DZone Refcardz

CONTENTS INCLUDE:

- Configuration Options
- Using the Shell
- Diagnosing What's Happening
- Quick Rules
- Query Operators
- Update Modifiers...and More!

# MongoDB

## Flexible NoSQL for Humongous Data

By Kristina Chodorow

## ABOUT THIS REFCARD

MongoDB is a document-oriented database that is easy to use from almost any language. This cheat sheet covers a bunch of handy and easily forgotten options, commands, and techniques for getting the most out of MongoDB.

## CONFIGURATION OPTIONS

### Setting Options

Startup options for MongoDB can be set on the command line or in a configuration file. The syntax is slightly different between the two. Here are the three types of options:

| Command-Line | Config File |
|---|---|
| --dbpath /path/to/db | dbpath=/path/to/db |
| --auth | auth=true |
| -vvv | vvv=true |

Run mongod --help for a full list of options, but here are some of the most useful ones:

| Option | Description |
|---|---|
| --config /path/to/config | Specifies config file where other options are set. |
| --dbpath /path/to/data | Path to data directory. |
| --port 27017 | Port for MongoDB to listen on. |
| --logpath /path/to/file.log | Where the log is stored. This is a path to the exact file, not a directory. |
| --logappend | On restart, appends to (does not truncate) the existing log file. Always use this when using the --logpath option. |
| --fork | Forks the mongod as a daemon process. |
| --auth | Enables authentication on a single server. |
| --keyFile /path/to/key.txt | Enables authentication on replica sets and sharding. Takes a path to a shared secret key |
| --nohttpinterface | Turns off HTTP interface. |
| --bind_ip address | Only allows connections from the specified network interface. |

To start mongod securely, use the nohttpinterface and bind_ip options and make sure it isn't accessible to the outside world. In particular, make sure that you do not have the rest option enabled. MongoDB requires the following network accessibility:

- Single server - ability to accept connections from clients.
- Replica set - ability to accept connections from any member of the set, including themselves. Clients must be able to connect to any member that can become primary.
- Sharding - mongos processes must be able to connect to config servers and shards. Shards must be able to connect to each other and config servers. Clients must be able to connect to mongos processes. Config servers do not need to connect to anyone, including each other.

All of the connections are over TCP.

### Seeing Options

If you started mongod with a bunch of options six months ago, how can you see which options you used? The shell has a helper:

```
> db.serverCmdLineOpts()
{ “argv” : [ “./mongod”, “--port”, “30000” ], “parsed” : { },
“ok” : 1 }
```

The parsed field is a list of arguments read from a config file.

## USING THE SHELL

### Shell Help

There are a number of functions that give you a little help if you forget a command:

```
> // basic help
> help
        db.help()                    help on db methods
        db.mycoll.help()             help on collection methods
        sh.help()                    sharding helpers
        rs.help()                    replica set helpers
        help admin                   administrative help
        help connect                 connecting to a db help
        ...
```

Note that there are separate help functions for databases, collections, replica sets, sharding, administration, and more. Although not listed explicitly, there is also help for cursors:

```
> // list common cursor functions
> db.foo.find().help()
```

You can use these functions and helpers as built-in cheat sheets.

### Seeing Function Definitions

If you don't understand what a function is doing, you can run it without the parentheses in the shell to see its source code:

```
> // run the function
> db.serverCmdLineOpts()
{ "argv" : [ "./mongod" ], "parsed" : { }, "ok" : 1 }
> // see its source
> db.serverCmdLineOpts
function () {
   return this._adminCommand("getCmdLineOpts");
}
```

This can be helpful for seeing what arguments it expects or what errors it can throw, as well as how to run it from another language.

### Using edit

The shell has limited multi-line support, so it can be difficult to program in. The shell helper edit makes this easier, opening up a text editor and allowing you to edit variables from there. For example:

```
> x = function() { /* some function we're going to fill in */ }
> edit x
<opens emacs with the contents of x>
```

Modify the variable in your editor, then save and exit. The variable will be set in the shell.

Either the EDITOR environment variable or a MongoDB shell variable EDITOR must be set to use edit. You can set it in the MongoDB shell as follows:

```
> EDITOR="/usr/bin/emacs"
```

Edit is not available from JavaScript scripts, only in the interactive shell.

### .mongorc.js

If a .mongorc.js file exists in your home directory, it will automatically be run on shell startup. Use it to initialize any helper functions you use regularly and remove functions you don't want to accidentally use.

For example, if you would prefer to not have dropDatabase() available by default, you could add the following lines to your .mongorc.js file:

```
DB.prototype.dropDatabase = function() {
    print("No dropping DBs!");
}
db.dropDatabase = DB.prototype.dropDatabase;
```

The example above will change the dropDatabase() helper to only print a message, and not to drop databases.

Note that this technique should not be used for security because a determined user can still drop a database without the helper. However, removing dangerous admin commands can help prevent fat-fingering.

A couple of suggestions for helpers you may want to remove from .mongorc.js are:

- DB.prototype.shutdownServer
- DBCollection.prototype.drop
- DBCollection.prototype.ensureIndex
- DBCollection.prototype.reIndex
- DBCollection.prototype.dropIndexes

### Changing the Prompt

The shell prompt can be customized by setting the prompt variable to a function that returns a string:

```
prompt = function() {
    try {
        db.getLastError();
    }
    catch (e) {
        print(e);
    }
    return (new Date())+"$ ";
}
```

If you set prompt, it will be executed each time the prompt is drawn (thus, the example above would give you the time the last operation completed).

Try to include the db.getLastError() function call in your prompt. This is included in the default prompt and takes care of server reconnection and returning errors from writes.

Also, always put any code that could throw an exception in a try/catch block. It's annoying to have your prompt turn into an exception!

## DIAGNOSING WHAT'S HAPPENING

### Viewing and Killing Operations

You can see current operations with the currentOp function:

```
> db.currentOp()
{
    "inprog" : [
        {
            "opid" : 123,
            "active" : false,
            "locktype" : "write",
            "waitingForLock" : false,
            "secs_running" : 200,
            "op" : "query",
            "ns" : "foo.bar",
            "query" : {
            }
            ...
        },
        ...
    ]
}
```

Using the opid field from above, you can kill operations:

```
> db.killOp(123)
```

Not all operations can be killed or will be killed immediately. In general, operations that are waiting for a lock cannot be killed until they acquire the lock.

The active field indicates whether this operation is currently running. If an operation is not running, it generally has either not started yet and is waiting for a lock or has yielded to other operations. You can see a count of the number of times an operation has yielded in the numYields field.

### Index Usage

Use explain() to see which index MongoDB is using for a query.

```
> db.foo.find(criteria).explain()
{
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 2,
        "nscannedObjects" : 2,
        "nscanned" : 2,
        "nscannedObjectsAllPlans" : 2,
        "nscannedAllPlans" : 2,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 0,
        "indexBounds" : {

        },
        "server" : "ubuntu:27017"
}
```

There are several important fields in the output of explain():

- **n:** the number of results returned.
- **nscanned:** the number of index entries read.
- **nscannedObjects:** the number of docs referred by the index.
- **indexOnly:** if the query never had to touch the collection itself.
- **nYields:** the number of times this query has released the read lock and waited for other operations to go.
- **indexBounds:** when an index is used, this shows the index scan ranges.

## Types of Cursors

A BasicCursor means that no index was used. A BtreeCursor means a normal index was used. Parallel cursors are used by sharding and geospatial indexes use their own special cursors.

Once an array has been indexed, that index has a special "multikey" flag set, which is what the isMultiKey field in the explain above indicates. This flag remains set for the lifetime of the index, even if it is no longer indexing any arrays.

If an index is used for a query, the explain will contain index bounds fields, which describe the portions of the index looked at. For example, if you know that your documents have an age field that's evenly distributed between 0 and 120 and the index bounds go from 3-5, you know that your index only needs to hit a few percent of the index entries to satisfy that query.

## Hinting

Use hint() to force a particular index to be used for a query:

```
> db.foo.find().hint({x:1})
```

The hint must exactly match the key of the index you want to use. You can see the available indexes for a collections by running:

```
> db.foo.getIndexes()
```

As a rule of thumb, you can create an index from the fields in your query. If you have a query and a sort, the best index depends a bit on the query. If the query is for a single value (e.g., {x: y}), the index should be {queryField: 1, sortField: 1}. If the query is a range or set, it may be more efficient to index {sortField: 1, queryField: 1}. If you use this index, MongoDB must scan the whole index to find all results, but it can return them in order without any in-memory sorting.

## System Profiling

You can turn on system profiling to see operations currently happening on a database. There is a performance penalty to profiling, but it can help isolate slow queries.

```
> db.setProfilingLevel(2) // profile all operations
> db.setProfilingLevel(1) // profile operations that take longer
than 100ms
> db.setProfilingLevel(1, 500) // profile operations that take
longer than 500ms
> db.setProfilingLevel(0) // turn off profiling
> db.getProfilingLevel(1) // see current profiling setting
```

Profile entries are stored in a capped collection called system.profile in the database in which profiling was enabled. Profiling can be turned on and off for each database.

## Replica Sets

To find replication lag, connect to a secondary and run this function:

```
> db.printReplicationStatus()
configured oplog size:   2000MB
log length start to end: 23091secs (6.4hrs)
oplog first event time:  Fri Aug 10 2012 04:33:03 GMT+0200 (CEST)
oplog last event time:   Mon Aug 20 2012 10:56:51 GMT+0200 (CEST)
now:                     Mon Aug 20 2012 10:56:51 GMT+0200 (CEST)
```

To see a member's view of the entire set, connect to it and run:

```
> rs.status()
```

This command will show you what it thinks the state and status of the other members are.

Running rs.status() on a secondary will show you who the secondary is syncing from in the (poorly named) syncingTo field.

## Sharding

To see your cluster's metadata (shards, databases, chunks, etc.), run the following function:

```
> db.printShardingStatus()
> db.printShardingStatus(true) // show all chunks
```

You can also connect to the mongos and see data about your shards, databases, collections, or chunks by using "use config" and then querying the relevant collections.

```
> use config
switched to db config
> show collections
chunks
databases
lockpings
locks
mongos
settings
shards
system.indexes
version
```

Always connect to a mongos to get sharding information. Never connect directly to a config server. Never directly write to a config server. Always use sharding commands and helpers.

After maintenance, sometimes mongos processes that were not actually performing the maintenance will not have an updated version of the config. Either bouncing these servers or running the flushRouterConfig command is generally a quick fix to this issue.

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

Often this problem will manifest as setShardVersion failed errors.

Don't worry about setShardVersion errors in the logs, but they should not trickle up to your application (you shouldn't get the errors from a driver unless the mongos it's connecting to cannot reach any config servers).

To add a new shard, run:

```
> db.addShard("rsName/seed1,seed2,seed3")
```

To enable sharing on a database, run:

```
> db.adminCommand({enableSharding: true})
```

To enable sharding on a collection, run:

```
> db.adminCommand({shardCollection: "dbName.collName",
unique: true, key: {fieldName: 1}})
```

dbName.collName should either not exist yet or already have an index on fieldName (the shard key). If you are using a unique shard key, it must be a unique index.

If you are not sharding on _id, _ids are not required to be unique across the cluster. However, they are on individual shards (that is, you could have a doc with _id:123 on shard1 and a doc with _id:123 on shard2, but you could not have two docs with _id:123 on shard1). As documents tend to move from shard to shard, you should make sure that your _ids are unique if you're generating your own. If you're using ObjectIds, you'll be fine.

To turn off the balancer, update the config.settings collection through the mongos:

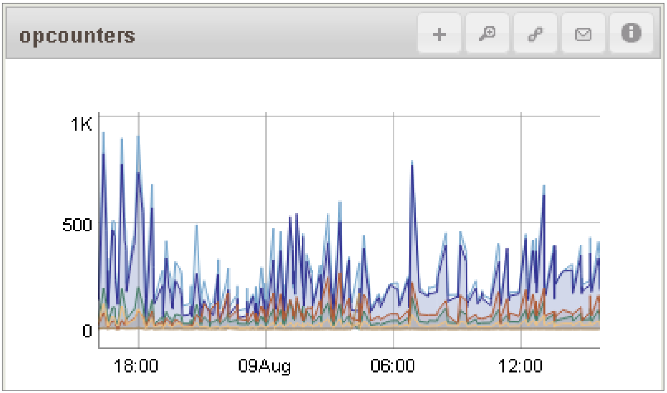
```
> sh.setBalancerState(false)
```

To turn it back on, run the same command passing in true.

## Mongo Monitoring Service (MMS)

MMS is a free, easily-setup way to monitor MongoDB.

To use it, create an account at http://mms.10gen.com.



See http://mms.10gen.com/help for more documentation.

## QUICK RULES

### Databases

Database names cannot contain ".", "$", or "\0" (the null character). Names can only contain characters that can be used on your filesystem as filenames. Admin, config, and local are reserved database names (you can store your own data in them, but you should never drop them).

### Collections

Collection names cannot contain "$" or "\0". Names prefixed with "system." are reserved by MongoDB and cannot be dropped (even if you created the collection). Dots are often used for organization in collection names, but they have no semantic importance. A collection named "che.se" has no more relation to a collection named "che" than one named "cheese" does.

### Field Names

Field names cannot contain "." nor "\0". Fields should only contain "$" when they are database references.

### Index Options

| | |
|---|---|
| background | Builds indexes in the background, while other connections can read and write. |
| unique | Every value for this key must be distinct. |
| sparse | Non-existent values are not indexed. Very handy for indexing unique fields that some documents might not have. |
| expireAfterSeconds | Takes a number of seconds and makes this a "time to live" collection. |
| dropDups | When creating unique indexes, drops duplicate values instead of erroring out. Note that this will delete documents with duplicate values! |

### Query Format

Queries are generally of the form:

```
{key : {$op : value}}
```

For example:

```
{age : {$gte : 18}}
```

There are three exceptions to this rule: $and, $or, and $nor, which are all top-level:

```
{$or : [{age: {$gte : 18}}, {age : {$lt : 18},
parentalConsent:true}}]}
```

### Update Format

Updates are always of the form:

```
{key : {$mod : value}}
```

For example:

```
{age : {$inc : 1}}
```

## QUERY OPERATORS

**√**: Matches
**x**: Does not match

| Operator | Example Query | Example Docs |
|---|---|---|
| $gt, $gte, $lt, $lte, $ne | {numSold : {$lt:3}} | **√** {numSold: 1}<br>**x** {numSold: "hello"}<br>**x** {x : 1} |
| $in, $nin | {age : {$in : [10, 14, 21]}} | **√** {age: 21}<br>**√** {age: [9, 10, 11]}<br>**x** {age: 9} |
| $all | {hand : {$all : ["10","J","Q","K","A"]}} | **√** {hand: ["7", "8", "9", "10", "J", "Q", "K", "A"]}<br>**x** {hand:["J","Q","K"]} |
| $not | {name : {$not : /jon/i}} | **√** {name: "Jon"}<br>**x** {name: "John"} |
| $mod | {age : {$mod : [10, 0]}} | **√** {age: 50}<br>**x** {age: 42} |
| $exists | {phone: {$exists: true}} | **√** {phone: "555-555-5555"}<br>**x** {phones: ["555-555-5555", "1-800-555-5555"]} |
| $type* | {age : {$type : 2}} | **√** {age : "42"}<br>**x** {age : 42} |
| $size | {"top-three":{$size:3}} | **√** {"top-three":["gold","silver","bronze"]}<br>**x** {"top-three":["blue ribbon"]} |

*See http://www.mongodb.org/display/DOCS/Advanced+Queries for a full list of types.

## UPDATE MODIFIERS

| Modifier | Start Doc | Example Mod | End Doc |
|---|---|---|---|
| $set | {x:"foo"} | {$set:{x:[1,2,3]}} | {x:[1,2,3]} |
| $unset | {x:"foo"} | {$unset:{x:true}} | {} |
| $inc | {countdown:5} | {$inc:{countdown: -1}} | {countdown:4} |
| $push, $pushAll | {votes:[-1,-1,1]} | {$push:{votes:-1 }} | {votes:[-1,-1,1, -1]} |
| $pull, $pullAll | {blacklist:["ip1", "ip2","ip3"]} | {$pull:{blacklist: "ip2"}} | {blacklist:"ip1", "ip3"} {blacklist: "ip1","ip3"} |
| $pop | {queue:["1pm", "3pm","8pm"]} | {$pop:{queue:--1}} | {queue:["3pm", "8pm"]} |
| $addToSet, $each | {ints:[0,1,3,4 ]} | {$addToSet:{ints:{ $each:[1,2,3]}}} | {ints:[0,1,2,3,4]} |
| $rename | {nmae:"sam"} | {$rename:{nmae: "name"}} | {name:"sam"} |
| $bit | {permission:6} | {$bit:{permissions :{or:1}}} | {permission:7} |

## AGGREGATION PIPELINE OPERATORS

The aggregation framework can be used to perform everything from simple queries to complex aggregations.

To use the aggregation framework, pass the aggregate() function a pipeline of aggregation operations:

```
> db.collection.aggregate({$match:{x:1}},
... {$limit:10},
... {$group:{_id : “$age”}})
```

A list of available operators:

| Operator | Description |
|---|---|
| {$project : projection} | Includes, exclude,s renames, and munges fields. |
| {$match : match} | Queries, takes an argument identical to that passed to find(). |
| {$limit : num} | Limits results to num. |
| {$skip : skip} | Skips num results. |
| {$sort : sort} | Sorts results by the given fields. |
| {$group : group} | Groups results using the expressions given (see table below). |
| {$unwind : field} | Explodes an embedded array into its own top-level documents. |

To refer to a field, use the syntax $fieldName. For example, this projection would return the existing "time" field with a new name, "time since epoch":

```
{$project: {“time since epoch”: “$time”}}
```

$project and $group can both take expressions, which can use this $fieldName syntax as shown below:

| Expression Op Example | Description |
|---|---|
| $add : ["$age", 1] | Adds 1 to the age field. |
| $divide : ["$sum", "$count"] | Divides the sum field by count. |
| $mod : ["$sum", "$count"] | The remainder of dividing sum by count. |
| $multiply : ["$mph", 24, 365] | Multiplies mph by 24*365. |
| $subtract : ["$price", "$discount"] | Subtracts discount from price. |
| $strcasecmp : ["ZZ", "$name"] | 1 if name is less than ZZ, 0 if name is ZZ, -1 if name is greater than ZZ. |
| $substr : ["$phone", 0, 3] | Gets the area code (first three characters) of phone. |
| $toLower : "$str" | Converts str to all lowercase. |
| $toUpper : "$str" | Converts str to all uppercase. |
| $ifNull : ["$mightExist", $add : ["$doesExist", 1]] | If mightExist is not null, returns mightExist. Otherwise returns the result of the second expression. |
| $cond : [exp1, exp2, exp3] | If exp1 evalutes to true, return exp2, otherwise return expr3. |

## MAKING BACKUPS

The best way to make a backup is to make a copy of the database files while they are in a consistent state (i.e., not in the middle of being read from/to).

1. Use the fsync+lock command. This flushes all in-flight writes to disk and prevents new ones.
   ```
   >  db.fsyncLock()
   ```
2. Copy data files to a new location.
3. Use the unlock command to unlock the database.
   ```
   > db.fsyncUnlock()
   ```

To restore from this backup, copy the files to the correct server’s dbpath and start the mongod.

If you have a filesystem that does filesystem snapshots and your journal is on the same volume and you haven’t done anything stripy with RAID, you can take a snapshot without locking. In this case, when you restore, the journal will replay operations to make the data files consistent.

Mongodump is only for backup in special cases. If you decide to use it anyway, don’t fsync+lock first.

## REPLICA SET MAINTENANCE

### Keeping Members from Being Elected

To permanently stop a member from being elected, change its priority to 0:

```
> var config = rs.config()
> config.members[2].priority = 0
> rs.reconfig(config)
```

To prevent a secondary from being elected temporarily, connect to it and issue the freeze command:

```
> rs.freeze(10*60) // # of seconds to not become primary
```

This can be handy if you don’t want to change priorities permanently but need to do maintenance on the primary.

### Demoting a Member

If a member is currently primary and you don’t want it to be, use stepDown:

```
> rs.stepDown(10*60) // # of seconds to not try to become primary
again
```

## Starting a Member as a Stand-Alone Server

For maintenance, often it is desirable to start up a secondary and be able to do writes on it (e.g., for building indexes). To accomplish this, you can start up a secondary as a stand-alone mongod temporarily.

If the secondary was originally started with the following arguments:

```
$ mongod --dbpath /data/db --replSet setName --port 30000
```

Shut it down cleanly and restart it with:

```
$ mongod --dbpath /data/db --port 30001
```

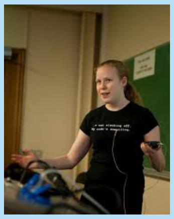
Note that the dbpath does not change, but the port does and the replSet option is removed (all other options can remain the same). This mongod will come up as a stand-alone server. The rest of the replica set will be looking for a member on port 30000, not 30001, so it will just appear to be "down" to the rest of the set.

When you are finished with maintenance, restart the server with the original arguments.

## MORE RESOURCES

- Download MongoDB at http://www.mongodb.org/downloads
- Documentation is available at http://docs.mongodb.org
- See the roadmap and file bugs and request features at http://jira.mongodb.org
- Ask questions on the mailing list: http://groups.google.com/group/mongodb-user
- Or, for a faster response with more variable quality, try the IRC chat: irc.freenode.net/#mongodb

## ABOUT THE AUTHORS



Kristina Chodorow works on MongoDB, with a focus on replication internals. She has written several books on MongoDB and given talks at conferences around the world. She lives in New York City and enjoys programming, writing, and cartooning. She blogs at http://www.kchodorow.com.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

**Refcardz Feedback Welcome**

refcardz@dzone.com

**Sponsorship Opportunities**

sales@dzone.com

ISBN-13: 978-1-936502-69-1
ISBN-10: 1-936502-69-0

50795

9 781936 502691

$7.95

Version 1.0