

Parameters

- **moveChunk** – The name of the *collection* where the *chunk* exists. Specify the collection's full namespace, including the database name.
- **find** – A document including the *shard key*.
- **to** – The identifier of the shard, that you want to migrate the chunk to.
- **_secondaryThrottle** – Optional. Set to false by default. Provides *write concern* support for chunk migrations.

If you set _secondaryThrottle to true, during chunk migrations when a *shard* hosted by a *replica set*, the mongod will wait until the *secondary* members replicate the migration operations continuing to migrate chunk data. You may also configure _secondaryThrottle in the balancer configuration.

Use the sh.moveChunk() helper in the mongo shell to migrate chunks manually.

The *chunk migration* section describes how chunks move between shards on MongoDB.

moveChunk (page 60) will return the following if another cursor is using the chunk you are moving:

```
errmsg: "The collection's metadata lock is already taken."
```

These errors usually occur when there are too many open *cursors* accessing the chunk you are migrating. You can either wait until the cursors complete their operation or close the cursors manually.

Note: Only use the moveChunk (page 60) in special circumstances such as preparing your *sharded cluster* for an initial ingestion of data, or a large bulk import operation. See *sharding-administration-create-chunks* for more information.

---

writebacklisten

writebacklisten (page 60) is an internal command.

dataSize

For internal use.

The dataSize (page 184) command returns the size 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 184) depends on the amount of data in the collection.

authenticate

Clients use `authenticate` (page 61) to authenticate a connection. When using the shell, use the command helper as follows:

```
db.authenticate( "username", "password" )
```

handshake

`handshake` (page 61) is an internal command.

mapreduce.**shardedfinish**

Provides internal functionality to support *map-reduce* in *sharded* environments.

See Also:

"`mapReduce` (page 18)"

_isSelf

`_isSelf` (page 61) is an internal command.

_migrateClone

`_migrateClone` (page 61) is an internal command. Do not call directly.

_recvChunkAbort

`_recvChunkAbort` (page 61) is an internal command. Do not call directly.

_recvChunkCommit

`_recvChunkCommit` (page 61) is an internal command. Do not call directly.

_recvChunkStatus

`_recvChunkStatus` (page 61) is an internal command. Do not call directly.

_recvChunkStart

`_recvChunkStart` (page 61) 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.

_skewClockCommand

`_skewClockCommand` (page 61) is an internal command. Do not call directly.

_testDistLockWithSkew

`_testDistLockWithSkew` (page 61) is an internal command. Do not call directly.

_testDistLockWithSyncCluster

`_testDistLockWithSyncCluster` (page 61) is an internal command. Do not call directly.

_transferMods

`_transferMods` (page 61) is an internal command. Do not call directly.

CHAPTER

FOUR

JAVASCRIPT INTERFACE

4.1 Data Manipulation

4.1.1 Query and Update Methods

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

The `find()` method selects documents in a collection and returns a *cursor* to the selected documents.

The `find()` method takes the following parameters.

Parameters

- **query** (*document*) – Optional. Specifies the selection criteria using *query operators* (page 111). Omit the `query` parameter or pass an empty document (e.g. `{}`) to return all documents in the collection.
- **projection** (*document*) – Optional. Controls the fields to return, or the *projection*. The `projection` argument will resemble the following prototype:

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

  The `boolean` can take the following include or exclude values:

  – `1` or `true` to include. The `find()` 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 `projection` cannot contain both include and exclude specifications except for the exclusion of the `_id` field.

Returns A *cursor* to the documents that match the `query` criteria and contain the `projection` fields.

Note: In the `mongo` shell, you can access the returned documents directly without explicitly using the JavaScript cursor handling method. Executing the query directly on the `mongo` shell prompt automatically iterates the cursor to display up to the first 20 documents. Type `it` to continue iteration.

Consider the following examples of the `find()` method:

- To select all documents in a collection, call the `find()` method with no parameters:

  ```
  db.products.find()
  ```

  This operation returns all the documents with all the fields from the collection `products`. By default, in the `mongo` shell, the cursor returns the first batch of 20 matching documents. In the `mongo` shell, iterate through the next batch by typing `it`. Use the appropriate cursor handling mechanism for your specific language driver.

- To select the documents that match a selection criteria, call the `find()` method with the `query` criteria:

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

This operation returns all the documents from the collection `products` where `qty` is greater than `25`, including all fields.

- To select the documents that match a selection criteria and return, or *project* only certain fields into the result set, call the `find()` method with the `query` criteria and the `projection` parameter, as in the following example:

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

This operation returns all the documents from the collection `products` where `qty` is greater than `25`. The documents in the result set only include the `_id`, `item`, and `qty` fields using "inclusion" projection. `find()` always returns the `_id` field, even when not explicitly included:

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

- To select the documents that match a query criteria and exclude a set of fields from the resulting documents, call the `find()` method with the `query` criteria and the `projection` parameter using the `exclude` syntax:

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

The query will return all the documents from the collection `products` where `qty` is greater than `25`. The documents in the result set will contain all fields *except* the `_id` and `qty` fields, as in the following:

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

`db.collection.`**`findOne`**`(`*query*`)`

Parameters

- **query** (*document*) – Optional. A *document* that specifies the *query* using the JSON-like syntax and *query operators* (page 111).

Returns One document that satisfies the query specified as the argument to this method.

Returns only one document that satisfies the specified query. 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 disc. In *capped collections*, natural order is the same as insertion order.

`db.collection.`**`findAndModify`**`()`

The `db.collection.findAndModify()` method atomically 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 `db.collection.findAndModify()` method takes a document parameter with the following sub-document fields:

Fields

- **query** (*document*) – Optional. Specifies the selection criteria for the modification. The `query` field employs the same *query selectors* (page 111) as used in the `db.collection.find()` method. Although the query may match multiple documents, `findAndModify()` will only select one document to modify.

  The `query` field has the following syntax:

```
query: { <query expression> }
```

- **sort** (*document*) – Optional. Determines which document the operation will modify if the query selects multiple documents. `findAndModify()` will modify the first document in the sort order specified by this argument.

  The `sort` field has the following syntax:

  ```
  sort: { field1: value1, field2: value2, ... }
  ```

- **remove** (*boolean*) – Optional if `update` field exists. When `true`, removes the selected document. The default is `false`.

  The `remove` field has the following syntax:

  ```
  remove: <boolean>
  ```

- **update** (*document*) – Optional if `remove` field exists. Performs an update of the selected document. The `update` field employs the same *update operators* (page 127) or `field: value` specifications to modify the selected document.

  ```
  update: { <update expression> }
  ```

- **new** (*boolean*) – Optional. When `true`, returns the modified document rather than the original. The `findAndModify` (page 25) method ignores the `new` option for `remove` operations. The default is `false`.

  ```
  new: <boolean>
  ```

- **fields** (*document*) – Optional. A subset of fields to return.

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

- **upsert** (*boolean*) – Optional. Used in conjunction with the `update` field. When `true`, `findAndModify` (page 25) creates a new document if the `query` returns no documents. The default is `false`. In version 2.2, the `findAndModify` (page 25) command returns `null` when `upsert` is `true`.

  ```
  upsert: <boolean>
  ```

Consider the following example:

```
db.people.findAndModify( {
    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` (page 112) 10.
2. The `sort` orders the results of the query in ascending order.
3. The update `increments` (page 127) the value of the `score` field by 1.
4. The command returns the original unmodified document selected for this update.

> **Warning:** When using `findAndModify` (page 25) in a *sharded* environment, the `query` must contain the *shard key* for all operations against the shard cluster. `findAndModify` (page 25) operations issued against `mongos` (page 256) instances for non-sharded collections function normally.

`db.collection.insert(document)`

The `insert()` method inserts a document or documents into a collection. Changed in version 2.2: The `insert()` method can accept an array of documents to perform a bulk insert of the documents into the collection. Consider the following behaviors of the `insert()` method:

- If the collection does not exist, then the `insert()` method will create the collection.
- If the document does not specify an *_id* field, then MongoDB will add the `_id` field and assign a unique *ObjectId* for the document before inserting. Most drivers create an ObjectId and insert the `_id` field, but the `mongod` will create and populate the `_id` if the driver or application does not.
- If the document specifies a new field, then the `insert()` method inserts the document with the new field. This requires no changes to the data model for the collection or the existing documents.

The `insert()` method takes one of the following parameters:

Parameters

- **document** – A document to insert into the collection.
- **documents** (*array*) – New in version 2.2. An array of documents to insert into the collection.

Consider the following examples of the `insert()` method:

- To insert a single document and have MongoDB generate the unique `_id`, omit the `_id` field in the document and pass the document to the `insert()` method as in the following:

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

This operation inserts a new document into the `products` collection with the `item` field set to `card`, the `qty` field set to `15`, and the `_id` field set to a unique `ObjectId`:

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

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

- To insert a single document, with a custom `_id` field, include the `_id` field set to a unique identifier and pass the document to the `insert()` method as follows:

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

This operation inserts a new document in the `products` collection with the `_id` field set to `10`, the `item` field set to `box`, the `qty` field set to `20`:

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

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

- To insert multiple documents, pass an array of documents to the `insert()` method as in the following:

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

The operation will insert three documents into the `products` collection:

  - A document with the fields `_id` set to `11`, `item` set to `pencil`, `qty` set to `50`, and the `type` set to `no.2`.
  - A document with the fields `_id` set to a unique `objectid`, `item` set to `pen`, and `qty` set to `20`.

–A document with the fields _id set to a unique objectid, item set to eraser, and qty set to 25.

```
{ "_id" : 11, "item" : "pencil", "qty" : 50, "type" : "no.2" }
{ "_id" : ObjectId("50631bc0be4617f17bb159ca"), "item" : "pen", "qty" : 20 }
{ "_id" : ObjectId("50631bc0be4617f17bb159cb"), "item" : "eraser", "qty" : 25 }
```

db.collection.**save**(*document*)

The save() method updates an existing document or inserts a document depending on the parameter.

The save() method takes the following parameter:

Parameters

- **document** – Specify a document to save to the collection.

  If the document does not contain an *_id* field, then the save() method performs an insert with the specified fields in the document as well as an _id field with a unique *objectid* value.

  If the document contains an _id field, then the save() method performs an upsert querying the collection on the _id field:

  – If a document does not exist with the specified _id value, the save() method performs an insert with the specified fields in the document.

  – If a document exists with the specified _id value, the save() method performs an update, replacing all field in the existing record with the fields from the document.

Consider the following examples of the save() method:

•Pass to the save() method a document without an _id field, so that to insert the document into the collection and have MongoDB generate the unique _id as in the following:

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

This operation inserts a new document into the products collection with the item field set to book, the qty field set to 40, and the _id field set to a unique ObjectId:

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

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

•Pass to the save() method a document with an _id field that holds a value that does not exist in the collection to insert the document with that value in the _id value into the collection, as in the following:

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

This operation inserts a new document into the products collection with the _id field set to 100, the item field set to water, and the field qty set to 30:

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

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

•Pass to the save() method a document with the _id field set to a value in the collection to replace all fields and values of the matching document with the new fields and values, as in the following:

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

This operation replaces the existing document with a value of `100` in the `_id` field. The updated document will resemble the following:

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

`db.collection.update(`*query*, *update*[, *options*]`)`

The `update()` method modifies an existing document or documents in a collection. By default the `update()` method updates a single document. To update all documents in the collection that match the update query criteria, specify the `multi` option. To insert a document if no document matches the update query criteria, specify the `upsert` option. Changed in version 2.2: The `mongo` shell provides an updated interface that accepts the options parameter in a document format to specify `multi` and `upsert` options. Prior to version 2.2, in the `mongo` shell, `upsert` and `multi` were positional boolean options:

```
db.collection.update(query, update, <upsert,> <multi>)
```

The `update()` method takes the following parameters:

Parameters

- **query** (*document*) – Specifies the selection criteria for the update. The `query` parameter employs the same *query selectors* (page 111) as used in the `db.collection.find()` method.
- **update** (*document*) – Specifies the modifications to apply.

  If the `update` parameter contains any *update operators* (page 127) expressions such as the `$set` (page 127) operator expression, then:

  – the `update` parameter must contain only `update operators` expressions.

  – the `update()` method updates only the corresponding fields in the document.

  If the `update` parameter consists only of `field: value` expressions, then:

  – the `update()` method *replaces* the document with the `updates` document. If the `updates` document is missing the *_id* field, MongoDB will add the `_id` field and assign to it a unique *objectid* .

  – the `update()` method updates cannot update multiple documents.
- **options** (*document*) – New in version 2.2. Optional. Specifies whether to perform an *upsert* and/or a multiple update. Use the `options` parameter instead of the individual `upsert` and `multi` parameters.
- **upsert** (*boolean*) – Optional. Specifies an *upsert* operation

  The default value is `false`. When `true`, the `update()` method will update an existing document that matches the `query` selection criteria **or** if no document matches the criteria, insert a new document with the fields and values of the `update` parameter and if the `update` included only `update operators`, the `query` parameter as well .

  In version 2.2 of the `mongo` shell, you may also specify `upsert` in the `options` parameter.

  Note: An upsert operation affects only *one* document, and cannot update multiple documents.

- **multi** (*boolean*) – Optional. Specifies whether to update multiple documents that meet the query criteria.

  When not specified, the default value is `false` and the `update()` method updates a single document that meet the `query` criteria.

  When `true`, the `update()` method updates all documents that meet the `query` criteria.

  In version 2.2 of the `mongo` shell, you may also specify `multi` in the `options` parameter.

  Note: The `multi` update operation may interleave with other write operations. For unsharded collections, you can override this behavior with the `$atomic` (page 133) isolation operator, which isolates the update operation and blocks other write operations during the update. See the `isolation operator`.

Although the update operation may apply mostly to updating the values of the fields, the `update()` method can also modify the name of the `field` in a document using the `$rename` (page 128) operator.

Consider the following examples of the `update()` method. These examples all use the 2.2 interface to specify options in the document form.

- To update specific fields in a document, call the `update()` method with an `update` parameter using `field: value` pairs and expressions using *update operators* (page 127) as in the following:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { $set: { x: 6 }, $inc: { y: 5} } )
```

  This operation updates a document in the `products` collection that matches the query criteria and sets the value of the field `x` to `6`, and increment the value of the field `y` by `5`. All other fields of the document remain the same.

- To replace all the fields in a document with the document as specified in the `update` parameter, call the `update()` method with an `update` parameter that consists of *only* `key: value` expressions, as in the following:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { x: 6, y: 15 } )
```

  This operation selects a document from the `products` collection that matches the query criteria sets the value of the field `x` to `6` and the value of the field `y` to `15`. All other fields of the matched document are *removed*, except the *_id* field.

- To update multiple documents, call the `update()` method and specify the `multi` option in the `options` argument, as in the following:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { $set: { x: 6, y: 15 } }, { multi: t
```

  This operation updates *all* documents in the `products` collection that match the query criteria by setting the value of the field `x` to `6` and the value of the field `y` to `15`. This operation does not affect any other fields in documents in the `products` collection.

  You can perform the same operation by calling the `update()` method with the `multi` parameter:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { $set: { x: 6, y: 15 } }, false, tr
```

- To update a document or to insert a new document if no document matches the query criteria, call the `update()` and specify the `upsert` option in the `options` argument, as in the following:

```
db.products.update( { item: "magazine", qty: { $gt: 5 } }, { $set: { x: 25, y: 50 } }, { up
```

  This operation, will:

– update a single document in the `products` collection that matches the query criteria, setting the value of the field `x` to `25` and the value of the field `y` to `50`, *or*

– if no matching document exists, insert a document in the `products` collection, with the field `item` set to `magazine`, the field `x` set to `25`, and the field `y` set to `50`.

4.1.2 Cursor Methods

Call cursor methods on cursors to modify how MongoDB returns objects to the cursor.

`cursor.next()`

Returns The next document in the cursor returned by the `db.collection.find()` method. See `cursor.hasNext()` related functionality.

`cursor.size()`

Returns A count of the number of documents that match the `db.collection.find()` query after applying any `cursor.skip()` and `cursor.limit()` methods.

`cursor.explain()`

The `cursor.explain()` method provides information on the query plan. The query plan is the plan the server uses to find the matches for a query. This information may be useful when optimizing a query.

Parameters

- **verbose** (*boolean*) – Specifies the level of detail to include in the output. If `true` or `1`, include the `allPlans` and `oldPlan` fields in the `output document`.

Returns A `document` that describes the process used to return the query results.

Retrieve the query plan by appending `explain()` to a `find()` query, as in the following example:

```
db.products.find().explain()
```

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

`explain` runs the actual query to determine the result. Although there are some differences between running the query with `explain` and running without, generally, the performance will be similar between the two. So, if the query is slow, the `explain` operation is also slow.

Additionally, the `explain` operation reevaluates a set of candidate query plans, which may cause the `explain` operation to perform differently than a normal query. As a result, these operations generally provide an accurate account of *how* MongoDB would perform the query, but do not reflect the length of these queries.

To determine the performance of a particular index, you can use `hint()` and in conjunction with `explain()`, as in the following example:

```
db.products.find().hint( { type: 1 } ).explain()
```

When you run `explain` with `hint()`, the query optimizer does not reevaluate the query plans.

Note: In some situations, the `explain()` operation may differ from the actual query plan used by MongoDB in a normal query.

The `explain()` operation evaluates the set of query plans and reports on the winning plan for the query. In normal operations the query optimizer caches winning query plans and uses them for similar related queries in the future. As a result MongoDB may sometimes select query plans from the cache that are different from the plan displayed using `explain`.

See Also:

- `$explain`
- Optimization wiki page for information regarding optimization strategies.
- Database Profiler wiki page for information regarding optimization strategies.
- *Current Operation Reporting* (page 199)

`cursor.`**`showDiskLoc`**`()`

Returns A modified cursor object that contains documents with appended information that describes the on-disk location of the document.

See Also:

`$showDiskLoc` for related functionality.

`cursor.`**`forEach`**`(`*function*`)`

Parameters

- **function** – function to apply to each document visited by the cursor.

Provides the ability to loop or iterate over the cursor returned by a `db.collection.find()` query and returns each result on the shell. Specify a JavaScript function as the argument for the `cursor.forEach()` function. Consider the following example:

```
db.users.find().forEach( function(u) { print("user: " + u.name); } );
```

See Also:

`cursor.map()` for similar functionality.

`cursor.`**`map`**`(`*function*`)`

Parameters

- **function** – function to apply to each document visited by the cursor.

Apply *function* to each document visited by the cursor, and collect the return values from successive application into an array. Consider the following example:

```
db.users.find().map( function(u) { return u.name; } );
```

See Also:

`cursor.forEach()` for similar functionality.

`cursor.`**`hasNext`**`()`

Returns Boolean.

`cursor.hasNext()` returns `true` if the cursor returned by the `db.collection.find()` query can iterate further to return more documents.

`cursor.`**`count`**`()`

The `count()` method counts the number of documents referenced by a cursor. Append the `count()` method to a `find()` query to return the number of matching documents, as in the following prototype:

```
db.collection.find().count()
```

This operation does not actually perform the `find()`; instead, the operation counts the results that would be returned by the `find()`.

The `count()` can accept the following argument:

Parameters

- **applySkipLimit** (*boolean*) – Optional. Specifies whether to consider the effects of the `cursor.skip()` and `cursor.limit()` methods in the count. By default, the `count()` method ignores the effects of the `cursor.skip()` and `cursor.limit()`. Set `applySkipLimit` to `true` to consider the effect of these methods.

See Also:

`cursor.size()`

MongoDB also provides the shell wrapper `db.collection.count()` for the `db.collection.find().count()` construct.

Consider the following examples of the `count()` method:

- Count the number of all documents in the `orders` collection:

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

- Count 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()
```

- Count 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)
```

`cursor.`**`limit`**`()`

Use the `cursor.limit()` method on a cursor to specify the maximum number of documents a the cursor will return. `cursor.limit()` is analogous to the `LIMIT` statement in a SQL database.

Note: You must apply `cursor.limit()` to the cursor before retrieving any documents from the database.

Use `cursor.limit()` to maximize performance and prevent MongoDB from returning more results than required for processing.

A `cursor.limit()` value of 0 (e.g. "`.limit(0)`") is equivalent to setting no limit.

`cursor.`**`skip`**`()`

Call the `cursor.skip()` 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()` to the cursor before retrieving any documents from the database.

Consider the following JavaScript function as an example of the sort function:

```
function printStudents(pageNumber, nPerPage) {
   print("Page: " + pageNumber);
   db.students.find().skip((pageNumber-1)*nPerPage).limit(nPerPage).forEach( function(student) {
}
```

The `cursor.skip()` 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 result. As offset (e.g. `pageNumber` above) increases, `cursor.skip()` will become slower and more CPU intensive. With larger collections, `cursor.skip()` 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.`**`readPref`**`()`

Parameters

- **mode** (*string*) – Read preference mode
- **tagSet** (*array*) – Optional. Array of tag set objects

Append the `readPref()` to a cursor to control how the client will route the query will route to members of the replica set.

The `mode` string should be one of:

- `primary`
- `primaryPreferred`
- `secondary`
- `secondaryPreferred`
- `nearest`

The `tagSet` parameter, if given, should consist of an array of tag set objects for filtering secondary read operations. For example, a secondary member tagged `{ dc: 'ny', rack: 2, size: 'large' }` will match the tag set `{ dc: 'ny', rack: 2 }`. Clients match tag sets first in the order they appear in the read preference specification. You may specify an empty tag set `{}` as the last element to default to any available secondary. See the *tag sets* documentation for more information.

Note: You must apply `cursor.readPref()` to the cursor before retrieving any documents from the database.

`cursor.`**`snapshot`**`()`

Append the `cursor.snapshot()` 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 `cursor.snapshot()` to the cursor before retrieving any documents from the database.
- You can only use `snapshot()` with **unsharded** collections.

The `snapshot()` does not guarantee isolation from insertion or deletions.

The `cursor.snapshot()` traverses the index on the `_id` field. As such, `snapshot()` **cannot** be used with `sort()` or `hint()`.

Queries with results of less than 1 megabyte are effectively implicitly snapshotted.

`cursor.`**`sort`**`(`*sort*`)`

Parameters

- **sort** – A document whose fields specify the attributes on which to sort the result set.

Append the `sort()` method to a cursor to control the order that the query returns matching documents. For each field in the sort document, if the field's corresponding value is positive, then `sort()` returns query results in ascending order for that attribute: if the field's corresponding value is negative, then `sort()` returns query results in descending order.

Note: You must apply `cursor.limit()` to the cursor before retrieving any documents from the database.

Consider the following example:

```
db.collection.find().sort( { age: -1 } );
```

Here, the query returns all documents in `collection` sorted by the `age` field in descending order. Specify a value of negative one (e.g. `-1`), as above, to sort in descending order or a positive value (e.g. `1`) to sort in ascending order.

Unless you have a index for the specified key pattern, use `cursor.sort()` in conjunction with `cursor.limit()` to avoid requiring MongoDB to perform a large, in-memory sort. `cursor.limit()` increases the speed and reduces the amount of memory required to return this query by way of an optimized algorithm.

> **Warning:** The sort function requires that the entire sort be able to complete within 32 megabytes. When the sort option consumes more than 32 megabytes, MongoDB will return an error. Use `cursor.limit()`, or create an index on the field that you're sorting to avoid this error.

The `$natural` parameter returns items according to their order on disk. Consider the following query:

```
db.collection.find().sort( { $natural: -1 } )
```

This will return documents in the reverse of the order on disk. Typically, the order of documents on disks reflects insertion order, *except* when documents move internal because of document growth due to update operations.

`cursor.`**`hint`**`(`*index*`)`

Arguments

- **index** – The specification for the index to "hint" or force MongoDB to use when performing the query.

Call this method on a query to override MongoDB's default index selection and query optimization process. The argument is an index specification, like the argument to `ensureIndex()`. Use `db.collection.getIndexes()` to return the list of current indexes on a collection.

See Also:

"`$hint`"

4.1.3 Data Aggregation Methods

`db.collection.`**`aggregate`**`(`*pipeline*`)`

New in version 2.1.0. Always call the `db.collection.aggregate()` method on a collection object.

Arguments

- **pipeline** – Specifies a sequence of data aggregation processes. See the *aggregation reference* (page 141) for documentation of these operators.

Consider the following example from the `aggregation documentation`.

```
db.article.aggregate(
  { $project : {
     author : 1,
     tags : 1,
  } },
  { $unwind : "$tags" },
  { $group : {
     _id : { tags : 1 },
     authors : { $addToSet : "$author" }
  } }
);
```

See Also:

"`aggregate` (page 29)," "`http://docs.mongodb.org/manual/applications/aggregation`," and "*Aggregation Framework Reference* (page 141)."

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

The `db.collection.group()` method 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()` method returns an array.

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

Fields

- **key** – Specifies one or more document fields to group by.
- **reduce** – Specifies a function for the group operation perform on the documents during the grouping operation, such as compute a sum or a count. The aggregation function takes two arguments: the current document and the aggregate result for the previous documents in the.
- **initial** – Initializes the aggregation result document.
- **keyf** – Optional. Alternative to the `key` field. Specifies a function that creates a "key object" for use as the grouping key. Use the `keyf` instead of `key` to group by calculated fields rather than existing document fields.
- **cond** – Optional. Specifies the selection criteria to determine which documents in the collection to process. If you omit the `cond` field, `db.collection.group()` processes all the documents in the collection for the group operation.
- **finalize** – Optional. Specifies a function that runs each item in the result set before `db.collection.group()` returns the final value. This function can either modify the result document or replace the result document as a whole.

The `db.collection.group()` method is a shell wrapper for the `group` (page 14) command; however, the `db.collection.group()` method takes the `keyf` field and the `reduce` field whereas the `group` (page 14) command takes the `$keyf` field and the `$reduce` field.

Warning:

- The `db.collection.group()` method does not work with *sharded clusters*. Use the *aggregation framework* or *map-reduce* in *sharded environments*.
- The `group` (page 14) command takes a read lock and does not allow any other threads to execute JavaScript while it is running.

Note:

- The result set must fit within the *maximum BSON document size* (page 265).
- 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 18). Previous versions had a limit of 10,000 elements.

Consider the following examples of the `db.collection.group()` 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 }
}
```

- 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
```

- 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
```

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

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

                              result.day_of_week = weekdays[result.day_of_week];
                              result.avg = Math.round(result.total / result.count);

                   }
                } )
```

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

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

See Also:

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

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

The `db.collection.mapReduce()` method provides a wrapper around the `mapReduce` (page 18) command.

```
db.collection.mapReduce(
                         mapfunction,
                         reducefunction,
                         {
```

```
                      out: <collection>,
                      query: <document>,
                      sort: <document>,
                      limit: <number>,
                      finalize: <function>,
                      scope: <document>,
                      jsMode: <boolean>,
                      verbose: <boolean>
                    }
                  )
```

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

Parameters

- **map** – A JavaScript function that associates or "maps" a value with a key.

  The `map` function has the following prototype:

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

  The `map` function process every input document for the map-reduce operation. All the `key` and `value` pairs emitted by the map function. In map-reduce operations, the operation groups the output from the map phase by the `key` value and passes these groupings to the `reduce` function.

  Note:

  – 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.)

  – The `emit(key,value)` function associates the `key` with a `value`.

    * A single emit can only hold half of MongoDB's *maximum BSON document size* (page 265).

    * There is no limit to the number of times you may call the `emit` function per document.

  – The `map` function can access the variables defined in the `scope` parameter.

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

  The `reduce` function has the following prototype:

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

  The `reduce` function accepts `key` and `values` arguments. The elements of the `values` array are the individual `value` objects emitted by the `<map>` function, grouped by the `key`.

Note:

– The `reduce` function should *not* access the database, even to perform read operations.

– The `reduce` function should *not* affect the outside system.

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

1. the *type* of the return object must be **identical** to the type of the `value` emitted by the `<map>` function to ensure that the following operations is true:

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

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

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

3. 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 ] )
```

– MongoDB will **not** call the `reduce` function for a key that has only a single value.

– The `reduce` function can access the variables defined in the `scope` parameter.

- **out** – Specifies the location of the result of the map-reduce operation. 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.

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

  – **Output to a collection**.

```
{ out: <collectionName> }
```

  – **Output to a collection and specify one of the following actions**. This option is only available when passing `out` a collection that already exists. This option is not available on secondary members of replica sets.

```
{ out: { <action>: <collectionName>[, db: <dbName>][, sharded: <boolean> ][, no
```

  * `<action>`: Specify one of the following actions:

    · `replace`

```
{ out: { replace: <collectionName> } }
```

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

    · `merge`

```
{ out: { merge: <collectionName> } }
```

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`

```
{ out: { reduce: <collectionName> } }
```

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 2.1. Optional. Specify output operation as non-atomic and is valid *only* for `merge` and `reduce` output modes which may take minutes to execute.

If `nonAtomic` is `true`, the post-processing step will prevent MongoDB from locking the database; however, other clients will be able to read intermediate states of the output collection. Otherwise the map reduce operation must lock the database during post-processing.

– **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 265).

- **query** – Optional. Specifies the selection criteria using *query operators* (page 111) for determining the documents input to the `map` function.
- **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.
- **limit** – Optional. Specifies a maximum number of documents to return from the collection.
- **finalize** – Optional. A JavaScript function that follows the `<reduce>` method and modifies the output and 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.

Note:

- 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.

- **scope** (*document*) – Optional. Specifies global variables that are accessible in the `map`, `reduce` and the `finalize` functions.
- **jsMode** (*Boolean*) – Optional. Specifies whether to convert intermediate data into BSON format between the execution of the `map` and `reduce` functions.

  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.

- **verbose** (*Boolean*) – Optional. Specifies whether to include the `timing` information in the result information. The `verbose` defaults to `true` to include the `timing` information.

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: 250,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

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

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

     – In the function, `this` refers to the document that the map-reduce operation is processing.

     – The function maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair.

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

  2. Define the corresponding reduce function with two arguments `keyCustId` and `valuesPrices`:

     – The `valuesPrices` is an array whose elements are the `price` values emitted by the map function and grouped by `keyCustId`.

     – The function reduces the `valuesPrice` array to the sum of its elements.

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

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

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

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

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

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

     – In the function, `this` refers to the document that the map-reduce operation is processing.

     – For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

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

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

   – `valuesCountObjects` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.

   – The function reduces the `valuesCountObjects` array to a single object `reducedValue` that also contains the `count` and the `qty` fields.

   – In `reducedValue`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

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

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

                          return reducedValue;
                      };
```

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

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

                          reducedValue.average = reducedValue.qty/reducedValue.count;

                          return reducedValue;
                      };
```

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

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

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

For more information and examples, see the `Map-Reduce` page.

See Also:

- *map-reduce* and `mapReduce` (page 18)
- `http://docs.mongodb.org/manual/applications/aggregation`

4.2 Administrative Functions

4.2.1 Database Methods

`db.addUser("username", "password"[, readOnly])`

Parameters

- **username** (*string*) – Specifies a new username.
- **password** (*string*) – Specifies the corresponding password.
- **readOnly** (*boolean*) – Optional. Restrict a user to read-privileges only. Defaults to false.

Use this function to create new database users, by specifying a username and password as arguments to the command. If you want to restrict the user to have only read-only privileges, supply a true third argument; however, this defaults to false.

`db.auth("username", "password")`

Parameters

- **username** (*string*) – Specifies an existing username with access privileges for this database.
- **password** (*string*) – Specifies the corresponding password.

Allows a user to authenticate to the database from within the shell. Alternatively use `mongo --username` (page 218) and `--password` (page 218) to specify authentication credentials.

`db.cloneDatabase("hostname")`

Parameters

- **hostname** (*string*) – Specifies the hostname to copy the current instance.

Use this function to copy a database from a remote to the current database. The command assumes that the remote database has the same name as the current database. For example, to clone a database named `importdb` on a host named `hostname`, do

```
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.

This function provides a wrapper around the MongoDB *database command* "`clone` (page 41)." The `copydb` (page 45) database command provides related functionality.

`db.commandHelp(command)`

Parameters

- **command** – Specifies a *database command name* (page 11).

Returns Help text for the specified *database command*. See the *database command reference* (page 11) for full documentation of these commands.

`db.copyDatabase(origin, destination, hostname)`

Parameters

- **origin** (*database*) – Specifies the name of the database on the origin system.
- **destination** (*database*) – Specifies the name of the database that you wish to copy the origin database into.

- **hostname** (*origin*) – Indicate the hostname of the origin database host. Omit the hostname to copy from one name to another on the same server.

Use this function to copy a specific database, named `origin` running on the system accessible via `hostname` into the local database named `destination`. The command creates destination databases implicitly when they do not exist. If you omit the hostname, MongoDB will copy data from one database into another on the same instance.

This function provides a wrapper around the MongoDB *database command* "`copydb` (page 45)." The `clone` (page 41) database command provides related functionality.

`db.createCollection(name[, {capped: <boolean>, size: <value>, max <bytes>} ])`

Parameters

- **name** (*string*) – Specifies the name of a collection to create.
- **capped** (*boolean*) – Optional. If this *document* is present, this command creates a capped collection. The capped argument is a *document* that contains the following three fields:
- **capped** – Enables a *collection cap*. False by default. If enabled, you must specify a `size` parameter.
- **size** (*bytes*) – If `capped` is `true`, `size` Specifies a maximum size in bytes, for the as a "*cap* for the collection. When `capped` is false, you may use `size`
- **max** (*int*) – Optional. Specifies a maximum "cap," in number of documents for capped collections. You must also specify `size` when specifying `max`.

Options

- **autoIndexId** – If `capped` is `true` you can specify `false` to disable the automatic index created on the `_id` field. Before 2.2, the default value for `autoIndexId` was `false`. See *_id Fields and Indexes on Capped Collections* (page 290) for more information.

Explicitly creates a new collation. Because MongoDB creates collections implicitly when referenced, this command is primarily used for creating new capped collections. In some circumstances, you may use this command to pre-allocate space for an ordinary 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, but may also specify a maximum document count. The collection will remove 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 or a maximum of 5000 documents.

The following command simply pre-allocates a 2 gigabyte, uncapped, collection named `people`:

```
db.createCollection("people", { size: 2147483648 })
```

This command provides a wrapper around the database command `create` (page 35). See the "Capped Collections" wiki page for more information about capped collections.

`db.currentOp()`

Returns A *document* that contains an array named `inprog`.

The `inprog` array reports the current operation in progress for the database instance. See *Current Operation Reporting* (page 199) for full documentation of the output of `db.currentOp()`.

`db.currentOp()` is only available for users with administrative privileges.

Consider the following JavaScript operations for the `mongo` shell that you can use to filter the output of identify specific types of operations:

- Return all pending write operations:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.waitingForLock && d.lockType != "read")
        printjson(d)
     })
```

- Return the active write operation:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.active && d.lockType == "write")
        printjson(d)
     })
```

- Return all active read operations:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.active && d.lockType == "read")
        printjson(d)
     })
```

> **Warning:** Terminate running operations with extreme caution. Only use `db.killOp()` to terminate operations initiated by clients and *do not* terminate internal database operations.

`db.`**`dropDatabase`**`()`

Removes the current database. Does not change the current database, so the insertion of any documents in this database will allocate a fresh set of data files.

`db.`**`eval`**`(`*function*, *arguments*`)`

The `db.eval()` provides the ability to run JavaScript code on the MongoDB server. It is a `mongo` shell wrapper around the `eval` (page 27) command.

The method accepts the following parameters:

Parameters

- **function** (*JavaScript*) – A JavaScript function.

  The function need not take any arguments, as in the first example, or may optionally take arguments as in the second:

  ```
  function () {
    // ...
  }

  function (arg1, arg2) {
     // ...
  }
  ```

- **arguments** – A list of arguments to pass to the JavaScript `function` if the function accepts arguments. Omit if the `function` does not take arguments.

Consider the following example of the `db.eval()` 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;
    },
    "<name>", 5 );
```

- The db in the function refers to the current database.
- "<name>" 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(). Otherwise, the mongo shell's JavaScript interpreter evaluates functions entered directly into the shell.

If an error occurs, db.eval() throws an exception. Consider the following invalid function that uses the variable x without declaring it as an argument:

```
db.eval( function() { return x + x; }, 3 );
```

The statement will result in the following exception:

```
{
   "errno" : -3,
   "errmsg" : "invoke failed: JS Error: ReferenceError: x is not defined nofile_b:1",
   "ok" : 0
}
```

Warning:

- The db.eval() operation takes a write lock by default. This means that db.eval() blocks all other read and write operations to the database while the db.eval() operation runs. However, if you are using the eval (page 27) command using db.runCommand() rather than the helper, you can disable the lock by setting the nolock flag to true if the eval (page 27) performs a strictly read-only operation.
- db.eval() also takes a JavaScript lock which blocks all
- Do not use db.eval() for long running operations, as db.eval() blocks all other operations. Consider using other server side code execution options.
- You can not use db.eval() with *sharded* data. In general, you should avoid using db.eval() in *sharded cluster*; nevertheless, it is possible to use db.eval() with non-sharded collections and databases stored in *sharded cluster*.

See Also:

http://docs.mongodb.org/manual/applications/server-side-javascript

db.**loadServerScripts**()

db.loadServerScripts() loads all scripts in the system.js collection for the current database into the mongo 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 121) clauses and `mapReduce` (page 18) operations.

`db.getCollection(name)`

Parameters

- **name** – The name of a collection.

Returns A collection.

Use this command to obtain a handle on a collection whose name might interact with the shell itself, including collections with names that begin with _ or mirror the *database commands* (page 11).

`db.getCollectionNames()`

Returns An array containing all collections in the existing database.

`db.getLastError()`

Returns The last error message string.

Sets the level of *write concern* for confirming the success of write operations.

See Also:

`getLastError` (page 50) for all options, *Write Concern* for a conceptual overview, `http://docs.mongodb.org/manual/core/write-operations` for information about all write operations in MongoDB, and *Replica Set Write Concern* for special considerations related to write concern for replica sets.

`db.getLastErrorObj()`

Returns A full *document* with status information.

`db.getMongo()`

Returns The current database connection.

`db.getMongo()` runs when the shell initiates. Use this command to test that the `mongo` shell has a connection to the proper database instance.

`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()`.

See the `readPref()` method for more fine-grained control over *read preference* in the `mongo` shell.

`db.getName()`

Returns the current database name.

`db.getProfilingLevel()`

This method provides a wrapper around the database command "`profile`" and returns the current profiling level. Deprecated since version 1.8.4: Use `db.getProfilingStatus()` for related functionality.

`db.getProfilingStatus()`

Returns The current `profile` level and `slowms` setting.

`db.getReplicationInfo()`

Returns A status document.

The output reports statistics related to replication.

See Also:

"`http://docs.mongodb.org/manual/reference/replication-info`" for full documentation of this output.

`db.getSiblingDB()`
Used to return another database without modifying the `db` variable in the shell environment.

`db.killOp(opid)`
Parameters

- **opid** – Specify an operation ID.

Terminates the specified operation. Use `db.currentOp()` to find operations and their corresponding ids. See *Current Operation Reporting* (page 199) for full documentation of the output of `db.currentOp()`.

> **Warning:** Terminate running operations with extreme caution. Only use `db.killOp()` to terminate operations initiated by clients and *do not* terminate internal database operations.

`db.listCommands()`
Provides a list of all database commands. See the "*Command Reference* (page 11)" document for a more extensive index of these options.

`db.logout()`
Ends the current authentication session. This function has no effect if the current session is not authenticated.

This function provides a wrapper around the database command "`logout` (page 45)".

`db.printCollectionStats()`
Provides a wrapper around the `db.collection.stats()` method. Returns statistics from every collection separated by three hyphen characters.

See Also:

"*Collection Statistics Reference* (page 185)"

`db.printReplicationInfo()`
Provides a formatted report of the status of a *replica set* from the perspective of the *primary* set member. See the "*Replica Set Status Reference* (page 193)" for more information regarding the contents of this output.

This function will return `db.printSlaveReplicationInfo()` if issued against a *secondary* set member.

`db.printSlaveReplicationInfo()`
Provides a formatted report of the status of a *replica set* from the perspective of the *secondary* set member. See the "*Replica Set Status Reference* (page 193)" for more information regarding the contents of this output.

`db.printShardingStatus()`
Provides a formatted report of the sharding configuration and the information regarding existing chunks in a *sharded cluster*.

Only use `db.printShardingStatus()` when connected to a `mongos` (page 256) instance.

This method is a wrapper around the `printShardingStatus` (page 14) command.

See Also:

`sh.status()`, `printShardingStatus` (page 14)

`db.removeUser(username)`
Parameters

- **username** – Specify a database username.

Removes the specified username from the database.

`db.repairDatabase()`

> **Warning:** In general, if you have an intact copy of your data, such as would exist on a very recent backup or an intact member of a *replica set*, **do not** use `repairDatabase` (page 43) or related options like `db.repairDatabase()` in the `mongo` shell or *`mongod --repair`* (page 211). Restore from an intact copy of your data.

Note: When using *journaling*, there is almost never any need to run `repairDatabase` (page 43). In the event of an unclean shutdown, the server will be able restore the data files to a pristine state automatically.

`db.repairDatabase()` provides a wrapper around the database command `repairDatabase` (page 43), and has the same effect as the run-time option *`mongod --repair`* (page 211) option, limited to *only* the current database. See `repairDatabase` (page 43) for full documentation.

`db.runCommand(command)`

Parameters

- **command** (*string*) – Specifies a *database command* in the form of a *document*.
- **command** – When specifying a *command* (page 11) as a string, `db.runCommand()` transforms the command into the form `{ command: 1 }`.

Provides a helper to run specified *database commands* (page 11). This is the preferred method to issue database commands, as it provides a consistent interface between the shell and drivers.

`db.serverStatus()`

Returns a *document* that provides an overview of the database process's state.

This command provides a wrapper around the database command `serverStatus` (page 52).

See Also:

"*Server Status Reference* (page 167)" for complete documentation of the output of this function.

`db.setProfilingLevel(level[, slowms])`

Parameters

- **level** – Specifies a profiling level, see list of possible values below.
- **slowms** – Optionally modify the threshold for the profile to consider a query or operation "slow."

Modifies the current *database profiler* 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, which might information security implications for your deployment.

The following profiling levels are available:

Level	Setting
0	Off. No profiling.
1	On. Only includes slow operations.
2	On. Includes all operations.

Also configure the `slowms` option to set the threshold for the profiler to consider a query "slow." Specify this value in milliseconds to override the default.

This command provides a wrapper around the *database command* `profile`.

`mongod` writes the output of the database profiler to the `system.profile` collection.

`mongod` prints information about queries that take longer than the `slowms` to the log even when the database profiler is not active.

Note: The database cannot be locked with `db.fsyncLock()` while profiling is enabled. You must disable profiling before locking the database with `db.fsyncLock()`. Disable profiling using `db.setProfilingLevel()` as follows in the `mongo` shell:

```
db.setProfilingLevel(0)
```

`db.shutdownServer()`

Shuts down the current `mongod` or `mongos` (page 256) 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 44).

`db.stats(scale)`

Parameters

- **scale** – Optional. Specifies the scale to deliver results. Unless specified, this command returns all data in bytes.

Returns A *document* that contains statistics reflecting the database system's state.

This function provides a wrapper around the database command "`dbStats` (page 48)". The `scale` option allows you to configure how the `mongo` shell scales the sizes of things in the output. For example, specify a `scale` value of `1024` to display kilobytes rather than bytes.

See the "*Database Statistics Reference* (page 183)" document for an overview of this output.

Note: The scale factor rounds values to whole numbers. This can produce unpredictable and unexpected results in some situations.

`db.version()`

Returns The version of the `mongod` instance.

`db.fsyncLock()`

Forces the `mongod` to flush pending all write operations to the disk and locks the *entire* `mongod` instance to prevent additional writes until the user releases the lock with the `db.fsyncUnlock()` command. `db.fsyncLock()` is an administrative command.

This command provides a simple wrapper around a `fsync` (page 39) database command with the following syntax:

```
{ fsync: 1, lock: true }
```

This function locks the database and create a window for `backup operations`.

Note: The database cannot be locked with `db.fsyncLock()` while profiling is enabled. You must disable profiling before locking the database with `db.fsyncLock()`. Disable profiling using `db.setProfilingLevel()` as follows in the `mongo` shell:

```
db.setProfilingLevel(0)
```

`db.fsyncUnlock()`

Unlocks a `mongod` instance to allow writes and reverses the operation of a `db.fsyncLock()` operation. Typically you will use `db.fsyncUnlock()` following a database `backup operation`.

`db.fsyncUnlock()` is an administrative command.

4.2.2 Collection Methods

These methods operate on collection objects. Also consider the "*Query and Update Methods* (page 63)" and "*Cursor Methods* (page 70)" documentation for additional methods that you may use with collection objects.

Note: Call these methods on a *collection* object in the shell (i.e. `db.collection.[method]()`, where `collection` is the name of the collection) to produce the documented behavior.

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

Returns The size of the collection. This method provides a wrapper around the `size` (page 186) output of the `collStats` (page 36) (i.e. `db.collection.stats()`) command.

`db.collection.`**`storageSize`**`()`

Returns The amount of storage space, calculated using the number of extents, used by the collection. This method provides a wrapper around the `storageSize` (page 184) output of the `collStats` (page 36) (i.e. `db.collection.stats()`) command.

`db.collection.`**`totalIndexSize`**`()`

Returns The total size of all indexes for the collection. This method provides a wrapper around the `db.collection.totalIndexSize()` output of the `collStats` (page 36) (i.e. `db.collection.stats()`) command.

`db.collection.`**`distinct`**`()`

The `db.collection.distinct()` method finds the distinct values for a specified field across a single collection and returns the results in an array. The method accepts the following argument:

Parameters

- **field** (*string*) – Specifies the field for which to return the distinct values.
- **query** (*document*) – Specifies the selection `query` to determine the subset of documents from which to retrieve the distinct values.

Note:

- The `db.collection.distinct()` method provides a wrapper around the `distinct` (page 26) command. Results must not be larger than the maximum *BSON size* (page 265).
- When possible to use covered indexes, the `db.collection.distinct()` method will use an index to find the documents in the query as well as to return the data.

Consider the following examples of the `db.collection.distinct()` method:

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

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

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

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

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

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

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

Call the `db.collection.drop()` method on a collection to drop it from the database.

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

`db.collection.dropIndex(index)`

Drops or removes the specified index from a collection. The `db.collection.dropIndex()` method provides a wrapper around the `dropIndexes` (page 41) command.

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

Parameters

- **index** – Specifies either the name or the key of the index to drop. You **must** use the name of the index if you specified a name during the index creation.

The `db.collection.dropIndex()` method cannot drop the `_id` index. Use the `db.collection.getIndexes()` method to view all indexes on a collection.

Consider the following examples of the `db.collection.dropIndex()` method that assumes the following indexes on the collection `pets`:

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

- To drop the index on the field `cat`, you must use the index name `catIdx`:

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

- To drop the index on the fields `cat` and `dog`, you use either the index name `cat_1_dog_-1` or the key `{ "cat" : 1, "dog" : -1 }`:

```
db.pets.dropIndex( 'cat_1_dog_-1' )

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

`db.collection.dropIndexes()`

Drops all indexes other than the required index on the `_id` field. Only call `dropIndexes()` (page 41) as a method on a collection object.

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

Parameters

- **keys** (*document*) – A *document* that contains pairs with the name of the field or fields to index and order of the index. A `1` specifies ascending and a `-1` specifies descending.
- **options** (*document*) – A *document* that controls the creation of the index. This argument is optional.

Warning: Index names, including their full namespace (i.e. `database.collection`) can be no longer than 128 characters. See the `db.collection.getIndexes()` field "`name`" for the names of existing indexes.

See Also:

The `http://docs.mongodb.org/manual/indexes` section of this manual for full documentation of indexes and indexing in MongoDB.

Creates an index on the field specified, if that index does not already exist. If the `keys` document specifies more than one field, than `db.collection.ensureIndex()` creates a *compound index*. For example:

```
db.collection.ensureIndex({ [key]: 1})
```

This command creates an index, in ascending order, on the field `[key]`. To specify a compound index use the following form:

```
db.collection.ensureIndex({ [key]: 1, [key1]: -1 })
```

This command creates a compound index on the `key` field (in ascending order) and `key1` field (in descending order.)

Note: Typically the order of an index is only important when doing `cursor.sort()` operations on the indexed fields.

The available options, possible values, and the default settings are as follows:

Option	Plugin	Default
background	`true` or `false`	`false`
unique	`true` or `false`	`false`
name	string	none
cache	`true` or `false`	`true`
dropDups	`true` or `false`	`false`
sparse	`true` or `false`	`false`
expireAfterSeconds	integer	none
v	index version	1

Options

- **background** (*Boolean*) – Specify `true` to build the index in the background so that building an index will *not* block other database activities.
- **unique** (*Boolean*) – Specify `true` to create a unique index so that the collection will not accept insertion of documents where the index key or keys matches an existing value in the index.
- **name** (*string*) – Specify the name of the index. If unspecified, MongoDB will generate an index name by concatenating the names of the indexed fields and the sort order.
- **cache** (*Boolean*) – Specify `false` to prevent caching of this `db.collection.ensureIndex()` call in the index cache.
- **dropDups** (*Boolean*) – Specify `true` when creating a unique index, on a field that *may* have duplicate to index only the first occurrence of a key, and **remove** all documents from the collection that contain subsequent occurrences of that key.

- **sparse** (*Boolean*) – If `true`, the index only references documents with the specified field. These indexes use less space, but behave differently in some situations (particularly sorts.)
- **expireAfterSeconds** (*integer*) – Specify a value, in seconds, as a *TTL* to control how long MongoDB will retain documents in this collection. See "`http://docs.mongodb.org/manual/tutorial/expire-data`" for more information on this functionality.
- **v** – Only specify a different index version in unusual situations. The latest index version (version 1) provides a smaller and faster index format.

Please be aware of the following behaviors of `ensureIndex()`:

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

  If you create an index with one set of options, and then issue `ensureIndex()` method command with the same index fields and different options without first dropping the index, `ensureIndex()` will *not* rebuild the existing index with the new options.
- If you call multiple `ensureIndex()` 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.
- You cannot stop a foreground index build once it's begun. See the *indexes-admin-stop-in-progress-build* for more information.

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

This method drops all indexes 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()
```

Change `collection` to the name of the collection that you want to rebuild the index.

`getDB``()`

Returns the name of the current database as a string.

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

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

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

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

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

`getIndexes.`**`v`**

Holds the version of the index.

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

`getIndexes.`**`key`**

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

`getIndexes.ns`
The namespace context for the index.

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

`db.collection.remove(query, justOne)`
The `remove` method removes documents from a collection.

The `remove()` method can take the following parameters:

Parameters

- **query** (*document*) – Optional. Specifies the deletion criteria using *query operators* (page 111). Omit the `query` parameter or pass an empty document (e.g. `{}`) to delete all *documents* in the *collection*.
- **justOne** (*boolean*) – Optional. A boolean that limits the deletion to just one document. The default value is `false`. Set to `true` to delete only the first result.

Note: You cannot apply the `remove()` method to a *capped collection*.

Consider the following examples of the `remove` method.

- To remove all documents in a collection, call the `remove` method with no parameters:

```
db.products.remove()
```

This operation will remove all the documents from the collection `products`.

- To remove the documents that match a deletion criteria, call the `remove` method with the `query` criteria:

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

This operation removes all the documents from the collection `products` where `qty` is greater than `20`.

- To remove the first document that match a deletion criteria, call the `remove` method with the `query` criteria and the `justOne` parameter set to `true` or `1`:

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

This operation removes all the documents from the collection `products` where `qty` is greater than `20`.

Note: If the `query` argument to the `remove()` method matches multiple documents in the collection, the delete operation may interleave with other write operations to that collection. For an unsharded collection, you have the option to override this behavior with the `$atomic` (page 133) isolation operator, effectively isolating the delete operation and blocking other write operations during the delete. To isolate the query, include `$atomic: 1` in the `query` parameter as in the following example:

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

`db.collection.renameCollection()`
`db.collection.renameCollection()` provides a helper for the `renameCollection` (page 37) *database command* in the `mongo` shell to rename existing collections.

Parameters

- **target** (*string*) – Specifies the new name of the collection. Enclose the string in quotes.
- **dropTarget** (*boolean*) – Optional. If `true`, `mongod` will drop the *target* of `renameCollection` (page 37) prior to renaming the collection.

Call the `db.collection.renameCollection()` method on a collection object, to rename a collection. Specify the new name of the collection as an argument. 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.

Consider the following limitations:

- `db.collection.renameCollection()` cannot move a collection between databases. Use `renameCollection` (page 37) for these rename operations.
- `db.collection.renameCollection()` cannot operation on sharded collections.

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

Refer to the documentation `renameCollection` (page 37) for additional warnings and messages.

> **Warning:** The `db.collection.renameCollection()` method and `renameCollection` (page 37) command will invalidate open cursors which interrupts queries that are currently returning data.

`db.collection.validate()`

Parameters

- **full** (*Boolean*) – Optional. Specify `true` to enable a full validation. MongoDB disables full validation by default because it is a potentially resource intensive operation.

Provides a wrapper around the `validate` (page 49) *database command*. Call the `db.collection.validate()` method on a collection object, to validate the collection itself. Specify the full option to return full statistics.

The `validation` (page 49) operation scans all of the data structures for correctness and returns a single *document* that describes the relationship between the logical collection and the physical representation of that data.

The output can provide a more 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.

See Also:

`http://docs.mongodb.org/manual/reference/collection-validation`

`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.

`getShardDistribution()`

See SERVER-4902 for more information.

`db.collection.stats(scale)`

Parameters

- **scale** – Optional. Specifies the scale to deliver results. Unless specified, this command returns all sizes in bytes.

Returns A *document* containing statistics that reflecting the state of the specified collection.

This function provides a wrapper around the database command `collStats` (page 36). The `scale` option allows you to configure how the `mongo` shell scales the sizes of things in the output. For example, specify a `scale` value of `1024` to display kilobytes rather than bytes.

Call the `db.collection.stats()` method on a collection object, to return statistics regarding that collection. For example, the following operation returns stats on the `people` collection:

```
db.people.stats()
```

See Also:

"*Collection Statistics Reference* (page 185)" for an overview of the output of this command.

4.2.3 Sharding Methods

See Also:

The "`http://docs.mongodb.org/manual/core/sharding`" page for more information on the sharding technology and using MongoDB's *sharded clusters*.

`sh.`**`addShard`**`(`*host*`)`

Parameters

- **host** (*string*) – Specify the hostname of a database instance or a replica set configuration.

Use this method to add a database instance or replica set to a *sharded cluster*. This method must be run on a `mongos` (page 256) instance. The `host` parameter can be in any of the following forms:

```
[hostname]
[hostname]:[port]
[set]/[hostname]
[set]/[hostname],[hostname]:port
```

You can specify shards using the hostname, or a hostname and port combination if the shard is running on a non-standard port.

> **Warning:** Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.

The optimal configuration is to deploy shards across *replica sets*. To add a shard on a replica set you must specify the name of the replica set and the hostname of at least one member of the replica set. You must specify at least one member of the set, but can specify all members in the set or another subset if desired. `sh.addShard()` takes the following form:

If you specify additional hostnames, all must be members of the same replica set.

```
sh.addShard("set-name/seed-hostname")
```

Example

```
sh.addShard("repl0/mongodb3.example.net:27327")
```

The `sh.addShard()` method is a helper for the `addShard` (page 12) command. The `addShard` (page 12) command has additional options which are not available with this helper.

See Also:

- `addShard` (page 12)
- `http://docs.mongodb.org/manual/administration/sharding`
- `http://docs.mongodb.org/manual/tutorial/add-shards-to-shard-cluster`
- `http://docs.mongodb.org/manual/tutorial/remove-shards-from-cluster`

`sh.enableSharding(database)`

Parameters

- **database** (*name*) – Specify a database name to shard.

Enables sharding on the specified database. This does not automatically shard any collections, but makes it possible to begin sharding collections using `sh.shardCollection()`.

`sh.shardCollection(collection, key, unique)`

Parameters

- **collection** (*name*) – The name of the collection to shard.
- **key** (*document*) – A *document* containing *shard key* that the sharding system uses to *partition* and distribute objects among the shards.
- **unique** (*boolean*) – When true, the `unique` option ensures that the underlying index enforces uniqueness so long as the unique index is a prefix of the shard key.

Shards the named collection, according to the specified *shard key*. Specify shard keys in the form of a *document*. Shard keys may refer to a single document field, or more typically several document fields to form a "compound shard key."

`sh.splitFind(namespace, query)`

Parameters

- **namespace** (*string*) – Specify the namespace (i.e. "`<database>.<collection>`") of the sharded collection that contains the chunk to split.
- **query** – Specify a query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Splits the chunk containing the document specified by the `query` at its median point, creating two roughly equal chunks. Use `sh.splitAt()` to split a collection in a specific point.

In most circumstances, you should leave chunk splitting to the automated processes. However, when initially deploying a *sharded cluster* it is necessary to perform some measure of *pre-splitting* using manual methods including `sh.splitFind()`.

`sh.splitAt(namespace, query)`

Parameters

- **namespace** (*string*) – Specify the namespace (i.e. "`<database>.<collection>`") of the sharded collection that contains the chunk to split.
- **query** (*document*) – Specify a query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Splits the chunk containing the document specified by the `query` as if that document were at the "middle" of the collection, even if the specified document is not the actual median of the collection. Use this command to manually split chunks unevenly. Use the "`sh.splitFind()`" function to split a chunk at the actual median.

In most circumstances, you should leave chunk splitting to the automated processes within MongoDB. However, when initially deploying a *sharded cluster* it is necessary to perform some measure of *pre-splitting* using manual methods including `sh.splitAt()`.

`sh.moveChunk(collection, query, destination)`

Parameters

- **collection** (*string*) – Specify the sharded collection containing the chunk to migrate.
- **query** – Specify a query to identify documents in a specific chunk. Typically specify the *shard key* for a document as the query.

- **destination** (*string*) – Specify the name of the shard that you wish to move the designated chunk to.

Moves the chunk containing the documents specified by the `query` to the shard described by `destination`.

This function provides a wrapper around the `moveChunk` (page 60). In most circumstances, allow the *balancer* to automatically migrate *chunks*, and avoid calling `sh.moveChunk()` directly.

See Also:

"`moveChunk` (page 60)" and "`http://docs.mongodb.org/manual/sharding`" for more information.

`sh.`**`setBalancerState`**`(`*state*`)`

Parameters

- **state** (*boolean*) – `true` enables the balancer if disabled, and `false` disables the balancer.

Enables or disables the *balancer*. Use `sh.getBalancerState()` to determine if the balancer is currently enabled or disabled and `sh.isBalancerRunning()` to check its current state.

`sh.`**`getBalancerState`**`()`

Returns boolean.

`sh.getBalancerState()` returns `true` when the *balancer* is enabled and false when the balancer is disabled. This does not reflect the current state of balancing operations: use `sh.isBalancerRunning()` to check the balancer's current state.

`sh.`**`isBalancerRunning`**`()`

Returns boolean.

Returns true if the *balancer* process is currently running and migrating chunks and false if the balancer process is not running. Use `sh.getBalancerState()` to determine if the balancer is enabled or disabled.

`sh.`**`status`**`()`

Returns a formatted report of the status of the *sharded cluster*, including data regarding the distribution of chunks.

`sh.`**`addShardTag`**`(`*shard*, *tag*`)`

New in version 2.2.

Parameters

- **shard** – Specifies the name of the shard that you want to give a specific tag.
- **tag** – Specifies the name of the tag that you want to add to the shard.

`sh.addShardTag()` associates a shard with a tag or identifier. MongoDB can use these identifiers, to "home" or attach (i.e. with `sh.addTagRange()`) specific data to a specific shard.

Always issue `sh.addShardTag()` when connected to a `mongos` (page 256) instance. The following example adds three tags, `LGA`, `EWR`, and `JFK`, to three shards:

```
sh.addShardTag("shard0000", "LGA")
sh.addShardTag("shard0001", "EWR")
sh.addShardTag("shard0002", "JFK")
```

`sh.`**`addTagRange`**`(`*namespace*, *minimum*, *maximum*, *tag*`)`

New in version 2.2.

Parameters

- **namespace** – Specifies the namespace, in the form of `<database>.<collection>` of the sharded collection that you would like to tag.

- **minimum** – Specifies the minimum value of the *shard key* range to include in the tag. Specify the minimum value in the form of `<fieldname>:<value>`.
- **maximum** – Specifies the maximum value of the shard key range to include in the tag. Specify the minimum value in the form of `<fieldname>:<value>`.
- **tag** – Specifies the name of the tag to attach the range specified by the `minimum` and `maximum` arguments to.

`sh.addTagRange()` attaches a range of values of the shard key to a shard tag created using the `sh.addShardTag()` helper. Use this operation to ensure that the documents that exist within the specified range exist on shards that have a matching tag.

Always issue `sh.addTagRange()` when connected to a `mongos` (page 256) instance.

`sh.`**`removeShardTag`**`(`*shard*, *tag*`)`

New in version 2.2.

Parameters

- **shard** – Specifies the name of the shard that you want to remove a tag from.
- **tag** – Specifies the name of the tag that you want to remove from the shard.

Removes the association between a tag and a shard. Always issue `sh.removeShardTag()` when connected to a `mongos` (page 256) instance.

`sh.`**`help`**`()`

Returns a basic help text for all sharding related shell functions.

4.2.4 Replica Set Methods

See Also:

`http://docs.mongodb.org/manual/core/replication` for more information regarding replication.

`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` *database command*.

See Also:

"*Replica Set Status Reference* (page 193)" for documentation of this output.

`rs.`**`initiate`**`(`*configuration*`)`

Parameters

- **configuration** – Optional. A *document* that specifies the configuration of a replica set. If not specified, MongoDB will use a default configuration.

Initiates a replica set. Optionally takes a configuration argument in the form of a *document* that holds the configuration of a replica set. Consider the following model of the most basic configuration for a 3-member replica set:

```
{
    _id : <setname>,
     members : [
         {_id : 0, host : <host0>},
         {_id : 1, host : <host1>},
         {_id : 2, host : <host2>},
```

```
        ]
}
```

This function provides a wrapper around the "`replSetInitiate`" *database command*.

`rs.conf()`

Returns a *document* that contains the current *replica set* configuration object.

`rs.config()`

`rs.config()` is an alias of `rs.conf()`.

`rs.reconfig(configuration[, force])`

Parameters

- **configuration** – A *document* that specifies the configuration of a replica set.
- **force** – Optional. Specify `{ force: true }` as the force parameter to force the replica set to accept the new configuration even if a majority of the members are not accessible. Use with caution, as this can lead to *rollback* situations.

Initializes a new *replica set* configuration. This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.reconfig()` provides a wrapper around the "`replSetReconfig`" *database command*.

`rs.reconfig()` overwrites the existing replica set configuration. Retrieve the current configuration object with `rs.conf()`, modify the configuration as needed and then use `rs.reconfig()` to submit the modified configuration object.

To reconfigure a replica set, use the following sequence of operations:

```
conf = rs.conf()

// modify conf to change configuration

rs.reconfig(conf)
```

If you want to force the reconfiguration if a majority of the set isn't connected to the current member, or you're issuing the command against a secondary, use the following form:

```
conf = rs.conf()

// modify conf to change configuration

rs.reconfig(conf, { force: true } )
```

> **Warning:** Forcing a `rs.reconfig()` can lead to *rollback* situations and other difficult to recover from situations. Exercise caution when using this option.

See Also:

"`http://docs.mongodb.org/manual/reference/replica-configuration`" and "`http://docs.mongodb.org/manual/administration/replica-sets`".

`rs.add(hostspec, arbiterOnly)`

Specify one of the following forms:

Parameters

- **host** (*string,document*) – Either a string or a document. If a string, specifies a host (and optionally port-number) for a new host member for the replica set; MongoDB will add this host with the default configuration. If a document, specifies any attributes about a member of a replica set.
- **arbiterOnly** (*Boolean*) – Optional. If `true`, this host is an arbiter. If the second argument evaluates to `true`, as is the case with some *documents*, then this instance will become an arbiter.

Provides a simple method to add a member to an existing *replica set*. You can specify new hosts in one of two ways:

1. as a "hostname" with an optional port number to use the default configuration as in the *replica-set-add-member* example.
2. as a configuration *document*, as in the *replica-set-add-member-alternate-procedure* example.

This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.add()` provides a wrapper around some of the functionality of the "`replSetReconfig`" *database command* and the corresponding shell helper `rs.reconfig()`. See the `http://docs.mongodb.org/manual/reference/replica-configuration` document for full documentation of all replica set configuration options.

Example

To add a `mongod` accessible on the default port `27017` running on the host `mongodb3.example.net`, use the following `rs.add()` invocation:

```
rs.add('mongodb3.example.net:27017')
```

If `mongodb3.example.net` is an arbiter, use the following form:

```
rs.add('mongodb3.example.net:27017', true)
```

To add `mongodb3.example.net` as a *secondary-only* member of set, use the following form of `rs.add()`:

```
rs.add( { "host": "mongodbd3.example.net:27017", "priority": 0 } )
```

See the `http://docs.mongodb.org/manual/reference/replica-configuration` and `http://docs.mongodb.org/manual/administration/replica-sets` documents for more information.

`rs.addArb(hostname)`

Parameters

- **host** (*string*) – Specifies a host (and optionally port-number) for a arbiter member for the replica set.

Adds a new *arbiter* to an existing replica set.

This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.stepDown(seconds)`

Parameters

- **seconds** (*init*) – Specify the duration of this operation. If not specified the command uses the default value of 60 seconds.

Returns disconnects shell.

Forces the current replica set member to step down as *primary* and then attempt to avoid election as primary for the designated number of seconds. Produces an error if the current node is not primary.

This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.stepDown()` provides a wrapper around the *database command* `replSetStepDown`.

`rs.`**`freeze`**`(`*seconds*`)`

Parameters

- **seconds** (*init*) – Specify the duration of this operation.

Forces the current node to become ineligible to become primary for the period specified.

`rs.freeze()` provides a wrapper around the *database command* `replSetFreeze`.

`rs.`**`remove`**`(`*hostname*`)`

Parameters

- **hostname** – Specify one of the existing hosts to remove from the current replica set.

Removes the node described by the `hostname` parameter from the current *replica set*. This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates negotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

Note: Before running the `rs.remove()` 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()`, but it remains good practice.

`rs.`**`slaveOk`**`()`

Provides a shorthand for the following operation:

```
db.getMongo().setSlaveOk()
```

This allows the current connection to allow read operations to run on *secondary* nodes. See the `readPref()` method for more fine-grained control over *read preference* in the `mongo` shell.

`db.`**`isMaster`**`()`

Returns a status document with fields that includes the `ismaster` field that reports if the current node is the *primary* node, as well as a report of a subset of current replica set configuration.

This function provides a wrapper around the *database command* `isMaster`

`rs.`**`help`**`()`

Returns a basic help text for all of the `replication` related shell functions.

`rs.`**`syncFrom`**`()`

New in version 2.2. Provides a wrapper around the `replSetSyncFrom`, 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 sync from in the form of `[hostname]:[port]`.

See `replSetSyncFrom` for more details.

4.3 Native Shell Methods

These methods provide a number of low level and internal functions that may be useful in the context of some advanced operations in the shell. The JavaScript standard library is accessible in the `mongo` shell.

`Date``()`

Returns Current date, as a string.

load (*"file"*)

Para string file Specify a path and file name containing JavaScript.

This native function loads and runs a JavaScript file into the current shell environment. To run JavaScript with the mongo shell, you can either:

- use the "*--eval* (page 218)" option when invoking the shell to evaluate a small amount of JavaScript code, or
- specify a file name with "*mongo* (page 219)". `mongo` will execute the script and then exit. Add the *--shell* (page 218) option to return to the shell after running the command.

Specify files loaded with the `load()` (page 105) function in relative terms to the current directory of the `mongo` shell session. Check the current directory using the "`pwd()` (page 105)" function.

quit ()

Exits the current shell session.

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*.

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.

pwd ()

Returns the current directory.

This function returns with output relative to the current shell session, and does not impact the server.

cd (*"path"*)

Parameters

- **file** (*string*) – Specify a path on the local file system.

Changes the current working directory to the specified path.

This function returns with output relative to the current shell session, and does not impact the server.

Note: This feature is not yet implemented.

---

cat (*"filename"*)

Parameters

- **filename** (*string*) – Specify a path and file name on the local file system.

Returns the contents of the specified file.

This function returns with output relative to the current shell session, and does not impact the server.

md5sumFile (*"filename"*)

Parameters

- **filename** (*string*) – a file name.

Returns The *md5* hash of the specified file.

Note: The specified filename must refer to a file located on the system running the `mongo` shell.

---

mkdir (*"path"*)

Parameters

- **path** (*string*) – A path on the local filesystem.

Creates a directory at the specified path. This command will create the entire path specified, if the enclosing directory or directories do not already exit.

Equivalent to **mkdir -p** with BSD or GNU utilities.

hostname()

Returns The hostname of the system running the mongo shell process.

getHostName()

Returns The hostname of the system running the mongo shell process.

removeFile(*"filename"*)

Parameters

- **filename** (*string*) – Specify a filename or path to a local file.

Returns boolean.

Removes the specified file from the local file system.

fuzzFile(*"filename"*)

Parameters

- **filename** (*string*) – Specify a filename or path to a local file.

Returns null

For internal use.

listFiles()

Returns an array, containing one document per object in the directory. This function operates in the context of the mongo process. The included fields are:

name

Returns a string which contains the name of the object.

isDirectory

Returns true or false if the object is a directory.

size

Returns the size of the object in bytes. This field is only present for files.

4.4 Non-User Functions and Methods

4.4.1 Deprecated Methods

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 53) (also db.resetError()) command.

This method provides a wrapper around the getPrevError (page 53) command.

db.**resetError**()

Deprecated since version 1.6. Resets the error message returned by db.getPrevError or getPrevError (page 53). Provides a wrapper around the resetError (page 53) command.

4.4.2 Native Methods

_srand()

For internal use.

_rand()

Returns A random number between `0` and `1`.

This function provides functionality similar to the `Math.rand()` function from the standard library.

_isWindows()

Returns boolean.

Returns "true" if the server is running on a system that is Windows, or "false" if the server is running on a Unix or Linux systems.

4.4.3 Internal Methods

These methods are accessible in the shell but exist to support other functionality in the environment. Do not call these methods directly.

_startMongoProgram()
For internal use.

runProgram()
For internal use.

run()
For internal use.

runMongoProgram()
For internal use.

stopMongod()
For internal use.

stopMongoProgram()
For internal use.

stopMongoProgramByPid()
For internal use.

rawMongoProgramOutput()
For internal use.

clearRawMongoProgramOutput()
For internal use.

waitProgram()
For internal use.

waitMongoProgramOnPort()
For internal use.

resetDbpath()
For internal use.

copyDbpath()
For internal use.

Part III

Query and Aggregation Operator Reference

CHAPTER

FIVE

OPERATOR REFERENCE

This document contains a reference to all *operators* used with MongoDB in version 2.2.2. See `http://docs.mongodb.org/manual/crud` for a higher level overview of the operations that use these operators, and `operator` for a more condensed index of these operators.

5.1 Query Selectors

5.1.1 Comparison

Note: To express `equal to` (e.g. `=`) in the MongoDB query language, use JSON `{ key:value }` structure. Consider the following prototype:

```
db.collection.find( { field: value } )
```

For example:

```
db.collection.find( { a: 42 } )
```

This query selects all the documents the where the `a` field holds a value of `42`.

$ne

Syntax: `{field: {$ne: value} }`

`$ne` (page 111) 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`.

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 111) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.state": { $ne: "NY" } }, { $set: { qty: 20 } } )
```

This `update()` operation will set the `qty` field value in the documents that contains the embedded document `carrier` whose `state` field value does not equal "NY", or where the `state` field or the `carrier` embedded document does not exist.

See Also:

`find()`, `update()`, `$set` (page 127).

$lt

Syntax: `{field: {$lt: value} }`

`$lt` (page 112) selects the documents where the value of the `field` is less than (i.e. <) the specified `value`.

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 112) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lt: 20 } }, { $set: { price: 9.99 } } )
```

This `update()` 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()`, `update()`, `$set` (page 127).

$lte

Syntax: `{ field: { $lte: value} }`

`$lte` (page 112) selects the documents where the value of the `field` is less than or equal to (i.e. <=) the specified `value`.

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 112) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lte: 5 } }, { $set: { price: 9.99 } } )
```

This `update()` 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()`, `update()`, `$set` (page 127).

$gt

Syntax: `{field: {$gt: value} }`

`$gt` (page 112) selects those documents where the value of the `field` is greater than (i.e. >) the specified `value`.

Consider the following example:

```
db.inventory.find( { qty: { $gt: 20 } } )
```

This query will select all documents in the `inventory` collection where the `qty` field value is greater than `20`.

Consider the following example which uses the `$gt` (page 112) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gt: 2 } }, { $set: { price: 9.99 } } )
```

This `update()` operation will set the value of the `price` field in the documents that contain the embedded document `carrier` whose `fee` field value is greater than `2`.

See Also:

`find()`, `update()`, `$set` (page 127).

$gte

Syntax: `{field: {$gte: value} }`

`$gte` (page 113) selects the documents where the value of the `field` is greater than or equal to (i.e. `>=`) a specified value (e.g. `value`.)

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 113) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gte: 2 } }, { $set: { price: 9.99 } } )
```

This `update()` 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()`, `update()`, `$set` (page 127).

$in

Syntax: `{ field: { $in: [<value1>, <value2>, ... <valueN> ] } }`

`$in` (page 113) selects the documents where the `field` value equals any value in the specified array (e.g. `<value1>`, `<value2>`, etc.)

Consider the following example:

```
db.inventory.find( { qty: { $in: [ 5, 15 ] } } )
```

This query will select to select all documents in the `inventory` collection where the `qty` field value is either `5` or `15`. Although you can express this query using the `$or` (page 116) operator, choose the `$in` (page 113) operator rather than the `$or` (page 116) operator when performing equality checks on the same field.

If the `field` holds an array, then the `$in` (page 113) 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.)

Consider the following example:

```
db.inventory.update( { tags: { $in: ["appliances", "school"] } }, { $set: { sale:true } } )
```

This `update()` operation will set the `sale` field value in the `inventory` collection where the `tags` field holds an array with at least one element matching an element in the array `["appliances", "school"]`.

See Also:

`find()`, `update()`, `$or` (page 116), `$set` (page 127).

`$nin`

Syntax: `{ field: { $nin: [ <value1>, <value2> ... <valueN> ]} }`

`$nin` (page 114) selects the documents where:

- the `field` value is not in the specified `array` **or**
- the `field` does not exist.

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 114) 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()` operation will set the `sale` field value in the `inventory` collection where the `tags` field holds an array with **no** elements matching an element in the array `["appliances", "school"]` or where a document does not contain the `tags` field.

See Also:

`find()`, `update()`, `$set` (page 127).

`$all`

Syntax: `{ field: { $all: [ <value> , <value1> ... ] }`

`$all` (page 114) selects the documents where the `field` holds an array and contains all elements (e.g. `<value>`, `<value1>`, etc.) in the array.

Consider the following example:

```
db.inventory.find( { tags: { $all: [ "appliances", "school", "book" ] } } )
```

This query selects all documents in the `inventory` collection where the `tags` field contains an array with the elements, `appliances`, `school`, and `technology`.

Therefore, the above query will match documents in the `inventory` collection that have a `tags` field that hold *either* of the following arrays:

```
[ "school", "book", "bag", "headphone", "appliances" ]
[ "appliances", "school", "book" ]
```

The `$all` (page 114) operator exists to describe and specify arrays in MongoDB queries. However, you may use the `$all` (page 114) operator to select against a non-array `field`, as in the following example:

```
db.inventory.find( { qty: { $all: [ 50 ] } } )
```

However, use the following form to express the same query:

```
db.inventory.find( { qty: 50 } )
```

Both queries will select all documents in the `inventory` collection where the value of the `qty` field equals `50`.

Note: In most cases, MongoDB does not treat arrays as sets. This operator provides a notable exception to this approach.

In the current release queries that use the `$all` (page 114) operator must scan all the documents that match the first element in the query array. As a result, even with an index to support the query, the operation may be long running, particularly when the first element in the array is not very selective.

See Also:

`find()`, `update()`, and `$set` (page 127).

You may combine comparison operators to specify ranges:

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

This statement returns all documents with `field` between `value1` and `value2`.

Note: Fields containing arrays match conditional operators, if only one item matches. Therefore, the following query:

```
db.collection.find( { field: { $gt:0, $lt:2 } } );
```

Will match a document that contains the following field:

```
{ field: [-1,3] }
```

5.1.2 Logical

`$and`

New in version 2.0. *Syntax*: `{ $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }`

`$and` (page 115) performs a logical `AND` operation on an array of *two or more* expressions (e.g. `<expression1>`, `<expression2>`, etc.) and selects the documents that satisfy *all* the expressions in the array. The `$and` (page 115) operator uses *short-circuit evaluation*. If the first expression (e.g. `<expression1>`) evaluates to `false`, MongoDB will not evaluate the remaining expressions.

Consider the following example:

```
db.inventory.find({ $and: [ { price: 1.99 }, { qty: { $lt: 20 } }, { sale: true } ] } )
```

This query will select all documents in the `inventory` collection where:

- `price` field value equals `1.99` **and**
- `qty` field value is less than `20` **and**
- `sale` field value is equal to `true`.

MongoDB provides an implicit `AND` operation when specifying a comma separated list of expressions. For example, you may write the above query as:

```
db.inventory.find( { price: 1.99, qty: { $lt: 20 } , sale: true } )
```

If, however, a query requires an `AND` operation on the same field such as `{ price: { $ne: 1.99 } } AND { price: { $exists: true } }`, then either use the `$and` (page 115) operator for the two separate expressions or combine the operator expressions for the field `{ price: { $ne: 1.99, $exists: true } }`.

Consider the following examples:

```
db.inventory.update( { $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] }, { $se

db.inventory.update( { price: { $ne: 1.99, $exists: true } } , { $set: { qty: 15 } } )
```

Both `update()` operations will set the value of the `qty` field in documents where:

- the `price` field value does not equal `1.99` **and**
- the `price` field exists.

See Also:

`find()`, `update()`, `$ne` (page 111), `$exists` (page 119), `$set` (page 127).

`$or`

New in version 1.6.Changed in version 2.0: You may nest `$or` (page 116) operations; however, these expressions are not as efficiently optimized as top-level. *Syntax*: `{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }`

The `$or` (page 116) 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>`.

Consider the following query:

```
db.inventory.find( { price:1.99, $or: [ { qty: { $lt: 20 } }, { sale: true } ] } )
```

This query will select all documents in the `inventory` collection where:

- the `price` field value equals `1.99` *and*
- either the `qty` field value is less than `20` **or** the `sale` field value is `true`.

Consider the following example which uses the `$or` (page 116) operator to select fields from embedded documents:

```
db.inventory.update( { $or: [ { price:10.99 }, { "carrier.state": "NY"} ] }, { $set: { sale: tru
```

This `update()` operation will set the value of the `sale` field in the documents in the `inventory` collection where:

- the `price` field value equals `10.99` **or**
- the `carrier` embedded document contains a field `state` whose value equals `NY`.

When using `$or` (page 116) with `<expressions>` that are equality checks for the value of the same field, choose the `$in` (page 113) operator over the `$or` (page 116) operator.

Consider the query to select all documents in the `inventory` collection where:

- either `price` field value equals `1.99` *or* the `sale` field value equals `true`, **and**
- either `qty` field value equals `20` *or* `qty` field value equals `50`,

The most effective query would be:

```
db.inventory.find ( { $or: [ { price: 1.99 }, { sale: true } ], qty: { $in: [20, 50] } } )
```

Consider the following behaviors when using the `$or` (page 116) operator:

- When using indexes with `$or` (page 116) queries, remember that each clause of an `$or` (page 116) query will execute in parallel. These clauses can each use their own index. Consider the following query:

  ```
  db.inventory.find ( { $or: [ { price: 1.99 }, { sale: true } ] } )
  ```

  For this query, you would create one index on `price` (`db.inventory.ensureIndex( { price: 1 } )`) and another index on `sale` (`db.inventory.ensureIndex( { sale: 1 } )`) rather than a compound index.
- Also, when using the `$or` (page 116) operator with the `sort()` method in a query, the query will **not** use the indexes on the `$or` (page 116) fields. Consider the following query which adds a `sort()` method to the above query:

  ```
  db.inventory.find ( { $or: [ { price: 1.99 }, { sale: true } ] } ).sort({item:1})
  ```

  This modified query will not use the index on `price` nor the index on `sale`.
- You cannot use the `$or` (page 116) with `2d geospatial queries`.

See Also:

`find()`, `update()`, `$set` (page 127), `$and` (page 115), `sort()`.

$nor

Syntax: `{ $nor: [ { <expression1> }, { <expression2> }, ... { <expressionN> } ] }`

`$nor` (page 117) performs a logical `NOR` operation on an array of *two or more* `<expressions>` and selects the documents that **fail** all the `<expressions>` in the array.

Consider the following example:

```
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 `$nor` (page 117) expression is when the `$nor` (page 117) operator is used with the `$exists` (page 119) operator.

Consider the following query which uses only the `$nor` (page 117) 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

Compare that with the following query which uses the `$nor` (page 117) operator with the `$exists` (page 119) 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`

See Also:

`find()`, `update()`, `$set` (page 127), `$exists` (page 119).

$not

Syntax: `{ field: { $not: { <operator-expression> } } }`

`$not` (page 118) 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 112) operator. `{ $lt: 1.99 }` returns *only* the documents where `price` field exists and its value is less than or equal to `1.99`.

Remember that the `$not` (page 118) operator only affects *other operators* and cannot check fields and documents independently. So, use the `$not` (page 118) operator for logical disjunctions and the `$ne` (page 111) operator to test the contents of fields directly.

Consider the following behaviors when using the `$not` (page 118) operator:

- The operation of the `$not` (page 118) operator is consistent with the behavior of other operators but may yield unexpected results with some data types like arrays.
- The `$not` (page 118) operator does **not** support operations with the `$regex` (page 122) operator. Instead use `http://docs.mongodb.org/manual//` or in your driver interfaces, use your language's regular expression capability to create regular expression objects.

  Consider the following example which uses the pattern match expression `http://docs.mongodb.org/manual//`:

  ```
  db.inventory.find( { item: { $not: /^p.*/ } } )
  ```

  The query will select all documents in the `inventory` collection where the `item` field value does *not* start with the letter `p`.

  If using PyMongo's `re.compile()`, you can write the above query as:

  ```
  import re
  for noMatch in db.inventory.find( { "item": { "$not": re.compile("^p.*") } } ):
      print noMatch
  ```

See Also:

`find()`, `update()`, `$set` (page 127), `$gt` (page 112), `$regex` (page 122), PyMongo, *driver*.

5.1.3 Element

`$exists`

Syntax: `{ field: { $exists: <boolean> } }`

`$exists` (page 119) selects the documents that contain the field if `<boolean>` is `true`. If `<boolean>` is `false`, the query only returns the documents that do not contain the field. Documents that contain the field but has the value `null` are *not* returned.

MongoDB *$exists* does **not** correspond to SQL operator `exists`. For SQL `exists`, refer to the `$in` (page 113) operator.

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 either `5` nor `15`.

See Also:

- `find()`
- `$nin` (page 114)
- `$and` (page 115)
- `$in` (page 113)
- *faq-developers-query-for-nulls*

`$type`

Syntax: `{ field: { $type: <BSON type> } }`

`$type` (page 119) selects the documents where the *value* of the `field` is the specified *BSON* type.

Consider the following example:

```
db.inventory.find( { price: { $type : 1 } } )
```

This query will select all documents in the `inventory` collection where the `price` field value is a Double.

If the `field` holds an array, the `$type` (page 119) operator performs the type check against the array elements and **not** the `field`.

Consider the following example where the `tags` field holds an array:

```
db.inventory.find( { tags: { $type : 4 } } )
```

This query will select all documents in the `inventory` collection where the `tags` array contains an element that is itself an array.

If instead you want to determine whether the `tags` field is an array type, use the `$where` (page 121) operator:

```
db.inventory.find( { $where : "Array.isArray(this.tags)" } )
```

See the SERVER-1475 for more information about the array type.

Refer to the following table for the available *BSON* types and their corresponding numbers.

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

`MinKey` and `MaxKey` compare less than and greater than all other possible *BSON* element values, respectively, and exist primarily for internal use.

Note: To query if a field value is a `MinKey`, you must use the `$type` (page 119) with `-1` as in the following example:

```
db.collection.find( { field: { $type: -1 } } )
```

Example

Consider the following example operation sequence that demonstrates both type comparison *and* the special `MinKey` and `MaxKey` values:

```
db.test.insert( {x : 3});
db.test.insert( {x : 2.9} );
db.test.insert( {x : new Date()} );
db.test.insert( {x : true } );
db.test.insert( {x : MaxKey } )
db.test.insert( {x : MinKey } )

db.test.find().sort({x:1})
{ "_id" : ObjectId("4b04094b7c65b846e2090112"), "x" : { $minKey : 1 } }
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Jul 25 2012 18:42:03 GMT-0500 (EST)"
{ "_id" : ObjectId("4b0409487c65b846e2090111"), "x" : { $maxKey : 1 } }
```

To query for the minimum value of a *shard key* of a *sharded cluster*, use the following operation when connected to the `mongos` (page 256):

```
use config
db.chunks.find( { "min.shardKey": { $type: -1 } } )
```

> **Warning:** Storing values of the different types in the same field in a collection is *strongly* discouraged.

See Also:

find(), insert(), $where (page 121), *BSON*, *shard key*, *sharded cluster* .

$mod

Syntax: { field: { $mod: [divisor, remainder]} }

$mod (page 121) selects the documents where the field value divided by the divisor has the specified remainder.

Consider the following example:

```
db.inventory.find( { qty: { $mod: [ 4, 0 ] } } )
```

This query will select all documents in the inventory collection where the qty field value modulo 4 equals 0, such as documents with qty value equal to 0 or 12.

In some cases, you can query using the $mod (page 121) operator rather than the more expensive $where (page 121) operator. Consider the following example using the $mod (page 121) operator:

```
db.inventory.find( { qty: { $mod: [ 4, 0 ] } } )
```

The above query is less expensive than the following query which uses the $where (page 121) operator:

```
db.inventory.find( { $where: "this.qty % 4 == 0" } )
```

See Also:

find(), update(), $set (page 127).

5.1.4 JavaScript

$where

Use the $where (page 121) operator to pass either a string containing a JavaScript expression or a full JavaScript function to the query system. The $where (page 121) 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 .

> **Warning:**
> - Do not write to the database within the $where (page 121) JavaScript function.
> - $where (page 121) 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 112), $in (page 113)).
> - In general, you should use $where (page 121) only when you can't express your query using another operator. If you must use $where (page 121), try to include at least one other standard query operator to filter the result set. Using $where (page 121) alone requires a table scan.

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 121) 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.de
```

You can include both the standard MongoDB operators and the `$where` (page 121) 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;
```

Using normal non-`$where` (page 121) query statements provides the following performance advantages:

- MongoDB will evaluate non-`$where` (page 121) components of query before `$where` (page 121) statements. If the non-`$where` (page 121) statements match no documents, MongoDB will not perform any query evaluation using `$where` (page 121).
- The non-`$where` (page 121) query statements may use an *index*.

$regex

The `$regex` (page 122) operator provides regular expression capabilities in queries. MongoDB uses Perl compatible regular expressions (i.e. "PCRE."))The following examples are equivalent:

```
db.collection.find( { field: /acme.*corp/i } );
db.collection.find( { field: { $regex: 'acme.*corp', $options: 'i' } } );
```

These expressions match all documents in `collection` where the value of `field` matches the case-insensitive regular expression `acme.*corp`.

`$regex` (page 122) uses "Perl Compatible Regular Expressions" (PCRE) as the matching engine.

$options

`$regex` (page 122) provides four option flags:

- `i` toggles case insensitivity, and allows all letters in the pattern to match upper and lower cases.
- `m` toggles multiline regular expression. Without this option, all regular expression match within one line.

  If there are no newline characters (e.g. `\n`) or no start/end of line construct, the `m` option has no effect.
- `x` toggles an "extended" capability. When set, `$regex` (page 122) ignores all white space characters unless escaped or included in a character class.

  Additionally, it ignores characters between an un-escaped `#` 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.)

New in version 1.9.0.

- `s` allows the dot (e.g. `.`) character to match all characters *including* newline characters.

`$regex` (page 122) only provides the `i` and `m` options in the short JavaScript syntax (i.e. `http://docs.mongodb.org/manual/acme.*corp/i`). To use `x` and `s` you must use the "`$regex` (page 122)" operator with the "`$options` (page 122)" syntax.

To combine a regular expression match with other operators, you need to specify the "`$regex` (page 122)" operator. For example:

```
db.collection.find( { field: $regex: /acme.*corp/i, $nin: [ 'acmeblahcorp' } );
```

This expression returns all instances of `field` in `collection` that match the case insensitive regular expression `acme.*corp` that *don't* match `acmeblahcorp`.

`$regex` (page 122) uses *indexes* only when the regular expression has an anchor for the beginning (i.e. `^`) of a string. Additionally, while `http://docs.mongodb.org/manual/^a/`, `http://docs.mongodb.org/manual/^a.*/`, and `http://docs.mongodb.org/manual/^a.*$/` are equivalent, they have different performance characteristics. All of these expressions use an index if an appropriate index exists; however, `http://docs.mongodb.org/manual/^a.*/`, and `http://docs.mongodb.org/manual/^a.*$/` are slower. `http://docs.mongodb.org/manual/^a/` can stop scanning after matching the prefix.

5.1.5 Geospatial

$near

The `$near` (page 123) operator takes an argument, coordinates in the form of `[x, y]`, and returns a list of objects sorted by distance from those coordinates. See the following example:

```
db.collection.find( { location: { $near: [100,100] } } );
```

This query will return 100 ordered records with a `location` field in `collection`. Specify a different limit using the `cursor.limit()`, or another *geolocation operator* (page 123), or a non-geospatial operator to limit the results of the query.

Note: Specifying a batch size (i.e. `batchSize()`) in conjunction with queries that use the `$near` (page 123) is not defined. See SERVER-5236 for more information.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$within

The `$within` (page 123) operator allows you to select items that exist within a shape on a coordinate system for *geospatial* queries. This operator uses the following syntax:

```
db.collection.find( { location: { $within: { shape } } } );
```

Replace `{ shape }` with a document that describes a shape. The `$within` (page 123) command supports three shapes. These shapes and the relevant expressions follow:

- Rectangles. Use the `$box` (page 124) operator, consider the following variable and `$within` (page 123) document:

```
db.collection.find( { location: { $within: { $box: [[100,0], [120,100]] } } } );
```

Here a box, `[[100,120], [100,0]]` describes the parameter for the query. As a minimum, you must specify the lower-left and upper-right corners of the box.

- Circles. Use the `$center` (page 124) operator. Specify circles in the following form:

```
db.collection.find( { location: { $within: { $center: [ center, radius } } } );
```

- Polygons. Use the `$polygon` (page 124) operator. Specify polygons with an array of points. See the following example:

```
db.collection.find( { location: { $within: { $polygon: [[100,120], [100,100], [120,100], [2
```

The last point of a polygon is implicitly connected to the first point.

All shapes include the border of the shape as part of the shape, although this is subject to the imprecision of floating point numbers.

Use `$uniqueDocs` (page 125) to control whether documents with many location fields show up multiple times when more than one of its fields match the query.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$box

New in version 1.4. The `$box` (page 124) operator specifies a rectangular shape for the `$within` (page 123) operator in *geospatial* queries. To use the `$box` (page 124) operator, you must specify the bottom left and top right corners of the rectangle in an array object. Consider the following example:

```
db.collection.find( { loc: { $within: { $box: [ [0,0], [100,100] ] } } } )
```

This will return all the documents that are within the box having points at: `[0,0]`, `[0,100]`, `[100,0]`, and `[100,100]`.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$polygon

New in version 1.9. Use `$polygon` (page 124) to specify a polygon for a bounded query using the `$within` (page 123) operator for *geospatial* queries. To define the polygon, you must specify an array of coordinate points, as in the following:

[[x1,y1], [x2,y2], [x3,y3]]

The last point specified is always implicitly connected to the first. You can specify as many points, and therefore sides, as you like. Consider the following bounded query for documents with coordinates within a polygon:

```
db.collection.find( { loc: { $within: { $polygon: [ [0,0], [3,6], [6,0]  ] } } } )
```

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$center

New in version 1.4. This specifies a circle shape for the `$within` (page 123) operator in *geospatial* queries. To define the bounds of a query using `$center` (page 124), you must specify:

- the center point, and
- the radius

Considering the following example:

```
db.collection.find( { location: { $within: { $center: [ [0,0], 10 ] } } } );
```

The above command returns all the documents that fall within a 10 unit radius of the point `[0,0]`.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$uniqueDocs

New in version 2.0. For *geospatial* queries, MongoDB may return a single document more than once for a single query, because geospatial indexes may include multiple coordinate pairs in a single document, and therefore return the same document more than once.

The `$uniqueDocs` (page 125) operator inverts the default behavior of the `$within` (page 123) operator. By default, the `$within` (page 123) operator returns the document only once. If you specify a value of `false` for `$uniqueDocs` (page 125), MongoDB will return multiple instances of a single document.

Example

Given an `addressBook` collection with a document in the following form:

```
{ addresses: [ { name: "Home", loc: [55.5, 42.3] }, { name: "Work", loc: [32.3, 44.2] } ] }
```

The following query would return the same document multiple times:

```
db.addressBook.find( { "addresses.loc": { "$within": { "$box": [ [0,0], [100,100] ], $uniqu
```

The following query would return each matching document, only once:

```
db.addressBook.find( { "address.loc": { "$within": { "$box": [ [0,0], [100,100] ], $uniqueD
```

You cannot specify `$uniqueDocs` (page 125) with `$near` (page 123) or haystack queries.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$maxDistance

The `$maxDistance` (page 125) operator specifies an upper bound to limit the results of a geolocation query. See below, where the `$maxDistance` (page 125) operator narrows the results of the `$near` (page 123) query:

```
db.collection.find( { location: { $near: [100,100], $maxDistance: 10 } } );
```

This query will return documents with `location` fields from `collection` that have values with a distance of 5 or fewer units from the point `[100,100]`. `$near` (page 123) returns results ordered by their distance from `[100,100]`. This operation will return the first 100 results unless you modify the query with the `cursor.limit()` method.

Specify the value of the `$maxDistance` (page 125) argument in the same units as the document coordinate system.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$nearSphere

New in version 1.8. The $nearSphere (page 126) operator is the spherical equivalent of the $near (page 123) operator. $nearSphere (page 126) returns all documents near a point, calculating distances using spherical geometry.

```
db.collection.find( { loc: { $nearSphere: [0,0] } } )
```

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$centerSphere

New in version 1.8. The $centerSphere (page 126) operator is the spherical equivalent of the $center (page 124) operator. $centerSphere (page 126) uses spherical geometry to calculate distances in a circle specified by a point and radius.

Considering the following example:

```
db.collection.find( { loc: { $centerSphere: { [0,0], 10 / 3959 } } } )
```

This query will return all documents within a 10 mile radius of `[0,0]` using a spherical geometry to calculate distances.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

5.1.6 Array

$size

The $size (page 126) 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 126) with a value of 1, as follows:

```
db.collection.find( { field: { $size: 1 } } );
```

$size (page 126) 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 126) portion of a query, although the other portions of a query can use indexes if applicable.

$elemMatch

New in version 1.4. The `$elemMatch` operator matches more than one component within an array element. For example,

```
db.collection.find( { array: { $elemMatch: { value1: 1, value2: { $gt: 1 } } } } );
```

returns all documents in `collection` where the array `array` satisfies all of the conditions in the `$elemMatch` expression, or where the value of `value1` is 1 and the value of `value2` is greater than 1. Matching arrays must have at least one element that matches all specified criteria. Therefore, the following document would not match the above query:

```
{ array: [ { value1:1, value2:0 }, { value1:2, value2:2 } ] }
```

while the following document would match this query:

```
{ array: [ { value1:1, value2:0 }, { value1:1, value2:2 } ] }
```

5.2 Update

5.2.1 Fields

$set

Use the `$set` (page 127) operator to set a particular value. The `$set` (page 127) operator requires the following syntax:

```
db.collection.update( { field: value1 }, { $set: { field1: value2 } } );
```

This statement updates in the document in `collection` where `field` matches `value1` by replacing the value of the field `field1` with `value2`. This operator will add the specified field or fields if they do not exist in this document *or* replace the existing value of the specified field(s) if they already exist.

$unset

The `$unset` (page 127) operator deletes a particular field. Consider the following example:

```
db.collection.update( { field: value1 }, { $unset: { field1: "" } } );
```

The above example deletes `field1` in `collection` from documents where `field` has a value of `value1`. The value of specified for the value of the field in the `$unset` (page 127) statement (i.e. `""` above,) does not impact the operation.

If documents match the initial query (e.g. `{ field: value1 }` above) but do not have the field specified in the `$unset` (page 127) operation, (e.g. `field1`) there the statement has no effect on the document.

$inc

The `$inc` (page 127) operator increments a value by a specified amount if field is present in the document. If the field does not exist, `$inc` (page 127) sets field to the number value. For example:

```
db.collection.update( { field: value }, { $inc: { field1: amount } } );
```

In this example, for documents in `collection` where `field` has the value `value`, the value of `field1` increments by the value of `amount`. The above operation only increments the *first* matching document *unless* you specify multi-update:

```
db.collection.update( { age: 20 }, { $inc: { age: 1 } } );
db.collection.update( { name: "John" }, { $inc: { age: 1 } } );
```

In the first example all documents that have an `age` field with the value of `20`, the operation increases `age` field by one. In the second example, in all documents where the `name` field has a value of `John` the operation increases the value of the `age` field by one.

`$inc` (page 127) accepts positive and negative incremental amounts.

`$rename`

New in version 1.7.2. *Syntax*: `{$rename: { <old name1>: <new name1>, <old name2>: <new name2>, ... } }`

The `$rename` (page 128) operator updates the name of a field. The new field name must differ from the existing field name.

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`.

If the document already has a field with the *new* field name, the `$rename` (page 128) operator removes that field and renames the field with the *old* field name to the *new* field name.

The `$rename` (page 128) operator will expand arrays and sub-documents to find a match for field names. When renaming a field in a sub-document to another sub-document or to a regular field, the sub-document itself remains.

Consider the following examples involving the sub-document of the following document:

```
{ "_id": 1,
  "alias": [ "The American Cincinnatus", "The American Fabius" ],
  "mobile": "555-555-5555",
  "nmae": { "first" : "george", "last" : "washington" }
}
```

- To rename a sub-document, call the `$rename` (page 128) operator with the name of the sub-document as you would any other field:

  ```
  db.students.update( { _id: 1 }, { $rename: { "nmae": "name" } } )
  ```

  This operation renames the sub-document `nmae` to `name`:

  ```
  { "_id": 1,
    "alias": [ "The American Cincinnatus", "The American Fabius" ],
    "mobile": "555-555-5555",
    "name": { "first" : "george", "last" : "washington" }
  }
  ```

- To rename a field within a sub-document, call the `$rename` (page 128) operator using the dot notation to refer to the field. Include the name of the sub-document in the new field name to ensure the field remains in the sub-document:

```
db.students.update( { _id: 1 }, { $rename: { "name.first": "name.fname" } } )
```

This operation renames the sub-document field `first` to `fname`:

```
{ "_id" : 1,
  "alias" : [ "The American Cincinnatus", "The American Fabius" ],
  "mobile" : "555-555-5555",
  "name" : { "fname" : "george", "last" : "washington" }
}
```

- To rename a field within a sub-document and move it to another sub-document, call the `$rename` (page 128) operator using the dot notation to refer to the field. Include the name of the new sub-document in the `new name`:

  ```
  db.students.update( { _id: 1 }, { $rename: { "name.last": "contact.lname" } } )
  ```

  This operation renames the sub-document field `last` to `lname` and moves it to the sub-document `contact`:

  ```
  { "_id" : 1,
    "alias" : [ "The American Cincinnatus", "The American Fabius" ],
    "contact" : { "lname" : "washington" },
    "mobile" : "555-555-5555",
    "name" : { "fname" : "george" }
  }
  ```

  If the new field name does not include a sub-document name, the field moves out of the subdocument and becomes a regular document field.

Consider the following behavior when the specified old field name does not exist:

- When renaming a single field and the existing field name refers to a non-existing field, the `$rename` (page 128) 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`.
- When renaming multiple fields and **all** of the old field names refer to non-existing fields, the `$rename` (page 128) operator does nothing, as in the following:

  ```
  db.students.update( { _id: 1 }, { $rename: { 'wife': 'spouse',
                                               'vice': 'vp',
                                               'office': 'term' } } )
  ```

  This operation does nothing because there are no fields named `wife`, `vice`, and `office`.
- When renaming multiple fields and **some** but not all old field names refer to non-existing fields, the `$rename` (page 128) operator performs the following operations: Changed in version 2.2.

  - Renames the fields that exist to the specified new field names.
  - Ignores the non-existing fields.

  Consider the following query that renames both an existing field `mobile` and a non-existing field `wife`. The field named `wife` does not exist and `$rename` (page 128) sets the field to a name that already exists `alias`.

  ```
  db.students.update( { _id: 1 }, { $rename: { 'wife': 'alias',
                                               'mobile': 'cell' } } )
  ```

  This operation renames the `mobile` field to `cell`, and has no other impact action occurs.

```
{ "_id" : 1,
  "alias" : [ "The American Cincinnatus", "The American Fabius" ],
  "cell" : "555-555-5555",
  "name" : { "lname" : "washington" },
  "places" : { "d" : "Mt Vernon", "b" : "Colonial Beach" }
}
```

Note: Before version 2.2, when renaming multiple fields and only some (but not all) old field names refer to non-existing fields:

- For the fields with the old names that do exist, the `$rename` (page 128) operator renames these fields to the specified new field names.
- For the fields with the old names that do **not** exist:
  - **if** no field exists with the new field name, the `$rename` (page 128) operator does nothing.
  - **if** fields already exist with the new field names, the `$rename` (page 128) operator drops these fields.

Consider the following operation that renames both the field `mobile`, which exists, and the field `wife`, which does not exist. The operation tries to set the field named `wife` to `alias`, which is the name of an existing field:

```
db.students.update( { _id: 1 }, { $rename: { 'wife': 'alias', 'mobile': 'cell' } } )
```

Before 2.2, the operation renames the field `mobile` to `cell` *and* drops the `alias` field even though the field `wife` does not exist:

```
{ "_id" : 1,
  "cell" : "555-555-5555",
  "name" : { "lname" : "washington" },
  "places" : { "d" : "Mt Vernon", "b" : "Colonial Beach" }
}
```

5.2.2 Array

$

Syntax: `{ "<array>.$" : value }`

The positional `$` (page 130) operator identifies an element in an `array` field to update without explicitly specifying the position of the element in the array. The positional `$` (page 130) operator, when used with the `update()` method and acts as a placeholder for the **first match** of the update `query selector`:

```
db.collection.update( { <query selector> }, { <update operator>: { "array.$" : value } } )
```

The `array` field **must** appear as part of the `query selector`.

Consider the following 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 130) 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 130) operator acts as a placeholder for the **first match** of the update `query selector`.

The positional `$` (page 130) operator facilitates updates to arrays that contain embedded documents. Use the positional `$` (page 130) operator to access the fields in the embedded documents with the dot notation on the `$` (page 130) operator.

```
db.collection.update( { <query selector> }, { <update operator>: { "array.$.field" : value } } )
```

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 } ] }
```

Use the positional `$` (page 130) 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 } } )
```

Consider the following behaviors when using the positional `$` (page 130) operator:

- Do not use the positional operator `$` (page 130) with *upsert* operations because, inserts will use the `$` as a field name in the inserted document.
- When used with the `$unset` (page 127) operator, the positional `$` (page 130) operator does not remove the matching element from the array but rather sets it to `null`.

See Also:

`update()`, `$set` (page 127) and `$unset` (page 127)

`$push`

The `$push` (page 131) operator appends a specified value to an array. For example:

```
db.collection.update( { field: value }, { $push: { field: value1 } } );
```

Here, `$push` (page 131) appends `value1` to the array identified by `value` in `field`. Be aware of the following behaviors:

- If the field specified in the `$push` (page 131) statement (e.g. `{ $push: { field: value1 } }`) does not exist in the matched document, the operation adds a new array with the specified field and value (e.g. `value1`) to the matched document.
- The operation will fail if the field specified in the `$push` (page 131) statement is *not* an array. `$push` (page 131) does not fail when pushing a value to a non-existent field.
- If `value1` is an array itself, `$push` (page 131) appends the whole array as an element in the identified array. To add multiple items to an array, use `$pushAll` (page 131).

`$pushAll`

The `$pushAll` (page 131) operator is similar to the `$push` (page 131) but adds the ability to append several values to an array at once.

```
db.collection.update( { field: value }, { $pushAll: { field1: [ value1, value2, value3 ] } } );
```

Here, `$pushAll` (page 131) appends the values in `[ value1, value2, value3 ]` to the array in `field1` in the document matched by the statement `{ field: value }` in `collection`.

If you specify a single value, `$pushAll` (page 131) will behave as `$push` (page 131).

$addToSet

The `$addToSet` (page 132) operator adds a value to an array only *if* the value is *not* in the array already. If the value *is* in the array, `$addToSet` (page 132) returns without modifying the array. Otherwise, `$addToSet` (page 132) behaves the same as `$push` (page 131). Consider the following example:

```
db.collection.update( { field: value }, { $addToSet: { field: value1 } } );
```

Here, `$addToSet` (page 132) appends `value1` to the array stored in `field`, *only if* `value1` is not already a member of this array.

$each (page 132) operator is only used with the `$addToSet` (page 132) see the documentation of `http://docs.mongodb.org/manual/reference/operator/addToSet` for more information.

$each

The `$each` (page 132) operator is available within the `$addToSet` (page 132), which allows you to add multiple values to the array if they do not exist in the `field` array in a single operation. Consider the following prototype:

```
db.collection.update( { field: value }, { $addToSet: { field: { $each : [ value1, value2, v
```

$pop

The `$pop` (page 132) operator removes the first or last element of an array. Pass `$pop` (page 132) a value of `1`` to remove the last element in an array and a value of `-1` to remove the first element of an array. Consider the following syntax:

```
db.collection.update( {field: value }, { $pop: { field: 1 } } );
```

This operation removes the last item of the array in `field` in the document that matches the query statement `{ field: value }`. The following example removes the *first* item of the same array:

```
db.collection.update( {field: value }, { $pop: { field: -1 } } );
```

Be aware of the following `$pop` (page 132) behaviors:

- The `$pop` (page 132) operation fails if `field` is not an array.
- `$pop` (page 132) will successfully remove the last item in an array. `field` will then hold an empty array.

New in version 1.1.

$pull

The `$pull` (page 132) operator removes all instances of a value from an existing array. Consider the following example:

```
db.collection.update( { field: value }, { $pull: { field: value1 } } );
```

`$pull` (page 132) removes the value `value1` from the array in `field`, in the document that matches the query statement `{ field: value }` in `collection`. If `value1` existed multiple times in the `field` array, `$pull` (page 132) would remove all instances of `value1` in this array.

$pullAll

The `$pullAll` (page 132) operator removes multiple values from an existing array. `$pullAll` (page 132) provides the inverse operation of the `$pushAll` (page 131) operator. Consider the following example:

```
db.collection.update( { field: value }, { $pullAll: { field1: [ value1, value2, value3 ] } } );
```

Here, `$pullAll` (page 132) removes `[ value1, value2, value3 ]` from the array in `field1`, in the document that matches the query statement `{ field: value }` in `collection`.

5.2.3 Bitwise

`$bit`

The `$bit` (page 133) operator performs a bitwise update of a field. Only use this with integer fields. For example:

```
db.collection.update( { field: 1 }, { $bit: { field: { and: 5 } } } );
```

Here, the `$bit` (page 133) operator updates the integer value of the field named `field` with a bitwise `and: 5` operation. This operator only works with number types.

5.2.4 Isolation

`$atomic`

`$atomic` (page 133) isolation operator **isolates** a write operation that affect multiple documents from other write operations.

Note: The `$atomic` (page 133) isolation operator does **not** provide "all-or-nothing" atomicity for write operations.

Consider the following example:

```
db.foo.update( { field1 : 1 , $atomic : 1 }, { $inc : { field2 : 1 } } , { multi: true } )
```

Without the `$atomic` (page 133) operator, multi-updates will allow other operations to interleave with this updates. If these interleaved operations contain writes, the update operation may produce unexpected results. By specifying `$atomic` (page 133) you can guarantee isolation for the entire multi-update.

See Also:

See `db.collection.update()` for more information about the `db.collection.update()` method.

5.3 Projection

`$slice`

The `$slice` operator controls the number of items of an array that a query returns. 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` accepts arguments in a number of formats, including negative values and arrays. Consider the following examples:

```
db.posts.find( {}, { comments: { $slice: 5 } } )
```

Here, `$slice` 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. 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.

CHAPTER

SIX

META QUERY OPERATORS

6.1 Introduction

In addition to the *MongoDB Query Operators* (page 111), there are a number of "meta" operators that you can modify the output or behavior of a query. On the server, MongoDB treats the query and the options as a single object. The `mongo` shell and driver interfaces may provide *cursor methods* (page 70) that wrap these options. When possible, use these methods; otherwise, you can add these options using either of the following syntax:

```
db.collection.find( { <query> } )._addSpecial( <option> )
db.collection.find( { $query: { <query> }, <option> } )
```

6.2 Modifiers

Many of these operators have corresponding *methods in the shell* (page 70). These methods provide a straightforward and user-friendly interface and are the preferred way to add these options.

`$returnKey`

Only return the index key or keys for the results of the query. If `$returnKey` is set to `true` and the query does not use an index to perform the read operation, the returned documents will not contain any fields. Use one of the following forms:

```
db.collection.find( { <query> } )._addSpecial( "$returnKey", true )
db.collection.find( { $query: { <query> }, $returnKey: true } )
```

`$maxScan`

Constrains the query to only scan the specified number of documents when fulfilling the query. Use one of the following forms:

```
db.collection.find( { <query> } )._addSpecial( "$maxScan" , <number> )
db.collection.find( { $query: { <query> }, $maxScan: <number> } )
```

Use this modifier to prevent potentially long running queries from disrupting performance by scanning through too much data.

`$showDiskLoc`

`$showDiskLoc` option adds a field `$diskLoc` to the returned documents. The `$diskLoc` field contains the disk location information.

The `mongo` shell provides the `cursor.showDiskLoc()` method:

```
db.collection.find().showDiskLoc()
```

You can also specify the option in either of the following forms:

```
db.collection.find( { <query> } )._addSpecial("$showDiskLoc" , true)
db.collection.find( { $query: { <query> }, $showDiskLoc: true } )
```

$comment

The `$comment` makes it possible to attach a comment to a query. Because these comments propagate to the `profile` log, adding `$comment` modifiers can make your profile data much easier to interpret and trace. Use one of the following forms:

```
db.collection.find( { <query> } )._addSpecial( "$comment", <comment> )
db.collection.find( { $query: { <query> }, $comment: <comment> } )
```

$max

Specify a `$max` value to specify the *exclusive* upper bound for a specific index in order to constrain the results of `find()`. The `mongo` shell provides the `cursor.max()` wrapper method:

```
db.collection.find( { <query> } ).max( { field1: <max value>, ... fieldN: <max valueN> } )
```

You can also specify the option with either of the two forms:

```
db.collection.find( { <query> } )._addSpecial( "$max", { field1: <max value1>, ... fieldN: <max
db.collection.find( { $query: { <query> }, $max: { field1: <max value1>, ... fieldN: <max valueN
```

The `$max` specifies the upper bound for *all* keys of a specific index *in order*.

Consider the following operations on a collection named `collection` that has an index `{ age: 1 }`:

```
db.collection.find( { <query> } ).max( { age: 100 } )
```

This operation limits the query to those documents where the field `age` is less than `100` using the index `{ age: 1 }`.

You can explicitly specify the corresponding index with `cursor.hint()`. 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.

Consider a collection named `collection` that has the following two indexes:

```
{ age: 1, type: -1 }
{ age: 1, type: 1 }
```

Without explicitly using `cursor.hint()`, MongoDB may select either index for the following operation:

```
db.collection.find().max( { age: 50, type: 'B' } )
```

Use `$max` alone or in conjunction with `$min` to limit results to a specific range for the *same* index, as in the following example:

```
db.collection.find().min( { age: 20 } ).max( { age: 25 } )
```

Note: Because `cursor.max()` requires an index on a field, and forces the query to use this index, you may prefer the `$lt` (page 112) operator for the query if possible. Consider the following example:

```
db.collection.find( { _id: 7 } ).max( { age: 25 } )
```

The query uses the index on the `age` field, even if the index on `_id` may be better.

$min

Specify a `$min` value to specify the *inclusive* lower bound for a specific index in order to constrain the results of `find()`. The `mongo` shell provides the `cursor.min()` wrapper method:

```
db.collection.find( { <query> } ).min( { field1: <min value>, ... fieldN: <min valueN>} )
```

You can also specify the option with either of the two forms:

```
db.collection.find( { <query> } )._addSpecial( "$min", { field1: <min value1>, ... fieldN: <min
db.collection.find( { $query: { <query> }, $min: { field1: <min value1>, ... fieldN: <min valueN
```

The `$min` specifies the lower bound for *all* keys of a specific index *in order*.

Consider the following operations on a collection named `collection` that has an index `{ age: 1 }`:

```
db.collection.find().min( { age: 20 } )
```

These operations limit the query to those documents where the field `age` is at least `20` using the index `{ age: 1 }`.

You can explicitly specify the corresponding index with `cursor.hint()`. 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.

Consider a collection named `collection` that has the following two indexes:

```
{ age: 1, type: -1 }
{ age: 1, type: 1 }
```

Without explicitly using `cursor.hint()`, it is unclear which index the following operation will select:

```
db.collection.find().min( { age: 20, type: 'C' } )
```

You can use `$min` in conjunction with `$max` to limit results to a specific range for the *same* index, as in the following example:

```
db.collection.find().min( { age: 20 } ).max( { age: 25 } )
```

Note: Because `cursor.min()` requires an index on a field, and forces the query to use this index, you may prefer the `$gte` (page 113) operator for the query if possible. Consider the following example:

```
db.collection.find( { _id: 7 } ).min( { age: 25 } )
```

The query will use the index on the `age` field, even if the index on `_id` may be better.

$orderby

The `$orderby` operator sorts the results of a query in ascending or descending order.

The `mongo` shell provides the `cursor.sort()` method:

```
db.collection.find().sort( { age: -1 } )
```

You can also specify the option in either of the following forms:

```
db.collection.find()._addSpecial( "$orderby", { age : -1 } )
db.collection.find( { $query: {}, $orderby: { age : -1 } } )
```

These examples return all documents in the collection named `collection` sorted by the `age` field in descending order. Specify a value to `$orderby` of negative one (e.g. `-1`, as above) to sort in descending order or a positive value (e.g. `1`) to sort in ascending order.

Unless you have a index for the specified key pattern, use `$orderby` in conjunction with `$maxScan` and/or `cursor.limit()` to avoid requiring MongoDB to perform a large in-memory sort. The `cursor.limit()` increases the speed and reduces the amount of memory required to return this query by way of an optimized algorithm.

$hint

The `$hint` operator forces the *query optimizer* to use a specific index to fulfill the query. Use `$hint` for testing query performance and indexing strategies. Consider the following form:

```
db.collection.find().hint( { age: 1 } )
```

This operation returns all documents in the collection named `collection` using the index on the `age` field. Use this operator to override MongoDB's default index selection process and pick indexes manually.

You can also specify the option in either of the following forms:

```
db.collection.find()._addSpecial( "$hint", { age : 1 } )
db.collection.find( { $query: {}, $hint: { age : 1 } } )
```

$explain

`$explain` operator provides information on the query plan. It returns a `document` that describes the process and indexes used to return the query. This may provide useful insight when attempting to optimize a query.

`mongo` shell also provides the `explain()` method:

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

You can also specify the option in either of the following forms:

```
db.collection.find()._addSpecial( "$explain", 1 )
db.collection.find( { $query: {}, $explain: 1 } )
```

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

`$explain` runs the actual query to determine the result. Although there are some differences between running the query with `$explain` and running without, generally, the performance will be similar between the two. So, if the query is slow, the `$explain` operation is also slow.

Additionally, the `$explain` operation reevaluates a set of candidate query plans, which may cause the `$explain` operation to perform differently than a normal query. As a result, these operations generally provide an accurate account of *how* MongoDB would perform the query, but do not reflect the length of these queries.

To determine the performance of a particular index, you can use `hint()` and in conjunction with `explain()`, as in the following example:

```
db.products.find().hint( { type: 1 } ).explain()
```

When you run `explain()` with `hint()`, the query optimizer does not reevaluate the query plans.

Note: In some situations, the `explain()` operation may differ from the actual query plan used by MongoDB in a normal query.

The `explain()` operation evaluates the set of query plans and reports on the winning plan for the query. In normal operations the query optimizer caches winning query plans and uses them for similar related queries in the future. As a result MongoDB may sometimes select query plans from the cache that are different from the plan displayed using `explain()`.

See Also:

- `cursor.explain()`
- Optimization wiki page for information regarding optimization strategies.
- Database Profiler wiki page for information regarding optimization strategies.
- *Current Operation Reporting* (page 199)

$snapshot

The `$snapshot` operator prevents the cursor from returning a document more than once because an intervening write operation results in a move of the document.

Even in snapshot mode, objects inserted or deleted during the lifetime of the cursor may or may not be returned.

The `mongo` shell provides the `cursor.snapshot()` method:

```
db.collection.find().snapshot()
```

You can also specify the option in either of the following forms:

```
db.collection.find()._addSpecial( "$snapshot", true )
db.collection.find( { $query: {}, $snapshot: true } )
```

The `$snapshot` operator traverses the index on the `_id` field [1].

> **Warning:**
> - You cannot use `$snapshot` with *sharded collections*.
> - Do **not** use `$snapshot` with `$hint` or `$orderby` (or the corresponding `cursor.hint()` and `cursor.sort()` methods.)

[1] You can achieve the `$snapshot` isolation behavior using any *unique* index on invariable fields.

CHAPTER

SEVEN

AGGREGATION FRAMEWORK REFERENCE

New in version 2.1.0. The aggregation framework provides the ability to project, process, and/or control the output of the query, without using *map-reduce*. Aggregation uses a syntax that resembles the same syntax and form as "regular" MongoDB database queries.

These aggregation operations are all accessible by way of the `aggregate()` (page 29) method. While all examples in this document use this method, `aggregate()` (page 29) is merely a wrapper around the *database command* `aggregate` (page 29). The following prototype aggregation operations are equivalent:

```
db.people.aggregate( <pipeline> )
db.people.aggregate( [<pipeline>] )
db.runCommand( { aggregate: "people", pipeline: [<pipeline>] } )
```

These operations perform aggregation routines on the collection named `people`. `<pipeline>` is a placeholder for the aggregation *pipeline* definition. `aggregate()` (page 29) accepts the stages of the pipeline (i.e. `<pipeline>`) as an array, or as arguments to the method.

This documentation provides an overview of all aggregation operators available for use in the aggregation pipeline as well as details regarding their use and behavior.

See Also:

`http://docs.mongodb.org/manual/applications/aggregation` overview, the *Aggregation Framework Documentation Index*, and the `http://docs.mongodb.org/manual/tutorial/aggregation-examples` for more information on the aggregation functionality.

Aggregation Operators:

7.1 Pipeline

> **Warning:** The pipeline cannot operate on values of the following types: `Binary`, `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, and `CodeWScope`.

Pipeline operators appear in an array. Conceptually, documents pass through these operators in a sequence. All examples in this section assume that the aggregation pipeline begins with a collection named `article` that contains documents that resemble the following:

```
{
  title : "this is my title" ,
  author : "bob" ,
  posted : new Date() ,
  pageViews : 5 ,
  tags : [ "fun" , "good" , "fun" ] ,
  comments : [
      { author :"joe" , text : "this is cool" } ,
      { author :"sam" , text : "this is bad" }
  ],
  other : { foo : 5 }
}
```

The current pipeline operators are:

$project

Reshapes a document stream by renaming, adding, or removing fields. Also use `$project` to create computed values or sub-objects. Use `$project` to:

- Include fields from the original document.
- Insert computed fields.
- Rename fields.
- Create and populate fields that hold sub-documents.

Use `$project` to quickly select the fields that you want to include or exclude from the response. Consider the following aggregation framework operation.

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        author : 1 ,
    }}
 );
```

This operation includes the `title` field and the `author` field in the document that returns from the aggregation *pipeline*.

> **Note:** The `_id` field is always included by default. You may explicitly exclude `_id` as follows:
>
> ```
> db.article.aggregate(
>     { $project : {
>         _id : 0 ,
>         title : 1 ,
>         author : 1
>     }}
> );
> ```

Here, the projection excludes the _id field but includes the title and author fields.

Projections can also add computed fields to the document stream passing through the pipeline. A computed field can use any of the *expression operators* (page 148). Consider the following example:

```
db.article.aggregate(
    { $project : {
        title : 1,
        doctoredPageViews : { $add:["$pageViews", 10] }
    }}
);
```

Here, the field doctoredPageViews represents the value of the pageViews field after adding 10 to the original field using the $add.

Note: You must enclose the expression that defines the computed field in braces, so that the expression is a valid object.

You may also use $project to rename fields. Consider the following example:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        page_views : "$pageViews" ,
        bar : "$other.foo"
    }}
);
```

This operation renames the pageViews field to page_views, and renames the foo field in the other sub-document as the top-level field bar. The field references used for renaming fields are direct expressions and do not use an operator or surrounding braces. All aggregation field references can use dotted paths to refer to fields in nested documents.

Finally, you can use the $project to create and populate new sub-documents. Consider the following example that creates a new object-valued field named stats that holds a number of values:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        stats : {
            pv : "$pageViews",
            foo : "$other.foo",
            dpv : { $add:["$pageViews", 10] }
        }
    }}
);
```

This projection includes the title field and places $project into "inclusive" mode. Then, it creates the stats documents with the following fields:

- pv which includes and renames the pageViews from the top level of the original documents.
- foo which includes the value of other.foo from the original documents.
- dpv which is a computed field that adds 10 to the value of the pageViews field in the original document using the $add aggregation expression.

$match

Provides a query-like interface to filter documents out of the aggregation *pipeline*. The $match drops documents that do not match the condition from the aggregation pipeline, and it passes documents that match along the pipeline unaltered.

The syntax passed to the $match is identical to the *query* syntax. Consider the following prototype form:

```
db.article.aggregate(
    { $match : <match-predicate> }
);
```

The following example performs a simple field equality test:

```
db.article.aggregate(
    { $match : { author : "dave" } }
);
```

This operation only returns documents where the author field holds the value dave. Consider the following example, which performs a range test:

```
db.article.aggregate(
    { $match : { score  : { $gt : 50, $lte : 90 } } }
);
```

Here, all documents return when the score field holds a value that is greater than 50 and less than or equal to 90.

Note: Place the $match as early in the aggregation *pipeline* as possible. Because $match limits the total number of documents in the aggregation pipeline, earlier $match operations minimize the amount of later processing. If you place a $match at the very beginning of a pipeline, the query can take advantage of *indexes* like any other db.collection.find() or db.collection.findOne().

Warning: You cannot use $where (page 121) or *geospatial operations* (page 123) in $match queries as part of the aggregation pipeline.

$limit

Restricts the number of *documents* that pass through the $limit in the *pipeline*.

$limit takes a single numeric (positive whole number) value as a parameter. Once the specified number of documents pass through the pipeline operator, no more will. Consider the following example:

```
db.article.aggregate(
    { $limit : 5 }
);
```

This operation returns only the first 5 documents passed to it from by the pipeline. $limit has no effect on the content of the documents it passes.

$skip

Skips over the specified number of *documents* that pass through the $skip in the *pipeline* before passing all of the remaining input.

$skip takes a single numeric (positive whole number) value as a parameter. Once the operation has skipped the specified number of documents, it passes all the remaining documents along the *pipeline* without alteration. Consider the following example:

```
db.article.aggregate(
    { $skip : 5 }
);
```

This operation skips the first 5 documents passed to it by the pipeline. `$skip` has no effect on the content of the documents it passes along the pipeline.

$unwind

Peels off the elements of an array individually, and returns a stream of documents. `$unwind` returns one document for every member of the unwound array within every source document. Take the following aggregation command:

```
db.article.aggregate(
    { $project : {
        author : 1 ,
        title : 1 ,
        tags : 1
    }},
    { $unwind : "$tags" }
);
```

Note: The dollar sign (i.e. `$`) must proceed the field specification handed to the `$unwind` operator.

In the above aggregation `$project` selects (inclusively) the `author`, `title`, and `tags` fields, as well as the `_id` field implicitly. Then the pipeline passes the results of the projection to the `$unwind` operator, which will unwind the `tags` field. This operation may return a sequence of documents that resemble the following for a collection that contains one document holding a `tags` field with an array of 3 items.

```
{
     "result" : [
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "fun"
             },
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "good"
             },
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "fun"
             }
     ],
     "OK" : 1
}
```

A single document becomes 3 documents: each document is identical except for the value of the `tags` field. Each value of `tags` is one of the values in the original "tags" array.

Note: $unwind has the following behaviors:

- $unwind is most useful in combination with $group.
- You may undo the effects of unwind operation with the $group pipeline operator.
- If you specify a target field for $unwind that does not exist in an input document, the pipeline ignores the input document, and will generate no result documents.
- If you specify a target field for $unwind that is not an array, aggregate() (page 29) generates an error.
- If you specify a target field for $unwind that holds an empty array ([]) in an input document, the pipeline ignores the input document, and will generates no result documents.

$group

Groups documents together for the purpose of calculating aggregate values based on a collection of documents. Practically, group often supports tasks such as average page views for each page in a website on a daily basis.

The output of $group depends on how you define groups. Begin by specifying an identifier (i.e. a _id field) for the group you're creating with this pipeline. You can specify a single field from the documents in the pipeline, a previously computed value, or an aggregate key made up from several incoming fields. Aggregate keys may resemble the following document:

```
{ _id : { author: '$author', pageViews: '$pageViews', posted: '$posted' } }
```

With the exception of the _id field, $group cannot output nested documents.

Every group expression must specify an _id field. You may specify the _id field as a dotted field path reference, a document with multiple fields enclosed in braces (i.e. { and }), or a constant value.

Note: Use $project as needed to rename the grouped field after an $group operation, if necessary.

Consider the following example:

```
db.article.aggregate(
    { $group : {
        _id : "$author",
        docsPerAuthor : { $sum : 1 },
        viewsPerAuthor : { $sum : "$pageViews" }
    }}
);
```

This groups by the author field and computes two fields, the first docsPerAuthor is a counter field that adds one for each document with a given author field using the $sum function. The viewsPerAuthor field is the sum of all of the pageViews fields in the documents for each group.

Each field defined for the $group must use one of the group aggregation function listed below to generate its composite value:

$addToSet

Returns an array of all the values found in the selected field among the documents in that group. *Every unique value only appears once* in the result set. There is no ordering guarantee for the output documents.

$first

Returns the first value it encounters for its group .

Note: Only use $first when the $group follows an $sort operation. Otherwise, the result of this operation is unpredictable.

$last

Returns the last value it encounters for its group.

Note: Only use `$last` when the `$group` follows an `$sort` operation. Otherwise, the result of this operation is unpredictable.

$max

Returns the highest value among all values of the field in all documents selected by this group.

$min

Returns the lowest value among all values of the field in all documents selected by this group.

$avg

Returns the average of all the values of the field in all documents selected by this group.

$push

Returns an array of all the values found in the selected field among the documents in that group. *A value may appear more than once* in the result set if more than one field in the grouped documents has that value.

$sum

Returns the sum of all the values for a specified field in the grouped documents, as in the second use above.

Alternately, if you specify a value as an argument, `$sum` will increment this field by the specified value for every document in the grouping. Typically, as in the first use above, specify a value of `1` in order to count members of the group.

> **Warning:** The aggregation system currently stores `$group` operations in memory, which may cause problems when processing a larger number of groups.

$sort

The `$sort` *pipeline* operator sorts all input documents and returns them to the pipeline in sorted order. Consider the following prototype form:

```
db.<collection-name>.aggregate(
    { $sort : { <sort-key> } }
);
```

This sorts the documents in the collection named `<collection-name>`, according to the key and specification in the `{ <sort-key> }` document.

Specify the sort in a document with a field or fields that you want to sort by and a value of `1` or `-1` to specify an ascending or descending sort respectively, as in the following example:

```
db.users.aggregate(
    { $sort : { age : -1, posts: 1 } }
);
```

This operation sorts the documents in the `users` collection, in descending order according by the `age` field and then in ascending order according to the value in the `posts` field.

Note: The `$sort` cannot begin sorting documents until previous operators in the pipeline have returned all output.

- `$skip`

`$sort` operator can take advantage of an index when placed at the **beginning** of the pipeline or placed **before** the following aggregation operators:

- `$project`
- `$unwind`
- `$group`.

Warning: Unless the `$sort` operator can use an index, in the current release, the sort must fit within memory. This may cause problems when sorting large numbers of documents.

7.2 Expressions

These operators calculate values within the *aggregation framework*.

7.2.1 Boolean Operators

The three boolean operators accept Booleans as arguments and return Booleans as results.

Note: These operators convert non-booleans to Boolean values according to the BSON standards. Here, `null`, `undefined`, and `0` values become `false`, while non-zero numeric values, and all other types, such as strings, dates, objects become `true`.

`$and`

Takes an array one or more values and returns `true` if *all* of the values in the array are `true`. Otherwise `$and` (page 115) returns `false`.

Note: `$and` (page 115) uses short-circuit logic: the operation stops evaluation after encountering the first `false` expression.

`$or`

Takes an array of one or more values and returns `true` if *any* of the values in the array are `true`. Otherwise `$or` (page 116) returns false.

Note: `$or` (page 116) uses short-circuit logic: the operation stops evaluation after encountering the first `true` expression.

`$not`

Returns the boolean opposite value passed to it. When passed a `true` value, `$not` (page 118) returns `false`; when passed a `false` value, `$not` (page 118) returns `true`.

7.2.2 Comparison Operators

These operators perform comparisons between two values and return a Boolean, in most cases, reflecting the result of that comparison.

All comparison operators take an array with a pair of values. You may compare numbers, strings, and dates. Except for `$cmp`, all comparison operators return a Boolean value. `$cmp` returns an integer.

$cmp

Takes two values in an array and returns an integer. The returned value is:

- A negative number if the first value is less than the second.
- A positive number if the first value is greater than the second.
- `0` if the two values are equal.

$eq

Takes two values in an array and returns a boolean. The returned value is:

- `true` when the values are equivalent.
- `false` when the values are **not** equivalent.

$gt

Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *greater than* the second value.
- `false` when the first value is *less than or equal to* the second value.

$gte

Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *greater than or equal* to the second value.
- `false` when the first value is *less than* the second value.

$lt

Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *less than* the second value.
- `false` when the first value is *greater than or equal to* the second value.

$lte

Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *less than or equal to* the second value.
- `false` when the first value is *greater than* the second value.

$ne

Takes two values in an array returns an integer. The returned value is:

- `true` when the values are **not equivalent**.
- `false` when the values are **equivalent**.

7.2.3 Arithmetic Operators

These operators only support numbers.

$add

Takes an array of one or more numbers and adds them together, returning the sum.

$divide

Takes an array that contains a pair of numbers and returns the value of the first number divided by the second number.

$mod

Takes an array that contains a pair of numbers and returns the *remainder* of the first number divided by the second number.

See Also:

`$mod` (page 121)

$multiply

Takes an array of one or more numbers and multiples them, returning the resulting product.

$subtract

Takes an array that contains a pair of numbers and subtracts the second from the first, returning their difference.

7.2.4 String Operators

These operators manipulate strings within projection expressions.

$strcasecmp

Takes in two strings. Returns a number. `$strcasecmp` is positive if the first string is "greater than" the second and negative if the first string is "less than" the second. `$strcasecmp` returns 0 if the strings are identical.

Note: `$strcasecmp` may not make sense when applied to glyphs outside the Roman alphabet.

`$strcasecmp` internally capitalizes strings before comparing them to provide a case-*insensitive* comparison. Use `$cmp` for a case sensitive comparison.

$substr

`$substr` takes a string and two numbers. The first number represents the number of bytes in the string to skip, and the second number specifies the number of bytes to return from the string.

Note: `$substr` is not encoding aware and if used improperly may produce a result string containing an invalid UTF-8 character sequence.

$toLower

Takes a single string and converts that string to lowercase, returning the result. All uppercase letters become lowercase.

Note: `$toLower` may not make sense when applied to glyphs outside the Roman alphabet.

$toUpper

Takes a single string and converts that string to uppercase, returning the result. All lowercase letters become uppercase.

Note: `$toUpper` may not make sense when applied to glyphs outside the Roman alphabet.

7.2.5 Date Operators

All date operators take a "Date" typed value as a single argument and return a number.

$dayOfYear

Takes a date and returns the day of the year as a number between 1 and 366.

$dayOfMonth

Takes a date and returns the day of the month as a number between 1 and 31.

$dayOfWeek

Takes a date and returns the day of the week as a number between 1 (Sunday) and 7 (Saturday.)

$year

Takes a date and returns the full year.

$month

Takes a date and returns the month as a number between 1 and 12.

$week

Takes a date and returns the week of the year as a number between 0 and 53.

Weeks begin on Sundays, and week 1 begins with the first Sunday of the year. Days preceding the first Sunday of the year are in week 0. This behavior is the same as the "%U" operator to the strftime standard library function.

$hour
Takes a date and returns the hour between 0 and 23.

$minute
Takes a date and returns the minute between 0 and 59.

$second
Takes a date and returns the second between 0 and 59, but can be 60 to account for leap seconds.

7.2.6 Conditional Expressions

$cond
Use the $cond operator with the following syntax:

```
{ $cond: [ <boolean-expression>, <true-case>, <false-case> ] }
```

Takes an array with three expressions, where the first expression evaluates to a Boolean value. If the first expression evaluates to true, $cond returns the value of the second expression. If the first expression evaluates to false, $cond evaluates and returns the third expression.

$ifNull
Use the $ifNull operator with the following syntax:

```
{ $ifNull: [ <expression>, <replacement-if-null> ] }
```

Takes an array with two expressions. $ifNull returns the first expression if it evaluates to a non-null value. Otherwise, $ifNull returns the second expression's value.

Part IV

MongoDB and SQL Interface Comparisons

CHAPTER

EIGHT

SQL TO MONGODB MAPPING CHART

In addition to the charts that follow, you might want to consider the `http://docs.mongodb.org/manual/faq` section for a selection of common questions about MongoDB.

8.1 Executables

The following table presents the MySQL/Oracle executables and the corresponding MongoDB executables.

	MySQL/Oracle	MongoDB
Database Server	mysqld/oracle	*mongod* (page 207)
Database Client	mysql/sqlplus	*mongo* (page 217)

8.2 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 Specify any unique column or column combination as primary key.	*primary key* In MongoDB, the primary key is automatically set to the *_id* field.
aggregation (e.g. group by)	aggregation framework See the *SQL to Aggregation Framework Mapping Chart* (page 161).

8.3 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'
}
```

8.3.1 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	Reference

`CREATE TABLE users ( id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1), PRIMARY KEY (id) )`	Implicitly created on first `insert` operation. The primary key `_id` is automatically added if `_id` field is not specified.
	`db.users.insert( { user_id: "abc123", age: 55, status: "A" } )`
	However, you can also explicitly create a collection:
	`db.createCollection("users")`
		See `insert()` and `createCollection()` for more information.

`ALTER TABLE users ADD join_date DATETIME`	Collections do not describe or enforce the structure of the constituent documents. See the Schema Design wiki page for more information.	See `update()` and `$set` (page 127) for more information on changing the structure of documents in a collection.
`ALTER TABLE users DROP COLUMN join_date`	Collections do not describe or enforce the structure of the constituent documents. See the Schema Design wiki page for more information.	See `update()` and `$set` (page 127) for more information on changing the structure of documents in a collection.
`CREATE INDEX idx_user_id_asc ON users(user_id)`	`db.users.ensureIndex( { user_id: 1 } )`	See `ensureIndex()` and `indexes` for more information.
`CREATE INDEX idx_user_id_asc_age_desc ON users(user_id, age DESC)`	`db.users.ensureIndex( { user_id: 1, age: -1 } )`	See `ensureIndex()` and `indexes` for more information.
`DROP TABLE users`	`db.users.drop()`	See `drop()` for more information.

8.3.2 Insert

The following table presents the various SQL statements related to inserting records into tables and the corresponding MongoDB statements.

<table>
<tr><th>SQL INSERT Statements</th><th>MongoDB insert() Statements</th><th>Reference</th></tr>
<tr><td><pre>INSERT INTO users(user_id,
                  age,
                  status)
VALUES ("bcd001",
        45,
        "A")</pre></td><td><pre>db.users.insert({
       user_id: "bcd001",
       age: 45,
       status: "A"
})</pre></td><td>See <code>insert()</code> for more information.</td></tr>
</table>

8.3.3 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	Reference
`SELECT *` `FROM users`	`db.users.find()`	See `find()` for more information.

`SELECT id, user_id, status`
`FROM users`
	`db.users.find(`
	`    { },`
	`    { user_id: 1, status: 1 }`
	`)`
		See `find()` for more information.

`SELECT user_id, status`
`FROM users`
	`db.users.find(`
	`    { },`
	`    { user_id: 1, status: 1, _id: 0 }`
	`)`
		See `find()` for more information.

`SELECT *`
`FROM users`
`WHERE status = "A"`
	`db.users.find(`
	`    { status: "A" }`
	`)`
		See `find()` for more information.

`SELECT user_id, status`
`FROM users`
`WHERE status = "A"`
	`db.users.find(`
	`    { status: "A" },`
	`    { user_id: 1, status: 1, _id: 0 }`
	`)`
		See `find()` for more information.

`SELECT *`
`FROM users`
`WHERE status != "A"`
	`db.users.find(`
	`    { status: { $ne: "A" } }`
	`)`
		See `find()` and `$ne` (page 111) for more information.

`SELECT *`
`FROM users`
`WHERE status = "A"`
`AND age = 50`
	`db.users.find(`
	`    { status: "A",`
	`      age: 50 }`
	`)`
		See `find()` and `$and` (page 115) for more information.

`SELECT *`
`FROM users`
`WHERE status = "A"`
`OR age = 50`
	`db.users.find(`
	`    { $or: [ { status: "A" } ,`
	`              { age: 50 } ] }`
	`)`
		See `find()` and `$or` (page 116) for more information.

`SELECT *`
`FROM users`
`WHERE age > 25`
	`db.users.find(`
	`    { age: { $gt: 25 } }`
	`)`
		See `find()` and `$gt` (page 112) for more information.

`SELECT *`
`FROM users`
`WHERE age < 25`
	`db.users.find(`
	`   { age: { $lt: 25 } }`
	`)`
		See `find()` and `$lt` (page 112) for more information.

`SELECT *`
`FROM users`
`WHERE age > 25`
`AND   age <= 50`
	`db.users.find(`
	`   { age: { $gt: 25, $lte: 50 } }`
	`)`
		See `find()`, `$gt` (page 112), and `$lte` (page 112) for more information.

`SELECT *`
`FROM users`
`WHERE user_id like "%bc%"`
	`db.users.find(`
	`   { user_id: /bc/ }`
	`)`
		See `find()` and `$regex` (page 122) for more information.

8.3.4 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	Reference

`UPDATE users`
`SET status = "C"`
`WHERE age > 25`
	`db.users.update(`
	`{ age: { $gt: 25 } },`
	`{ $set: { status: "C" } },`
	`{ multi: true }`
	`)`
		See `update()`, `$gt` (page 112), and `$set` (page 127) for more information.

`UPDATE users`
`SET age = age + 3`
`WHERE status = "A"`
	`db.users.update(`
	`{ status: "A" } ,`
	`{ $inc: { age: 3 } },`
	`{ multi: true }`
	`)`
		See `update()`, `$inc` (page 127), and `$set` (page 127) for more information.

8.3.5 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	Reference
`DELETE FROM users` `WHERE status = "D"`	`db.users.remove( { status: "D" } )`	See `remove()` for more information.
`DELETE FROM users`	`db.users.remove( )`	See `remove()` for more information.

CHAPTER

NINE

SQL TO AGGREGATION FRAMEWORK MAPPING CHART

The `aggregation framework` allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL. If you're new to MongoDB you might want to consider the `http://docs.mongodb.org/manual/faq` section for a selection of common questions.

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB *aggregation operators* (page 142):

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match`
GROUP BY	`$group`
HAVING	`$match`
SELECT	`$project`
ORDER BY	`$sort`
LIMIT	`$limit`
SUM()	`$sum`
COUNT()	`$sum`
join	No direct corresponding operator; *however*, the `$unwind` operator allows for somewhat similar functionality, but with fields embedded within the document.

9.1 Examples

The following table presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume *two* tables, `orders` and `order_lineitem` that join by the `order_lineitem.order_id` and the `orders.id` columns.
- The MongoDB examples assume *one* collection `orders` that contain documents of the following prototype:

```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
```

```
            { sku: "yyy", qty: 25, price: 1 } ]
}
```

- The MongoDB statements prefix, the names of the fields from the *documents* in the collection `orders` with a `$` character when they appear as operands to the aggregation operations.

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: "$ord_date" },`
	`               total: { $sum: "$price" } } }`
	`] )`
		For each unique `cust_id`, `ord_date` grouping, sum the `price` field.

`SELECT cust_id, count(*)`
`FROM orders`
`GROUP BY cust_id`
`HAVING count(*) > 1`
	`db.orders.aggregate( [`
	`   { $group: { _id: "$cust_id",`
	`               count: { $sum: 1 } } },`
	`   { $match: { count: { $gt: 1 } } }`
	`] )`
		For `cust_id` with multiple records, return the `cust_id` and the corresponding record count.

`SELECT cust_id,`
`       ord_date,`
`       SUM(price) AS total`
`FROM orders`
`GROUP BY cust_id, ord_date`
`HAVING total > 250`
	`db.orders.aggregate( [`
	`   { $group: { _id: { cust_id: "$cust_id",`
	`                      ord_date: "$ord_date" },`
	`               total: { $sum: "$price" } } },`
	`   { $match: { total: { $gt: 250 } } }`
	`] )`
		For each unique `cust_id`, `ord_date` grouping, sum the `price` field and return only where the sum is greater than 250.

`SELECT cust_id,`
`       SUM(price) as total`
`FROM orders`
`WHERE status = 'A'`
`GROUP BY cust_id`
	`db.orders.aggregate( [`
	`   { $match: { status: 'A' } },`
	`   { $group: { _id: "$cust_id",`
	`               total: { $sum: "$price" } } }`
	`] )`
		For each unique `cust_id` with status `A`, sum the `price` field.

`SELECT cust_id,`
`       SUM(price) as total`
`FROM orders`
`WHERE status = 'A'`
`GROUP BY cust_id`
`HAVING total > 250`
	`db.orders.aggregate( [`
	`   { $match: { status: 'A' } },`
	`   { $group: { _id: "$cust_id",`
	`               total: { $sum: "$price" } } },`
	`   { $match: { total: { $gt: 250 } } }`
	`] )`
		For each unique `cust_id` with status `A`, sum the `price` field and return only where the sum is greater than 250.

`SELECT cust_id,`	`db.orders.aggregate( [`	For each unique `cust_id`, sum the

Part V

Status, Monitoring, and Reporting Output

CHAPTER

TEN

SERVER STATUS REFERENCE

The `serverStatus` (page 52) command returns a collection of information that reflects the database's status. These data are useful for diagnosing and assessing the performance of your MongoDB instance. This reference catalogs each datum included in the output of this command and provides context for using this data to more effectively administer your database.

See Also:

Much of the output of `serverStatus` (page 52) is also displayed dynamically by `mongostat`. See the *mongostat* (page 238) command for more information.

For examples of the `serverStatus` (page 52) output, see `http://docs.mongodb.org/manual/reference/server-sta`

10.1 Instance Information

Example

output of the instance information fields.

host

The `host` (page 167) field contains the system's hostname. In Unix/Linux systems, this should be the same as the output of the `hostname` command.

version

The `version` (page 256) field contains the version of MongoDB running on the current `mongod` or `mongos` (page 256) instance.

process

The `process` (page 167) field identifies which kind of MongoDB instance is running. Possible values are:

- `mongos` (page 256)
- `mongod`

uptime

The value of the `uptime` (page 167) field corresponds to the number of seconds that the `mongos` (page 256) or `mongod` process has been active.

uptimeEstimate

`uptimeEstimate` (page 167) provides the uptime as calculated from MongoDB's internal course-grained time keeping system.

localTime

The `localTime` (page 167) value is the current time, according to the server, in UTC specified in an ISODate format.

10.2 locks

New in version 2.1.2: All `locks` (page 255) statuses first appeared in the 2.1.2 development release for the 2.2 series.

Example

output of the locks fields.

---

locks

The `locks` (page 255) document contains sub-documents that provides a granular report on MongoDB database-level lock use. All values are of the `NumberLong()` type.

Generally, fields named:

- `R` refer to the global read lock,
- `W` refer to the global write lock,
- `r` refer to the database specific read lock, and
- `w` refer to the database specific write lock.

If a document does not have any fields, it means that no locks have existed with this context since the last time the `mongod` started.

`locks..`

A field named `.` holds the first document in `locks` (page 255) that contains information about the global lock as well as aggregated data regarding lock use in all databases.

`locks...`**timeLockedMicros**

The `locks...timeLockedMicros` (page 168) document reports the amount of time in microseconds that a lock has existed in all databases in this `mongod` instance.

`locks...timeLockedMicros.`**R**

The `R` field reports the amount of time in microseconds that any database has held the global read lock.

`locks...timeLockedMicros.`**W**

The `W` field reports the amount of time in microseconds that any database has held the global write lock.

`locks...timeLockedMicros.`**r**

The `r` field reports the amount of time in microseconds that any database has held the local read lock.

`locks...timeLockedMicros.`**w**

The `w` field reports the amount of time in microseconds that any database has held the local write lock.

`locks...`**timeAcquiringMicros**

The `locks...timeAcquiringMicros` (page 168) document reports the amount of time in microseconds that operations have spent waiting to acquire a lock in all databases in this `mongod` instance.

`locks...timeAcquiringMicros.`**R**

The `R` field reports the amount of time in microseconds that any database has spent waiting for the global read lock.

`locks...timeAcquiringMicros.`**W**

The `W` field reports the amount of time in microseconds that any database has spent waiting for the global write lock.

`locks.`**admin**

The `locks.admin` (page 168) document contains two sub-documents that report data regarding lock use in the *admin database*.

`locks.admin.`**timeLockedMicros**

The `locks.admin.timeLockedMicros` (page 168) document reports the amount of time in microseconds that locks have existed in the context of the *admin database*.

`locks.admin.timeLockedMicros.r`
The `r` field reports the amount of time in microseconds that the *admin database* has held the read lock.

`locks.admin.timeLockedMicros.w`
The `w` field reports the amount of time in microseconds that the *admin database* has held the write lock.

`locks.admin.timeAcquiringMicros`
The `locks.admin.timeAcquiringMicros` (page 169) document reports on the amount of field time in microseconds that operations have spent waiting to acquire a lock for the *admin database*.

`locks.admin.timeAcquiringMicros.r`
The `r` field reports the amount of time in microseconds that operations have spent waiting to acquire a read lock on the *admin database*.

`locks.admin.timeAcquiringMicros.w`
The `w` field reports the amount of time in microseconds that operations have spent waiting to acquire a write lock on the *admin database*.

`locks.local`
The `locks.local` (page 169) document contains two sub-documents that report data regarding lock use in the `local` database. The local database contains a number of instance specific data, including the *oplog* for replication.

`locks.local.timeLockedMicros`
The `locks.local.timeLockedMicros` (page 169) document reports on the amount of time in microseconds that locks have existed in the context of the `local` database.

`locks.local.timeLockedMicros.r`
The `r` field reports the amount of time in microseconds that the `local` database has held the read lock.

`locks.local.timeLockedMicros.w`
The `w` field reports the amount of time in microseconds that the `local` database has held the write lock.

`locks.local.timeAcquiringMicros`
The `locks.local.timeAcquiringMicros` (page 169) document reports on the amount of time in microseconds that operations have spent waiting to acquire a lock for the `local` database.

`locks.local.timeAcquiringMicros.r`
The `r` field reports the amount of time in microseconds that operations have spent waiting to acquire a read lock on the `local` database.

`locks.local.timeAcquiringMicros.w`
The `w` field reports the amount of time in microseconds that operations have spent waiting to acquire a write lock on the `local` database.

`locks.<database>`
For each additional database `locks` (page 255) includes a document that reports on the lock use for this database. The names of these documents reflect the database name itself.

`locks.<database>.timeLockedMicros`
The `locks.<database>.timeLockedMicros` (page 169) document reports on the amount of time in microseconds that locks have existed in the context of the `<database>` database.

`locks.<database>.timeLockedMicros.r`
The `r` field reports the amount of time in microseconds that the `<database>` database has held the read lock.

`locks.<database>.timeLockedMicros.w`
The `w` field reports the amount of time in microseconds that the `<database>` database has held the write lock.

`locks.<database>.timeAcquiringMicros`
The `locks.<database>.timeAcquiringMicros` (page 169) document reports on the amount of time in microseconds that operations have spent waiting to acquire a lock for the `<database>` database.

locks.<database>.timeAcquiringMicros.**r**
The `r` field reports the amount of time in microseconds that operations have spent waiting to acquire a read lock on the `<database>` database.

locks.<database>.timeAcquiringMicros.**w**
The `w` field reports the amount of time in microseconds that operations have spent waiting to acquire a write lock on the `<database>` database.

10.3 globalLock

Example

output of the globalLock fields.

globalLock
The `globalLock` (page 170) data structure contains information regarding the database's current lock state, historical lock status, current operation queue, and the number of active clients.

globalLock.**totalTime**
The value of `globalLock.totalTime` (page 170) represents the time, in microseconds, since the database last started and creation of the `globalLock` (page 170). This is roughly equivalent to total server uptime.

globalLock.**lockTime**
The value of `globalLock.lockTime` (page 170) represents the time, in microseconds, since the database last started, that the `globalLock` (page 170) has been *held*.

Consider this value in combination with the value of `globalLock.totalTime` (page 170). MongoDB aggregates these values in the `globalLock.ratio` (page 170) value. If the `globalLock.ratio` (page 170) value is small but `globalLock.totalTime` (page 170) is high the `globalLock` (page 170) has typically been held frequently for shorter periods of time, which may be indicative of a more normal use pattern. If the `globalLock.lockTime` (page 170) is higher and the `globalLock.totalTime` (page 170) is smaller (relatively,) then fewer operations are responsible for a greater portion of server's use (relatively.)

globalLock.**ratio**
Changed in version 2.2: `globalLock.ratio` (page 170) was removed. See `locks` (page 255). The value of `globalLock.ratio` (page 170) displays the relationship between `globalLock.lockTime` (page 170) and `globalLock.totalTime` (page 170).

Low values indicate that operations have held the `globalLock` (page 170) frequently for shorter periods of time. High values indicate that operations have held `globalLock` (page 170) infrequently for longer periods of time.

10.3.1 globalLock.currentQueue

globalLock.**currentQueue**
The `globalLock.currentQueue` (page 170) data structure value provides more granular information concerning the number of operations queued because of a lock.

globalLock.currentQueue.**total**
The value of `globalLock.currentQueue.total` (page 170) provides a combined total of operations queued waiting for the lock.

A consistently small queue, particularly of shorter operations should cause no concern. Also, consider this value in light of the size of queue waiting for the read lock (e.g. `globalLock.currentQueue.readers` (page 170)) and write-lock (e.g. `globalLock.currentQueue.writers` (page 171)) individually.

globalLock.currentQueue.**readers**

The value of globalLock.currentQueue.readers (page 170) is the number of operations that are currently queued and waiting for the read-lock. A consistently small read-queue, particularly of shorter operations should cause no concern.

globalLock.currentQueue.**writers**

The value of globalLock.currentQueue.writers (page 171) is the number of operations that are currently queued and waiting for the write-lock. A consistently small write-queue, particularly of shorter operations is no cause for concern.

10.3.2 globalLock.activeClients

globalLock.**activeClients**

The globalLock.activeClients (page 171) data structure provides more granular information about the number of connected clients and the operation types (e.g. read or write) performed by these clients.

Use this data to provide context for the *currentQueue* (page 170) data.

globalLock.activeClients.**total**

The value of globalLock.activeClients.total (page 171) is the total number of active client connections to the database. This combines clients that are performing read operations (e.g. globalLock.activeClients.readers (page 171)) and clients that are performing write operations (e.g. globalLock.activeClients.writers (page 171)).

globalLock.activeClients.**readers**

The value of globalLock.activeClients.readers (page 171) contains a count of the active client connections performing read operations.

globalLock.activeClients.**writers**

The value of globalLock.activeClients.writers (page 171) contains a count of active client connections performing write operations.

10.4 mem

Example

output of the memory fields.

mem

The mem data structure holds information regarding the target system architecture of mongod and current memory use.

mem.**bits**

The value of mem.bits (page 171) is either 64 or 32, depending on which target architecture specified during the mongod compilation process. In most instances this is 64, and this value does not change over time.

mem.**resident**

The value of mem.resident (page 171) is roughly equivalent to the amount of RAM, in bytes, currently used by the database process. In normal use this value tends to grow. In dedicated database servers this number tends to approach the total amount of system memory.

mem.**virtual**

mem.virtual (page 171) displays the quantity, in megabytes (MB), of virtual memory used by the mongod process. In typical deployments this value is slightly larger than mem.mapped (page 172). If this value is significantly (i.e. gigabytes) larger than mem.mapped (page 172), this could indicate a memory leak.

With *journaling* enabled, the value of `mem.virtual` (page 171) is twice the value of `mem.mapped` (page 172).

`mem.`**`supported`**
`mem.supported` (page 172) is true when the underlying system supports extended memory information. If this value is false and the system does not support extended memory information, then other `mem` values may not be accessible to the database server.

`mem.`**`mapped`**
The value of `mem.mapped` (page 172) provides the amount of mapped memory, in megabytes (MB), by the database. Because MongoDB uses memory-mapped files, this value is likely to be to be roughly equivalent to the total size of your database or databases.

`mem.`**`mappedWithJournal`**
`mem.mappedWithJournal` (page 172) provides 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 172). This field is only included if journaling is enabled.

10.5 connections

Example

output of the connections fields.

`connections`
The `connections` sub document data regarding the current connection status and availability of the database server. Use these values to asses the current load and capacity requirements of the server.

`connections.`**`current`**
The value of `connections.current` (page 172) corresponds to the number of connections to the database server from clients. This number includes the current shell session. Consider the value of `connections.available` (page 172) to add more context to this datum.

This figure will include the current shell connection as well as any inter-node connections to support a *replica set* or *sharded cluster*.

`connections.`**`available`**
`connections.available` (page 172) provides a count of the number of unused available connections that the database can provide. Consider this value in combination with the value of `connections.current` (page 172) to understand the connection load on the database, and the `http://docs.mongodb.org/manual/administration/ulimit` document for more information about system thresholds on available connections.

10.6 extra_info

Example

output of the extra_info fields.

`extra_info`
The `extra_info` (page 172) data structure holds data collected by the `mongod` instance about the underlying system. Your system may only report a subset of these fields.

`extra_info.`**`note`**

The field `extra_info.note` (page 172) reports that the data in this structure depend on the underlying platform, and has the text: "fields vary by platform."

`extra_info.`**`heap_usage_bytes`**

The `extra_info.heap_usage_bytes` (page 173) field is only available on Unix/Linux systems, and reports the total size in bytes of heap space used by the database process.

`extra_info.`**`page_faults`**

The `extra_info.page_faults` (page 173) field is only available on Unix/Linux systems, and reports the total number of page faults that require disk operations. Page faults refer to operations that require the database server to access data which isn't available in active memory. The `page_fault` (page 173) 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.

10.7 indexCounters

Example

output of the indexCounters fields.

`indexCounters`

Changed in version 2.2: Previously, data in the `indexCounters` (page 173) document reported sampled data, and were only useful in relative comparison to each other, because they could not reflect absolute index use. In 2.2 and later, these data reflect actual index use. The `indexCounters` (page 173) data structure reports information regarding the state and use of indexes in MongoDB.

`indexCounters.`**`btree`**

The `indexCounters.btree` (page 173) data structure contains data regarding MongoDB's *btree* indexes.

`indexCounters.btree.`**`accesses`**

`indexCounters.btree.accesses` (page 173) reports the number of times that operations have accessed indexes. This value is the combination of the `indexCounters.btree.hits` (page 173) and `indexCounters.btree.misses` (page 173). Higher values indicate that your database has indexes and that queries are taking advantage of these indexes. If this number does not grow over time, this might indicate that your indexes do not effectively support your use.

`indexCounters.btree.`**`hits`**

The `indexCounters.btree.hits` (page 173) value reflects the number of times that an index has been accessed and `mongod` is able to return the index from memory.

A higher value indicates effective index use. `indexCounters.btree.hits` (page 173) values that represent a greater proportion of the `indexCounters.btree.accesses` (page 173) value, tend to indicate more effective index configuration.

`indexCounters.btree.`**`misses`**

The `indexCounters.btree.misses` (page 173) value represents the number of times that an operation attempted to access an index that was not in memory. These "misses," do not indicate a failed query or operation, but rather an inefficient use of the index. Lower values in this field indicate better index use and likely overall performance as well.

`indexCounters.btree.`**`resets`**

The `indexCounters.btree.resets` (page 173) value reflects the number of times that the index counters have been reset since the database last restarted. Typically this value is `0`, but use this value to provide context for the data specified by other `indexCounters` (page 173) values.

indexCounters.btree.**missRatio**

The `indexCounters.btree.missRatio` (page 173) value is the ratio of `indexCounters.btree.hits` (page 173) to `indexCounters.btree.misses` (page 173) misses. This value is typically `0` or approaching `0`.

10.8 backgroundFlushing

Example

output of the backgroundFlushing fields.

backgroundFlushing

`mongod` periodically flushes writes to disk. In the default configuration, this happens every 60 seconds. The `backgroundFlushing` (page 174) data structure contains data regarding these operations. Consider these values if you have concerns about write performance and *journaling* (page 179).

backgroundFlushing.**flushes**

`backgroundFlushing.flushes` (page 174) is a counter that collects the number of times the database has flushed all writes to disk. This value will grow as database runs for longer periods of time.

backgroundFlushing.**total_ms**

The `backgroundFlushing.total_ms` (page 174) value provides the total number of milliseconds (ms) that the `mongod` processes have spent writing (i.e. flushing) data to disk. Because this is an absolute value, consider the value of `backgroundFlushing.flushes` (page 174) and `backgroundFlushing.average_ms` (page 174) to provide better context for this datum.

backgroundFlushing.**average_ms**

The `backgroundFlushing.average_ms` (page 174) value describes the relationship between the number of flushes and the total amount of time that the database has spent writing data to disk. The larger `backgroundFlushing.flushes` (page 174) is, the more likely this value is likely to represent a "normal," time; however, abnormal data can skew this value.

Use the `backgroundFlushing.last_ms` (page 174) to ensure that a high average is not skewed by transient historical issue or a random write distribution.

backgroundFlushing.**last_ms**

The value of the `backgroundFlushing.last_ms` (page 174) field is the amount of time, in milliseconds, that the last flush operation took to complete. Use this value to verify that the current performance of the server and is in line with the historical data provided by `backgroundFlushing.average_ms` (page 174) and `backgroundFlushing.total_ms` (page 174).

backgroundFlushing.**last_finished**

The `backgroundFlushing.last_finished` (page 174) field provides a timestamp of the last completed flush operation in the *ISODate* format. If this value is more than a few minutes old relative to your server's current time and accounting for differences in time zone, restarting the database may result in some data loss.

Also consider ongoing operations that might skew this value by routinely block write operations.

10.9 cursors

Example

output of the cursors fields.

cursors
The `cursors` data structure contains data regarding cursor state and use.

`cursors.`**totalOpen**
`cursors.totalOpen` (page 175) provides the number of cursors that MongoDB is maintaining for clients. Because MongoDB exhausts unused cursors, typically this value small or zero. However, if there is a queue, stale tailable cursors, or a large number of operations this value may rise.

`cursors.`**clientCursors_size**
Deprecated since version 1.x: See `cursors.totalOpen` (page 175) for this datum.

`cursors.`**timedOut**
`cursors.timedOut` (page 175) provides a counter of the total number of cursors that have timed out since the server process started. If this number is large or growing at a regular rate, this may indicate an application error.

10.10 network

Example

output of the network fields.

network
The `network` data structure contains data regarding MongoDB's network use.

`network.`**bytesIn**
The value of the `network.bytesIn` (page 175) field reflects the amount of network traffic, in bytes, received *by* this database. Use this value to ensure that network traffic sent to the `mongod` process is consistent with expectations and overall inter-application traffic.

`network.`**bytesOut**
The value of the `network.bytesOut` (page 175) field reflects the amount of network traffic, in bytes, sent *from* this database. Use this value to ensure that network traffic sent by the `mongod` process is consistent with expectations and overall inter-application traffic.

`network.`**numRequests**
The `network.numRequests` (page 175) field is a counter of the total number of distinct requests that the server has received. Use this value to provide context for the `network.bytesIn` (page 175) and `network.bytesOut` (page 175) values to ensure that MongoDB's network utilization is consistent with expectations and application use.

10.11 repl

Example

output of the repl fields.

repl
The `repl` data structure contains status information for MongoDB's replication (i.e. "replica set") configuration. These values only appear when the current host has replication enabled.

See `http://docs.mongodb.org/manual/core/replication` for more information on replication.

`repl.`**`setName`**

The `repl.setName` (page 176) field contains a string with the name of the current replica set. This value reflects the `--replSet` (page 212) command line argument, or `replSet` value in the configuration file.

See `http://docs.mongodb.org/manual/core/replication` for more information on replication.

`repl.`**`ismaster`**

The value of the `repl.ismaster` (page 176) field is either `true` or `false` and reflects whether the current node is the master or primary node in the replica set.

See `http://docs.mongodb.org/manual/core/replication` for more information on replication.

`repl.`**`secondary`**

The value of the `repl.secondary` (page 176) field is either `true` or `false` and reflects whether the current node is a secondary node in the replica set.

See `http://docs.mongodb.org/manual/core/replication` for more information on replication.

`repl.`**`hosts`**

`repl.hosts` (page 176) is an array that lists the other nodes in the current replica set. Each member of the replica set appears in the form of `hostname:port`.

See `http://docs.mongodb.org/manual/core/replication` for more information on replication.

10.12 opcountersRepl

Example

output of the opcountersRepl fields.

`opcountersRepl`

The `opcountersRepl` (page 176) data structure, similar to the `opcounters` data structure, provides an overview of database replication operations by type and makes it possible to analyze the load on the replica in more granular manner. These values only appear when the current host has replication enabled.

These values will differ from the `opcounters` values because of how MongoDB serializes operations during replication. See `http://docs.mongodb.org/manual/core/replication` for more information on replication.

These numbers will grow over time in response to database use. Analyze these values over time to track database utilization.

`opcountersRepl.`**`insert`**

`opcountersRepl.insert` (page 176) provides a counter of the total number of replicated insert operations since the `mongod` instance last started.

`opcountersRepl.`**`query`**

`opcountersRepl.query` (page 176) provides a counter of the total number of replicated queries since the `mongod` instance last started.

`opcountersRepl.`**`update`**

`opcountersRepl.update` (page 176) provides a counter of the total number of replicated update operations since the `mongod` instance last started.

`opcountersRepl.`**`delete`**

`opcountersRepl.delete` (page 176) provides a counter of the total number of replicated delete operations since the `mongod` instance last started.

opcountersRepl.**getmore**
opcountersRepl.getmore (page 176) provides a counter of the total number of "getmore" operations since the mongod 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**
opcountersRepl.command (page 177) provides a counter of the total number of replicated commands issued to the database since the mongod instance last started.

10.13 replNetworkQueue

New in version 2.1.2.

Example

output of the replNetworkQueue fields.

---

replNetworkQueue
The replNetworkQueue (page 177) document reports on the network replication buffer, which permits replication operations to happen in the background. This feature is internal.

This document only appears on *secondary* members of *replica sets*.

replNetworkQueue.**waitTimeMs**
replNetworkQueue.waitTimeMs (page 177) reports the amount of time that a *secondary* waits to add operations to network queue. This value is cumulative.

replNetworkQueue.**numElems**
replNetworkQueue.numElems (page 177) reports the number of operations stored in the queue.

replNetworkQueue.**numBytes**
replNetworkQueue.numBytes (page 177) reports the total size of the network replication queue.

10.14 opcounters

Example

output of the opcounters fields.

---

opcounters
The opcounters data structure provides an overview of database operations by type and makes it possible to analyze the load on the database in more granular manner.

These numbers will grow over time and in response to database use. Analyze these values over time to track database utilization.

opcounters.**insert**
opcounters.insert (page 177) provides a counter of the total number of insert operations since the mongod instance last started.

opcounters.**query**
opcounters.query (page 177) provides a counter of the total number of queries since the mongod instance last started.

`opcounters.update`
: `opcounters.update` (page 177) provides a counter of the total number of update operations since the `mongod` instance last started.

`opcounters.delete`
: `opcounters.delete` (page 178) provides a counter of the total number of delete operations since the `mongod` instance last started.

`opcounters.getmore`
: `opcounters.getmore` (page 178) provides a counter of the total number of "getmore" operations since the `mongod` 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`
: `opcounters.command` (page 178) provides a counter of the total number of commands issued to the database since the `mongod` instance last started.

10.15 asserts

Example

output of the asserts fields.

asserts
: The `asserts` document reports the number of asserts on the database. While assert errors are typically uncommon, if there are non-zero values for the `asserts`, you should check the log file for the `mongod` process for more information. In many cases these errors are trivial, but are worth investigating.

`asserts.regular`
: The `asserts.regular` (page 178) counter tracks the number of regular assertions raised since the server process started. Check the log file for more information about these messages.

`asserts.warning`
: The `asserts.warning` (page 178) counter tracks the number of warnings raised since the server process started. Check the log file for more information about these warnings.

`asserts.msg`
: The `asserts.msg` (page 178) counter tracks the number of message assertions raised since the server process started. Check the log file for more information about these messages.

`asserts.user`
: The `asserts.user` (page 178) counter reports the number of "user asserts" that have occurred since the last time the server process started. These are errors that user may generate, such as out of disk space or duplicate key. You can prevent these assertions by fixing a problem with your application or deployment. Check the MongoDB log for more information.

`asserts.rollovers`
: The `asserts.rollovers` (page 178) counter displays the number of times that the rollover counters have rolled over since the last time the server process started. The counters will rollover to zero after 2^{30} assertions. Use this value to provide context to the other values in the `asserts` data structure.

10.16 writeBacksQueued

Example

output of the writeBacksQueued fields.

`writeBacksQueued`

The value of `writeBacksQueued` (page 179) is `true` when there are operations from a `mongos` (page 256) instance queued for retrying. Typically this option is false.

See Also:

writeBacks

10.17 dur

New in version 1.8.

10.17.1 Journaling

Example

output of the journaling fields.

`dur`

The `dur` (for "durability") document contains data regarding the `mongod`'s journaling-related operations and performance. `mongod` must be running with journaling for these data to appear in the output of "`serverStatus` (page 52)".

Note: The data values are **not** cumulative but are reset on a regular basis as determined by the journal group commit interval. This interval is ~100 milliseconds (ms) by default (or 30ms if the journal file is on the same file system as your data files) and is cut by 1/3 when there is a `getLastError` (page 50) command pending. The interval is configurable using the `--journalCommitInterval` option.

See Also:

"Journaling" for more information about journaling operations.

`dur.`**`commits`**

The `dur.commits` (page 179) provides the number of transactions written to the *journal* during the last journal group commit interval.

`dur.`**`journaledMB`**

The `dur.journaledMB` (page 179) provides the amount of data in megabytes (MB) written to *journal* during the last journal group commit interval.

`dur.`**`writeToDataFilesMB`**

The `dur.writeToDataFilesMB` (page 179) provides the amount of data in megabytes (MB) written from *journal* to the data files during the last journal group commit interval.

`dur.`**`compression`**

New in version 2.0. The `dur.compression` (page 179) represents the compression ratio of the data written to the *journal*:

```
( journaled_size_of_data / uncompressed_size_of_data )
```

`dur.commitsInWriteLock`
The `dur.commitsInWriteLock` (page 179) provides a count of the commits that occurred while a write lock was held. Commits in a write lock indicate a MongoDB node under a heavy write load and call for further diagnosis.

`dur.earlyCommits`
The `dur.earlyCommits` (page 180) value reflects the number of times MongoDB requested a commit before the scheduled journal group commit interval. Use this value to ensure that your journal group commit interval is not too long for your deployment.

`dur.timeMS`
The `dur.timeMS` (page 180) document provides information about the performance of the `mongod` instance during the various phases of journaling in the last journal group commit interval.

`dur.timeMS.dt`
The `dur.timeMS.dt` (page 180) value provides, in milliseconds, the amount of time over which MongoDB collected the `dur.timeMS` (page 180) data. Use this field to provide context to the other `dur.timeMS` (page 180) field values.

`dur.timeMS.prepLogBuffer`
The `dur.timeMS.prepLogBuffer` (page 180) value provides, in milliseconds, the amount of time spent preparing to write to the journal. Smaller values indicate better journal performance.

`dur.timeMS.writeToJournal`
The `dur.timeMS.writeToJournal` (page 180) value provides, in milliseconds, the amount of time spent actually writing to the journal. File system speeds and device interfaces can affect performance.

`dur.timeMS.writeToDataFiles`
The `dur.timeMS.writeToDataFiles` (page 180) value provides, in milliseconds, the amount of time spent writing to data files after journaling. File system speeds and device interfaces can affect performance.

`dur.timeMS.remapPrivateView`
The `dur.timeMS.remapPrivateView` (page 180) value provides, in milliseconds, the amount of time spent remapping copy-on-write memory mapped views. Smaller values indicate better journal performance.

10.18 recordStats

Example

output of the recordStats fields.

`recordStats`
The `recordStats` (page 180) document provides fine grained reporting on page faults on a per database level.

`recordStats.accessesNotInMemory`
`recordStats.accessesNotInMemory` (page 180) reflects the number of times `mongod` needed to access a memory page that was *not* resident in memory for *all* databases managed by this `mongod` instance.

`recordStats.pageFaultExceptionsThrown`
`recordStats.pageFaultExceptionsThrown` (page 180) reflects the number of page fault exceptions thrown by `mongod` when accessing data for *all* databases managed by this `mongod` instance.

`recordStats.local.accessesNotInMemory`
`recordStats.local.accessesNotInMemory` (page 180) reflects the number of times `mongod` needed to access a memory page that was *not* resident in memory for the `local` database.

recordStats.local.**pageFaultExceptionsThrown**
: `recordStats.local.pageFaultExceptionsThrown` (page 180) reflects the number of page fault exceptions thrown by `mongod` when accessing data for the `local` database.

recordStats.admin.**accessesNotInMemory**
: `recordStats.admin.accessesNotInMemory` (page 181) reflects the number of times `mongod` needed to access a memory page that was *not* resident in memory for the *admin database*.

recordStats.admin.**pageFaultExceptionsThrown**
: `recordStats.admin.pageFaultExceptionsThrown` (page 181) reflects the number of page fault exceptions thrown by `mongod` when accessing data for the *admin database*.

recordStats.<database>.**accessesNotInMemory**
: `recordStats.<database>.accessesNotInMemory` (page 181) reflects the number of times `mongod` needed to access a memory page that was *not* resident in memory for the `<database>` database.

recordStats.<database>.**pageFaultExceptionsThrown**
: `recordStats.<database>.pageFaultExceptionsThrown` (page 181) reflects the number of page fault exceptions thrown by `mongod` when accessing data for the `<database>` database.

CHAPTER

ELEVEN

DATABASE STATISTICS REFERENCE

11.1 Synopsis

MongoDB can report data that reflects the current state of the "active" database. In this context "database," refers to a single MongoDB database. To run `dbStats` (page 48) issue this command in the shell:

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

The `mongo` shell provides the helper function `db.stats()`. Use the following form:

```
db.stats()
```

The above commands are equivalent. Without any arguments, `db.stats()` returns values in bytes. To convert the returned values to kilobytes, use the scale argument:

```
db.stats(1024)
```

Or:

```
db.runCommand( { dbStats: 1, scale: 1024 } )
```

Note: Because scaling rounds values to whole number, scaling may return unlikely or unexpected results.

The above commands are equivalent. See the `dbStats` (page 48) *database command* and the `db.stats()` helper for the `mongo` shell for additional information.

11.2 Fields

db
Contains the name of the database.

collections
Contains a count of the number of collections in that database.

objects
Contains a count of the number of objects (i.e. *documents*) in the database across all collections.

avgObjSize
The average size of each object. The `scale` argument affects this value. This is the `dataSize` (page 184) divided by the number of objects.

dataSize
The total size of the data held in this database including the *padding factor*. The `scale` argument affects this value. The `dataSize` (page 184) will not decrease when *documents* shrink, but will decrease when you remove documents.

storageSize
The total amount of space allocated to collections in this database for *document* storage. The `scale` argument affects this value. The `storageSize` (page 184) does not decrease as you remove or shrink documents.

numExtents
Contains a count of the number of extents in the database across all collections.

indexes
Contains a count of the total number of indexes across all collections in the database.

indexSize
The total size of all indexes created on this database. The `scale` arguments affects this value.

fileSize
The total size of the data files that hold the database. This value includes preallocated space and the *padding factor*. The value of `fileSize` (page 184) only reflects the size of the data files for the database and not the namespace file.

The `scale` argument affects this value.

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.

See Also:

The `nssize` option, and *Maximum Namespace File Size* (page 265)

CHAPTER

TWELVE

COLLECTION STATISTICS REFERENCE

12.1 Synopsis

To fetch collection statistics, call the `db.collection.stats()` method on a collection object in the `mongo` shell:

```
db.collection.stats()
```

You may also use the literal command format:

```
db.runCommand( { collStats: "collection" } )
```

Replace `collection` in both examples with the name of the collection you want statistics for. By default, the return values will appear in terms of bytes. You can, however, enter a `scale` argument. For example, you can convert the return values to kilobytes like so:

```
db.collection.stats(1024)
```

Or:

```
db.runCommand( { collStats: "collection", scale: 1024 } )
```

Note: The `scale` argument rounds values to whole numbers. This can produce unpredictable and unexpected results in some situations.

See Also:

The documentation of the "`collStats` (page 36)" command and the "`db.collection.stats()`," method in the `mongo` shell.

12.2 Example Document

The output of `db.collection.stats()` resembles the following:

```
{
     "ns" : "<database>.<collection>",
     "count" : <number>,
     "size" : <number>,
     "avgObjSize" : <number>,
```

```
    "storageSize" : <number>,
    "numExtents" : <number>,
    "nindexes" : <number>,
    "lastExtentSize" : <number>,
    "paddingFactor" : <number>,
    "systemFlags" : <bit>,
    "userFlags" : <bit>,
    "totalIndexSize" : <number>,
    "indexSizes" : {
            "_id_" : <number>,
            "a_1" : <number>
    },
    "ok" : 1
}
```

12.3 Fields

ns
The namespace of the current collection, which follows the format `[database].[collection]`.

count
The number of objects or documents in this collection.

size
The size of the data stored in this collection. This value does not include the size of any indexes associated with the collection, which the `totalIndexSize` (page 187) field reports.

The `scale` argument affects this value.

avgObjSize
The average size of an object in the collection. The `scale` argument affects this value.

storageSize
The total amount of storage allocated to this collection for *document* storage. The `scale` argument affects this value. The `storageSize` (page 184) does not decrease as you remove or shrink documents.

numExtents
The total number of contiguously allocated data file regions.

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` may not have an `_id` index.

lastExtentSize
The size of the last extent allocated. The `scale` argument affects this value.

paddingFactor
The amount of space added to the end of each document at insert time. The document padding provides a small amount of extra space on disk to allow a document to grow slightly without needing to move the document. `mongod` automatically calculates this padding factor

flags
Changed in version 2.2: Removed in version 2.2 and replaced with the `userFlags` (page 187) and `systemFlags` (page 186) fields. Indicates the number of flags on the current collection. In version 2.0, the only flag notes the existence of an *index* on the *_id* field.

systemFlags
New in version 2.2. Reports the flags on this collection that reflect internal server options. Typically this value is `1` and reflects the existence of an *index* on the `_id` field.

userFlags
New in version 2.2. Reports the flags on this collection set by the user. In version 2.2 the only user flag is `usePowerOf2Sizes` (page 38). If `usePowerOf2Sizes` (page 38) is enabled, `userFlags` (page 187) will be set to `1`, otherwise `userFlags` (page 187) will be `0`.

See the `collMod` (page 38) command for more information on setting user flags and *usePowerOf2Sizes* (page 38).

totalIndexSize
The total size of all indexes. The `scale` argument affects this value.

indexSizes
This field specifies the key and size of every existing index on the collection. The `scale` argument affects this value.

CHAPTER

THIRTEEN

CONNECTION POOL STATISTICS REFERENCE

13.1 Synopsis

mongos (page 256) instances maintain a pool of connections for interacting with constituent members of the *sharded cluster*. Additionally, mongod instances maintain connection with other shards in the cluster for migrations. The connPoolStats (page 48) command returns statistics regarding these connections between the mongos (page 256) and mongod instances or between the mongod instances in a shard cluster.

Note: connPoolStats (page 48) only returns meaningful results for mongos (page 256) instances and for mongod instances in sharded clusters.

13.2 Output

hosts

The sub-documents of the hosts (page 189) *document* report connections between the mongos (page 256) or mongod instance and each component mongod of the *sharded cluster*.

hosts.[host].**available**

hosts.[host].available (page 189) reports the total number of connections that the mongos (page 256) or mongod could use to connect to this mongod.

hosts.[host].**created**

hosts.[host].created (page 189) reports the number of connections that this mongos (page 256) or mongod has ever created for this host.

replicaSets

replicaSets (page 189) is a *document* that contains *replica set* information for the *sharded cluster*.

replicaSets.**shard**

The replicaSets.shard (page 189) *document* reports on each *shard* within the *sharded cluster*

replicaSets.[shard].**host**

The replicaSets.[shard].host (page 189) 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 193) values.

replicaSets.[shard].host[n].**addr**
replicaSets.[shard].host[n].addr (page 189) reports the address for the host in the *sharded cluster* in the format of "[hostname]:[port]".

replicaSets.[shard].host[n].**ok**
replicaSets.[shard].host[n].ok (page 190) reports false when:

- the mongos (page 256) or mongod cannot connect to instance.
- the mongos (page 256) or mongod received a connection exception or error.

This field is for internal use.

replicaSets.[shard].host[n].**ismaster**
replicaSets.[shard].host[n].ismaster (page 190) reports true if this replicaSets.[shard].host (page 189) is the *primary* member of the *replica set*.

replicaSets.[shard].host[n].**hidden**
replicaSets.[shard].host[n].hidden (page 190) reports true if this replicaSets.[shard].host (page 189) is a *hidden member* of the *replica set*.

replicaSets.[shard].host[n].**secondary**
replicaSets.[shard].host[n].secondary (page 190) reports true if this replicaSets.[shard].host (page 189) is a *secondary* member of the *replica set*.

replicaSets.[shard].host[n].**pingTimeMillis**
replicaSets.[shard].host[n].pingTimeMillis (page 190) reports the ping time in milliseconds from the mongos (page 256) or mongod to this replicaSets.[shard].host (page 189).

replicaSets.[shard].host[n].**tags**
New in version 2.2. replicaSets.[shard].host[n].tags (page 190) reports the members[n].tags, if this member of the set has tags configured.

replicaSets.[shard].**master**
replicaSets.[shard].master (page 190) reports the ordinal identifier of the host in the replicaSets.[shard].host (page 189) array that is the *primary* of the *replica set*.

replicaSets.[shard].**nextSlave**
Deprecated since version 2.2. replicaSets.[shard].nextSlave (page 190) reports the *secondary* member that the mongos (page 256) will use to service the next request for this *replica set*.

createdByType
createdByType (page 190) *document* reports the number of each type of connection that mongos (page 256) or mongod has created in all connection pools.

mongos (page 256) connect to mongod instances using one of three types of connections. The following sub-document reports the total number of connections by type.

createdByType.**master**
createdByType.master (page 190) reports the total number of connections to the *primary* member in each *cluster*.

createdByType.**set**
createdByType.set (page 190) reports the total number of connections to a *replica set* member.

createdByType.**sync**
createdByType.sync (page 190) reports the total number of *config database* connections.

totalAvailable
totalAvailable (page 190) reports the running total of connections from the mongos (page 256) or

mongod to all mongod instances in the *sharded cluster* available for use. This value does not reflect those connections that

totalCreated
totalCreated (page 191) reports the total number of connections ever created from the mongos (page 256) or mongod to all mongod instances in the *sharded cluster*.

numDBClientConnection
numDBClientConnection (page 191) reports the total number of connections from the mongos (page 256) or mongod to all of the mongod instances in the *sharded cluster*.

numAScopedConnection
numAScopedConnection (page 191) reports the number of exception safe connections created from mongos (page 256) or mongod to all mongod in the *sharded cluster*. The mongos (page 256) or mongod releases these connections after receiving a socket exception from the mongod.

CHAPTER

FOURTEEN

REPLICA SET STATUS REFERENCE

The `replSetGetStatus` provides an overview of the current status of a *replica set*. Issue the following command against the *admin database*, in the `mongo` shell:

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

You can also use the following helper in the `mongo` shell to access this functionality

```
rs.status()
```

The value specified (e.g `1` above,) does not impact 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.

Note: The `mongod` must have replication enabled and be a member of a replica set for the for `replSetGetStatus` to return successfully.

See Also:

"`rs.status()`" shell helper function, "`http://docs.mongodb.org/manual/replication`".

14.1 Fields

`rs.status.`**`set`**

The `set` value is the name of the replica set, configured in the `replSet` setting. This is the same value as `_id` in `rs.conf()`.

`rs.status.`**`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 `members.lastHeartbeat` (page 195) to find the operational lag between the current host and the other hosts in the set.

`rs.status.`**`myState`**

The value of `myState` (page 193) reflects state of the current replica set member. An integer between `0` and `10` represents the state of the member. These integers map to states, as described in the following table:

Number	State
0	Starting up, phase 1 (parsing configuration)
1	Primary
2	Secondary
3	Recovering (initial syncing, post-rollback, stale members)
4	Fatal error
5	Starting up, phase 2 (forking threads)
6	Unknown state (the set has never connected to the member)
7	Arbiter
8	Down
9	Rollback
10	Removed

`rs.status.`**`members`**

The `members` field holds an array that contains a document for every member in the replica set. See the "*Member Statuses* (page 194)" for an overview of the values included in these documents.

`rs.status.`**`syncingTo`**

The `syncingTo` field is only present on the output of `rs.status()` on *secondary* and recovering members, and holds the hostname of the member from which this instance is syncing.

14.2 Member Statuses

`members.`**`name`**

The `name` field holds the name of the server.

`members.`**`self`**

The `self` field is only included in the document for the current `mongod` instance in the members array. It's value is `true`.

`members.`**`errmsg`**

This field contains the most recent error or status message received from the member. This field may be empty (e.g. `""`) in some cases.

`members.`**`health`**

The `health` value is only present for the other members of the replica set (i.e. not the member that returns `rs.status`.) This field conveys if the member is up (i.e. `1`) or down (i.e. `0`.)

`members.`**`state`**

The value of the `members.state` (page 194) reflects state of this replica set member. An integer between `0` and `10` represents the state of the member. These integers map to states, as described in the following table:

Number	State
0	Starting up, phase 1 (parsing configuration)
1	Primary
2	Secondary
3	Recovering (initial syncing, post-rollback, stale members)
4	Fatal error
5	Starting up, phase 2 (forking threads)
6	Unknown state (the set has never connected to the member)
7	Arbiter
8	Down
9	Rollback
10	Removed

members.**stateStr**
A string that describes members.state (page 194).

members.**uptime**
The members.uptime (page 195) 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() data.

members.**optime**
A document that contains information regarding the last operation from the operation log that this member has applied.

members.optime.**t**
A 32-bit timestamp of the last operation applied to this member of the replica set from the *oplog*.

members.optime.**i**
An incremented field, which reflects the number of operations in since the last time stamp. This value only increases if there is more than one operation per second.

members.**optimeDate**
An *ISODate* formatted date string that reflects the last entry from the *oplog* that this member applied. If this differs significantly from members.lastHeartbeat (page 195) 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.

members.**lastHeartbeat**
The lastHeartbeat value provides an *ISODate* formatted date of the last heartbeat received from this member. Compare this value to the value of the date (page 193) field to track latency between these members.

This value does not appear for the member that returns the rs.status() data.

members.**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() data.

CHAPTER

FIFTEEN

EXIT CODES AND STATUSES

MongoDB will return one of the following codes and statuses when exiting. Use this guide to interpret logs and when troubleshooting issues with `mongod` and `mongos` (page 256) instances.

0
: Returned by MongoDB applications upon successful exit.

2
: The specified options are in error or are incompatible with other options.

3
: Returned by `mongod` if there is a mismatch between hostnames specified on the command line and in the `local.sources` (page 260) collection. `mongod` may also return this status if *oplog* collection in the `local` database is not readable.

4
: The version of the database is different from the version supported by the `mongod` (or `mongod.exe`) instance. The instance exits cleanly. Restart `mongod` with the *--upgrade* (page 212) option to upgrade the database to the version supported by this `mongod` instance.

5
: Returned by `mongod` if a `moveChunk` (page 60) operation fails to confirm a commit.

12
: Returned by the `mongod.exe` process on Windows when it receives a Control-C, Close, Break or Shutdown event.

14
: Returned by MongoDB applications which encounter an unrecoverable error, an uncaught exception or uncaught signal. The system exits without performing a clean shut down.

20
: *Message:* `ERROR: wsastartup failed <reason>`

  Returned by MongoDB applications on Windows following an error in the WSAStartup function.

  Message: `NT Service Error`

  Returned by MongoDB applications for Windows due to failures installing, starting or removing the NT Service for the application.

45
: Returned when a MongoDB application cannot open a file or cannot obtain a lock on a file.

47
: MongoDB applications exit cleanly following a large clock skew (32768 milliseconds) event.

48

`mongod` exits cleanly if the server socket closes. The server socket is on port `27017` by default, or as specified to the *--port* (page 208) run-time option.

49

Returned by `mongod.exe` or `mongos.exe` on Windows when either receives a shutdown message from the *Windows Service Control Manager*.

100

Returned by `mongod` when the process throws an uncaught exception.

CHAPTER

SIXTEEN

CURRENT OPERATION REPORTING

Changed in version 2.2.

16.1 Example Output

The `db.currentOp()` helper in the `mongo` shell reports on the current operations running on the `mongod` instance. The operation returns the `inprog` array, which contains a document for each in progress operation. Consider the following example output:

```
{
  "inprog": [
              {
                      "opid" : 3434473,
                      "active" : <boolean>,
                      "secs_running" : 0,
                      "op" : "<operation>",
                      "ns" : "<database>.<collection>",
                      "query" : {
                      },
                      "client" : "<host>:<outgoing>",
                      "desc" : "conn57683",
                      "threadId" : "0x7f04a637b700",
                      "connectionId" : 57683,
                      "locks" : {
                              "^" : "w",
                              "^local" : "W",
                              "^<database>" : "W"
                      },
                      "waitingForLock" : false,
                      "msg": "<string>"
                      "numYields" : 0,
                      "progress" : {
                              "done" : <number>,
                              "total" : <number>
                      }
                      "lockStats" : {
                              "timeLockedMicros" : {
                                      "R" : NumberLong(),
                                      "W" : NumberLong(),
                                      "r" : NumberLong(),
                                      "w" : NumberLong()
                              },
```

```
                                "timeAcquiringMicros" : {
                                        "R" : NumberLong(),
                                        "W" : NumberLong(),
                                        "r" : NumberLong(),
                                        "w" : NumberLong()
                                }
                        }
                },
            ]
}
```

Optional

You may specify the `true` argument to `db.currentOp()` to return a more verbose output including idle connections and system operations. For example:

```
db.currentOp(true)
```

Furthermore, active operations (i.e. where `active` (page 201) is `true`) will return additional fields.

16.2 Operations

You can use the `db.killOp()` in conjunction with the `opid` (page 201) field to terminate a currently running operation. Consider the following JavaScript operations for the `mongo` shell that you can use to filter the output of identify specific types of operations:

- Return all pending write operations:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.waitingForLock && d.lockType != "read")
        printjson(d)
     })
```

- Return the active write operation:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.active && d.lockType == "write")
        printjson(d)
     })
```

- Return all active read operations:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.active && d.lockType == "read")
        printjson(d)
     })
```

16.3 Output Reference

opid

Holds an identifier for the operation. You can pass this value to `db.killOp()` in the `mongo` shell to terminate the operation.

active

A boolean value, that is `true` if the operation is currently running or `false` if the operation is queued and waiting for a lock to run.

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.

op

A string that identifies the type of operation. The possible values are:

- `insert`
- `query`
- `update`
- `remove`
- `getmore`
- `command`

ns

The *namespace* the operation targets. MongoDB forms namespaces using the name of the *database* and the name of the *collection*.

query

A document containing the current operation's query. The document is empty for operations that do not have queries: `getmore`, `insert`, and `command`.

client

The IP address (or hostname) and the ephemeral port of the client connection where the operation originates. If your `inprog` array has operations from many different clients, use this string to relate operations to clients.

For some commands, including `findAndModify` (page 25) and `db.eval()`, the client will be `0.0.0.0:0`, rather than an actual client.

desc

A description of the client. This string includes the `connectionId` (page 201).

threadId

An identifier for the thread that services the operation and its connection.

connectionId

An identifier for the connection where the operation originated.

locks

New in version 2.2. The `locks` (page 255) document reports on the kinds of locks the operation currently holds. The following kinds of locks are possible:

`locks.`**^**

`locks.^` (page 201) reports on the global lock state for the `mongod` instance. The operation must hold this for some global phases of an operation.

`locks.`**^local**

`locks.^` (page 201) reports on the lock for the `local` database. MongoDB uses the `local` database for a number of operations, but the most frequent use of the `local` database is for the *oplog* used in replication.

locks.**^<database>**

locks.^ reports on the lock state for the database that this operation targets.

locks (page 255) replaces lockType in earlier versions.

lockType

Changed in version 2.2: The locks (page 255) replaced the lockType (page 202) field in 2.2. Identifies the type of lock the operation currently holds. The possible values are:

- read
- write

waitingForLock

Returns a boolean value. waitingForLock (page 202) is true if the operation is waiting for a lock and false if the operation has the required lock.

msg

The msg (page 202) 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.

progress

Reports on the progress of mapReduce or indexing operations. The progress (page 202) fields corresponds to the completion percentage in the msg (page 202) field. The progress (page 202) specifies the following information:

progress.**done**

Reports the number completed.

progress.**total**

Reports the total number.

killed

Returns true if mongod instance is in the process of killing the operation.

numYields

numYields (page 202) 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.

lockStats

New in version 2.2. The lockStats document reflects the amount of time the operation has spent both acquiring and holding locks. lockStats reports data on a per-lock type, with the following possible lock types:

- R represents the global read lock,
- W represents the global write lock,
- r represents the database specific read lock, and
- w represents the database specific write lock.

timeLockedMicros

The timeLockedMicros (page 202) document reports the amount of time the operation has spent holding a specific lock.

timeLockedMicros.**R**

Reports the amount of time in microseconds the operation has held the global read lock.

timeLockedMicros.**W**

Reports the amount of time in microseconds the operation has held the global write lock.

`timeLockedMicros.r`
Reports the amount of time in microseconds the operation has held the database specific read lock.

`timeLockedMicros.w`
Reports the amount of time in microseconds the operation has held the database specific write lock.

timeAcquiringMicros
The `timeAcquiringMicros` (page 203) document reports the amount of time the operation has spent *waiting* to acquire a specific lock.

`timeAcquiringMicros.R`
Reports the mount of time in microseconds the operation has waited for the global read lock.

`timeAcquiringMicros.W`
Reports the mount of time in microseconds the operation has waited for the global write lock.

`timeAcquiringMicros.r`
Reports the mount of time in microseconds the operation has waited for the database specific read lock.

`timeAcquiringMicros.w`
Reports the mount of time in microseconds the operation has waited for the database specific write lock.

Part VI

Program and Tool Reference Pages

CHAPTER

SEVENTEEN

MONGODB PACKAGE COMPONENTS

17.1 Core Processes

The core components in the MongoDB package are: `mongod`, the core database process; `mongos` (page 256) the controller and query router for *sharded clusters*; and `mongo` the interactive MongoDB Shell.

17.1.1 `mongod`

Synopsis

`mongod` is the primary daemon process for the MongoDB system. It handles data requests, manages data format, and performs background management operations.

This document provides a complete overview of all command line options for `mongod`. These options are primarily useful for testing purposes. In common operation, use the `configuration file options` to control the behavior of your database, which is fully capable of all operations described below.

Options

`mongod`

`--help, -h`
Returns a basic help and usage text.

`--version`
Returns the version of the `mongod` daemon.

`--config` `<filename>`**`, -f`** `<filename>`
Specifies a configuration file, that you can use to specify runtime-configurations. While the options are equivalent and accessible via the other command line arguments, the configuration file is the preferred method for runtime configuration of mongod. See the "`http://docs.mongodb.org/manual/reference/configuration-options`" document for more information about these options.

`--verbose, -v`
Increases the amount of internal reporting returned on standard output or in the log file specified by *`--logpath`* (page 208). Use the `-v` form to control the level of verbosity by including the option multiple times, (e.g. `-vvvvv`.)

`--quiet`
Runs the `mongod` instance in a quiet mode that attempts to limit the amount of output. This option suppresses:

- output from *database commands*, including `drop` (page 34), `dropIndexes` (page 41), `diagLogging` (page 56), `validate` (page 49), and `clean` (page 57).
- replication activity.
- connection accepted events.
- connection closed events.

--port <port>
Specifies a TCP port for the `mongod` to listen for client connections. By default `mongod` listens for connections on port 27017.

UNIX-like systems require root privileges to use ports with numbers lower than 1000.

--bind_ip <ip address>
The IP address that the `mongod` process will bind to and listen for connections. By default `mongod` listens for connections on the localhost (i.e. `127.0.0.1` address.) You may attach `mongod` to any interface; however, if you attach `mongod` to a publicly accessible interface ensure that you have implemented proper authentication and/or firewall restrictions to protect the integrity of your database.

--maxConns <number>
Specifies the maximum number of simultaneous connections that `mongod` will accept. This setting will have no effect if it is higher than your operating system's configured maximum connection tracking threshold.

Note: You cannot set `maxConns` to a value higher than *20000*.

--objcheck
Forces the `mongod` to validate all requests from clients upon receipt to ensure that invalid objects are never inserted into the database. Enabling this option will produce some performance impact, and is not enabled by default.

--logpath <path>
Specify a path for the log file that will hold all diagnostic logging information.

Unless specified, `mongod` will output all log information to the standard output. Additionally, unless you also specify *`--logappend`* (page 208), the logfile will be overwritten when the process restarts.

Note: The behavior of the logging system may change in the near future in response to the SERVER-4499 case.

--logappend
When specified, this option ensures that `mongod` appends new entries to the end of the logfile rather than overwriting the content of the log when the process restarts.

--syslog
New in version 2.1.0. Sends all logging output to the host's *syslog* system rather than to standard output or a log file as with *`--logpath`* (page 208).

Warning: You cannot use *`--syslog`* (page 208) with *`--logpath`* (page 208).

--pidfilepath <path>
Specify a file location to hold the "*PID*" or process ID of the `mongod` process. Useful for tracking the `mongod` process in combination with the *`mongod --fork`* (page 209) option.

If this option is not set, `mongod` will create no PID file.

--keyFile <file>
Specify the path to a key file to store authentication information. This option is only useful for the connection between replica set members.

See Also:

"*Replica Set Security*" and "`http://docs.mongodb.org/manual/administration/replica-sets`."

`--nounixsocket`
Disables listening on the UNIX socket. Unless set to false, `mongod` and `mongos` (page 256) provide a UNIX-socket.

`--unixSocketPrefix` `<path>`
Specifies a path for the UNIX socket. Unless this option has a value, `mongod` and `mongos` (page 256), create a socket with the `http://docs.mongodb.org/manual/tmp` as a prefix.

`--fork`
Enables a *daemon* mode for `mongod` that runs the process to the background. This is the normal mode of operation, in production and production-like environments, but may *not* be desirable for testing.

`--auth`
Enables database authentication for users connecting from remote hosts. configure users via the *mongo shell shell* (page 217). If no users exist, the localhost interface will continue to have access to the database until the you create the first user.

See the "Security and Authentication wiki page for more information regarding this functionality.

`--cpu`
Forces `mongod` to report the percentage of CPU time in write lock. `mongod` generates output every four seconds. MongoDB writes this data to standard output or the logfile if using the `logpath` option.

`--dbpath` `<path>`
Specify a directory for the `mongod` instance to store its data. Typical locations include: `http://docs.mongodb.org/manual/srv/mongodb`, `http://docs.mongodb.org/manual/var/lib/mongodb` or `http://docs.mongodb.org/manual/opt/mon`

Unless specified, `mongod` will look for data files in the default `http://docs.mongodb.org/manual/data/db` directory. (Windows systems use the `\data\db` directory.) If you installed using a package management system. Check the `http://docs.mongodb.org/manual/etc/mongodb.conf` file provided by your packages to see the configuration of the `dbpath`.

`--diaglog` `<value>`
Creates a very verbose, *diagnostic log* for troubleshooting and recording various errors. MongoDB writes these log files in the `dbpath` directory in a series of files that begin with the string `diaglog` and end with the initiation time of the logging as a hex string.

The specified value configures the level of verbosity. Possible values, and their impact are as follows.

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` tool to replay this output for investigation. Given a typical diaglog file, located at `http://docs.mongodb.org/manual/data/db/diaglog.4f76a58c`, you might use a command in the following form to read these files:

```
mongosniff --source DIAGLOG /data/db/diaglog.4f76a58c
```

`--diaglog` (page 209) is for internal use and not intended for most users.

> **Warning:** Setting the diagnostic level to `0` will cause `mongod` to stop writing data to the *diagnostic log* file. However, the `mongod` 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` instance before doing so.

--directoryperdb
Alters the storage pattern of the data directory to store each database's files in a distinct folder. This option will create directories within the *--dbpath* (page 209) named for each directory.

Use this option in conjunction with your file system and device configuration so that MongoDB will store data on a number of distinct disk devices to increase write throughput or disk capacity.

--journal
Enables operation journaling to ensure write durability and data consistency. `mongod` enables journaling by default on 64-bit builds of versions after 2.0.

--journalOptions <arguments>
Provides functionality for testing. Not for general use, and may affect database integrity.

--journalCommitInterval <value>
Specifies the maximum amount of time for `mongod` to allow between journal operations. The default value is 100 milliseconds, while possible values range from 2 to 300 milliseconds. Lower values increase the durability of the journal, at the expense of disk performance.

To force `mongod` to commit to the journal more frequently, you can specify `j:true`. When a write operation with `j:true` pending, `mongod` will reduce `journalCommitInterval` to a third of the set value.

--ipv6
Specify this option to enable IPv6 support. This will allow clients to connect to `mongod` using IPv6 networks. `mongod` disables IPv6 support by default in `mongod` and all utilities.

--jsonp
Permits *JSONP* access via an HTTP interface. Consider the security implications of allowing this activity before enabling this option.

--noauth
Disable authentication. Currently the default. Exists for future compatibility and clarity.

--nohttpinterface
Disables the HTTP interface.

--nojournal
Disables the durability journaling. By default, `mongod` enables journaling in 64-bit versions after v2.0.

--noprealloc
Disables the preallocation of data files. This will shorten the start up time in some cases, but can cause significant performance penalties during normal operations.

--noscripting
Disables the scripting engine.

--notablescan
Forbids operations that require a table scan.

--nssize <value>
Specifies the default size for namespace files (i.e `.ns`). This option has no impact on the size of existing namespace files. The maximum size is 2047 megabytes.

The default value is 16 megabytes; this provides for approximately 24,000 namespaces. Each collection, as well as each index, counts as a namespace.

--profile <level>

Changes the level of database profiling, which inserts information about operation performance into output of `mongod` or the log file. The following levels are available:

Level	Setting
0	Off. No profiling.
1	On. Only includes slow operations.
2	On. Includes all operations.

Profiling is off by default. Database profiling can impact database performance. Enable this option only after careful consideration.

--quota

Enables a maximum limit for the number data files each database can have. When running with *--quota* (page 211), there are a maximum of 8 data files per database. Adjust the quota with the *--quotaFiles* (page 211) option.

--quotaFiles <number>

Modify limit on the number of data files per database. This option requires the *--quota* (page 211) setting. The default value for *--quotaFiles* (page 211) is 8.

--rest

Enables the simple *REST* API.

--repair

Runs a repair routine on all databases. This is equivalent to shutting down and running the `repairDatabase` (page 43) database command on all databases.

> **Warning:** In general, if you have an intact copy of your data, such as would exist on a very recent backup or an intact member of a *replica set*, **do not** use `repairDatabase` (page 43) or related options like `db.repairDatabase()` in the `mongo` shell or *mongod --repair* (page 211). Restore from an intact copy of your data.

Note: When using *journaling*, there is almost never any need to run `repairDatabase` (page 43). In the event of an unclean shutdown, the server will be able 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, `mongod` will refuse to start. In these cases you should start `mongod` without the *--repair* (page 211) option to allow `mongod` to recover data from the journal. This will complete more quickly and will result in a more consistent and complete data set.

To continue the repair operation despite the journal files, shut down `mongod` cleanly and restart with the *--repair* (page 211) option.

--repairpath <path>

Specifies the root directory containing MongoDB data files, to use for the *--repair* (page 211) operation. Defaults to the value specified by *--dbpath* (page 209).

--slowms <value>

Defines the value of "slow," for the *--profile* (page 210) option. The database logs all slow queries to the log, even when the profiler is not turned on. When the database profiler is on, `mongod` the profiler writes to the `system.profile` collection. See the `profile` command for more information on the database profiler.

--smallfiles

Enables a mode where MongoDB uses a smaller default file size. Specifically, *--smallfiles* (page 211) reduces the initial size for data files and limits them to 512 megabytes. *--smallfiles* (page 211) also reduces the size of each *journal* files from 1 gigabyte to 128 megabytes.

Use *--smallfiles* (page 211) if you have a large number of databases that each holds a small quantity of data. *--smallfiles* (page 211) can lead your mongod to create a large number of files, which may affect performance for larger databases.

--shutdown
Used in *control scripts*, the *--shutdown* (page 212) will cleanly and safely terminate the mongod process. When invoking mongod with this option you must set the *--dbpath* (page 209) option either directly or by way of the configuration file and the *--config* (page 207) option.

--syncdelay <value>
mongod writes data very quickly to the journal, and lazily to the data files. *--syncdelay* (page 212) controls how much time can pass before MongoDB flushes data to the datafiles via an *fsync* operation. The default setting is 60 seconds. We recommend almost always using the default setting of 60.

The serverStatus command reports the background flush thread's status via the backgroundFlushing field.

Note: If *--syncdelay* (page 212) is 0, mongod flushes all operations to disk immediately, which has a significant impact on performance. Run with journal enabled, which is the default for 64-bit MongoDB builds.

--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.

--upgrade
Upgrades the on-disk data format of the files specified by the *--dbpath* (page 209) to the latest version, if needed.

This option only affects the operation of mongod if the data files are in an old format.

Note: 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 (on the download page) for more information about the upgrade process.

--traceExceptions
For internal diagnostic use only.

Replication Options

--replSet <setname>
Use this option to configure replication with replica sets. Specify a setname as an argument to this set. All hosts must have the same set name.

See Also:

"http://docs.mongodb.org/manual/replication," "http://docs.mongodb.org/manual/administrat
and "http://docs.mongodb.org/manual/reference/replica-configuration"

--oplogSize <value>
Specifies a maximum size in megabytes for the replication operation log (e.g. *oplog*.) By mongod creates an *oplog* based on the maximum amount of space available. For 64-bit systems, the op log is typically 5% of available disk space.

Once the mongod has created the oplog for the first time, changing *--oplogSize* (page 212) will not affect the size of the oplog.

--fastsync

In the context of *replica set* replication, set this option if you have seeded this replica with a snapshot of the *dbpath* of another member of the set. Otherwise the `mongod` will attempt to perform a full sync.

> **Warning:** If the data is not perfectly synchronized *and* `mongod` starts with `fastsync`, then the secondary or slave will be permanently out of sync with the primary, which may cause significant consistency problems.

--replIndexPrefetch

New in version 2.2. You must use *--replIndexPrefetch* (page 213) in conjunction with `replSet`. 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` from loading *any* index into memory.

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.

--master

Configures `mongod` to run as a replication *master*.

--slave

Configures `mongod` to run as a replication *slave*.

--source `<host>:<port>`

For use with the *--slave* (page 213) option, the `--source` option designates the server that this instance will replicate.

--only `<arg>`

For use with the *--slave* (page 213) option, the `--only` option specifies only a single *database* to replicate.

--slavedelay `<value>`

For use with the *--slave* (page 213) option, the `--slavedelay` option configures a "delay" in seconds, for this slave to wait to apply operations from the *master* node.

--autoresync

For use with the *--slave* (page 213) option, the *--autoresync* (page 213) option allows this slave to automatically resync if the local data is more than 10 seconds behind the master. This option may be problematic if the *oplog* is too small (controlled by the *--oplogSize* (page 212) option.) If the *oplog* not large enough to store the difference in changes between the master's current state and the state of the slave, this node will forcibly resync itself unnecessarily. When you set the If the *--autoresync* (page 213) option the slave will not attempt an automatic resync more than once in a ten minute period.

Sharding Cluster Options

--configsvr

Declares that this `mongod` instance serves as the *config database* of a sharded cluster. When running with this option, clients will not be able to write data to any database other than `config` and `admin`. The default port for `mongod` with this option is `27019` and `mongod` writes all data files to the http://docs.mongodb.org/manual/configdb sub-directory of the *--dbpath* (page 209) directory.

--shardsvr
Configures this mongod instance as a shard in a partitioned cluster. The default port for these instances is 27018. The only effect of *--shardsvr* (page 214) is to change the port number.

--noMoveParanoia
Disables a "paranoid mode" for data writes for chunk migration operation. See the *chunk migration* and moveChunk (page 60) command documentation for more information.

By default mongod will save copies of migrated chunks on the "from" server during migrations as "paranoid mode." Setting this option disables this paranoia.

Usage

In common usage, the invocation of mongod will resemble the following in the context of an initialization or control script:

```
mongod --config /etc/mongodb.conf
```

See the "http://docs.mongodb.org/manual/reference/configuration-options" for more information on how to configure mongod using the configuration file.

17.1.2 mongos

Synopsis

mongos (page 256) 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 256) instance behaves identically to any other MongoDB instance.

See Also:

See the "Sharding" wiki page for more information regarding MongoDB's sharding functionality.

Note: Changed in version 2.1. Some aggregation operations using the aggregate (page 29) will cause mongos (page 256) instances to require more CPU resources than in previous versions. This modified performance profile may dictate alternate architecture decisions if you make use the *aggregation framework* extensively in a sharded environment.

Options

mongos

--help, -h
Returns a basic help and usage text.

--version
Returns the version of the mongod daemon.

--config <filename>**, -f** <filename>
Specifies a configuration file, that you can use to specify runtime-configurations. While the options are equivalent and accessible via the other command line arguments, the configuration file is the preferred method for runtime configuration of mongod. See the "`http://docs.mongodb.org/manual/reference/configuration-options`" document for more information about these options.

Not all configuration options for `mongod` make sense in the context of `mongos` (page 256).

--verbose, -v
Increases the amount of internal reporting returned on standard output or in the log file specified by *`--logpath`* (page 215). Use the `-v` form to control the level of verbosity by including the option multiple times, (e.g. `-vvvvv`.)

--quiet
Runs the `mongos` (page 256) instance in a quiet mode that attempts to limit the amount of output.

--port <port>
Specifies a TCP port for the `mongos` (page 256) to listen for client connections. By default `mongos` (page 256) listens for connections on port 27017.

UNIX-like systems require root access to access ports with numbers lower than 1000.

--bind_ip <ip address>
The IP address that the `mongos` (page 256) process will bind to and listen for connections. By default `mongos` (page 256) listens for connections on the localhost (i.e. `127.0.0.1` address.) You may attach `mongos` (page 256) to any interface; however, if you attach `mongos` (page 256) to a publicly accessible interface you must implement proper authentication or firewall restrictions to protect the integrity of your database.

--maxConns <number>
Specifies the maximum number of simultaneous connections that `mongos` (page 256) will accept. This setting will have no effect if the value of this setting is higher than your operating system's configured maximum connection tracking threshold.

This is particularly useful for `mongos` (page 256) if you have a client that creates a number of collections but allows them to timeout rather than close the collections. When you set `maxConns`, ensure the value is slightly higher than the size of the connection pool or the total number of connections to prevent erroneous connection spikes from propagating to the members of a *shard* cluster.

Note: You cannot set `maxConns` to a value higher than *20000*.

--objcheck
Forces the `mongos` (page 256) to validate all requests from clients upon receipt to ensure that invalid objects are never inserted into the database. This option has a performance impact, and is not enabled by default.

--logpath <path>
Specify a path for the log file that will hold all diagnostic logging information.

Unless specified, `mongos` (page 256) will output all log information to the standard output. Additionally, unless you also specify *`--logappend`* (page 215), the logfile will be overwritten when the process restarts.

--logappend
Specify to ensure that `mongos` (page 256) appends additional logging data to the end of the logfile rather than overwriting the content of the log when the process restarts.

--syslog
New in version 2.1.0. Sends all logging output to the host's *syslog* system rather than to standard output or a log file as with *`--logpath`* (page 215).

Warning: You cannot use `--syslog` (page 215) with `--logpath` (page 215).

--pidfilepath `<path>`
Specify a file location to hold the "*PID*" or process ID of the `mongod` process. Useful for tracking the `mongod` process in combination with the `mongos --fork` (page 216) option.

Without this option, `mongos` (page 256) will create a PID file.

--keyFile `<file>`
Specify the path to a key file to store authentication information. This option is only useful for the connection between `mongos` (page 256) instances and components of the *sharded cluster*.

See Also:

"*Replica Set Security*" and "`http://docs.mongodb.org/manual/administration/replica-sets`."

--nounixsocket
Disables listening on the UNIX socket. Without this option `mongos` (page 256) creates a UNIX socket.

--unixSocketPrefix `<path>`
Specifies a path for the UNIX socket. Unless specified, `mongos` (page 256) creates a socket in the `http://docs.mongodb.org/manual/tmp` path.

--fork
Enables a *daemon* mode for `mongod` which forces the process to the background. This is the normal mode of operation, in production and production-like environments, but may *not* be desirable for testing.

--configdb `<config1>,<config2><:port>,<config3>`
Set this option to specify a configuration database (i.e. *config database*) for the *sharded cluster*. You must specify either 1 configuration server or 3 configuration servers, in a comma separated list.

Note: `mongos` (page 256) instances read from the first *config server* in the list provided. All `mongos` (page 256) instances **must** specify the hosts to the `--configdb` (page 216) setting in the same order.

If your configuration databases reside in more that one data center, order the hosts in the `--configdb` (page 216) argument so that the config database that is closest to the majority of your `mongos` (page 256) instances is first servers in the list.

Warning: Never remove a config server from the `--configdb` (page 216) parameter, even if the config server or servers are not available, or offline.

--test
This option is for internal testing use only, and runs unit tests without starting a `mongos` (page 256) instance.

--upgrade
This option updates the meta data format used by the *config database*.

--chunkSize `<value>`
The value of the `--chunkSize` (page 216) determines the size of each *chunk* of data distributed around thee *sharded cluster*. The default value is 64 megabytes, which is the ideal size for chunks in most deployments: larger chunk size can lead to uneven data distribution, smaller chunk size often leads to inefficient movement of chunks between nodes. However, in some circumstances it may be necessary to set a different chunk size.

This option *only* sets the chunk size when initializing the cluster for the first time. If you modify the run-time option later, the new value will have no effect. See the "*sharding-balancing-modify-chunk-size*" procedure if you need to change the chunk size on an existing sharded cluster.

--ipv6

Enables IPv6 support to allow clients to connect to `mongos` (page 256) using IPv6 networks. MongoDB disables IPv6 support by default in `mongod` and all utilities.

--jsonp

Permits *JSONP* access via an HTTP interface. Consider the security implications of allowing this activity before enabling this option.

--noscripting

Disables the scripting engine.

--nohttpinterface

New in version 2.1.2. Disables the HTTP interface.

--localThreshold

New in version 2.2. `--localThreshold` (page 217) affects the logic that program:*mongos* uses when selecting *replica set* members to pass reads operations to from clients. Specify a value to `--localThreshold` (page 217) in milliseconds. The default value is `15`, which corresponds to the default value in all of the client `drivers`.

When `mongos` (page 256) receives a request that permits reads to *secondary* members, the `mongos` (page 256) 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 `--localThreshold` (page 217), `mongos` (page 256) will construct the list of replica members that are within the latency allowed by this value.
- The `mongos` (page 256) will select a member to read from at random from this list.

The ping time used for a set member compared by the `--localThreshold` (page 217) 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 256) recalculates the average.

See the *replica-set-read-preference-behavior-member-selection* section of the *read preference* documentation for more information.

17.1.3 mongo

Synopsis

mongo [-shell] [-nodb] [-norc] [-quiet] [-port <port>] [-host <host>] [-eval <JavaScript>]

Description

mongo

`mongo` is an interactive JavaScript shell interface to MongoDB. The `mongo` command provides a powerful interface for systems administrators as well as a way to test queries and operations directly with the database. To increase the flexibility of the `mongo` command, the shell provides a fully functional JavaScript environment. This document addresses the basic invocation of the `mongo` shell and an overview of its usage.

Interface

Options

--shell
Enables the shell interface after evaluating a *JavaScript* file. If you invoke the `mongo` command and specify a JavaScript file as an argument, or use *mongo --eval* (page 218) to specify JavaScript on the command line, the *mongo --shell* (page 218) option provides the user with a shell prompt after the file finishes executing.

--nodb
Prevents the shell from connecting to any database instances.

--norc
Prevents the shell from sourcing and evaluating `~/.mongorc.js` on startup.

--quiet
Silences output from the shell during the connection process.

--port <PORT>
Specifies the port where the `mongod` or `mongos` (page 256) instance is listening. Unless specified `mongo` connects to `mongod` instances on port 27017, which is the default `mongod` port.

--host <HOSTNAME>
specifies the host where the `mongod` or `mongos` (page 256) is running to connect to as `<HOSTNAME>`. By default `mongo` will attempt to connect to a MongoDB process running on the localhost.

--eval <JAVASCRIPT>
Evaluates a JavaScript expression specified as an argument to this option. `mongo` does not load its own environment when evaluating code: as a result many options of the shell environment are not available.

--username <USERNAME>, **-u** <USERNAME>
Specifies a username to authenticate to the MongoDB instance. Use in conjunction with the *mongo --password* (page 218) option to supply a password. If you specify a username and password but the default database or the specified database do not require authentication, `mongo` will exit with an exception.

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongo --username* (page 218) option to supply a username. If you specify a *--username* (page 218) without the *mongo --password* (page 218) option, `mongo` will prompt for a password interactively, if the `mongod` or `mongos` (page 256) requires authentication.

--help, -h
Returns a basic help and usage text.

--version
Returns the version of the shell.

--verbose
Increases the verbosity of the output of the shell during the connection process.

--ipv6
Enables IPv6 support that allows `mongo` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongo`, 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` shell to the *admin database* on the local machine. You may specify a remote database instance, with the resolvable hostname or IP address. Separate the database name from the hostname using a `http://docs.mongodb.org/manual/` character. See the following examples:

```
mongo mongodb1.example.net
mongo mongodb1/admin
mongo 10.8.8.10/test
```

`<file.js>`
Specifies a JavaScript file to run and then exit. Must be the last option specified. Use the *`mongo --shell`* (page 218) option to return to a shell after the file finishes running.

Files

`~/.dbshell`

`mongo` maintains a history of commands in the `.dbshell` file.

Note: Interaction related to authentication, including `authenticate` (page 61) and `db.addUser()` are not saved in the history file.

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` will read *.mongorc.js* from the home directory of the user invoking `mongo`. Specify the *`mongo --norc`* (page 218) option to disable reading *.mongorc.js*.

`http://docs.mongodb.org/manual/tmp/mongo_edit`*`<time_t>`*`.js`

Created by `mongo` when editing a file. If the file exists `mongo` 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` 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 219).

`HOME`
Specifies the path to the home directory where `mongo` `mongo` will read the `.mongorc.js` file and write the `.dbshell` file.

`HOMEDRIVE`
On Windows systems, `HOMEDRIVE` (page 219) specifies the path the directory where `mongo` will read the `.mongorc.js` file and write the `.dbshell` file.

`HOMEPATH`
Specifies the Windows path to the home directory where `mongo` will read the `.mongorc.js` file and write the `.dbshell` file.

Use

Typically users invoke the shell with the `mongo` 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> --hostname <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 218) 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 print return a query as *JSON*, from the system prompt using the *`--eval`* (page 218) 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.

17.2 Windows Services

The `mongod.exe` and `mongos.exe` describe the options available for configuring MongoDB when running as a Windows Service. The `mongod.exe` and `mongos.exe` binaries provide a superset of the `mongod` and `mongos` (page 256) options.

17.2.1 `mongod.exe`

Synopsis

`mongod.exe` is the build of the MongoDB daemon (i.e. `mongod`) for the Windows platform. `mongod.exe` has all of the features of `mongod` on Unix-like platforms and is completely compatible with the other builds of `mongod`. In addition, `mongod.exe` provides several options for interacting with the Windows platform itself.

This document only references options that are unique to `mongod.exe`. All `mongod` options are available. See the "*mongod* (page 207)" and the "`http://docs.mongodb.org/manual/reference/configuration-options`" documents for more information regarding `mongod.exe`.

To install and use `mongod.exe`, read the "`http://docs.mongodb.org/manual/tutorial/install-mongodb-on-wir`
document.

Options

--install
Installs `mongod.exe` as a Windows Service and exits.

--remove
Removes the `mongod.exe` Windows Service. If `mongod.exe` is running, this operation will stop and then remove the service.

Note: *--remove* (page 221) requires the *--serviceName* (page 221) if you configured a non-default *--serviceName* (page 221) during the *--install* (page 221) operation.

--reinstall
Removes `mongod.exe` and reinstalls `mongod.exe` as a Windows Service.

--serviceName `<name>`
Default: "MongoDB"

Set the service name of `mongod.exe` when running as a Windows Service. Use this name with the `net start <name>` and `net stop <name>` operations.

You must use *--serviceName* (page 221) in conjunction with either the *--install* (page 221) or *--remove* (page 221) install option.

--serviceDisplayName `<name>`
Default: "Mongo DB"

Sets the name listed for MongoDB on the Services administrative application.

--serviceDescription `<description>`
Default: "MongoDB Server"

Sets the `mongod.exe` service description.

You must use *--serviceDescription* (page 221) in conjunction with the *--install* (page 221) option.

Note: For descriptions that contain spaces, you must enclose the description in quotes.

--serviceUser `<user>`
Runs the `mongod.exe` service in the context of a certain user. This user must have "Log on as a service" privileges.

You must use *--serviceUser* (page 221) in conjunction with the *--install* (page 221) option.

--servicePassword `<password>`
Sets the password for `<user>` for `mongod.exe` when running with the *--serviceUser* (page 221) option.

You must use *--servicePassword* (page 221) in conjunction with the *--install* (page 221) option.

17.2.2 mongos.exe

Synopsis

`mongos.exe` is the build of the MongoDB Shard (i.e. `mongos` (page 256)) for the Windows platform. `mongos.exe` has all of the features of `mongos` (page 256) on Unix-like platforms and is completely compatible with the other builds of `mongos` (page 256). In addition, `mongos.exe` provides several options for interacting with the Windows platform itself.

This document only references options that are unique to `mongos.exe`. All `mongos` (page 256) options are available. See the “*mongos* (page 214)” and the “`http://docs.mongodb.org/manual/reference/configuration-options`” documents for more information regarding `mongos.exe`.

To install and use `mongos.exe`, read the “`http://docs.mongodb.org/manual/tutorial/install-mongodb-on-wir` document.

Options

--install
Installs `mongos.exe` as a Windows Service and exits.

--remove
Removes the `mongos.exe` Windows Service. If `mongos.exe` is running, this operation will stop and then remove the service.

Note: *--remove* (page 222) requires the *--serviceName* (page 222) if you configured a non-default *--serviceName* (page 222) during the *--install* (page 222) operation.

--reinstall
Removes `mongos.exe` and reinstalls `mongos.exe` as a Windows Service.

--serviceName <name>
Default: “MongoS”

Set the service name of `mongos.exe` when running as a Windows Service. Use this name with the `net start <name>` and `net stop <name>` operations.

You must use *--serviceName* (page 222) in conjunction with either the *--install* (page 222) or *--remove* (page 222) install option.

--serviceDisplayName <name>
Default: “Mongo DB Router”

Sets the name listed for MongoDB on the Services administrative application.

--serviceDescription <description>
Default: “Mongo DB Sharding Router”

Sets the `mongos.exe` service description.

You must use *--serviceDescription* (page 222) in conjunction with the *--install* (page 222) option.

Note: For descriptions that contain spaces, you must enclose the description in quotes.

--serviceUser <user>
Runs the `mongos.exe` service in the context of a certain user. This user must have “Log on as a service” privileges.

You must use *--serviceUser* (page 222) in conjunction with the *--install* (page 222) option.

--servicePassword <password>
Sets the password for <user> for `mongos.exe` when running with the *--serviceUser* (page 222) option.

You must use *--servicePassword* (page 222) in conjunction with the *--install* (page 222) option.

17.3 Binary Import and Export Tools

`mongodump` provides a method for creating *BSON* dump files from the `mongod` instances, while `mongorestore` makes it possible to restore these dumps. `bsondump` converts BSON dump files into *JSON*. The `mongooplog` utility provides the ability to stream *oplog* entries outside of normal replication.

17.3.1 mongodump

Synopsis

`mongodump` is a utility for creating a binary export of the contents of a database. Consider using this utility as part an effective `backup strategy`. Use in conjunction with `mongorestore` to provide restore functionality.

Note: The format of data created by `mongodump` tool from the 2.2 distribution or later is different and incompatible with earlier versions of `mongod`.

See Also:

"`mongorestore`" and "`http://docs.mongodb.org/manual/administration/backups`".

Options

mongodump

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Returns the version of the `mongodump` utility and exits.

--host <hostname><:port>
Specifies a resolvable hostname for the `mongod` that you wish to use to create the database dump. By default `mongodump` will attempt to connect to a MongoDB process ruining on the localhost port number `27017`.

Optionally, specify a port number to connect a MongoDB instance running on a port other than `27017`.

To connect to a replica set, use the *--host* (page 223) argument with a setname, followed by a slash and a comma-separated list of host names and port numbers. The `mongodump` utility will, given the seed of at least one connected set member, connect to the primary member of that set. This option would resemble:

```
mongodump --host repl0/mongo0.example.net,mongo0.example.net:27018,mongo1.example.net,mongo2.exa
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

--port <port>
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *--host* (page 223) option.

--ipv6
Enables IPv6 support that allows `mongodump` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongodump`, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *--password* (page 224) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* (page 223) option to supply a username.

If you specify a *--username* (page 223) without the *--password* (page 224) option, mongodump will prompt for a password interactively.

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 224) option enables mongodump to attach directly to local data files and copy the data without the mongod. To run with *--dbpath* (page 224), mongodump needs to restrict access to the data directory: as a result, no mongod can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 224) in conjunction with the corresponding option to mongod. This option allows mongodump to read data files organized with each database located in a distinct directory. This option is only relevant when specifying the *--dbpath* (page 224) option.

--journal
Allows mongodump operations to use the durability *journal* to ensure that the export is in a consistent state. This option is only relevant when specifying the *--dbpath* (page 224) option.

--db <db>, **-d** <db>
Use the *--db* (page 224) option to specify a database for mongodump to backup. If you do not specify a DB, mongodump copies all databases in this instance into the dump files. Use this option to backup or copy a smaller subset of your data.

--collection <collection>, **-c** <collection>
Use the *--collection* (page 224) option to specify a collection for mongodump to backup. If you do not specify a collection, this option copies all collections in the specified database or instance to the dump files. Use this option to backup or copy a smaller subset of your data.

--out <path>, **-o** <path>
Specifies a path where mongodump and store the output the database dump. To output the database dump to standard output, specify a - rather than a path.

--query <json>, **-q** <json>
Provides a query to limit (optionally) the documents included in the output of mongodump.

--oplog
Use this option to ensure that mongodump creates a dump of the database that includes an *oplog*, to create a point-in-time snapshot of the state of a mongod instance. To restore to a specific point-in-time backup, use the output created with this option in conjunction with *mongorestore --oplogReplay* (page 227).

Without *--oplog* (page 224), 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 224) has no effect when running mongodump against a mongos (page 256) instance to dump the entire contents of a sharded cluster. However, you can use *--oplog* (page 224) to dump individual shards.

Note: *--oplog* (page 224) only works against nodes that maintain a *oplog*. This includes all members of a replica set, as well as *master* nodes in master/slave replication deployments.

--repair

Use this option to run a repair option in addition to dumping the database. The repair option attempts to repair a database that may be in an inconsistent state as a result of an improper shutdown or mongod crash.

--forceTableScan

Forces mongodump to scan the data store directly: typically, mongodump saves entries as they appear in the index of the _id field. Use *--forceTableScan* (page 225) 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 225), mongodump does not use $snapshot. As a result, the dump produced by mongodump can reflect the state of the database at many different points in time.

> **Warning:** Use *--forceTableScan* (page 225) with extreme caution and consideration.

> **Warning:** Changed in version 2.2: When used in combination with fsync (page 39) or db.fsyncLock(), mongod may block some reads, including those from mongodump, when queued write operation waits behind the fsync (page 39) lock.

Behavior

When running mongodump against a mongos (page 256) instance where the *sharded cluster* consists of *replica sets*, the *read preference* of the operation will prefer reads from *secondary* members of the set.

Usage

See the "*backup guide section on database dumps*" for a larger overview of mongodump usage. Also see the "*mongorestore* (page 226)" document for an overview of the mongorestore, which provides the related inverse functionality.

The following command, creates a dump file that contains only the collection named collection in the database named test. In this case the database is running on the local interface on port 27017:

```
mongodump --collection collection --db test
```

In the next example, mongodump creates a backup of the database instance stored in the http://docs.mongodb.org/manual/srv/mongodb directory on the local machine. This requires that no mongod instance is using the http://docs.mongodb.org/manual/srv/mongodb directory.

```
mongodump --dbpath /srv/mongodb
```

In the final example, mongodump creates a database dump located at http://docs.mongodb.org/manual/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 /opt/backup/mongod
```

17.3.2 mongorestore

Synopsis

The mongorestore tool imports content from binary database dump, created by mongodump into a specific database. mongorestore can import content to an existing database or create a new one.

mongorestore, and only performs inserts into the existing database, and does not perform updates or *upserts*. If existing data with the same _id already exists on the target database, mongorestore will *not* replace it.

mongorestore will recreate indexes from the dump

The behavior of mongorestore has the following properties:

- all operations are inserts, not updates.
- all inserts are "fire and forget," mongorestore does not wait for a response from a mongod to ensure that the MongoDB process has received or recorded the operation.

  The mongod will record any errors to its log that occur during a restore operation but mongorestore will not receive errors.

Note: The format of data created by mongodump tool from the 2.2 distribution or later is different and incompatible with earlier versions of mongod.

Options

mongorestore

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

--version
Returns the version of the mongorestore tool.

--host <hostname><:port>
Specifies a resolvable hostname for the mongod to which you want to restore the database. By default mongorestore will attempt 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.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

--port <port>
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. 27017) You may also specify a port number using the *--host* (page 226) command.

--ipv6
Enables IPv6 support that allows mongorestore to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including mongorestore, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *--password* (page 227) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongorestore --username* (page 226) option to supply a username.

If you specify a *--username* (page 226) without the *--password* (page 227) option, mongorestore will prompt for a password interactively.

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 227) option enables mongorestore to attach directly to local data files and insert the data without the mongod. To run with *--dbpath* (page 227), mongorestore needs to lock access to the data directory: as a result, no mongod can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 227) in conjunction with the corresponding option to mongod, which allows mongorestore to import data into MongoDB instances that have every database's files saved in discrete directories on the disk. This option is only relevant when specifying the *--dbpath* (page 227) option.

--journal
Allows mongorestore write to the durability *journal* to ensure that the data files will remain in a consistent state during the write process. This option is only relevant when specifying the *--dbpath* (page 227) option.

--db <db>, **-d** <db>
Use the *--db* (page 227) option to specify a database for mongorestore to restore data *into*. If the database doesn't exist, mongorestore will create the specified database. If you do not specify a <db>, mongorestore 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 227) does *not* control which *BSON* files mongorestore restores. You must use the mongorestore *path option* (page 228) to limit that restored data.

--collection <collection>, **-c** <collection>
Use the *--collection* (page 227) option to specify a collection for mongorestore to restore. If you do not specify a <collection>, mongorestore imports all collections created. Existing data may be overwritten. Use this option to restore data into a MongoDB instance that already has data, or to restore only some data in the specified imported data set.

--objcheck
Verifies each object as a valid *BSON* object before inserting it into the target database. If the object is not a valid *BSON* object, mongorestore will not insert the object into the target database and stop processing remaining documents for import. This option has some performance impact.

--filter '<JSON>'
Limits the documents that mongorestore imports to only those documents that match the JSON document specified as '<JSON>'. Be sure to include the document in single quotes to avoid interaction with your system's shell environment.

--drop
Modifies the restoration procedure to drop every collection from the target database before restoring the collection from the dumped backup.

--oplogReplay
Replays the *oplog* after restoring the dump to ensure that the current state of the database reflects the point-in-time backup captured with the "*mongodump --oplog* (page 224)" command.

--keepIndexVersion
Prevents mongorestore from upgrading the index to the latest version during the restoration process.

--w <number of replicas per write>
New in version 2.2. Specifies the *write concern* for each write operation that mongorestore writes to the target database. By default, mongorestore does not wait for a response for *write acknowledgment*.

--noOptionsRestore
New in version 2.2. Prevents mongorestore from setting the collection options, such as those specified by the collMod (page 38) *database command*, on restored collections.

--noIndexRestore
New in version 2.2. Prevents mongorestore from restoring and building indexes as specified in the corresponding mongodump output.

--oplogLimit <timestamp>
New in version 2.2. Prevents mongorestore from applying *oplog* entries newer than the <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 228) in conjunction with the *--oplogReplay* (page 227) option.

<path>
The final argument of the mongorestore command is a directory path. This argument specifies the location of the database dump from which to restore.

Usage

See the "*backup guide section on database dumps*" for a larger overview of mongorestore usage. Also see the "*mongodump* (page 223)" document for an overview of the mongodump, which provides the related inverse functionality.

Consider the following example:

```
mongorestore --collection people --db accounts dump/accounts/
```

Here, mongorestore 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 restores data to the instance running on the localhost interface on port 27017.

In the next example, mongorestore restores a backup of the database instance located in dump to a database instance stored in the http://docs.mongodb.org/manual/srv/mongodb on the local machine. This requires that there are no active mongod instances attached to http://docs.mongodb.org/manual/srv/mongodb data directory.

```
mongorestore --dbpath /srv/mongodb
```

In the final example, mongorestore restores a database dump located at http://docs.mongodb.org/manual/opt/backup/mongodump-2011-10-24, from a database running on port 37017 on the host mongodb1.example.net. mongorestore` authenticates to the this 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
```

17.3.3 bsondump

Synopsis

The bsondump converts *BSON* files into human-readable formats, including *JSON*. For example, bsondump is useful for reading the output files generated by mongodump.

Options

bsondump

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

--version
Returns the version of the bsondump utility.

--objcheck
Validates each *BSON* object before outputting it in *JSON* format. Use this option to filter corrupt objects from the output. This option has some performance impact.

--filter '<JSON>'
Limits the documents that bsondump exports to only those documents that match the *JSON document* specified as '<JSON>'. Be sure to include the document in single quotes to avoid interaction with your system's shell environment.

--type <=json|=debug>
Changes the operation of bsondump from outputting "*JSON*" (the default) to a debugging format.

<bsonfilename>
The final argument to bsondump is a document containing *BSON*. This data is typically generated by mongodump or by MongoDB in a *rollback* operation.

Usage

By default, bsondump 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
```

17.3.4 mongooplog

New in version 2.1.1.

Synopsis

mongooplog 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 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* (page 231) host running during the migration, consider using mongodump and mongorestore or another backup operation, which may be better suited to your operation.

Note: If the mongod instance specified by the *--from* (page 231) argument is running with authentication, then mongooplog will not be able to copy oplog entries.

See Also:

mongodump, mongorestore, "http://docs.mongodb.org/manual/administration/backups," "*Oplog Internals Overview*, and "*Replica Set Oplog Sizing*".

Options

mongooplog

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

--version
Returns the version of the mongooplog utility.

--host <hostname><:port>**, -h**
Specifies a resolvable hostname for the mongod instance to which mongooplog will apply *oplog* operations retrieved from the serve specified by the *--from* (page 231) option.

mongooplog assumes that all target mongod instances are accessible by way of port 27017. You may, optionally, declare an alternate port number as part of the hostname argument.

You can always connect directly to a single mongod instance by specifying the host and port number directly.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

--port
Specifies the port number of the mongod instance where mongooplog will apply *oplog* entries. Only specify this option if the MongoDB instance that you wish to connect to is not running on the standard port. (i.e. 27017) You may also specify a port number using the *--host* (page 230) command.

--ipv6
Enables IPv6 support that allows `mongooplog` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongooplog`, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *--password* (page 231) option to supply a password.

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* (page 231) option to supply a username.

If you specify a *--username* (page 231) without the *--password* (page 231) option, `mongooplog` will prompt for a password interactively.

--dbpath <path>
Specifies a directory, containing MongoDB data files, to which `mongooplog` will apply operations from the *oplog* of the database specified with the *--from* (page 231) option. When used, the *--dbpath* (page 231) option enables `mongo` to attach directly to local data files and write data without a running `mongod` instance. To run with *--dbpath* (page 231), `mongooplog` needs to restrict access to the data directory: as a result, no `mongod` can be access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 231) in conjunction with the corresponding option to `mongod`. This option allows `mongooplog` to write to data files organized with each database located in a distinct directory. This option is only relevant when specifying the *--dbpath* (page 231) option.

--journal
Allows `mongooplog` operations to use the durability *journal* to ensure that the data files will remain in a consistent state during the writing process. This option is only relevant when specifying the *--dbpath* (page 231) option.

--fields [field1[,field2]], **-f** [field1[,field2]]
Specify a field or number fields to constrain which data `mongooplog` will migrate. All other fields will be *excluded* from the migration. Comma separate a list of fields to limit the applied fields.

--fieldFile <file>
As an alternative to "*--fields* (page 231)" the *--fieldFile* (page 231) option allows you to specify a file (e.g. `<file>`) that holds a list of field names to *include* in the migration. All other fields will be *excluded* from the migration. Place one field per line.

--seconds <number>, **-s** <number>
Specify a number of seconds of operations for `mongooplog` to pull from the *remote host* (page 231). Unless specified the default value is `86400` seconds, or 24 hours.

--from <host[:port]>
Specify the host for `mongooplog` to retrieve *oplog* operations from. `mongooplog` *requires* this option.

Unless you specify the *--host* (page 230) option, `mongooplog` will apply the operations collected with this option to the oplog of the `mongod` instance running on the localhost interface connected to port `27017`.

--oplogns <namespace>
Specify a namespace in the *--from* (page 231) 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, use this option to copy oplog entries stored in another location.

Namespaces take the form of `[database].[collection]`.

Usage

Consider the following prototype mongooplog command:

```
mongooplog  --from mongodb0.example.net --host mongodb1.example.net
```

Here, entries from the *oplog* of the mongod running on port 27017. This only pull entries from the last 24 hours.

In the next command, the parameters limit this operation to only apply operations to the database people in the collection usage on the target host (i.e. mongodb1.example.net):

```
mongooplog  --from mongodb0.example.net --host mongodb1.example.net --database people --collection us
```

This operation only applies oplog entries from the last 24 hours. Use the *--seconds* (page 231) 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 captures 2 full days of operations. To migrate 12 hours of *oplog* entries, use the following form:

```
mongooplog  --from mongodb0.example.net --seconds 43200
```

For the previous two examples, mongooplog migrates entries to the mongod process running on the localhost interface connected to the 27017 port. mongooplog can also operate directly on MongoDB's data files if no mongod is running on the *target* host. Consider the following example:

```
mongooplog  --from mongodb0.example.net --dbpath /srv/mongodb --journal
```

Here, mongooplog imports *oplog* operations from the mongod host connected to port 27017. This migrates operations to the MongoDB data files stored in the http://docs.mongodb.org/manual/srv/mongodb directory. Additionally mongooplog will use the durability *journal* to ensure that the data files remain in a consistent state.

17.4 Data Import and Export Tools

mongoimport provides a method for taking data in *JSON*, *CSV*, or *TSV* and importing it into a mongod instance. mongoexport provides a method to export data from a mongod instance into JSON, CSV, or TSV.

Note: The conversion between BSON and other formats lacks full type fidelity. Therefore you cannot use mongoimport and mongoexport for round-trip import and export operations.

17.4.1 mongoimport

Synopsis

The mongoimport tool provides a route to import content from a JSON, CSV, or TSV export created by mongoexport, or potentially, another third-party export tool. See the "http://docs.mongodb.org/manual/administration/import-export" document for a more in depth usage overview, and the "*mongoexport* (page 235)" document for more information regarding mongoexport, which provides the inverse "importing" capability.

Note: Do not use `mongoimport` and `mongoexport` for full instance, production backups because they will not reliably capture data type information. Use `mongodump` and `mongorestore` as described in "`http://docs.mongodb.org/manual/administration/backups`" for this kind of functionality.

Options

mongoimport

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Returns the version of the `mongoimport` program.

--host `<hostname><:port>`**, -h**
Specifies a resolvable hostname for the `mongod` to which you want to restore the database. By default `mongoimport` will attempt to connect to a MongoDB process ruining on the localhost port numbered `27017`.

Optionally, specify a port number to connect a MongoDB instance running on a port other than `27017`.

To connect to a replica set, use the *`--host`* (page 233) argument with a setname, followed by a slash and a comma-separated list of host and port names. `mongoimport` will, given the seed of at least one connected set member, connect to primary node of that set. This option would resemble:

```
--host repl0/mongo0.example.net,mongo0.example.net,27018,mongo1.example.net,mongo2.example.net
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

--port `<port>`
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *`mongoimport --host`* (page 233) command.

--ipv6
Enables IPv6 support that allows `mongoimport` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongoimport`, disable IPv6 support by default.

--username `<username>`**, -u** `<username>`
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *`mongoimport --password`* (page 233) option to supply a password.

--password `<password>`
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *`mongoimport --username`* (page 233) option to supply a username.

If you specify a *`--username`* (page 233) without the *`--password`* (page 233) option, `mongoimport` will prompt for a password interactively.

--dbpath `<path>`
Specifies the directory of the MongoDB data files. If used, the *`--dbpath`* (page 233) option enables `mongoimport` to attach directly to local data files and insert the data without the `mongod`. To run with `--dbpath`, `mongoimport` needs to lock access to the data directory: as a result, no `mongod` can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 233) in conjunction with the corresponding option to `mongod`, which allows `mongoimport` to import data into MongoDB instances that have every database's files saved in discrete directories on the disk. This option is only relevant when specifying the *--dbpath* (page 233) option.

--journal
Allows `mongoexport` write to the durability *journal* to ensure that the data files will remain in a consistent state during the write process. This option is only relevant when specifying the *--dbpath* (page 233) option.

--db <db>**, -d** <db>
Use the *--db* (page 234) option to specify a database for `mongoimport` to restore data. If you do not specify a <db>, `mongoimport` 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, or to restore only some data in the specified backup.

--collection <collection>**, -c** <collection>
Use the *--collection* (page 234) option to specify a collection for `mongorestore` to restore. If you do not specify a `<collection>`, `mongoimport` imports all collections created. Existing data may be overwritten. Use this option to restore data into a MongoDB instance that already has data, or to restore only some data in the specified imported data set.

--fields <field1[,filed2]>**, -f** <field1[,filed2]>
Specify a field or number fields to *import* from the specified file. All other fields present in the export will be *excluded* during importation. Comma separate a list of fields to limit the fields imported.

--fieldFile <filename>
As an alternative to "*mongoimport --fields* (page 234)" the *--fieldFile* (page 234) option allows you to specify a file (e.g. `<file>`') to hold a list of field names to specify a list of fields to *include* in the export. All other fields will be *excluded* from the export. Place one field per line.

--ignoreBlanks
In *csv* and *tsv* exports, ignore empty fields. If not specified, `mongoimport` creates fields without values in imported documents.

--type <json|csv|tsv>
Declare the type of export format to import. The default format is *JSON*, but it's possible to import *csv* and *tsv* files.

--file <filename>
Specify the location of a file containing the data to import. `mongoimport` will read data from standard input (e.g. "stdin.") if you do not specify a file.

--drop
Modifies the importation procedure so that the target instance drops every collection before restoring the collection from the dumped backup.

--headerline
If using "*--type csv* (page 234)" or "*--type tsv* (page 234)," use the first line as field names. Otherwise, `mongoimport` will import the first line as a distinct document.

--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 234) `mongoimport` will upsert on the basis of the `_id` field.

--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.

To ensure adequate performance, indexes should exist for this field or fields.

--stopOnError

New in version 2.2. Forces mongoimport to halt the import operation at the first error rather than continuing the operation despite errors.

--jsonArray

Changed in version 2.2: The limit on document size increased from 4MB to 16MB. Accept import of data expressed with multiple MongoDB document within a single *JSON* array.

Use in conjunction with *mongoexport --jsonArray* (page 237) to import data written as a single *JSON* array. Limited to imports of 16 MB or smaller.

Usage

In this example, mongoimport imports the *csv* formatted data in the http://docs.mongodb.org/manual/opt/backups/contacts.csv into the collection contacts in the users database on the MongoDB instance running on the localhost port numbered 27017.

```
mongoimport --db users --collection contacts --type csv --file /opt/backups/contacts.csv
```

In the following example, mongoimport imports the data in the *JSON* formatted file contacts.json into the collection contacts on the MongoDB instance running on the localhost port number 27017. Journaling is explicitly enabled.

```
mongoimport --collection contacts --file contacts.json --journal
```

In the next example, mongoimport takes data passed to it on standard input (i.e. with a | pipe.) and imports it into the collection contacts in the sales database is the MongoDB datafiles located at http://docs.mongodb.org/manual/srv/mongodb/. if the import process encounters an error, the mongoimport will halt because of the *--stopOnError* (page 235) option.

```
mongoimport --db sales --collection contacts --stopOnError --dbpath /srv/mongodb/
```

In the final example, mongoimport imports data from the file http://docs.mongodb.org/manual/opt/backups/mdb1-e into the collection contacts within the database marketing on a remote MongoDB database. This mongoimport accesses the mongod instance running on the host mongodb1.example.net over port 37017, which requires the username user and the password pass.

```
mongoimport --host mongodb1.example.net --port 37017 --username user --password pass --collection co
```

17.4.2 mongoexport

Synopsis

mongoexport is a utility that produces a JSON or CSV export of data stored in a MongoDB instance. See the "http://docs.mongodb.org/manual/administration/import-export" document for a more in depth usage overview, and the "*mongoimport* (page 232)" document for more information regarding the mongoimport utility, which provides the inverse "importing" capability.

Note: Do not use mongoimport and mongoexport for full-scale backups because they may not reliably capture data type information. Use mongodump and mongorestore as described in "http://docs.mongodb.org/manual/administration/backups" for this kind of functionality.

Options

mongoexport

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

--version
Returns the version of the mongoexport utility.

--host <hostname><:port>
Specifies a resolvable hostname for the mongod from which you want to export data. By default mongoexport attempts to connect to a MongoDB process ruining on the localhost port number 27017.

Optionally, specify a port number to connect a MongoDB instance running on a port other than 27017.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

--port <port>
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. 27017) You may also specify a port number using the *mongoexport --host* (page 236) command.

--ipv6
Enables IPv6 support that allows mongoexport to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including mongoexport, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *mongoexport --password* (page 236) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* (page 236) option to supply a username.

If you specify a *--username* (page 236) without the *--password* (page 236) option, mongoexport will prompt for a password interactively.

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the --dbpath option enables mongoexport to attach directly to local data files and insert the data without the mongod. To run with --dbpath, mongoexport needs to lock access to the data directory: as a result, no mongod can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 236) in conjunction with the corresponding option to mongod, which allows mongoexport to export data into MongoDB instances that have every database's files saved in discrete directories on the disk. This option is only relevant when specifying the *--dbpath* (page 236) option.

--journal
Allows mongoexport operations to access the durability *journal* to ensure that the export is in a consistent state. This option is only relevant when specifying the *--dbpath* (page 236) option.

--db <db>, **-d** <db>
Use the *--db* (page 237) option to specify the name of the database that contains the collection you want to export.

--collection <collection>, **-c** <collection>
Use the *--collection* (page 237) option to specify the collection that you want mongoexport to export.

--fields <field1[,field2]>, **-f** <field1[,field2]>
Specify a field or number fields to *include* in the export. All other fields will be *excluded* from the export. Comma separate a list of fields to limit the fields exported.

--fieldFile <file>
As an alternative to "*--fields* (page 237)" the *--fieldFile* (page 237) option allows you to specify a file (e.g. <file>`) to hold a list of field names to specify a list of fields to *include* in the export. All other fields will be *excluded* from the export. Place one field per line.

--query <JSON>
Provides a *JSON document* as a query that optionally limits the documents returned in the export.

--csv
Changes the export format to a comma separated values (CSV) format. By default mongoexport writes data using one *JSON* document for every MongoDB document.

--jsonArray
Modifies the output of mongoexport to write the entire contents of the export as a single *JSON* array. By default mongoexport writes data using one JSON document for every MongoDB document.

--slaveOk, -k
Allows mongoexport to read data from secondary or slave nodes when using mongoexport with a replica set. This option is only available if connected to a mongod or mongos (page 256) and is not available when used with the "*mongoexport --dbpath* (page 236)" option.

This is the default behavior.

--out <file>, **-o** <file>
Specify a file to write the export to. If you do not specify a file name, the mongoexport writes data to standard output (e.g. stdout).

Usage

In the following example, mongoexport exports the collection contacts from the users database from the mongod instance running on the localhost port number 27017. This command writes the export data in *CSV* format into a file located at http://docs.mongodb.org/manual/opt/backups/contacts.csv.

```
mongoexport --db users --collection contacts --csv --out /opt/backups/contacts.csv
```

The next example creates an export of the collection contacts from the MongoDB instance running on the localhost port number 27017, with journaling explicitly enabled. This writes the export to the contacts.json file in *JSON* format.

```
mongoexport --db sales --collection contacts --out contacts.json --journal
```

The following example exports the collection `contacts` from the `sales` database located in the MongoDB data files located at `http://docs.mongodb.org/manual/srv/mongodb/`. This operation writes the export to standard output in *JSON* format.

```
mongoexport --db sales --collection contacts --dbpath /srv/mongodb/
```

> **Warning:** The above example will only succeed if there is no `mongod` connected to the data files located in the `http://docs.mongodb.org/manual/srv/mongodb/` directory.

The final example exports the collection `contacts` from the database `marketing` . This data resides on the MongoDB instance located on the host `mongodb1.example.net` running on port `37017`, which requires the username `user` and the password `pass`.

```
mongoexport --host mongodb1.example.net --port 37017 --username user --password pass --collection cor
```

17.5 Diagnostic Tools

`mongostat`, `mongotop`, and `mongosniff` provide diagnostic information related to the current operation of a `mongod` instance.

Note: Because `mongosniff` depends on *libpcap*, most distributions of MongoDB do *not* include `mongosniff`.

17.5.1 `mongostat`

Synopsis

The `mongostat` utility provides a quick overview of the status of a currently running `mongod` or `mongos` (page 256) instance. `mongostat` is functionally similar to the UNIX/Linux file system utility `vmstat`, but provides data regarding `mongod` and `mongos` (page 256) instances.

See Also:

For more information about monitoring MongoDB, see `http://docs.mongodb.org/manual/administration/monitori`

For more background on various other MongoDB status outputs see:

- *Server Status Reference* (page 167)
- *Replica Set Status Reference* (page 193)
- *Database Statistics Reference* (page 183)
- *Collection Statistics Reference* (page 185)

For an additional utility that provides MongoDB metrics see "*mongotop* (page 242)."

`mongostat` connects to the `mongod` instance running on the local host interface on TCP port `27017`; however, `mongostat` can connect to any accessible remote `mongod` instance.

Options

mongostat

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Returns the version of the `mongostat` utility.

--host `<hostname><:port>`
Specifies a resolvable hostname for the `mongod` from which you want to export data. By default `mongostat` attempts to connect to a MongoDB instance running on the localhost port number `27017`.

Optionally, specify a port number to connect a MongoDB instance running on a port other than `27017`.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

--port `<port>`
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *`mongostat --host`* (page 239) command.

--ipv6
Enables IPv6 support that allows `mongostat` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongostat`, disable IPv6 support by default.

--username `<username>`**, -u** `<username>`
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *`mongostat --password`* (page 239) option to supply a password.

--password `<password>`
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *`mongostat --username`* (page 239) option to supply a username.

If you specify a *`--username`* (page 239) without the *`--password`* (page 239) option, `mongostat` will prompt for a password interactively.

--noheaders
Disables the output of column or field names.

--rowcount `<number>`**, -n** `<number>`
Controls the number of rows to output. Use in conjunction with `mongostat` to control the duration of a `mongostat` operation.

Unless *`--rowcount`* (page 239) is specified, `mongostat` will return an infinite number of rows (e.g. value of `0`.)

--http
Configures `mongostat` to collect data using the HTTP interface rather than a raw database connection.

--discover
With this option `mongostat` 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 239) all non-*hidden members* of the replica set. When connected to a `mongos` (page 256), `mongostat` will return data from all *shards* in the

cluster. If a replica set provides a shard in the sharded cluster, `mongostat` will report on non-hidden members of that replica set.

The *`mongostat --host`* (page 239) option is not required but potentially useful in this case.

`--all`
Configures `mongostat` to return all optional *fields* (page 240).

`<sleeptime>`
The final argument is the length of time, in seconds, that `mongostat` waits in between calls. By default `mongostat` returns one call every second.

`mongostat` returns values that reflect the operations over a 1 second period. For values of `<sleeptime>` greater than 1, `mongostat` averages data to reflect average operations per second.

Fields

`mongostat` returns values that reflect the operations over a 1 second period. When **mongostat <sleeptime>** has a value greater than 1, `mongostat` averages the statistics to reflect average operations per second.

`mongostat` 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` presents two values separated by a pipe character (e.g. `|`), in the form of `local|replicated` commands.

`flushes`
The number of *fsync* operations per second.

`mapped`
The total amount of data mapped in megabytes. This is the total data size at the time of the last `mongostat` call.

`size`
The amount of (virtual) memory in megabytes used by the process at the time of the last `mongostat` call.

`res`
The amount of (resident) memory in megabytes used by the process at the time of the last `mongostat` call.

`faults`
Changed in version 2.1. The number of page faults per second.

Before version 2.1 this value was only provided for MongoDB instances running on Linux hosts.

locked
The percent of time in a global write lock. Changed in version 2.2: The `locked db` field replaces the `locked %` field to more appropriate data regarding the database specific locks in version 2.2.

locked db
New in version 2.2. The percent of time in the per-database context-specific lock. `mongostat` will report the database that has spent the most time since the last `mongostat` call with a write lock.

This value represents the amount of time the database had a database specific lock *and* the time that the `mongod` spent in the global lock. Because of this, and the sampling method, you may see some values greater than 100%.

idx miss
The percent of index access attempts that required a page fault to load a btree node. This is a sampled value.

qr
The length of the queue of clients waiting to read data from the MongoDB instance.

qw
The length of the queue of clients waiting to write data from the MongoDB instance.

ar
The number of active clients performing read operations.

aw
The number of active clients performing write operations.

netIn
The amount of network traffic, in *bytes*, received by the MongoDB instance.

This includes traffic from `mongostat` itself.

netOut
The amount of network traffic, in *bytes*, sent by the MongoDB instance.

This includes traffic from `mongostat` itself.

conn
The total number of open connections.

set
The name, if applicable, of the replica set.

repl
The replication status of the node.

Value	Replication Type
M	*master*
SEC	*secondary*
REC	recovering
UNK	unknown
SLV	*slave*

Usage

In the first example, `mongostat` will return data every second for 20 seconds. `mongostat` collects data from the `mongod` 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 returns data every 5 minutes (or 300 seconds) for as long as the program runs. mongostat collects data from the mongod 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 returns data every 5 minutes for an hour (12 times.) mongostat collects data from the mongod 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* (page 239) will help provide a more complete snapshot of the state of an entire group of machines. If a mongos (page 256) 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
```

17.5.2 mongotop

Synopsis

mongotop provides a method to track the amount of time a MongoDB instance spends reading and writing data. mongotop provides statistics on a per-collection level. By default, mongotop returns values every second.

See Also:

For more information about monitoring MongoDB, see http://docs.mongodb.org/manual/administration/monitori

For additional background on various other MongoDB status outputs see:

- *Server Status Reference* (page 167)
- *Replica Set Status Reference* (page 193)
- *Database Statistics Reference* (page 183)
- *Collection Statistics Reference* (page 185)

For an additional utility that provides MongoDB metrics see "*mongostat* (page 238)."

Options

mongotop

--help

Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Print the version of the `mongotop` utility and exit.

--host <hostname><:port>
Specifies a resolvable hostname for the `mongod` from which you want to export data. By default `mongotop` 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`.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

--port <port>
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *mongotop --host* (page 243) command.

--ipv6
Enables IPv6 support that allows `mongotop` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongotop`, disable IPv6 support by default.

--username <username>**, -u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *mongotop* (page 243) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* (page 243) option to supply a username.

If you specify a *--username* (page 243) without the *--password* (page 243) option, `mongotop` will prompt for a password interactively.

--locks
New in version 2.2. Toggles the mode of `mongotop` to report on use of per-database *locks* (page 168). These data are useful for measuring concurrent operations and lock percentage.

<sleeptime>
The final argument is the length of time, in seconds, that `mongotop` waits in between calls. By default `mongotop` returns data every second.

Fields

`mongotop` returns time values specified in milliseconds (ms.)

`mongotop` only reports active namespaces or databases, depending on the *--locks* (page 243) 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` shell to generate activity to affect the output of `mongotop`.

ns
Contains the database namespace, which combines the database name and collection. Changed in version 2.2: If you use the *--locks* (page 243), the `ns` (page 243) field does not appear in the `mongotop` output.

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` with the *--locks* (page 243) option.

total
Provides the total amount of time that this `mongod` spent operating on this namespace.

read
Provides the amount of time that this `mongod` spent performing read operations on this namespace.

write
Provides the amount of time that this `mongod` spent performing write operations on this namespace.

<timestamp>
Provides a time stamp for the returned data.

Use

By default `mongotop` connects to the MongoDB instance running on the localhost port `27017`. However, `mongotop` can optionally connect to remote `mongod` instances. See the *mongotop options* (page 242) for more information.

To force `mongotop` to return less frequently specify a number, in seconds at the end of the command. In this example, `mongotop` will return every 15 seconds.

```
mongotop 15
```

This command produces the following output:

```
connected to: 127.0.0.1

                    ns       total        read       write           2012-08-13T15:45:40
test.system.namespaces         0ms         0ms         0ms
  local.system.replset         0ms         0ms         0ms
  local.system.indexes         0ms         0ms         0ms
  admin.system.indexes         0ms         0ms         0ms
                admin.         0ms         0ms         0ms

                    ns       total        read       write           2012-08-13T15:45:55
test.system.namespaces         0ms         0ms         0ms
  local.system.replset         0ms         0ms         0ms
  local.system.indexes         0ms         0ms         0ms
  admin.system.indexes         0ms         0ms         0ms
                admin.         0ms         0ms         0ms
```

To return a `mongotop` report every 5 minutes, use the following command:

```
mongotop 300
```

To report the use of per-database locks, use *mongotop --locks* (page 243), 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
```

17.5.3 `mongosniff`

Synopsis

`mongosniff` provides a low-level operation tracing/sniffing view into database activity in real time. Think of `mongosniff` as a MongoDB-specific analogue of `tcpdump` for TCP/IP network traffic. Typically, `mongosniff` is most frequently used in driver development.

Note: `mongosniff` requires `libpcap` and is only available for Unix-like systems. Furthermore, the version distributed with the MongoDB binaries is dynamically linked against aversion 0.9 of `libpcap`. If your system has a different version of `libpcap`, you will need to compile `mongosniff` yourself or create a symbolic link pointing to `libpcap.so.0.9` to your local version of `libpcap`. Use an operation that resembles the following:

```
ln -s /usr/lib/libpcap.so.1.1.1 /usr/lib/libpcap.so.0.9
```

Change the path's and name of the shared library as needed.

As an alliterative to `mongosniff`, Wireshark, a popular network sniffing tool is capable of inspecting and parsing the MongoDB wire protocol.

Options

`mongosniff`

`--help`
Returns a basic help and usage text.

`--forward` `<host>:<port>`
Declares a host to forward all parsed requests that the `mongosniff` intercepts to another `mongod` 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, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

`--source` `<NET [interface]>, <FILE [filename]>, <DIAGLOG [filename]>`
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`* (page 209) option.

`--objcheck`
Modifies the behavior to *only* display invalid BSON objects and nothing else. Use this option for troubleshooting driver development. This option has some performance impact on the performance of `mongosniff`.

`<port>`
Specifies alternate ports to sniff for traffic. By default, `mongosniff` watches for MongoDB traffic on port 27017. Append multiple port numbers to the end of `mongosniff` to monitor traffic on multiple ports.

Usage

Use the following command to connect to a mongod or mongos (page 256) 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 or mongos (page 256) running on the localhost interface and port 27018, for driver development and troubleshooting:

```
mongosniff --objcheck --source NET lo 27018
```

Build mongosniff

To build mongosniff yourself, Linux users can use the following procedure:

1. Obtain prerequisites using your operating systems package management software. Dependencies include:
   - libpcap - to capture network packets.
   - git - to download the MongoDB source code.
   - scons and a C++ compiler - to build mongosniff.
2. Download a copy of the MongoDB source code using git:

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

3. Issue the following sequence of commands to change to the mongo/ directory and build mongosniff:

   ```
   cd mongo
   scons mongosniff
   ```

17.6 GridFS

mongofiles provides a command-line interact to a MongoDB *GridFS* storage system.

17.6.1 mongofiles

Synopsis

The mongofiles 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 commands take arguments in three groups:

1. *Options* (page 247). You may use one or more of these options to control the behavior of mongofiles.
2. *Commands* (page 247). Use one of these commands to determine the action of mongofiles.
3. A file name representing either the name of a file on your system's file system, a GridFS object.

`mongofiles`, like `mongodump`, `mongoexport`, `mongoimport`, and `mongorestore`, can access data stored in a MongoDB data directory without requiring a running `mongod` instance, if no other `mongod` is running.

Note: For *replica sets*, `mongofiles` can only read from the set's '*primary*.

Commands

`mongofiles`

`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` assumes that this reflects the name the file has on the local file system. If the local filename is different use the *`mongofiles --local`* (page 248) option.

`get <filename>`
Copy the specified file from GridFS storage to the local file system.

Here, `<filename>` refers to the name the object will have in GridFS, and `mongofiles` assumes that this reflects the name the file has on the local file system. If the local filename is different use the *`mongofiles --local`* (page 248) option.

`delete <filename>`
Delete the specified file from GridFS storage.

Options

`--help`
Returns a basic help and usage text.

`--verbose, -v`
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

`--version`
Returns the version of the `mongofiles` utility.

`--host` `<hostname><:port>`
Specifies a resolvable hostname for the `mongod` that holds your GridFS system. By default `mongofiles` attempts to connect to a MongoDB process ruining on the localhost port number `27017`.

Optionally, specify a port number to connect a MongoDB instance running on a port other than 27017.

`--port` `<port>`
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *`mongofiles --host`* (page 247) command.

`--ipv6`
Enables IPv6 support that allows `mongofiles` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongofiles`, disable IPv6 support by default.

--username <username>**, -u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *mongofiles --password* (page 248) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongofiles --username* (page 247) option to supply a username.

If you specify a *--username* (page 247) without the *--password* (page 248) option, mongofiles will prompt for a password interactively.

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 248) option enables mongofiles to attach directly to local data files interact with the GridFS data without the mongod. To run with *--dbpath* (page 248), mongofiles needs to lock access to the data directory: as a result, no mongod can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 248) in conjunction with the corresponding option to mongod, which allows mongofiles when running with the *--dbpath* (page 248) option and MongoDB uses an on-disk format where every database has a distinct directory. This option is only relevant when specifying the *--dbpath* (page 248) option.

--journal
Allows mongofiles operations to use the durability *journal* when running with *--dbpath* (page 248) to ensure that the database maintains a recoverable state. This forces mongofiles to record all data on disk regularly.

--db <db>**, -d** <db>
Use the *--db* (page 248) option to specify the MongoDB database that stores or will store the GridFS files.

--collection <collection>**, -c** <collection>
This option has no use in this context and a future release may remove it. See SERVER-4931 for more information.

--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 assumes that this reflects the file's name on the local file system. This setting overrides this default.

--type <MIME>, t <MIME>
Provides the ability to specify a *MIME* type to describe the file inserted into GridFS storage. mongofiles omits this option in the default operation.

Use only with **mongofiles put** operations.

--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.

Use

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` instance will connect to the `mongod` 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` 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
```

Part VII

Internal Metadata

CHAPTER EIGHTEEN

CONFIG DATABASE CONTENTS

The `config` database supports *sharded cluster* operation. See the `http://docs.mongodb.org/manual/sharding` section of this manual for full documentation of sharded clusters.

To access a the `config` database, connect to a `mongos` (page 256) instance in a sharded cluster, and issue the following command:

```
use config
```

You can return a list of the databases, by issuing the following command:

```
show collections
```

18.1 Collections

changelog

The `changelog` (page 253) 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 `config.changelog <changelog>`` 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 253) collection contains the following fields:

`changelog.`**`_id`**
The value of `changelog._id` is: `<hostname>-<timestamp>-<increment>`.

`changelog.`**`server`**
The hostname of the server that holds this data.

`changelog.`**`clientAddr`**
A string that holds the address of the client, a `mongos` (page 256) instance that initiates this change.

`changelog.`**`time`**
A *ISODate* timestamp that reflects when the change occurred.

`changelog.`**`what`**
Reflects the type of change recorded. Possible values are:

- `dropCollection`
- `dropCollection.start`
- `dropDatabase`
- `dropDatabase.start`
- `moveChunk.start`
- `moveChunk.commit`
- `split`
- `multi-split`

`changelog.`**`ns`**
Namespace where the change occured.

`changelog.details`

A *document* that contains additional details regarding the change. The structure of the `details` (page 254) document depends on the type of change.

chunks

The `chunks` (page 255) 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.

collections

The `collections` (page 183) 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 183) 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")
}
```

databases

The `databases` (page 255) collection stores a document for each database in the cluster, and tracks if the database has sharding enabled. `databases` (page 255) 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" }
```

lockpings

The `lockpings` (page 255) collection keeps track of the active components in the sharded cluster. Given a cluster with a `mongos` (page 256) running on `example.com:30000`, the document in the `lockpings` (page 255) collection would resemble:

```
{ "_id" : "example.com:30000:1350047994:16807", "ping" : ISODate("2012-10-12T18:32:54.892Z") }
```

locks

The locks (page 255) collection stores a distributed lock. This ensures that only one mongos (page 256) instance can perform administrative tasks on the cluster at once. The mongos (page 256) 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 256) 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.

mongos

The mongos (page 256) collection stores a document for each mongos (page 256) instance affiliated with the cluster. mongos (page 256) instances send pings to all members of the cluster every 30 seconds so the cluster can verify that the mongos (page 256) is active. The ping field shows the time of the last ping. The cluster maintains this collection for reporting purposes.

The following document shows the status of the mongos (page 256) running on example.com:30000.

```
{ "_id" : "example.com:30000", "ping" : ISODate("2012-10-12T17:08:13.538Z"), "up" : 13699, "wait
```

settings

The settings (page 256) collection holds the following sharding configuration settings:

- Chunk size. To change chunk size, see *sharding-balancing-modify-chunk-size*.
- Balancer status. To change status, see *sharding-balancing-disable-temporally*.

The following is an example settings collection:

```
{ "_id" : "chunksize", "value" : 64 }
{ "_id" : "balancer", "stopped" : false }
```

shards

The shards (page 256) 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" }
```

version

The version (page 256) collection holds the current metadata version number. This collection contains only one document:

To access the version (page 256) collection you must use the db.getCollection() method. For example, to display the collection's document:

```
mongos> db.getCollection("version").find()
{ "_id" : 1, "version" : 3 }
```

Note: Like all databases in MongoDB, the `config` database contains a `system.indexes` collection contains metadata for all indexes in the database for information on indexes, see `http://docs.mongodb.org/manual/indexes`.

CHAPTER

NINETEEN

THE `LOCAL` DATABASE

19.1 Overview

Every `mongod` instance has its own `local` database, which stores data used in the replication process, and other instance-specific data. The `local` database is invisible to replication: collections in the `local` database are not replicated.

When running with authentication (i.e. `auth`), authenticating against the `local` database is equivalent to authenticating against the `admin` database. This authentication gives access to all databases.

In replication, the `local` database store stores internal replication data for each member of a *replica set*. The `local` database contains the following collections used for replication:

19.2 Collections on Replica Set Members

`local.system.`**`replset`**

`local.system.replset` (page 259) holds the replica set's configuration object as its single document. To view the object's configuration information, issue `rs.conf()` from the `mongo` shell. You can also query this collection directly.

`local.oplog.`**`rs`**

`local.oplog.rs` (page 259) is the capped collection that holds the *oplog*. You set its size at creation using the `oplogSize` setting. To resize the oplog after replica set initiation, use the `http://docs.mongodb.org/manual/tutorial/change-oplog-size` procedure. For additional information, see the *replica-set-internals-oplog* topic in this document and the *replica-set-oplog-sizing* topic in the `http://docs.mongodb.org/manual/core/replication` document.

`local.replset.`**`minvalid`**

This contains an object used internally by replica sets to track sync status.

`local.`**`slaves`**

This contains information about each member of the set.

19.3 Collections used in Master/Slave Replication

In *master*/*slave* replication, the `local` database contains the following collections:

- On the master:

`local.oplog.$main`
: This is the oplog for the master-slave configuration.

`local.slaves`
: This contains information about each slave.

- On each slave:

`local.sources`
: This contains information about the slave's master server.

CHAPTER

TWENTY

SYSTEM COLLECTIONS

20.1 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 259), specifically for replication purposes.

20.2 Collections

System collections include these collections stored directly in the database:

`<database>.system.`**`namespaces`**

The `<database>.system.namespaces` (page 261) collection contains information about all of the database's collections. Additional namespace metadata exists in the `database.ns` files and is opaque to database users.

`<database>.system.`**`indexes`**

The `<database>.system.indexes` (page 261) collection lists all the indexes in the database. Add and remove data from this collection via the `ensureIndex()` and `dropIndex()`

`<database>.system.`**`profile`**

The `<database>.system.profile` (page 261) collection stores database profiling information. For information on profiling, see *database-profiling*.

`<database>.system.`**`users`**

The `<database>.system.users` (page 261) collection stores credentials for users who have access to the database. For more information on this collection, see *security-authentication*.

`<database>.system.`**`js`**

The `<database>.system.js` (page 261) collection holds special JavaScript code for use in `server side JavaScript`. See *storing-server-side-javascript* for more information.

Part VIII

General System Reference

CHAPTER

TWENTYONE

MONGODB LIMITS AND THRESHOLDS

21.1 Synopsis

This document provides a collection of hard and soft limitations of the MongoDB system.

21.2 Limits

21.2.1 BSON Documents

`BSON Document Size`

The maximum BSON document size is 16 megabytes.

The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth. To store documents larger than the maximum size, MongoDB provides the GridFS API. See `mongofiles` 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*.

21.2.2 Namespaces

`Namespace Length`

Each namespace, including database and collection name, must be shorter than 123 bytes.

`Number of Namespaces`

The limitation on the number of namespaces is the size of the namespace file divided by 628.

A 16 megabyte namespace file can support approximately 24,000 namespaces. Each index also counts as a namespace.

`Size of Namespace File`

Namespace files can be no larger than 2047 megabytes.

By default namespace files are 16 megabytes. You can configure the size using the `nssize`.

21.2.3 Indexes

`Index Size`
Indexed items can be *no larger* than 1024 bytes. This value is the indexed content (i.e. the field value, or compound field value.)

`Number of Indexes per Collection`
A single collection can have *no more* than 64 indexes.

`Index Name Length`
The names of indexes, including their namespace (i.e database and collection name) cannot be longer than 128 characters. The default index name is the concatenation of the field names and index directions.

You can explicitly specific a name to the `ensureIndex()` helper if the default index name is too long.

`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 Also:

`http://docs.mongodb.org/manual/tutorial/enforce-unique-keys-for-sharded-collections` for an alternate approach.

`Number of Indexed Fields in a Compound Index`
There can be no more than 31 fields in a compound index.

21.2.4 Replica Sets

`Number of Members of a Replica Set`
Replica sets can have no more than 12 members.

`Number of Voting Members of a Replica Set`
Only 7 members of a replica set can have votes at any given time. See can vote *replica-set-non-voting-members* for more information

21.2.5 Operations

`Sorted Documents`
MongoDB will only return sorted results on fields without an index *if* the sort operation uses less than 32 megabytes of memory.

`Operations Unavailable in Sharded Environments`
The `group` (page 14) does not work with sharding. Use `mapReduce` (page 18) or `aggregate` (page 29) instead.

`db.eval()` is incompatible with sharded collections. You may use `db.eval()` with un-sharded collections in a shard cluster.

`$where` (page 121) does not permit references to the `db` object from the `$where` (page 121) function. This is uncommon in un-sharded collections.

The `$atomic` (page 133) update modifier does not work in sharded environments.

`$snapshot` queries do not work in sharded environments.

2d Geospatial queries cannot use the $or operator

See Also:

`$or` (page 116) and `http://docs.mongodb.org/manual/core/geospatial-indexes`.

21.2.6 Naming Restrictions

Restrictions on Database Names

The dot (i.e. `.`) character is not permissible in database names.

Database names are case sensitive even if the underlying file system is case insensitive. Changed in version 2.2: For MongoDB instances running on Windows. In 2.2 the following characters are not permissible in database names:

```
/\. "*<>:|?
```

See *Restrictions on Database Names for Windows* (page 289) for more information.

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.)

See *faq-restrictions-on-collection-names* and *Restrictions on Collection Names* (page 289) for more information.

Restrictions on Field Names

Field names cannot contain dots (i.e. `.`), dollar signs (i.e. `$`), or null characters. See *faq-dollar-sign-escaping* for an alternate approach.

CHAPTER

TWENTYTWO

GLOSSARY

$cmd A virtual *collection* that exposes *MongoDB*'s *database commands*.

_id A field containing a unique ID, typically a BSON *ObjectId*. If not specified, this value is automatically assigned upon the creation of a new document. You can think of the `_id` as the document's *primary key*.

accumulator An *expression* in the *aggregation framework* that maintains state between documents in the *aggregation pipeline*. See: `$group` for a list of accumulator operations.

admin database A privileged database named `admin`. Users must have access to this database to run certain administrative commands. See *administrative commands* (page 11) for more information and *Administration Commands* (page 39) for a list of these commands.

aggregation Any of a variety of operations that reduce and summarize large sets of data. SQL's `GROUP` and MongoDB's map-reduce are two examples of aggregation functions.

aggregation framework The MongoDB aggregation framework provides a means to calculate aggregate values without having to use *map-reduce*.

See Also:

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

arbiter A member of a *replica set* that exists solely to vote in *elections*. Arbiter nodes do not replicate data.

See Also:

Delayed Nodes

balancer An internal MongoDB process that runs in the context of a *sharded cluster* and manages the splitting and migration of *chunks*. Administrators must disable the balancer for all maintenance operations on a sharded cluster.

box MongoDB's *geospatial* indexes and querying system allow you to build queries around rectangles on two-dimensional coordinate systems. These queries use the `$box` (page 124) operator to define a shape using the lower-left and the upper-right coordinates.

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. For a detailed spec, see bsonspec.org.

See Also:

The *bson-json-type-conversion-fidelity* section.

BSON types The set of types supported by the *BSON* serialization format. The following types are available:

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

btree A data structure used by most database management systems for to store indexes. MongoDB uses b-trees for its indexes.

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*. Once they reach their fixed size, capped collections automatically overwrite their oldest entries. MongoDB's *oplog* replication mechanism depends on capped collections. Developers may also use capped collections in their applications.

See Also:

The Capped Collections wiki page.

checksum A calculated value used to ensure data integrity. The *md5* algorithm is sometimes used as a checksum.

chunk In the context of a *sharded cluster*, a chunk is a contiguous range of *shard key* values assigned to a particular *shard*. By default, chunks are 64 megabytes or less. When they grow beyond the configured chunk size, a `mongos` (page 256) splits the chunk into two chunks.

circle MongoDB's *geospatial* indexes and querying system allow you to build queries around circles on two-dimensional coordinate systems. These queries use the `$circle` operator to define circle using the center and the radius of the circle.

client The application layer that uses a database for data persistence and storage. *Drivers* provide the interface level between the application layer and the database server.

cluster A set of `mongod` instances running in conjunction to increase database availability and performance. See *sharding* and *replication* for more information on the two different approaches to clustering with MongoDB.

collection Collections are groupings of *BSON documents*. Collections do not enforce a schema, but they are otherwise mostly analogous to *RDBMS* tables.

The documents within a collection may not need the exact same set of fields, but typically all documents in a collection have a similar or related purpose for an application.

All collections exit within a single *database*. The namespace within a database for collections are flat.

See *faq-dev-namespace* and `http://docs.mongodb.org/manual/core/document` for more information.

compound index An *index* consisting of two or more keys. See `http://docs.mongodb.org/manual/core/indexes` for more information.

config database One of three `mongod` instances that store all of the metadata associated with a *sharded cluster*.

control script A simple shell script, typically located in the `http://docs.mongodb.org/manual/etc/rc.d` or `http://docs.mongodb.org/manual/etc/init.d` directory and used by the system's initialization process to start, restart and stop a *daemon* process.

control script A script used by a UNIX-like operating system to start, stop, or restart a *daemon* process. On most systems, you can find these scripts in the `http://docs.mongodb.org/manual/etc/init.d/` or `http://docs.mongodb.org/manual/etc/rc.d/` directories.

CRUD Create, read, update, and delete. The fundamental operations of any database.

CSV A text-based data format consisting of comma-separated values. This format is commonly used to exchange database between relational databases, since the format is well-suited to tabular data. You can import CSV files using `mongoimport`.

cursor In MongoDB, a cursor is a pointer to the result set of a *query*, that clients can iterate through to retrieve results. By default, cursors will timeout after 10 minutes of inactivity.

daemon The conventional name for a background, non-interactive process.

data-center awareness A property that allows clients to address nodes in a system to based upon their location.

Replica sets implement data-center awareness using *tagging*.

See Also:

`members[n].tags` and *replica-set-configuration-tag-sets* for more information about tagging and replica sets.

database A physical container for *collections*. Each database gets its own set of files on the file system. A single MongoDB server typically servers multiple databases.

database command Any MongoDB operation other than an insert, update, remove, or query. MongoDB exposes commands as queries against the special *$cmd* collection. For example, the implementation of `count` (page 17) for MongoDB is a command.

See Also:

Command Reference (page 11) for a full list of database commands in MongoDB

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 Also:

Monitoring Database Systems.

dbpath Refers to the location of MongoDB's data file storage. The default `dbpath` is `http://docs.mongodb.org/manual/data/db`. Other common data paths include `http://docs.mongodb.org/manual/srv/mongodb` and `http://docs.mongodb.org/manual/var/lib/mongodb`.

See Also:

`dbpath` or `--dbpath` (page 209).

delayed member A member of a *replica set* that cannot become primary and applies operations at a specified delay. This 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 Also:

Delayed Members

diagnostic log `mongod` can create a verbose log of operations with the *mongod --diaglog* (page 209) option or through the `diagLogging` (page 56) command. The `mongod` creates this log in the directory specified to *mongod --dbpath* (page 209). The name of the is `diaglog.<time in hex>`, where "`<time-in-hex>`" reflects the initiation time of logging as a hexadecimal string.

> **Warning:** Setting the diagnostic level to `0` will cause `mongod` to stop writing data to the *diagnostic log* file. However, the `mongod` 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` instance before doing so.

See Also:

mongod --diaglog (page 209), `diaglog`, and `diagLogging` (page 56).

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*.

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

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

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

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

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

draining The process of removing or "shedding" *chunks* from one *shard* to another. Administrators must drain shards before removing them from the cluster.

See Also:

`removeShard` (page 13), *sharding*.

driver A client implementing the communication protocol required for talking to a server. The MongoDB drivers provide language-idiomatic methods for interfacing with MongoDB.

See Also:

`http://docs.mongodb.org/manual/applications/drivers`

election In the context of *replica sets*, an election is the process by which members of a replica set select primary nodes on startup and in the event of failures.

See Also:

Replica Set Elections and *priority*.

eventual consistency A property of a distributed system allowing changes to the system to propagate gradually. In a database system, this means that readable nodes are not required to reflect the latest writes at all times. In MongoDB, reads to a primary have *strict consistency*; reads to secondary nodes have *eventual consistency*.

expression In the context of the *aggregation framework*, expressions are the stateless transformations that operate on the data that passes through the *pipeline*.

See Also:

http://docs.mongodb.org/manual/applications/aggregation.

failover The process that allows one of the *secondary* nodes in a *replica set* to become *primary* in the event of a failure.

See Also:

Replica Set Failover.

field A name-value pair in a *document*. Documents have zero or more fields. Fields are analogous to columns in relational databases.

firewall A system level networking filter that restricts access based on, among other things, IP address. Firewalls form part of effective network security strategy.

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.

Geohash A value is a binary representation of the location on a coordinate grid.

geospatial Data that relates to geographical location. In MongoDB, you may index or store geospatial data according to geographical parameters and reference specific coordinates in queries.

GridFS A convention for storing large files in a MongoDB database. All of the official MongoDB drivers support this convention, as does the `mongofiles` program.

See Also:

mongofiles (page 246).

haystack index In the context of *geospatial* queries, haystack indexes enhance searches by creating "bucket" of objects grouped by a second criterion. For example, you might want all geographical searches to also include the type of location being searched for. In this case, you can create a haystack index that includes a document's position and type:

```
db.places.ensureIndex( { position: "geoHaystack", type: 1 } )
```

You can then query on position and type:

```
db.places.find( { position: [34.2, 33.3], type: "restaurant" } )
```

hidden member A member of a *replica set* that cannot become primary and is not advertised as part of the set in the *database command* `isMaster`, which prevents it from receiving read-only queries depending on *read preference*.

See Also:

Hidden Member, `isMaster`, `db.isMaster`, and `members[n].hidden`.

idempotent When calling an idempotent operation on a value or state, the operation only affects the value once. Thus, the operation can safely run multiple times without unwanted side effects. In the context of MongoDB, *oplog* entries must be idempotent to support initial synchronization and recovery from certain failure situations. Thus, MongoDB can safely apply oplog entries more than once without any ill effects.

index A data structure that optimizes queries. See `http://docs.mongodb.org/manual/core/indexes` for more information.

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`. to display dates. E.g. `YYYY-MM-DD HH:MM.SS.milis`.

JavaScript A popular scripting language original designed for web browsers. The MongoDB shell and certain server-side functions use a JavaScript interpreter.

journal A sequential, binary transaction used to bring the database into a consistent state in the event of a hard shutdown. MongoDB enables journaling by default for 64-bit builds of MongoDB version 2.0 and newer. Journal files are pre-allocated and will exist as three 1GB file in the data directory. To make journal files smaller, use `smallfiles`.

When enabled, MongoDB writes data first to the journal and after to the core data files. MongoDB commits to the journal every 100ms and this is configurable using the `journalCommitInterval` runtime option.

To force `mongod` to commit to the journal more frequently, you can specify `j:true`. When a write operation with `j:true` pending, `mongod` will reduce `journalCommitInterval` to a third of the set value.

See Also:

The Journaling wiki page.

JSON JavaScript Object Notation. A human-readable, plain text format for expressing structured data with support in many programming languages.

JSON document A *JSON* document is a collection of fields and values in a structured format. The following is a sample *JSON document* with two fields:

```
{ name: "MongoDB",
  type: "database" }
```

JSONP *JSON* with Padding. Refers to a method of injecting JSON into applications. Presents potential security concerns.

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.

map-reduce A data and 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.

See Also:

The Map Reduce wiki page for more information regarding MongoDB's map-reduce implementation, and `http://docs.mongodb.org/manual/applications/aggregation` for another approach to data aggregation in MongoDB.

master In conventional master/*slave* replication, the master database receives all writes. The *slave* instances replicate from the master instance in real time.

md5 `md5` is 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*.

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.

mongo The MongoDB Shell. `mongo` connects to `mongod` and `mongos` (page 256) instances, allowing administration, management, and testing. `mongo` has a JavaScript interface.

See Also:

mongo (page 217) and *JavaScript Interface* (page 63).

mongod The program implementing the MongoDB database server. This server typically runs as a *daemon*.

See Also:

mongod (page 207).

MongoDB The document-based database server described in this manual.

mongos The routing and load balancing process that acts an interface between an application and a MongoDB *sharded cluster*.

See Also:

mongos (page 214).

multi-master replication A *replication* method where multiple database instances can accept write operations to the same data set at any time. Multi-master replication exchanges increased concurrency and availability for a relaxed consistency semantic. MongoDB ensures consistency and, therefore, does not provide multi-master replication.

namespace A canonical name for a collection or index in MongoDB. Namespaces consist of a concatenation of the database and collection or index name, like so: `[database-name].[collection-or-index-name]`. All documents belong to a namespace.

natural order The order in which a database stores documents on disk. Typically, the order of documents on disks reflects insertion order, *except* when documents move internal because of document growth due to update operations. However, *Capped collections* guarantee that insertion order and natural order are identical.

When you execute `find()` with no parameters, the database returns documents in forward natural order. When you execute `find()` and include `sort()` with a parameter of `$natural:-1`, the database returns documents in reverse natural order.

ObjectId A special 12-byte *BSON* type that has a high probability of being unique when generated. The most significant digits in an ObjectId represent the time when the Object. MongoDB uses ObjectId values as the default values for *_id* fields.

operator A keyword beginning with a `$` used to express a complex query, update, or data transformation. For example, `$gt` is the query language's "greater than" operator. See the *Operator Reference* (page 111) for more information about the available operators.

See Also:

Operator Reference (page 111).

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 Also:

Oplog Sizes and `http://docs.mongodb.org/manual/tutorial/change-oplog-size`.

padding The extra space allocated to document on the disk to prevent moving a document when it grows as the result of `update()` operations.

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.

page fault The event that occurs when a process requests stored data (i.e. a page) from memory that the operating system has moved to disk.

See Also:

Storage FAQ: What are page faults?

partition A distributed system architecture that splits data into ranges. *Sharding* is a kind of partitioning.

pcap A packet capture format used by `mongosniff` to record packets captured from network interfaces and display them as human-readable MongoDB operations.

PID A process identifier. On UNIX-like systems, a unique integer PID is assigned to each running process. You can use a PID to inspect a running process and send signals to it.

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 The series of operations in the *aggregation* process.

See Also:

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

polygon MongoDB's *geospatial* indexes and querying system allow you to build queries around multi-sided polygons on two-dimensional coordinate systems. These queries use the `$within` (page 123) operator and a sequence of points that define the corners of the polygon.

powerOf2Sizes A per-*collection* setting that changes and normalizes the way that MongoDB allocates space for each *document* in an effort to maximize storage reuse reduce fragmentation. This is the default for `TTL Collections`. See `collMod` (page 38) and `usePowerOf2Sizes` (page 38) for more information. New in version 2.2.

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. When deploying a *sharded cluster*, in some cases pre-splitting will expedite the initial distribution of documents among shards by manually dividing the collection into chunks rather than waiting for the MongoDB *balancer* to create chunks during the course of normal operation.

primary In a *replica set*, the primary member is the current *master* instance, which receives all write operations.

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 For a database where *sharding* is enabled, the primary shard holds all un-sharded collections.

priority In the context of *replica sets*, priority is a configurable value that helps determine which nodes in a replica set are most likely to become *primary*.

See Also:

Replica Set Node Priority

projection A document given to a *query* that specifies which fields MongoDB will return from the documents in the result set.

query A read request. MongoDB queries use a *JSON*-like query language that includes a variety of *query operators* with names that begin with a `$` character. In the `mongo` shell, you can issue queries using the `db.collection.find()` and `db.collection.findOne()` methods.

query optimizer For each query, the MongoDB query optimizer generates a query plan that matches the query to the index that produces the fastest results. The optimizer then uses the query plan each time the `mongod` receives the query. If a collection changes significantly, the optimizer creates a new query plan.

See Also:

read-operations-query-optimization

RDBMS Relational Database Management System. A database management system based on the relational model, typically using *SQL* as the query language.

read preference A setting on the MongoDB `drivers` that determines how the clients direct read operations. Read preference affects all replica sets including shards. By default, drivers direct all reads to *primary* nodes for *strict consistency*. However, you may also direct reads to secondary nodes for *eventually consistent* reads.

See Also:

Read Preference

read-lock In the context of a reader-writer lock, a lock that while held allows concurrent readers, but no writers.

record size The space allocated for a document including the padding.

recovering A *replica set* member status indicating that a member is synchronizing or re-synchronizing its data from the primary node. Recovering nodes 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 Also:

`http://docs.mongodb.org/manual/replication` and `http://docs.mongodb.org/manual/core/repli`

replication A feature allowing multiple database servers to share the same data, thereby ensuring redundancy and facilitating load balancing. MongoDB supports two flavors of replication: master-slave replication and replica sets.

See Also:

replica set, *sharding*, `http://docs.mongodb.org/manual/replication`. and `http://docs.mongodb.org/manual/core/replication`.

replication lag The length of time between the last operation in the primary's *oplog* last operation applied to a particular *secondary* or *slave* node. In general, you want to keep replication lag as small as possible.

See Also:

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.

REST An API design pattern centered around the idea of resources and the *CRUD* operations that apply to them. Typically implemented over HTTP. MongoDB provides a simple HTTP REST interface that allows HTTP clients to run commands against the server.

rollback A process that, in certain replica set situations, reverts writes operations to ensure the consistency of all replica set members.

secondary In a *replica set*, the `secondary` members are the current *slave* instances that replicate the contents of the master database. Secondary members may handle read requests, but only the *primary* members can handle write operations.

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.

set name In the context of a *replica set*, the `set name` refers to an arbitrary name given to a replica set when it's first configured. All members of a replica set must have the same name specified with the `replSet` setting (or *`--replSet`* (page 212) option for `mongod`.)

See Also:

replication, `http://docs.mongodb.org/manual/replication` and `http://docs.mongodb.org/manual/core/replication`.

shard A single replica set that stores some portion of a sharded cluster's total data set. See *sharding*.

See Also:

The documents in the `http://docs.mongodb.org/manual/sharding` section of manual.

shard key In a sharded collection, a shard key is the field that MongoDB uses to distribute documents among members of the *sharded cluster*.

sharded cluster The set of nodes comprising a *sharded* MongoDB deployment. A sharded cluster consists of three config processes, one or more replica sets, and one or more `mongos` (page 256) routing processes.

See Also:

The documents in the `http://docs.mongodb.org/manual/sharding` section of manual.

sharding A database architecture that enable horizontal scaling by splitting data into key ranges among two or more replica sets. This architecture is also known as "range-based partitioning." See *shard*.

See Also:

The documents in the `http://docs.mongodb.org/manual/sharding` section of manual.

shell helper A number of *database commands* (page 11) have "helper" methods in the `mongo` shell that provide a more concise syntax and improve the general interactive experience.

See Also:

mongo (page 217) and *JavaScript Interface* (page 63).

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.

slave In conventional *master*/slave replication, slaves are read-only instances that replicate operations from the *master* database. Data read from slave instances may not be completely consistent with the master. Therefore, applications requiring consistent reads must read from the master database instance.

split The division between *chunks* in a *sharded cluster*.

SQL Structured Query Language (SQL) is a common special-purpose programming language used for interaction with a relational database including access control as well as inserting, updating, querying, and deleting data. 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. Often, `SQL` is often used as a 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.

standalone In MongoDB, a standalone is an instance of `mongod` that is running as a single server and not as part of a *replica set*.

strict consistency A property of a distributed system requiring that all nodes always reflect the latest changes to the system. In a database system, this means that any system that can provide data must reflect the latest writes at all times. In MongoDB, reads to a primary have *strict consistency*; reads to secondary nodes have *eventual consistency*.

syslog On UNIX-like systems, a logging process that provides a uniform standard for servers and processes to submit logging information.

tag One or more labels applied to a given replica set member that clients may use to issue data-center aware operations.

TSV A text-based data format consisting of tab-separated values. This format is commonly used to exchange database between relational databases, since the format is well-suited to tabular data. You can import TSV files using `mongoimport`.

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 system before the system deletes it or ages it out.

unique index An index that enforces uniqueness for a particular field across a single collection.

upsert A kind of update that either updates the first document matched in the provided query selector or, if no document matches, inserts a new document having the fields implied by the query selector and the update operation.

virtual memory An application's working memory, typically residing on both disk an in physical RAM.

working set The collection of data that MongoDB uses regularly. This data is typically (or preferably) held in RAM.

write concern Specifies whether a write operation has succeeded. Write concern allows your application to detect insertion errors or unavailable `mongod` instances. For *replica sets*, you can configure write concern to confirm replication to a specified number of members.

See Also:

Write Concern, `http://docs.mongodb.org/manual/core/write-operations`, and *Write Concern for Replica Sets*.

write-lock A lock on the database for a given writer. When a process writes to the database, it takes an exclusive write-lock to prevent other processes from writing or reading.

writeBacks The process within the sharding system that ensures that writes issued to a *shard* that isn't responsible for the relevant chunk, get applied to the proper shard.

Part IX

Release Notes

Always install the latest, stable version of MongoDB. See the following release notes for an account of the changes in major versions. Release notes also include instructions for upgrade.

Current stable release (v2.2-series):

CHAPTER

TWENTYTHREE

RELEASE NOTES FOR MONGODB 2.2

See the `full index of this page` for a complete list of changes included in 2.2.

23.1 Upgrading

MongoDB 2.2 is a production release series and succeeds the 2.0 production release series.

MongoDB 2.0 data files are compatible with 2.2-series binaries without any special migration process. However, always perform the upgrade process for replica sets and sharded clusters using the procedures that follow.

Always upgrade to the latest point release in the 2.2 point release. Currently the latest release of MongoDB is 2.2.2.

23.1.1 Synopsis

- `mongod`, 2.2 is a drop-in replacement for 2.0 and 1.8.
- Check your `driver` documentation for information regarding required compatibility upgrades, and always run the recent release of your driver.

  Typically, only users running with authentication, will need to upgrade drivers before continuing with the upgrade to 2.2.
- For all deployments using authentication, upgrade the drivers (i.e. client libraries), before upgrading the `mongod` instance or instances.
- For all upgrades of sharded clusters:
  - turn off the balancer during the upgrade process. See the *sharding-balancing-disable-temporally* section for more information.
  - upgrade all `mongos` (page 256) instances before upgrading any `mongod` instances.

Other than the above restrictions, 2.2 processes can interoperate with 2.0 and 1.8 tools and processes. You can safely upgrade the `mongod` and `mongos` (page 256) 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.

23.1.2 Upgrading a Standalone `mongod`

1. Download binaries of the latest release in the 2.2 series from the MongoDB Download Page.
2. Shutdown your `mongod` instance. Replace the existing binary with the 2.2 `mongod` binary and restart MongoDB.

23.1.3 Upgrading a Replica Set

You can upgrade to 2.2 by performing a "rolling" upgrade of the set by upgrading the members individually while the other members are available to minimize downtime. Use the following procedure:

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` and replacing the 2.0 binary with the 2.2 binary. After upgrading a `mongod` instance, wait for the member to recover to `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` in the `mongo` shell.
2. Use the `mongo` shell method `rs.stepDown()` to step down the *primary* to allow the normal *failover* procedure. `rs.stepDown()` expedites the failover procedure and is preferable to shutting down the primary directly.

   Once the primary has stepped down and another member has assumed `PRIMARY` state, as observed in the output of `rs.status()`, shut down the previous primary and replace `mongod` binary with the 2.2 binary and start the new process.

   Note: Replica set failover is not instant but will render the set unavailable to read or accept writes until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the upgrade during a predefined maintenance window.

23.1.4 Upgrading a Sharded Cluster

Use the following procedure to upgrade a sharded cluster:

- *Disable the balancer*.
- Upgrade all `mongos` (page 256) instances *first*, in any order.
- Upgrade all of the `mongod` config server instances using the *stand alone* (page 286) procedure. To keep the cluster online, be 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 286) detailed above.
- re-enable the balancer.

Note: Balancing is not currently supported in *mixed* 2.0.x and 2.2.0 deployments. Thus you will want to reach a consistent version for all shards within a reasonable period of time, e.g. same-day. See SERVER-6902 for more information.

23.2 Changes

23.2.1 Major Features

Aggregation Framework

The aggregation framework makes it possible to do aggregation operations without needing to use *map-reduce*. The `aggregate` (page 29) command exposes the aggregation framework, and the `db.collection.aggregate()` helper in the `mongo` shell provides an interface to these operations. Consider the following resources for background on the aggregation framework and its use:

- Documentation: `http://docs.mongodb.org/manual/applications/aggregation`
- Reference: *Aggregation Framework Reference* (page 141)
- Examples: `http://docs.mongodb.org/manual/tutorial/aggregation-examples`

TTL Collections

TTL collections remove expired data from a collection, using a special index and a background thread that deletes expired documents every minute. These collections are useful as an alternative to *capped collections* in some cases, such as for data warehousing and caching cases, including: machine generated event data, logs, and session information that needs to persist in a database for only a limited period of time.

For more information, see the `http://docs.mongodb.org/manual/tutorial/expire-data` tutorial.

Concurrency Improvements

MongoDB 2.2 increases the server's capacity for concurrent operations with the following improvements:

1. DB Level Locking
2. Improved Yielding on Page Faults
3. Improved Page Fault Detection on Windows

To reflect these changes, MongoDB now provides changed and improved reporting for concurrency and use, see *locks* (page 168) and *recordStats* (page 180) in *server status* (page 167) and see *current operation output* (page 199), `db.currentOp()`, *mongotop* (page 242), and *mongostat* (page 238).

Improved Data Center Awareness with Tag Aware Sharding

MongoDB 2.2 adds additional support for geographic distribution or other custom partitioning for sharded collections in *clusters*. By using this "tag aware" sharding, you can automatically ensure that data in a sharded database system is always on specific shards. For example, with tag aware sharding, you can ensure that data is closest to the application servers that use that data most frequently.

Shard tagging controls data location, and is complementary but separate from replica set tagging, which controls *read preference* and *write concern*. For example, shard tagging can pin all "USA" data to one or more logical shards, while replica set tagging can control which `mongod` instances (e.g. "`production`" or "`reporting`") the application uses to service requests.

See the documentation for the following helpers in the `mongo` shell that support tagged sharding configuration:

- `sh.addShardTag()`
- `sh.addTagRange()`

- `sh.removeShardTag()`

Also, see the wiki page for tag aware sharding.

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 256) and applies identically to single replica sets and to the replica sets for each shard in a *sharded cluster*.

Additional read preference support now exists in the `mongo` shell using the `readPref()` cursor method.

23.2.2 Compatibility Changes

Authentication Changes

MongoDB 2.2 provides more reliable and robust support for authentication clients, including drivers and `mongos` (page 256) 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 286).

`findAndModify` Returns Null Value for Upserts

In version 2.2, for *upsert* operations, `findAndModify` (page 25) commands will now return the following output:

```
{'ok': 1.0, 'value': null}
```

In the `mongo` shell, `findAndModify` (page 25) operations running as upserts will only output a `null` value.

Previously, in version 2.0 these operations would return an empty document, e.g. `{ }`.

See: SERVER-6226 for more information.

`mongodump` Output can only Restore to 2.2 MongoDB Instances

If you use the `mongodump` tool from the 2.2 distribution to create a dump of a database, you may only restore that dump to a 2.2 database.

See: SERVER-6961 for more information.

`ObjectId().toString()` Returns String Literal `ObjectId("...")`

In version 2.2, the `ObjectId.toString()` method returns the string representation of the *ObjectId()* object and has the format `ObjectId("...")`.

Consider the following example that calls the `toString()` method on the `ObjectId("507c7f79bcf86cd7994f6c0e")` object:

```
ObjectId("507c7f79bcf86cd7994f6c0e").toString()
```

The method now returns the *string* `ObjectId("507c7f79bcf86cd7994f6c0e")`.

Previously, in version 2.0, the method would return the *hexadecimal string* `507c7f79bcf86cd7994f6c0e`.

If compatibility between versions 2.0 and 2.2 is required, use *ObjectId().str*, which holds the hexadecimal string value in both versions.

`ObjectId().valueOf()` Returns hexadecimal string

In version 2.2, the `ObjectId.valueOf()` method returns the value of the *ObjectId()* object as a lowercase hexadecimal string.

Consider the following example that calls the `valueOf()` method on the `ObjectId("507c7f79bcf86cd7994f6c0e")` object:

```
ObjectId("507c7f79bcf86cd7994f6c0e").valueOf()
```

The method now returns the *hexadecimal string* `507c7f79bcf86cd7994f6c0e`.

Previously, in version 2.0, the method would return the *object* `ObjectId("507c7f79bcf86cd7994f6c0e")`.

If compatibility between versions 2.0 and 2.2 is required, use *ObjectId().str* attribute, which holds the hexadecimal string value in both versions.

23.2.3 Behavioral Changes

Restrictions on Collection Names

In version 2.2, collection names cannot:

- contain the `$`.
- be an empty string (e.g. `""`).

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 and the *faq-restrictions-on-collection-names* FAQ item.

Restrictions on Database Names for Windows

Database names running on Windows can no longer contain the following characters:

```
/\. "*<>:|?
```

The names of the data files include the database name. If you attempt to upgrade a database instance with one or more of these characters, `mongod` will refuse to start.

Change the name of these databases before upgrading. See SERVER-4584 and SERVER-6729 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 for more information.

New `$elemMatch` Projection Operator

The `$elemMatch` operator allows applications to narrow the data returned from queries so that the query operation will only return the first matching element in an array. See the `http://docs.mongodb.org/manual/reference/projection/elemMatch` documentation and the SERVER-2238 and SERVER-828 issues for more information.

23.2.4 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 for more information.

Service Support for `mongos.exe`

You may now run `mongos.exe` instances as a Windows Service. See the *mongos.exe* (page 221) reference and *tutorial-mongod-as-windows-service* and SERVER-1589 for more information.

Log Rotate Command Support

MongoDB for Windows now supports log rotation by way of the `logRotate` (page 46) database command. See SERVER-2612 for more information.

New Build Using SlimReadWrite Locks for Windows Concurrency

Labeled "2008+" on the Downloads Page, 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 for more information.

23.2.5 Tool Improvements

Index Definitions Handled by `mongodump` and `mongorestore`

When you specify the `--collection` (page 224) option to `mongodump`, `mongodump` will now backup the definitions for all indexes that exist on the source database. When you attempt to restore this backup with `mongorestore`, the target `mongod` will rebuild all indexes. See SERVER-808 for more information.

mongorestore now includes the *--noIndexRestore* (page 228) option to provide the preceding behavior. Use *--noIndexRestore* (page 228) to prevent mongorestore from building previous indexes.

mongooplog for Replaying Oplogs

The mongooplog tool makes it possible to pull *oplog* entries from mongod instance and apply them to another mongod instance. You can use mongooplog to achieve point-in-time backup of a MongoDB data set. See the SERVER-3873 case and the *mongooplog* (page 229) documentation.

Authentication Support for mongotop and mongostat

mongotop and mongostat now contain support for username/password authentication. See SERVER-3875 and SERVER-3871 for more information regarding this change. Also consider the documentation of the following options for additional information:

- *mongotop --username* (page 243)
- *mongotop --password* (page 243)
- *mongostat --username* (page 239)
- *mongostat --password* (page 239)

Write Concern Support for mongoimport and mongorestore

mongoimport now provides an option to halt the import if the operation encounters an error, such as a network interruption, a duplicate key exception, or a write error. The *--stopOnError* (page 235) option will produce an error rather than silently continue importing data. See SERVER-3937 for more information.

In mongorestore, the *--w* (page 228) option provides support for configurable write concern.

mongodump Support for Reading from Secondaries

You can now run mongodump when connected to a *secondary* member of a *replica set*. See SERVER-3854 for more information.

mongoimport Support for full 16MB Documents

Previously, mongoimport would only import documents that were less than 4 megabytes in size. This issue is now corrected, and you may use mongoimport to import documents that are at least 16 megabytes ins size. See SERVER-4593 for more information.

Timestamp() Extended JSON format

MongoDB extended JSON now includes a new Timestamp() type to represent the Timestamp type that MongoDB uses for timestamps in the *oplog* among other contexts.

This permits tools like mongooplog and mongodump to query for specific timestamps. Consider the following mongodump operation:

```
mongodump --db local --collection oplog.rs --query '{"ts":{"$gt":{"$timestamp" : {"t": 1344969612000,
```

See SERVER-3483 for more information.

23.2.6 Shell Improvements

Improved Shell User Interface

2.2 includes a number of changes that improve the overall quality and consistency of the user interface for the `mongo` shell:

- Full Unicode support.
- Bash-like line editing features. See SERVER-4312 for more information.
- Multi-line command support in shell history. See SERVER-3470 for more information.
- Windows support for the `edit` command. See SERVER-3998 for more information.

Helper to load Server-Side Functions

The `db.loadServerScripts()` loads the contents of the current database's `system.js` collection into the current `mongo` shell session. See SERVER-1651 for more information.

Support for Bulk Inserts

If you pass an array of *documents* to the `insert()` method, the `mongo` shell will now perform a bulk insert operation. See SERVER-3819 and SERVER-2395 for more information.

23.2.7 Operations

Support for Logging to Syslog

See the SERVER-2957 case and the documentation of the `syslog` run-time option or the *`mongod --syslog`* (page 208) and *`mongos --syslog`* (page 215) command line-options.

`touch` Command

Added the `touch` (page 39) command to read the data and/or indexes from a collection into memory. See: SERVER-2023 and `touch` (page 39) for more information.

`indexCounters` No Longer Report Sampled Data

`indexCounters` (page 173) now report actual counters that reflect index use and state. In previous versions, these data were sampled. See SERVER-5784 and `indexCounters` (page 173) for more information.

Padding Specifiable on `compact` Command

See the documentation of the `compact` (page 41) and the SERVER-4018 issue for more information.

Added Build Flag to Use System Libraries

The Boost library, version 1.49, is now embedded in the MongoDB code base.

If you want to build MongoDB binaries using system Boost libraries, you can pass `scons` using the `--use-system-boost` flag, as follows:

```
scons --use-system-boost
```

When building MongoDB, you can also pass `scons` a flag to compile MongoDB using only system libraries rather than the included versions of the libraries. For example:

```
scons --use-system-all
```

See the SERVER-3829 and SERVER-5172 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 and SERVER-4683. For more information about TCMalloc, see the documentation of TCMalloc itself.

23.2.8 Replication

Improved Logging for Replica Set Lag

When *secondary* members of a replica set fall behind in replication, `mongod` now provides better reporting in the log. This makes it possible to track replication in general and identify what process may produce errors or halt replication. See SERVER-3575 for more information.

Replica Set Members can Sync from Specific Members

The new `replSetSyncFrom` command and new `rs.syncFrom()` helper in the `mongo` shell make it possible for you to manually configure from which member of the set a replica will poll *oplog* entries. Use these commands to override the default selection logic if needed. Always exercise caution with `replSetSyncFrom` when overriding the default behavior.

Replica Set Members will not Sync from Members Without Indexes Unless `buildIndexes: false`

To prevent inconsistency between members of replica sets, if the member of a replica set has `members[n].buildIndexes` set to `true`, other members of the replica set will *not* sync from this member, unless they also have `members[n].buildIndexes` set to `true`. See SERVER-4160 for more information.

New Option To Configure Index Pre-Fetching during Replication

By default, when replicating options, *secondaries* will pre-fetch *indexes* associated with a query to improve replication throughput in most cases. The `replIndexPrefetch` setting and *`--replIndexPrefetch`* (page 213) option allow administrators to disable this feature or allow the `mongod` to pre-fetch only the index on the `_id` field. See SERVER-6718 for more information.

23.2.9 Map Reduce Improvements

In 2.2 Map Reduce received the following improvements:

- Improved support for sharded MapReduce, and
- MapReduce will retry jobs following a config error.

23.2.10 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 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.

23.3 Licensing Changes

Added License notice for Google Perftools (TCMalloc Utility). See the License Notice and the SERVER-4683 for more information.

23.4 Resources

- MongoDB Downloads
- All JIRA Issues resolved in 2.2
- All Backwards Incompatible Changes
- All Third Party License Notices

23.4.1 What's New in MongoDB 2.2 Online Conference

- Introduction and Welcome
- The Aggregation Framework
- Concurrency
- Data Center Awareness
- TTL Collections
- Closing Remarks and Q&A

See `http://docs.mongodb.org/manual/release-notes/2.2-changes` for an overview of all changes in 2.2.

Previous stable releases:

CHAPTER TWENTYFOUR

RELEASE NOTES FOR MONGODB 2.0

See the `full index of this page` for a complete list of changes included in 2.0.

24.1 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.

24.1.1 Preparation

Read through all release notes before upgrading, and ensure that no changes will affect your deployment.

If you create new indexes in 2.0, then downgrading to 1.8 is possible but you must reindex the new collections.

`mongoimport` and `mongoexport` now correctly adhere to the CSV spec for handling CSV input/output. This may break existing import/export workflows that relied on the previous behavior. For more information see SERVER-1097.

Journaling is **enabled by default** in 2.0 for 64-bit builds. If you still prefer to run without journaling, start `mongod` with the *`--nojournal`* (page 210) run-time option. Otherwise, MongoDB creates journal files during startup. The first time you start `mongod` with journaling, you will see a delay the `mongod` creates new files. In addition, you may see reduced write throughput.

2.0 `mongod` instances are interoperable with 1.8 `mongod` instances; however, for best results, upgrade your deployments using the following procedures:

24.1.2 Upgrading a Standalone `mongod`

1. Download the v2.0.x binaries from the MongoDB Download Page.
2. Shutdown your `mongod` instance. Replace the existing binary with the 2.0.x `mongod` binary and restart MongoDB.

24.1.3 Upgrading a Replica Set

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` and replacing the 1.8 binary with the 2.0.x binary from the MongoDB Download Page.
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()` to step down the primary to allow the normal *failover* procedure.

   `rs.stepDown()` and `replSetStepDown` provide for shorter and more consistent failover procedures than simply shutting down the primary directly.

   When the primary has stepped down, shut down its instance and upgrade by replacing the `mongod` binary with the 2.0.x binary.

24.1.4 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 256) routers in any order.

24.2 Changes

24.2.1 Compact Command

A `compact` (page 41) command is now available for compacting a single collection and its indexes. Previously, the only way to compact was to repair the entire database.

24.2.2 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 for more information.

The specific operations yield in 2.0 are:

- Updates by `_id`
- Removes
- Long cursor iterations

24.2.3 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.

24.2.4 Index Performance Enhancements

v2.0 includes significant improvements to the index structures. 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.
- The `repair` database command converts indexes to a 2.0 indexes.

To convert all indexes for a given collection to the *2.0 type* (page 297), invoke the `compact` (page 41) command.

Once you create new indexes, downgrading to 1.8.x will require a re-index of any indexes created using 2.0. See `http://docs.mongodb.org/manual/tutorial/roll-back-to-v1.8-index`.

24.2.5 Sharding Authentication

Applications can now use authentication with *sharded clusters*.

24.2.6 Replica Sets

Hidden Nodes in Sharded Clusters

In 2.0, `mongos` (page 256) instances can now determine when a member of a replica set becomes "hidden" without requiring a restart. In 1.8, `mongos` (page 256) if you reconfigured a member as hidden, you *had* to restart `mongos` (page 256) 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 *Member 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 Tagging.

Drivers may also support tag-aware reads. Instead of specifying `slaveOk`, you specify `slaveOk` with tags indicating which data-centers to read from. For details, see the `http://docs.mongodb.org/manual/applications/drivers` documentation.

`w : majority`

You can also set `w` to `majority` to ensure that the write propagates to a majority of nodes, effectively committing it. The value for "majority" will automatically adjust as you add or remove nodes from the set.

For more information, see *replica-set-write-concern*.

Reconfiguration with a Minority Up

If the majority of servers in a set has been permanently lost, you can now force a reconfiguration of the set to bring it back online.

For more information see `http://docs.mongodb.org/manual/tutorial/reconfigure-replica-set-with-unava`

Primary Checks for a Caught up Secondary before Stepping Down

To minimize time without a *primary*, the `rs.stepDown()` method will now fail if the primary does not see a *secondary* within 10 seconds of its latest optime. You can force the primary to step down anyway, but by default it will return an error message.

See also `http://docs.mongodb.org/manual/tutorial/force-member-to-be-primary`.

Extended Shutdown on the Primary to Minimize Interruption

When you call the `shutdown` (page 44) 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 44) 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 41) 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.

24.2.7 Geospatial Features

Multi-Location Documents

Indexing is now supported on documents which have multiple location objects, embedded either inline or in nested sub-documents. Additional command options are also supported, allowing results to return with not only distance but the location used to generate the distance.

For more information, see Multi-location Documents.

Polygon searches

Polygonal `$within` (page 123) queries are also now supported for simple polygon shapes. For details, see the `$within` (page 123) operator documentation.

24.2.8 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` (page 210) run-time option exists for specifying your own group commit interval. 100ms is the default (same as in 1.8).
- A new `{ getLastError: { j: true } }` (page 50) 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.

24.2.9 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 50) 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 50) results.

See OP_INSERT.

24.2.10 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 MapReduce Output Options and `http://docs.mongodb.org/manual/reference/command/map`

Performance Improvements

Map/reduce performance will benefit from the following:

- Larger in-memory buffer sizes, reducing the amount of disk I/O needed during a job
- Larger javascript heap size, allowing for larger objects and less GC

- Supports pure JavaScript execution with the `jsMode` flag. See `http://docs.mongodb.org/manual/reference/command/mapReduce`.

24.2.11 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 Regular Expressions and `$regex` (page 122).

$and

A special boolean `$and` (page 115) query operator is now available.

24.2.12 Command Output Changes

The output of the `validate` (page 49) 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.

24.2.13 Shell Features

Custom Prompt

You can define a custom prompt for the `mongo` shell. You can change the prompt at any time by setting the prompt variable to a string or a custom JavaScript function returning a string. For examples, see 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` (page 218).

For more information, see *mongo* (page 217).

24.2.14 Most Commands Require Authentication

In 2.0, when running with authentication (e.g. `auth`) *all* database commands require authentication, *except* the following commands.

- `isMaster`
- `authenticate` (page 61)
- `getnonce` (page 57)
- `buildInfo` (page 48)
- `ping` (page 52)
- `isdbgrid` (page 55)

24.3 Resources

- MongoDB Downloads
- All JIRA Issues resolved in 2.0
- All Backward Incompatible Changes

CHAPTER

TWENTYFIVE

RELEASE NOTES FOR MONGODB 1.8

See the `full index of this page` for a complete list of changes included in 1.8.

25.1 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 303).
- The `mapReduce` (page 18) command has changed in 1.8, causing incompatibility with previous releases. `mapReduce` (page 18) 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 18) document. If you use MapReduce, this also likely means you need a recent version of your client driver.

25.1.1 Preparation

Read through all release notes before upgrading and ensure that no changes will affect your deployment.

25.1.2 Upgrading a Standalone `mongod`

1. Download the v1.8.x binaries from the MongoDB Download Page.
2. Shutdown your `mongod` instance.
3. Replace the existing binary with the 1.8.x `mongod` binary.
4. Restart MongoDB.

25.1.3 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.

2. Change your config (optional) to prevent election of a new primary.

   It is possible that, when you start shutting down members of the set, a new primary will be elected. To prevent this, you can give all of the secondaries a priority of `0` before upgrading, and then change them back afterwards. To do so:

   (a) Record your current config. Run `rs.config()` and paste the results into a text file.

   (b) Update your config so that all secondaries have priority `0`. For example:

```
config = rs.conf()
{
     "_id" : "foo",
     "version" : 3,
     "members" : [
             {
                     "_id" : 0,
                     "host" : "ubuntu:27017"
             },
             {
                     "_id" : 1,
                     "host" : "ubuntu:27018"
             },
             {
                     "_id" : 2,
                     "host" : "ubuntu:27019",
                     "arbiterOnly" : true
             }
             {
                     "_id" : 3,
                     "host" : "ubuntu:27020"
             },
             {
                     "_id" : 4,
                     "host" : "ubuntu:27021"
             },
     ]
}
config.version++
3
rs.isMaster()
{
     "setName" : "foo",
     "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.
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.

25.1.4 Upgrading a Sharded Cluster

1. Turn off the balancer:

```
mongo <a_mongos_hostname>
use config
db.settings.update({_id:"balancer"},{$set : {stopped:true}}, true)
```

2. For each *shard*:
   - If the shard is a *replica set*, follow the directions above for *Upgrading a Replica Set* (page 303).
   - If the shard is a single `mongod` process, shut it down and then restart it with the 1.8.x binary from the MongoDB Download Page.
3. For each `mongos` (page 256):
   (a) Shut down the `mongos` (page 256) process.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page.
4. For each config server:
   (a) Shut down the config server process.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page.
5. Turn on the balancer:

```
use config
db.settings.update({_id:"balancer"},{$set : {stopped:false}})
```

25.1.5 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.

25.2 Changes

25.2.1 Journaling

MongoDB now supports write-ahead journaling to facilitate fast crash recovery and durability in the storage engine. With journaling enabled, a `mongod` can be quickly restarted following a crash without needing to repair the *collections*. The aggregation framework makes it possible to do aggregation

25.2.2 Sparse and Covered Indexes

Sparse Indexes are indexes that only include documents that contain the fields specified in the index. Documents missing the field will not appear in the index at all. This can significantly reduce index size for indexes of fields that contain only a subset of documents within a *collection*.

Covered Indexes enable MongoDB to answer queries entirely from the index when the query only selects fields that the index contains.

25.2.3 Incremental MapReduce Support

The `mapReduce` (page 18) 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 18) document.

25.2.4 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 256).
- `getLastError` (page 50) improvements for *sharding*.

1.7.4

- `mongos` (page 256) routes `slaveOk` queries to *secondaries* in *replica sets*.
- New `mapReduce` (page 18) output options.
- *index-type-sparse*.

1.7.3

- Initial *covered index* support.
- Distinct can use data from indexes when possible.
- `mapReduce` (page 18) can merge or reduce results into an existing collection.
- `mongod` tracks and `mongostat` displays network usage. See *mongostat* (page 238).
- Sharding stability improvements.

1.7.2

- `$rename` (page 128) operator allows renaming of fields in a document.
- `db.eval()` not to block.
- Geo queries with sharding.
- *`mongostat --discover`* (page 239) option
- Chunk splitting enhancements.
- Replica sets network enhancements for servers behind a nat.

1.7.1

- Many sharding performance enhancements.
- Better support for `$elemMatch` on primitives in embedded arrays.
- Query optimizer enhancements on range queries.
- Window service enhancements.
- Replica set setup improvements.
- `$pull` (page 132) 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.

25.2.5 Release Announcement Forum Pages

- 1.8.1, 1.8.0
- 1.7.6, 1.7.5, 1.7.4, 1.7.3, 1.7.2, 1.7.1, 1.7.0

25.3 Resources

- MongoDB Downloads
- All JIRA Issues resolved in 1.8

Current Development series:

CHAPTER

TWENTYSIX

RELEASE NOTES FOR MONGODB 2.4 (2.3 DEVELOPMENT SERIES)

MongoDB 2.4 is currently in development, as part of the 2.3 development release series. While 2.3-series releases are currently available, these versions of MongoDB are for **testing** *only*, and are *not for production use* under any circumstances.

Important: *All* interfaces *and* functionality described in this document are subject to change before the 2.4.0 release.

This document will eventually contain the full release notes for MongoDB 2.4; during the development cycle this document will contain documentation of new features and functionality only available in the 2.3 releases.

See the `full index of this page` for a complete list of changes included in 2.4.

26.1 Downloading

You can download the 2.3 release on the downloads page in the *Development Release (Unstable)* section. There are no distribution packages for development releases, but you can use the binaries provided for testing purposes. See `http://docs.mongodb.org/manual/tutorial/install-mongodb-on-linux`, `http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows`, or `http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x` for the basic installation process.

26.2 Changes

26.2.1 Additional Authentication Features

Note: These features are only present in the MongoDB Subscriber Edition. To download 2.3 development releases of the Subscriber Edition, use the following resources:

- Amazon Linux 6.4
- Red Hat Enterprise Linux 6.2
- Ubuntu 11.04
- SUSE 11

An improved authentication system is a core focus of the entire 2.3 cycle, as of 2.3.1, the following components of the new authentication system are available for use in MongoDB:

- SASL Support in `mongod`, the `mongo` shell, and C++ client library (driver.)
- New `acquirePrivilege` (page 311), `saslBegin` (page 311), and `saslContinue` (page 311) database commands.

Note: As of 2.3.1 support for SASL/Kerberos in `mongos` (page 256) is forthcoming. Test Kerberos with standalone `mongod` instances and *replica sets*.

Initial SASL/Kerberos Support

Development work on this functionality is ongoing, and additional related functionality is forthcoming. To use Kerberos with MongoDB as of the current 2.3-series release, consider the following requirements:

- add users to MongoDB as with the existing authentication mechanism:
  - Usernames *must* correspond to the Kerberos principal (e.g. `<username>@<REALM>` as in `mongodbuser@EXAMPLE.COM`),
  - You *must* have a user document in the `system.users` (page 261) collection with the Kerberos principal for any database that you want to grant access.
- every `mongod` using Kerberos must have a fully resolvable fully qualified domain name. This includes all members of replica sets.
- every `mongod` using Kerberos must have a Kerberos service principal, in the form of: `mongodb/<fqdn>@<REALM>`.
- each system running a `mongod` with Kerberos must have a key tab file that holds key data granting access to it's principal that the `mongod` can read.

To start `mongod` with support for Kerberos, use the following form:

```
env KRB5_KTNAME=<path to keytab file> <mongod invocation>
```

You must start `mongod` with `auth` or `keyfile`, [1] so that an actual command would resemble:

```
env KRB5_KTNAME=/opt/etc/mongodb.keytab \
    /opt/bin/mongod --auth --dbpath /opt/data/db --logpath /opt/log/mongod.log --fork
```

Replace the paths as needed for your test deployment.

To use Kerberos with the `mongo` shell, begin by initializing a Kerberos session with `kinit`. Then start a 2.3.1 or greater `mongo` shell instance, and run the following operations to associate the current connection with the Kerberos session:

[1] `keyfile` implies `auth`, and you *must* use `keyfile` for replica sets.

```
db.getMongo().saslAuthenticate( { mechanism: "GSSAPI",
                                  principal: "<username>@<REALM>" } )
```

The value of the `principal` field *must* be the same principal that you initialized with `kinit`. Continue to gain privileges using the `acquirePrivilege` (page 311) in an operation that resembles the following:

```
db.adminCommand( { acquirePrivilege: 1,
                   resource: <dbname>,
                   principal: <principalName>,
                   actions: [ <actionString> ] } )
```

Replace the `<dbname>` with the name of the database you want privileges, replace `<principalName>` with the Kerberos principal you initialized with `kinit`. The `<actionString>` list, contains the privileges you are acquiring, currently this value must be either:

- `oldRead`, or
- `oldWrite`.

The `oldRead` action string corresponds to the "*read only*" privileges in the existing authentication system, while `oldWrite` corresponds to the existing "*read/write*" privileges.

See Also:

http://docs.mongodb.org/manual/security

New Authentication Control Database Commands

In the 2.3 series, MongoDB adds the following database commands:

acquirePrivilege

saslBegin

saslContinue

26.2.2 Default Java Script Engine Switched to v8 from SpiderMonkey

The default JavaScript engine used throughout MongoDB, for the `mongo` shell, `mapReduce` (page 18), `$where` (page 121), and `eval` (page 27) is now v8.

`serverBuildInfo.`**`interpreterVersion`**
: The `interpreterVersion` (page 311) field of the document output by `db.serverBuildInfo()` in the `mongo` shell reports which JavaScript interpreter the `mongod` instance is running.

`interpreterVersion``()`
: The `interpreterVersion()` (page 311) in the `mongo` shell reports which JavaScript interpreter this `mongo` shell uses.

26.2.3 New Geospatial Indexes with GeoJSON and Improved Spherical Geometry

Note: In 2.3.2, the index type for Spherical Geospatial Indexes will become `2dsphere`

The 2.3 series adds a new type of geospatial index that supports improved spherical queries and GeoJSON. Create the index by specifying `s2d` as the value of the field in the index specification, as any of the following:

```
db.collection.ensureIndex( { geo: "s2d" } )
db.collection.ensureIndex( { type: 1, geo: "s2d" } )
db.collection.ensureIndex( { geo: "s2d", type: 1 } )
```

In the first example you create a spherical geospatial index on the field named `geo`, in the second example, you create a compound index where the first field is a normal index, and the index of the second field is a spherical geospatial index. Unlike `2d` indexes, fields indexed using the `s2d` type can do not have to be the first field in a compound index.

You must store data in the fields indexed using the `s2d` index using the GeoJSON specification, at the moment. Support for storing points, in the form used by the existing `2d` (i.e. geospatial) indexes is forthcoming. Currently, `s2d` indexes only support the following GeoJSON shapes:

- `Point`, as in the following:

  ```
  { "type": "Point", "coordinates": [ 40, 5 ] }
  ```

- `LineString`, as in the following:

  ```
  { "type": "LineString", "coordinates": [ [ 40, 5 ], [ 41, 6 ] ] }
  ```

- `Polygon`, as in the following:

  ```
  {
    "type": "Polygon",
    "coordinates": [ [ 40, 5 ], [ 40, 6 ], [ 41, 6 ], [ 41, 5 ], [ 40, 5 ] ]
  }
  ```

To query `s2d` indexes, all current geospatial *query operators* (page 123) with an additional `$intersect` (page 312) operator. Currently, all queries using the `s2d` index must pass the query selector (e.g. `$near` (page 123), `$intersect` (page 312)) a GeoJSON document. With the exception of the GeoJSON requirement, the operation of `$near` (page 123) is the same for `s2d` indexes as `2d` indexes.

`$intersect`

The `$intersect` (page 312) selects all indexed points that intersect with provided geometry. (i.e. `Point`, `LineString`, and `Polygon`.) You must pass `$intersect` (page 312) a document in GeoJSON format.

```
db.collection.find( { $intersect: { "type": "Point", "coordinates": [ 40, 5 ] } } )
```

This query will select all indexed objects that intersect with the `Point` with the coordinates `[ 40, 5 ]`. MongoDB will return documents as intersecting if they have a shared edge.

26.2.4 `mongod` Automatically Continues in Progress Index Builds Following Restart

If your `mongod` instance was building an index when it shutdown or terminated, `mongod` will now continue building the index when the `mongod` restarts. Previously, the index build *had* to finish building before `mongod` shutdown.

To disable this behavior the 2.3 series adds a new run time option, `noIndexBuildRetry` (page 312) (or via,q `--noIndexBuildRetry` on the command line,) for `mongod`. `noIndexBuildRetry` (page 312) prevents `mongod` from continuing rebuilding indexes that did were not finished building when the `mongod` last shut down.

`noIndexBuildRetry`

By default, `mongod` will attempt to rebuild indexes upon start-up *if* `mongod` shuts down or stops in the middle of an index build. When enabled, run time option prevents this behavior.

26.2.5 New Hashed Index and Sharding with a Hashed Shard Key

To support an easy to configure and evenly distributed shard key, version 2.3 adds a new "`hashed`" index type that indexes based on hashed values. This section introduces and documents both the new index type and its use in sharding:

Hashed Index

The new `hashed` index exists primarily to support automatically hashed shard keys. Consider the following properties of hashed indexes:

- Hashed indexes must only have a single field, and cannot be compound indexes.
- Fields indexed with hashed indexes must *not* hold arrays. Hashed indexes cannot be multikey indexes.
- Hashed indexes cannot have a `unique` constraint.

  You *may* create hashed indexes with the `sparse` property.
- MongoDB can use the hashed index to support equality queries, but cannot use these indexes for range queries.
- Hashed indexes offer no performance advantage over normal indexes. *However*, hashed indexes may be smaller than a normal index when the values of the indexed field are larger than 64 bits. [2]
- it's possible to have a hashed and non-hased index on the same field: MongoDB will use the non-hashed for range queries.

> **Warning:** Hashed indexes round floating point numbers to 64-bit integers before hashing. For example, a hashed index would store the same value for a field that held a value of `2.3` and `2.2`. To prevent collisions do not use a hashed index for floating point numbers that cannot be consistently converted to 64-bit integers (and then back to floating point.) Hashed indexes do not support floating point values larger than 2^{53}.

Create a hashed index using an operation that resembles the following:

```
db.records.ensureIndex( { a: "hashed" } )
```

This operation creates a hashed index for the `records` collection on the `a` field.

Hashed Sharding

To shard a collection using a hashed shard key, issue an operation in the `mongo` shell that resembles the following:

```
sh.shardCollection( "records.active", { a: "hashed" } )
```

This operation shards the `active` collection in the `records` database, using a hash of the `a` field as the shard key. Consider the following properties when using a hashed shard key:

- As with other kinds of shard key indexes, if your collection has data, you must create the hashed index before sharding. If your collection does not have data, sharding the collection will create the appropriate index.
- The `mongos` (page 256) will route all equality queries to a specific shard or set of shards; however, the `mongos` (page 256) must route range queries to all shards.

[2] The hash stored in the hashed index is 64 bits long.

- When using a hashed shard key on a new collection, MongoDB automatically pre-splits the range of 64-bit hash values into chunks. By default, the initial number of chunks is equal to twice the number of shards at creation time. You can change the number of chunks created, using the `numInitialChunks` option, as in the following invocation of `shardCollection` (page 13):

  ```
  db.adminCommand( { shardCollection: "test.collection",
                     key: { a: "hashed"},
                     numInitialChunks: 2001 } )
  ```

  MongoDB will only pre-split chunks in a collection when sharding empty collections. MongoDB will not create chunk splits in a collection sharding collections that have data.

> **Warning:** Avoid using hashed shard keys when the hashed field has non-integral floating point values, see *hashed indexes* (page 313) for more information.

Other MongoDB release notes:

CHAPTER

TWENTYSEVEN

DEFAULT WRITE CONCERN CHANGE

These release notes outline a change to all driver interfaces released in November 2012. See release notes for specific drivers for additional information.

27.1 Changes

As of the releases listed below, there are two major changes to all drivers:

1. All drivers will add a new top-level connection class that will increase consistency for all MongoDB client interfaces.

   This change is non-backward breaking: existing connection classes will remain in all drivers for a time, and will continue to operate as expected. However, those previous connection classes are now deprecated as of these releases, and will eventually be removed from the driver interfaces.

   The new top-level connection class is named `MongoClient`, or similar depending on how host languages handle namespacing.

2. The default write concern on the new `MongoClient` class will be to acknowledge all write operations [1]. This will allow your application to receive acknowledgment of all write operations.

   See the documentation of *Write Concern* for more information about write concern in MongoDB.

   Please migrate to the new `MongoClient` class expeditiously.

27.2 Releases

The following driver releases will include the changes outlined in *Changes* (page 315). See each driver's release notes for a full account of each release as well as other related driver-specific changes.

- C#, version 1.7
- Java, version 2.10.0
- Node.js, version 1.2
- Perl, version 0.601.1
- PHP, version 1.4
- Python, version 2.4

[1] The drivers will call `getLastError` (page 50) without arguments, which is logically equivalent to the `w: 1` option; however, this operation allows *replica set* users to override the default write concern with the `getLastErrorDefaults` setting in the `http://docs.mongodb.org/manual/reference/replica-configuration`.

- Ruby, version 1.8

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

N

O

S

T

U

V

W