# JavaScript Interface

## Data Manipulation

### Query and Update Methods

`db.collection.find`

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*. 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" : "bot
{ "_id" : ObjectId("50634dbcbe4617f17bb159d0"), "item" : "pap
```

- 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`**

**Parameters:**

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

**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 subdocument fields:

**Fields:**

- **query** (*document*) –

  Optional. Specifies the selection criteria for the modification. The `query` field employs the same *query selectors* 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* 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`** 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`** creates a new document if the `query` returns no documents. The default is `false`. In version 2.2, the **`findAndModify`** 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 : operator:*greater than <$gt>* 10.
2. The `sort` orders the results of the query in ascending order.
3. The update **`increments`** the value of the `score` field by 1.
4. The command returns the original unmodified document selected for this update.

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

`db.collection.`**`insert`**

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" : "car
  ```

  **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, typ
                        {          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
  { "_id" : ObjectId("50631bc0be4617f17bb159cb"), "item" : "era
  ```

`db.collection.save`

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" : "boo
  ```

  **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**

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* as used in the **db.collection.find()** method.
- **update** (*document*) –

  Specifies the modifications to apply.

  **If** the `update` parameter contains any *update operators* expressions such as the **$set** 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.

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`** 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* as in the following:

  ```
  db.products.update( { item: "book", qty: { $gt: 5 } }, { $set
  ```

  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
  ```

  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
  ```

  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
  ```

- 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 the `options` argument, as in the following:

  ```
  db.products.update( { item: "magazine", qty: { $gt: 5 } }, {
  ```

  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`.

## Cursor Methods

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

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`

**Returns:** A document that describes the process used to return the query.

This method may provide useful insight when attempting to optimize a query. When you call the `explain` on a query, the query system reevaluates the query plan. As a result, these operations only provide a realistic account of *how* MongoDB would perform the query, and *not* how long the query would take.

The method's output includes these fields:

- `cursor`: The value for cursor can be either `BasicCursor` or `BtreeCursor`. The second of these indicates that the given query is using an index.
- `nscanned`: The number of index entries scanned.
- `n`: the number of documents returned by the query. You want the value of `n` to be close to the value of `nscanned`. You want to avoid a "collection scan," which is a scan where every document in the collection is accessed. This is the case when `nscanned` is equal to the number of documents in the collection.
- `millis`: the number of milliseconds require to complete the query. This value is useful for comparing indexing strategies.

**See also:** `$explain` for related functionality and the "Optimization" wiki page for information regarding optimization strategies.

`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`

**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`

**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()`

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

cursor.**count**

**Parameters:**

- **override** (*boolean*) – Override the effects of the **cursor.skip()** and **cursor.limit()** methods on the cursor.

Append the **count()** method on a "**find()**" query to return the number of matching objects for any query.

In normal operation, **cursor.count()** ignores the effects of the **cursor.skip()** and **cursor.limit()**. To consider these effects specify "**count(true)**".

**See also:** **cursor.size()**.

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
}
```

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

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:ref:*tag sets <replica-set-read-preference-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 miss any documents and return no duplicates, even if other operations modify objects while the query runs.

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

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

`cursor.`**`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`**

**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`"

## Data Aggregation Methods

`db.collection.aggregate`

*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* 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`," "*Aggregation Framework*," and "*Aggregation Framework Reference*."

`db.collection.group`

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

**Fields:**

- **key** – Specifies one or more fields to group by.
- **reduce** – Specifies a reduce function that operates over all the iterated objects. Typically these aggregator functions perform some sort of summing or counting. The reduce function takes two arguments: the current document and an aggregation counter object.
- **initial** – The starting value of the aggregation counter object.
- **keyf** – Optional. An optional function that returns a "key object" for use as the grouping key. Use `keyf` instead of `key` to specify a key that is not a single/multiple existing fields. For example, use `keyf` to group by day or week in place of a fixed `key`.
- **cond** – Optional. A statement that must evaluate to true for the `db.collection.group()` to process this document. Simply, this argument specifies a query document (as for `db.collection.find()`). Unless specified, `db.collection.group()` runs the "reduce" function against all documents in the collection.
- **finalize** – Optional. A function that runs each item in the result set before `db.collection.group()` returns the final value. This function can either modify the document by computing and adding an average field, or return compute and return a new document.

**Warning:** `db.collection.group()` does not work in *shard environments*. Use the *aggregation framework* or *map-reduce* in *sharded environments*.

**Note:** The result set of the `db.collection.group()` must fit within a single *BSON* object.
Furthermore, you must ensure that there are fewer then 10,000 unique keys. If you have more than this, use `mapReduce`.

`db.collection.group()` provides a simple aggregation capability similar to the function of `GROUP BY` in SQL statements. Use `db.collection.group()` to return counts and

averages from collections of MongoDB documents. Consider the following example **db.collection.group()** command:

```
db.collection.group(
               {key: { a:true, b:true },
                cond: { active: 1 },
                reduce: function(obj,prev) { prev.csum += obj.c; },
                initial: { csum: 0 }
               });
```

This command in the **mongo** shell groups the documents in the collection where the field `active` equals `1` into sets for all combinations of combinations values of the `a` and `b` fields. Then, within each group, the reduce function adds up each document's c field into the csum field in the aggregation counter document. This is equivalent to the following SQL statement.

```
SELECT a,b,sum(c) csum FROM collection WHERE active=1 GROUP BY a,b
```

**See also:** The "Aggregation" wiki page and "*Aggregation Framework*."

`db.collection.`**mapReduce**

The **db.collection.mapReduce()** provides a wrapper around the **mapReduce** *database command*. Always call the **db.collection.mapReduce()** method on a collection. The following argument list specifies a *document* with 3 required and 8 optional fields:

**Parameters:**

- **map** – A JavaScript function that performs the "map" step of the MapReduce operation. This function references the current input document and calls the `emit(key,value)` method to supply the value argument to the reduce function, grouped by the key argument. Map functions may call `emit()`, once, more than once, or not at all depending on the type of aggregation.
- **reduce** – A JavaScript function that performs the "reduce" step of the MapReduce operation. The reduce function receives a key value and an array of emitted values from the map function, and returns a single value. Because it's possible to invoke the reduce function more than once for the same key, the structure of the object returned by function must be identical to the structure of the emitted function.
- **out** – Specifies the location of the out of the reduce stage of the operation. Specify a string to write the output of the map-reduce job to a collection with that name. The map-reduce operation will replace the content of the specified collection in the current database by default. See below for additional options.
- **query** (*document*) – Optional. A query object, like the query used by the **db.collection.find()** method. Use this to specify which documents should enter the map phase of the aggregation.
- **sort** – Optional. Sorts the input objects using this key. This option is useful for optimizing the job. Common uses include sorting by the emit key so that there are fewer reduces.
- **limit** – Optional. Specifies a maximum number of objects to return from the collection.
- **finalize** – Optional. Specifies an optional "finalize" function to run on a result, following the reduce stage, to modify or control the output of the **db.collection.mapReduce()** operation.
- **scope** – Optional. Place a *document* as the contents of this field, to place fields into the global javascript scope for the execution of the map-reduce command.
- **jsMode** (*Boolean*) –

  Optional. Specifies whether to convert intermediate data into BSON format between the mapping and reducing steps.

If false, map-reduce execution internally converts the values emitted during the map function from JavaScript objects into BSON objects, and so must convert those BSON objects into JavaScript objects when calling the reduce function. When this argument is false, **`db.collection.mapReduce()`** places the *BSON* objects used for intermediate values in temporary, on-disk storage, allowing the map-reduce job to execute over arbitrarily large data sets.

If true, map-reduce execution retains the values emitted by the map function and returned as JavaScript objects, and so does not need to do extra conversion work to call the reduce function. When this argument is true, the map-reduce job can execute faster, but can only work for result sets with less than 500K distinct key arguments to the mapper's emit function.

The `jsMode` option defaults to true.

- **verbose** (*Boolean*) – Optional. The `verbose` option provides statistics on job execution times.

The `out` field of the **`db.collection.mapReduce()`**, provides a number of additional configuration options that you may use to control how MongoDB returns data from the map-reduce job. Consider the following 4 output possibilities.

**Parameters:**

- **replace** – Optional. Specify a collection name (e.g. `{ out: { replace: collectionName } }`) where the output of the map-reduce overwrites the contents of the collection specified (i.e. `collectionName`) if there is any data in that collection. This is the default behavior if you only specify a collection name in the `out` field.
- **merge** – Optional. Specify a collection name (e.g. `{ out: { merge: collectionName } }`) where the map-reduce operation writes output to an existing collection (i.e. `collectionName`,) and only overwrites existing documents in the collection when a new document has the same key as a document that existed before the map-reduce operation began.
- **reduce** – Optional. This operation behaves like the `merge` option above, except that when an existing document has the same key as a new document, `reduce` function from the map reduce job will run on both values and MongoDB will write the result of this function to the new collection. The specification takes the form of `{ out: { reduce: collectionName } }`, where `collectionName` is the name of the results collection.
- **inline** – Optional. Indicate the inline option (i.e. `{ out: { inline: 1 } }`) to perform the map reduce job in memory and return the results at the end of the function. This option is only possible when the entire result set will fit within the *maximum size of a BSON document*. When performing map-reduce jobs on secondary members of replica sets, this is the only available `out` option.
- **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* the output mode writes to a collection, and the output database has sharding enabled, the map-reduce operation will shard the results collection according to the `_id` field.
- **nonAtomic** –

  *New in version 2.1.*

  Optional. Specify output operation as non-atomic such that the output behaves like a normal `multi` **`update()`**. If `true`, the post processing step will not execute inside of a database lock so that partial results will be visible during processing . `nonAtomic` is valid only for `merge` and `reduce` output operations where post-processing may be a long-running operation.

**See also:** *map-reduce*, provides a greater overview of MongoDB's map-reduce functionality.
Also consider "*Aggregation Framework*" for a more flexible approach to data aggregation in MongoDB, and the "Aggregation" wiki page for an over view of aggregation in MongoDB.

# Administrative Functions

## Database Methods

`db.addUser`

**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`

**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* and *--password* to specify authentication credentials.

`db.cloneDatabase`

**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`." The `copydb` database command provides related functionality.

`db.commandHelp`

**Parameters:**

- **command** – Specifies a *database command name*.

**Returns:**

Help text for the specified *database command*. See the *database command reference* for full documentation of these commands.

`db.copyDatabase`

**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`**." The **`clone`** database command provides related functionality.

`db.`**`createCollection`**

**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* 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 : 536870912, max
```

This command creates a collection named log with a maximum size of 5 megabytes (512 kilobytes) 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`**. 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* 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)
     })
```

`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`

**Parameters:**

- **function** (*JavaScript*) – A JavaScript function.
- **arguments** – A list of arguments to pass to the JavaScript function.

Provides the ability to run JavaScript code using the JavaScript engine embedded in the MongoDB instance. In this environment the value of the `db` variable on the server is the name of the current database.

Unless you use **`db.eval()`**, the **`mongo`** shell itself will evaluate all JavaScript entered into **`mongo`** shell itself.

> **Warning:** Do not use **`db.eval()`** for long running operations, as **`db.eval()`** blocks all other operations. Consider using *map-reduce* for similar functionality in these situations.
> The **`db.eval()`** method cannot operate on sharded data. However, you may use **`db.eval()`** with non-sharded collections and databases stored in *sharded cluster*.

`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`** clauses and **`mapReduce`** operations.

`db.getCollection`

**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*.

`db.getCollectionNames`

**Returns:** An array containing all collections in the existing database.

`db.getLastError`

**Returns:** The last error message string.

In many situation MongoDB drivers and users will follow a write operation with this command in order to ensure that the write succeeded. Use “safe mode” for most write operations.

**See also:** "*Replica Set Write Concern*" and "`getLastError`."

`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:** "*Replication Info Reference*" for full documentation of this output.

`db.getSiblingDB`

Used to return another database without modifying the `db` variable in the shell environment.

`db.killOp`

**Parameters:**

- **opid** – Specify an operation ID.

Terminates the specified operation. Use `db.currentOp()` to find operations and their corresponding ids. See *Current Operation Reporting* for full documentation of the output of `db.currentOp()`.

`db.listCommands`

Provides a list of all database commands. See the "*Command Reference*" 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`".

`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*"

`db.printReplicationInfo`

Provides a formatted report of the status of a *replica set* from the perspective of the *primary*

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*" 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*" 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*.

**See also:** **sh.status()**

db.**removeUser**

**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** or related options like **db.repairDatabase()** in the **mongo** shell or *mongod --repair*. Restore from an intact copy of your data.

**Note:** When using *journaling*, there is almost never any need to run **repairDatabase**. 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**, and has the same effect as the run-time option *mongod --repair* option, limited to *only* the current database. See **repairDatabase** for full documentation.

db.**runCommand**

**Parameters:**

- **command** (*string*) – Specifies a *database command* in the form of a *document*.
- **command** – When specifying a *command* as a string, **db.runCommand()** transforms the command into the form `{ command: 1 }`.

Provides a helper to run specified *database commands*. 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**.

**See also:** "*Server Status Reference*" for complete documentation of the output of this function.

db.**setProfilingLevel**

**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:

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** process cleanly and safely.

This operation fails when the current database *is not* the *admin database*.

This command provides a wrapper around the **shutdown**.

db.**stats**

**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**". The `scale` option allows you to configure how the **mongo** shell scales the 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*" 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 database to flush all write operations to the disk and locks the database 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** 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 database server 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.

## Collection Methods

These methods operate on collection objects. Also consider the "*Query and Update Methods*" and "*Cursor Methods*" 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** output of the **collStats** (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** output of the **collStats** (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** (i.e. **db.collection.stats()**) command.

db.collection.**distinct**

**Parameters:**

- **string** (*field*) – A field that exists in a document or documents within the *collection*.

Returns an array that contains a list of the distinct values for the specified field.

> **Note:** The **db.collection.distinct()** method provides a wrapper around the **distinct**. Results must not be larger than the maximum *BSON size*.

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**

Drops or removes the specified index from a collection. The **db.collection.dropIndex()** method provides a wrapper around the **dropIndexes** 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()`** as a method on a collection object.

`db.collection.`**`ensureIndex`**

**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.

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 | Value | 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 [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 ignore 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 "*Expire Data from Collections by Setting TTL*" 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 *Monitor and Control Index Building* for more information.

[1] The default index version depends on the version of **`mongod`** running when creating the index. Before version 2.0, the this value was 0; versions 2.0 and later use version 1.

`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`**

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*. 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 **`$atomic`** isolation operator, effectively isolating the delete operation and blocking all other operations during the delete. To isolate the query, include `$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`** *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`** 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`** 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`** for additional warnings and messages.

**Warning:** The **`db.collection.renameCollection()`** method and **`renameCollection`** 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`** *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`** 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:** "*Collection Validation Data*"

`getShardVersion`

**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**

**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**. The `scale` option allows you to configure how the **mongo** shell scales the 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*" for an overview of the output of this command.

## Sharding Methods

**See also:** The "*Sharding Fundamentals*" page for more information on the sharding technology and using MongoDB's *sharded clusters*.

sh.**addShard**

**Parameters:**

- **host** – Specify the hostname of a new shard server.

Use this to add shard instances to the current *cluster*. 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. A *replica set* can also function as a shard member. In these cases supply **addShard** with the set name, followed by at least one existing member of the set as a seed in a comma separated list, as in the final two examples.

This function provides a wrapper around the administrative command **addShard**.

sh.**enableSharding**

**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**

**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.

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`

**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`

**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`

**Parameters:**

- **collection** (*string*) – Specify the sharded collection containing the chunk to migrate.
- **query** – Specify a query to identify documents in a specific chunk. Typically specify the *shard key* for a document as the query.
- **destination** (*string*) – Specify the name of the shard that you wish to move the designated chunk to.

Moves the chunk containing the documents specified by the `query` to the shard described by `destination`.

This function provides a wrapper around the **`moveChunk`**. In most circumstances, allow the *balancer* to automatically migrate *chunks*, and avoid calling **`sh.moveChunk()`** directly.

**See also:** "**`moveChunk`**" and "*Sharding*" for more information.

`sh.setBalancerState`

**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`**

*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`** 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`**

*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`** instance.

`sh.`**`removeShardTag`**

*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`** instance.

`sh.`**`help`**

**Returns:** a basic help text for all sharding related shell functions

a basic help text for all sharding related shell functions.

### Replica Set Methods

**See also:** *Replication Fundamentals* 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*" for documentation of this output.

`rs.initiate`

**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`

**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:** "*Replica Set Configuration*" and "*Replica Set Administration*".

rs.**add**

Specify one of the following forms:

**Parameters:**

- **host** (*string*) – Either a string or a document. If a string, specifies a host (and optionally port-number) for a new host member for the replica set; MongoDB will add this host with the default configuration. If a document, specifies any attributes about a member of a replica set.
- **arbiterOnly** (*Boolean*) – Optional. If `true`, this host is an arbiter. If the second argument evaluates to `true`, as is the case with some *documents*, then this instance will become an arbiter.

Provides a simple method to add a member to an existing *replica set*. You can specify new hosts in one of two ways:

1. as a "hostname" with an optional port number to use the default configuration as in the *Add a Member to an Existing Replica Set* example.
2. as a configuration *document*, as in the *Add a Member to an Existing Replica Set (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 *Replica Set Configuration* document for full documentation of all replica set configuration options.

rs.**addArb**

**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**

**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**

**Parameters:**

- **seconds** (*init*) – Specify the duration of this operation

- **seconds** (*int*) – 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`

**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.

## 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`

**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*" option when invoking the shell to evaluate a small amount of JavaScript code, or
- specify a file name with "*mongo*". `mongo` will execute the script and then exit. Add the *--shell* option to return to the shell after running the command.

Specify files loaded with the `load()` function in relative terms to the current directory of the `mongo` shell session. Check the current directory using the "`pwd()`" function.

mongo shell session. Check the current directory using the pwd() 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**

**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**

**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**

**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**

**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**

**Parameters:**

- **filename** (*string*) – Specify a filename or path to a local file.

**Returns:**

boolean.

Removes the specified file from the local file system.

**fuzzFile**

**Parameters:**

- **filename** (*string*) – Specify a filename or path to a local file.

- **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.

# Non-User Functions and Methods

## 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** (also **db.resetError()**) command.

This method provides a wrapper around the **getPrevError** command.

db.**resetError**

*Deprecated since version 1.6.*

Resets the error message returned by **db.getPrevError** or **getPrevError**. Provides a wrapper around the **resetError** command.

## 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.

## 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.