

MongoDB Cheat Sheet v2.11

Test

```
$ mongosh
test> use bunny
bunny> db.rabbit.insertOne({ name:"Bugs" })
bunny> db.rabbit.find({})
```

(Make sure you get the Bugs document back.)

Meta

```
test> show dbs
test> use bunny
bunny> show collections
```

Be *careful* with these!

```
test> db.dropDatabase(dbname)
test> db.collName.drop()
```

Importing data from a JSON file

```
$ mongoimport --db nameOfDatabase --collection nameOfCollection --file pathToFile.json
```

If it's a list, add the `--jsonArray` flag at the end.

Common queries

Query based on field value

```
db.awards.find({ year: 1999 })
db.awards.find({ year: { $lt: 1990 } })
db.awards.find({ year: { $in: [ 1992, 1997 ] } })
db.awards.find({ year: { $not: { $in: [ 1992, 1997 ] } } })
```

In addition to `$lt`, also available are `$lte`, `$gt`, `$gte`, and `$ne`, all with guessable meanings.

Query based on existence of field

```
db.heroes.find({ sidekick: { $exists: true } })
```

Retrieve only certain fields

Pass a *second* argument to `.find()`, which is a dict of boolean values (or 0's/1's). You can pass either 0's, indicating the fields you *don't* want, or 1's, indicating those you do.

```
db.heroes.find({ gender: "male" }, { name:true, planet:true })  <-- or 1 instead of true
```

(To get rid of `_id`, add `_id:false`.)

Retrieve only unique values

```
db.starwars.distinct("lightsaber color")
```

Size of collection or query result

```
db.starwars.countDocuments()  
db.starwars.countDocuments(...any query...)
```

Multiple conditions

```
db.awards.find({ year: 1994, award: "Cy Young" })           <- AND  
db.awards.find({ $or: [ { year: 1990 }, { year:1984 } ] })  <- OR  
db.awards.find({ $and: [ { year: 1990 }, { award: "MVP" } ] }) <- AND alternative
```

Queries in nested elements

In nested dictionary

```
db.starwars.find({ "relations.Leia": "sister" })
```

In this case, we're finding documents with a `relations` field which is a dictionary and which has a `"Leia":"sister"` key-value pair. (You must have the quotes when using this dot syntax.)

In nested list

```
db.starwars.find({ episodes: "VI" })
```

Somewhat counterintuitively, but happily, the above query matches any document whose `episodes` list has a `"VI"` element in it somewhere. (If you want to match the entire list, in order, specify a list instead of a string as the value.)

In nested list of dictionaries

```
db.courses.find({ "roster.first": "Brian" })
```

This finds documents with a `roster` field which is a list of documents, at least one of which contains a `first` field whose value is `Brian`.

Based on (exact) size of nested list

```
db.courses.find({ "roster": { $size: 20 } })
```

This finds all courses with exactly 20 students in them. To query on a range, you'll have to create a "counter" field that you `$inc` every time elements are added/removed.

Regular expressions

```
db.awards.find({ name: /regex/ })
```

Limit and skip (like SQL's LIMIT and OFFSET)

```
db.awards.find(...any query...).limit(10).skip(25)
```

This will retrieve documents 26 through 36, according to whatever order MongoDB has the documents in internally (no guarantee this will be associated with any attribute, the order in which they were inserted, or anything else).

Aggregate (like SQL’s “GROUP BY”)¹

Count the number of characters of each distinct race:

```
db.starwars.aggregate([{$group: {_id:'$race', count:{$sum:1}}]})
```

Get a total cartons ordered for each base flavor:

```
db.bj.aggregate([{$group: {_id:'$baseFlavor', totcarts:{$sum:"$cartonsordered"}}]})
```

Also available are the aggregation operators `$min`, `$max`, and `$avg`.

It’s also common to first “match” (select) only certain documents, and then do the aggregation on that subset:

```
db.bj.aggregate([{$match: {"releaseDate":{$gt:2015}}, {$group: {_id:'$baseFlavor', totcarts:{$sum:1}}]})
```

This technique is called an “aggregation pipeline” and has many other options. The reason the syntax calls for a *list* inside the `aggregate()` call is that it contains the list of stages in the pipeline, to be executed in order.

Inserting/updating/deleting documents

```
db.stadiums.insertOne({ name:"Wrigley Field", loc:"Chicago"})
db.stadiums.insertMany([ { name:"Wrigley Field", loc:"Chicago"},
  { name:"Comiskey Park", loc:"Chicago"} ])
db.stadiums.deleteOne({loc:"Chicago"})
db.stadiums.deleteMany({loc:"Chicago"})
```

For updates, the operators `$set`, `$inc`, `$unset`, `$push`, and `$pull` are useful.

```
db.stadiums.updateOne({loc:"Chicago"}, {$set: {year:1908}})
db.stadiums.updateMany({loc:"Chicago"}, {$set: {year:1908}})
db.stadiums.updateMany({loc:"Chicago"}, {$inc: {year:1}})
db.stadiums.updateMany({loc:"Chicago"}, {$unset: {year:""}})
db.stadiums.updateMany({loc:"Chicago"}, {$push: {sections:"305"}})
db.stadiums.updateMany({loc:"Chicago"}, {$pull: {sections:"415"}})
```

You can set (or inc, or unset, or push, or pull) multiple fields at once by including additional key/value pairs in the dictionary for the `$set` (or `$inc`, or `$unset`, etc.) key:

```
db.sw.updateOne({name:'Obi-wan'},
  {$set: {'current residence':'Tatooine', 'on medicare':true},
  $inc: {'age':1, 'credits':-400}})
```

Python support

Big gotcha warning

Don’t forget that when you are accessing Mongo via *Python*, rather than the Mongo CLI, you must:

1. Put strings in quotes (like `$set`, or even `height` where “height” is the name of a field)
2. Use underscores instead of camelCase for things like `insert_many()` and `count_documents()`

¹Seems inconsistent to me that in these examples, things like `'$race'`, `'$baseflavor'`, and `"$cartonsordered"` must not only be in quotes, but also be preceded by a cash sign (`$`), even though they are obviously not Mongo operators like `$sum` or `$group`. Oh well.)

3. Call `.next()` on single-item responses to actually get the document
4. Iterate through (with a `for` loop, say) multi-item responses

Connecting

Install `pymongo` package (with `pip`), then:

```
from pymongo import MongoClient, ASCENDING, DESCENDING
mongo_client = MongoClient("mongodb://localhost:27017")

db = mongo_client.db_name          (or mongo_client['db_name'] if db_name is not a legal Python name)
collections = db.list_collection_names()
```

Example usage

```
db.collname.count_documents({...any query...})
result_set = db.collname.find({...any query...})
```

Note: the return value from `.find()` is a *cursor* object. You can deal with it in any of the following ways:

- If you only retrieved a single document, you can call `.next()` on the cursor object to get the document itself.
- If you retrieved multiple documents, you can wrap the cursor in a `list()` to fetch all the result documents into a list.
- If you retrieved multiple documents, you can use the cursor as the argument of a `for` loop, like so:

```
for result in result_set:
    ...do something with result['field2'], result['field3'] etc...
```

Other stuff You can use `skip/limit` by passing those values to `.find()`:

```
result_set = db.collname.find({...any query...}, skip=?, limit=?)
```

You can sort by multiple fields with:

```
result_set = result_set.sort([("field1",ASCENDING), ("field2",DESCENDING)])
```

(You'll need to import `ASCENDING` and/or `DESCENDING` from the `pymongo` package.)