

# MongoDB Shell Cheat Sheet

To get started, install the **MongoDB Shell** (mongosh).

## Basic Commands

These basic help commands are available in the MongoDB Shell.

<code>mongosh</code>	Open a connection to your local MongoDB instance. All other commands will be run within this mongosh connection.
<code>db.help()</code>	Show help for database methods.
<code>db.&lt;collection&gt;.help()</code>	Show help on collection methods. The <collection> can be the name of an existing collection or a non-existing collection.
<code>db.users.help()</code>	Shows help on methods related to the users collection.
<code>show dbs</code>	Print a list of all databases on the server.
<code>use &lt;db&gt;</code>	Switch current database to <db>. The mongo shell variable db is set to the current database.
<code>show collections</code>	Print a list of all collections for the current database.
<code>show users</code>	Print a list of users for the current database.
<code>show roles</code>	Print a list of all roles, both user-defined and built-in, for the current database.
<code>show profile</code>	Print the five most recent operations that took 1 millisecond or more on databases with profiling enabled.

`show databases`

Print a list of all existing databases available to the current user.

---

`exit`

Exit the mongosh session.

## Create Operations

Create or insert operations add new documents to a collection. If the collection does not exist, create operations also create the collection.

`db.collection.insertOne()`

Inserts a single document into a collection.

`db.users.insertOne( { name: "Chris"} )`

Add a new document with the name of Chris into the users collection

---

`db.collection.insertMany()`

Inserts multiple documents into a collection.

`db.users.insertMany( { age: "24"}, {age: "38"} )`

Add two new documents with the age of 24 and 38 into the users collection

## Read Operations

Read operations retrieve documents from a collection; i.e. query a collection for documents.

`db.collection.find()`

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

`db.users.find()`

Returns all users.

---

`db.collection.find(<filterobject>)`

Find all documents that match the filter object

`db.users.find({place: "NYC"})`

Returns all users with the place NYC.

---

`db.collection.find({<field>:1,<field>:1})`

Returns all documents that match the query after you explicitly include several fields by setting the <field> to 1 in the projection document.

`db.users.find({status:1,item:1})`

Returns matching documents only from state field, item field and, by default, the `_id` field.

```
db.collection.find(<{<field>:1,<
field>:0, _id:0}>)
```

Returns all documents that match the query and removes the `_id` field from the results by setting it to 0 in the projection.

```
db.users.find({status:1,item:1,_id:0}
)
```

Returns matching documents only from state field and item field. Does not return the `_id` field.

## Update Operations

Update operations modify existing documents in a collection.

```
db.collection.updateOne()
```

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

```
db.users.updateOne({ age: 25 },
{ $set: { age: 32 } })
```

Updates all users from the age of 25 to 32.

```
db.collection.updateMany()
```

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

```
db.users.updateMany({ age: 27 },
{ $inc: { age: 3 } })
```

Updates all users with an age of 27 with an increase of 3.

```
db.collection.replaceOne()
```

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

```
db.users.replaceOne({ name: Kris }, {
name: Chris })
```

Replaces the first user with the name Kris with a document that has the name Chris in its name field.

## Delete Operations

Delete operations remove documents from a collection.

```
db.collection.deleteOne()
```

Removes a single document from a collection.

```
db.users.deleteOne({ age: 37 })
```

Deletes the first user with the age 37.

```
db.collection.deleteMany()
```

Removes all documents that match the filter from a collection.

```
db.users.deleteMany({ age: {$lt:18 } })
```

Deletes all users with the age less than 18..

## Comparison Query Operators

Use the following inside an filter object to make complex queries

### \$eq

```
db.users.find({ system: { $eq: "macOS" } })
```

Matches values that are equal to a specified value.

Finds all users with the operating system macOS.

### \$gt

```
db.users.deleteMany({ age: { $gt: 99} })
```

Matches values that are greater than a specified value.

Deletes all users with an age greater than 99.

### \$gte

```
db.users.updateMany({ age": {$gte:21 },{access: "valid"})
```

Matches values that are greater than or equal to a specified value.

Updates all access to "valid" for all users with an age greater than or equal to 21.

### \$in

```
db.users.find( { place: { $in: [ "NYC", "SF" ] } )
```

Matches any of the values specified in an array.

Find all users with the place field that is either NYC or SF.

### \$lt

```
db.users.deleteMany({ "age": {$lt:18 } })
```

Matches values that are less than a specified value.

Deletes all users with the age less than 18..

### \$lte

```
db.users.updateMany({ age: { $lte: 17 }, {access: "invalid"})
```

Matches values that are less than or equal to a specified value.

Updates all access to "invalid" for all users with an age less than or equal to 17.

### \$ne

```
db.users.find({ "place": {$ne: 'NYC'})
```

Matches all values that are not equal to a specified value.

Find all users with the place field set to anything other than NYC.

## `$nin`

```
db.users.find( { place: { $nin: [ "NYC", "SF" ] } } )
```

Matches none of the values specified in an array.

Find all users with the place field that does not equal NYC or SF.

## Field Update Operators

Use the following inside an update object to make complex updates

### `$inc`

```
db.users.updateOne( { age: 22 }, { $inc: { age: 3 } } )
```

Increments the value of the field by the specified amount.

Adds 3 to the age of the first user with the age of 22.

---

### `$min`

```
db.scores.insertOne( { _id: 1, highScore: 800, lowScore: 200 } )
```

Only updates the field if the specified value is less than the existing field value.

Creates a scores collection and sets the value of highScore to 800 and lowScore to 200.

```
db.scores.updateOne( { _id: 1 }, { $min: { lowScore: 150 } } )
```

`$min` compares 200 (the current value of lowScore) to the specified value of 150. Because 150 is less than 200, `$min` will update lowScore to 150.

### `$max`

```
db.scores.updateOne( { _id: 1 }, { $max: { highScore: 1000 } } )
```

Only updates the field if the specified value is greater than the existing field value.

`$max` compares 800 (the current value of highScore) to the specified value of 1000. Because 1000 is more than 800, `$max` will update highScore to 1000.

---

### `$rename`

```
db.scores.updateOne( { $rename: { 'highScore': 'high' } } )
```

Renames a field.

Renames the field 'highScores' to 'high',

## `$set`

```
db.users.updateOne({ $set: { name:
"valid user" } })
```

Sets the value of a field in a document.

Replaces the value of the name field with the specified value valid user.

---

## `$unset`

```
db.users.updateOne({ $unset: { name:
"" } })
```

Removes the specified field from a document.

Deletes the specified value valid user from the name field.

---

## Read Modifiers

Add any of the following to the end of any read operation

### `cursor.sort()`

```
db.users.find().sort({ name: 1, age:
-1 })
```

Orders the elements of an array during a `$push` operation.

Sorts all users by name in alphabetical order and then if any names are the same sort by age in reverse order

---

### `cursor.limit()`

Specifies the maximum number of documents the cursor will return.

---

### `cursor.skip()`

Controls where MongoDB begins returning results.

---

### `cursor.push()`

```
db.users.updateMany({}, { $push: {
friends: "Chris" } })
```

Appends a specified value to an array.

Add Chris to the friends array for all users

## Aggregation Operations

The Aggregation Framework provides a specific language that can be used to execute a set of aggregation operations (processing & computation) against data held in MongoDB.

`db.collection.aggregate()`

A method that provides access to the aggregation pipeline.

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

Selects documents in the users collection with `acddb.orders.estimatedDocumentCount({})_id` field from the sum of the amount field, and sorts the results by the total field in descending order:

## Aggregation Operations

Aggregation pipelines consist of one or more stages that process documents and can return results for groups of documents.

`count`

Counts the number of documents in a collection or a view.

`distinct`

Displays the distinct values found for a specified key in a collection or a view.

`mapReduce`

Run map-reduce aggregation operations over a collection

## Aggregation Operations

Single Purpose Aggregation Methods aggregate documents from a single collection.

`db.collection.estimatedDocumentCount()`

Returns an approximate count of the documents in a collection or a view.

```
db.users.estimatedDocumentCount({})
```

Retrieves an approximate count of all the documents in the users collection.

### `db.collection.count()`

```
db.users.count({})
```

Returns a count of the number of documents in a collection or a view.

Returns the distinct values for the age field from all documents in the users collection.

---

### `db.collection.distinct()`

```
db.users.distinct("age")
```

Returns an array of documents that have distinct values for the specified field.

Returns the distinct values for the age field from all documents in the users collection.

---

## Indexing Commands

Indexes support the efficient execution of queries in MongoDB. Indexes are special data structures that store a small portion of the data set in an easy-to-traverse form.

### `db.collection.createIndex()`

```
db.users.createIndex("account  
creation date")
```

Builds an index on a collection.

Creates the account creation date index in the users collection.

---

### `db.collection.dropIndex()`

```
db.users.dropIndex("account creation  
date")
```

Removes a specified index on a collection.

Removes the account creation date index from the users collection.

---

### `db.collection.dropIndexes()`

```
db.users.dropIndexes()
```

Removes all indexes but the `_id` (no parameters) or a specified set of indexes on a collection.

Drop all but the `_id` index from a collection.

```
db.users.dropIndex("account creation  
date", "account termination date")
```

Removes the account creation date index and the account termination date index from the users collection.



`db.collection.getIndexes()`

Returns an array of documents that describe the existing indexes on a collection.

`db.users.getIndexes()`

Returns an array of documents that hold index information for the users collection.

---

`db.collection.reIndex()`

Rebuilds all existing indexes on a collection

`db.users.reIndex()`

Drops all indexes on the users collection and recreates them.

---

`db.collection.totalIndexSize()`

Reports the total size used by the indexes on a collection. Provides a wrapper around the `totalIndexSize` field of the `collStats` output.

`db.users.totalIndexSize()`

Returns the total size of all indexes for the users collection.

---

## Replication Commands

Replication refers to the process of ensuring that the same data is available on more than one MongoDB Server.

`rs.add()`

`rs.add( "mongodb4.example.net:27017" )`

Adds a member to a replica set.

Adds a new secondary member, `mongodb4.example.net:27017`, with default vote and priority settings to a new replica set

---

`rs.conf()`

Returns a document that contains the current replica set configuration.

---

`rs.status()`

Returns the replica set status from the point of view of the member where the method is run.

`rs.stepDown()`

Instructs the primary of the replica set to become a secondary. After the primary steps down, eligible secondaries will hold an election for primary.

---

`rs.remove()`

Removes the member described by the hostname parameter from the current replica set.

---

`rs.reconfig()`

Reconfigures an existing replica set, overwriting the existing replica set configuration.

## Sharding Commands

Sharding is a method for distributing or partitioning data across multiple computers. This is done by partitioning the data by key ranges and distributing the data among two or more database instances.

`sh.abortReshardCollection()`

`sh.abortReshardCollection("users")`

Ends a **resharding operation**

Aborts a running reshard operation on the users collection.

---

`sh.addShard()`

`sh.addShard("cluster"/mongodb3.example.net:27327")`

Adds a shard to a sharded cluster.

Adds the cluster replica set and specifies one member of the replica set.

---

`sh.commitReshardCollection()`

`sh.commitReshardCollection("records.users")`

Forces a resharding operation to block writes and complete.

Forces the resharding operation on the records.users to block writes and complete.

### `sh.disableBalancing()`

```
sh.disableBalancing("records.users")
```

Disable balancing on a single collection in a sharded database. Does not affect balancing of other collections in a sharded cluster.

Disables the balancer for the specified sharded collection.

---

### `sh.enableAutoSplit()`

Enables auto-splitting for the sharded cluster.

---

### `sh.disableAutoSplit()`

Disables auto-splitting for the sharded cluster.

---

### `sh.enableSharding()`

```
sh.enablingSharding("records")
```

Creates a database.

Creates the records database.

---

### `sh.help()`

Returns help text for the sh methods.

---

### `sh.moveChunk()`

```
sh.moveChunk("records.users", {  
  zipcode: "10003" }, "shardexample")
```

Migrates a chunk in a sharded cluster.

Finds the chunk that contains the documents with the zipcode field set to 10003 and then moves that chunk to the shard named shardexample.

---

### `sh.reshardCollection()`

```
sh.reshardCollection("records.users",  
{ order_id: 1 })
```

Initiates a resharding operation to change the shard key for a collection, changing the distribution of your data.

Reshards the users collection with the new shard key { order\_id: 1 }

---

### `sh.shardCollection()`

```
sh.shardCollection("records.users",  
{ zipcode: 1 } )
```

Enables sharding for a collection.

Shards the users collection by the zipcode field.

## sh.splitAt()

Divides an existing chunk into two chunks using a specific value of the shard key as the dividing point.

```
sh.splitAt( "records.users", { x: 70 } )
```

Splits a chunk of the records.users collection at the shard key value x: 70

---

## sh.splitFind()

Divides an existing chunk that contains a document matching a query into two approximately equal chunks.

```
sh.splitFind( "records.users", { x:70 } )
```

Splits, at the median point, a chunk that contains the shard key value x: 70.

---

## sh.status()

Reports on the status of a sharded cluster, as [db.printShardingStatus\(\)](#).

---

## sh.waitForPingChange()

Internal. Waits for a change in ping state from one of the [mongos](#) in the sharded cluster.

---

## refineCollectionShardKey

Modifies the collection's shard key by adding new field(s) as a suffix to the existing key.

```
db.adminCommand( { shardCollection: "test.orders", key: { customer_id: 1 } } )
```

Shard the orders collection in the test database. The operation uses the customer\_id field as the initial shard key.

```
db.getSiblingDB("test").orders.createIndex( { customer_id: 1, order_id: 1 } )
```

Create the index to support the new shard key if the index does not already exist.

```
db.adminCommand( { refineCollectionShardKey: "test.orders", key: { customer_id: 1, order_id: 1 } } )
```

Run refineCollectionShardKey command to add the order\_id field as a suffix

## convertShardKeyToHashed()

Returns the hashed value for the input.

```
use test
db.orders.createIndex( { _id:
"hashed" } )
sh.shardCollection( "test.orders", {
_id : "hashed" } )
```

Consider a sharded collection that uses a hashed shard key.

```
{
  _id:
ObjectId("5b2be413c06d924ab26ff9ca"),
  "item" : "Chocolates",
  "qty" : 25
}
```

If the following document exists in the collection, the hashed value of the `_id` field is used to distribute the document:

```
convertShardKeyToHashed(
ObjectId("5b2be413c06d924ab26ff9ca")
)
```

Determine the hashed value of `_id` field used to distribute the document across the shards,

## Database Methods

### `db.runCommand()`

Run a command against the current database

---

### `db.adminCommand()`

Provides a helper to run specified database commands against the admin database.

## User Management Commands

Make updates to users in the MongoDB Shell.

<code>db.auth()</code>	Authenticates a user to a database.
<code>db.changeUserPassword()</code>	Updates a user's password.
<code>db.createUser()</code>	Creates a new user for the database on which the method is run.
<code>db.dropUser()</code> <code>db.dropAllUsers()</code>	Removes user/all users from the current database.
<code>db.getUser()</code> <code>db.getUsers()</code>	Returns information for a specified user/all users in the database.
<code>db.grantRolesToUser()</code>	Grants a role and its privileges to a user.
<code>db.removeUser()</code>	Removes the specified username from the database.
<code>db.revokeRolesFromUser()</code>	Removes one or more roles from a user on the current database.
<code>db.updateUser()</code>	Updates the user's profile on the database on which you run the method.
<code>passwordPrompt()</code>	Prompts for the password in mongosh.

## Role Management Commands

Make updates to roles in the MongoDB Shell.

<code>db.createRole()</code>	Authenticates a user to a database.
<code>db.dropRole()</code> <code>db.dropAllRoles()</code>	Deletes a user-defined role/all user-defined roles associated with a database.
<code>db.getRole()</code> <code>db.getRoles()</code>	Returns information for the specified role/all the user-defined roles in a database.
<code>db.grantPrivilegesToRole()</code>	Assigns privileges to a user-defined role.
<code>db.revokePrivilegesFromRole()</code>	Removes the specified privileges from a user-defined role.
<code>db.grantRolesToRole()</code>	Specifies roles from which a user-defined role inherits privileges.
<code>db.revokeRolesFromRole()</code>	Removes inherited roles from a role.
<code>db.updateRole()</code>	Updates a user-defined role.