codecentric

MongoDB - Cheat Sheet

Version 1.0 / 27th December 2012 / Page 1 of 4

More information can be found from the MongoDB Tutorial
@ http://blog.codecentric.de/en/2012/12/mongodb-tutorial/

About this Cheat Sheet

The idea behind this is to have all (well, most) information from the above mentioned Tutorial immediately available in a very compact format. All commands can be used on a small data basis created in the insert-section. All information in this sheet comes **without the slightest warranty for correctness**. Use at your own risk. Have fun ☺!

Basic Information

Download MongoDB	http://www.mongodb.org/downloads
JSON Specification	http://www.json.org/
BSON Specification	http://bsonspec.org/
Java Tutorial	http://www.mongodb.org/display/DOCS/Java+Tutorial

Inserting Documents

```
db.ships.insert({name:'USS Enterprise-D',operator:'Starfleet',type:'Explorer',class:'Galaxy',crew:750,codes:[10,11,12]})
db.ships.insert({name:'USS Prometheus',operator:'Starfleet',class:'Prometheus',crew:4,codes:[1,14,17]})
db.ships.insert({name:'USS Defiant',operator:'Starfleet',class:'Defiant',crew:50,codes:[10,17,19]})
db.ships.insert({name:'IKS Buruk',operator:' Klingon Empire',class:'Warship',crew:40,codes:[100,110,120]})
db.ships.insert({name:'IKS Somraw',operator:' Klingon Empire',class:'Raptor',crew:50,codes:[101,111,120]})
db.ships.insert({name:'Scimitar',operator:'Romulan Star Empire',type:'Warbird',class:'Warbird',crew:25,codes:[201,211,220]})
db.ships.insert({name:'Narada',operator:'Romulan Star Empire',type:'Warbird',class:'Warbird',crew:65,codes:[251,251,220]})
```

Finding Documents

`db.ships.findOne()`	Finds one arbitrary document
`db.ships.find().prettyPrint()`	Finds all documents and using nice formatting
`db.ships.find({}, {name:true, _id:false})`	Shows only the names of the ships
`db.ships.findOne({'name':'USS Defiant'})`	Finds one document by attribute

Basic Concepts & Shell Commands

`db.ships.<command>`	`db` – implicit handle to the used database `ships` – name of the used collection
`use <database>`	Switch to another database
`show collections`	Lists the available collections
`help`	Prints available commands and help

Finding Documents using Operators

`$gt / $gte`	greater than / greater than equals	`db.ships.find({class:{$gt:’P'}`
`$lt / $lte`	lesser than / lesser than equals	`db.ships.find({class:{$lte:’P'}`
`$exists`	does an attribute exist or not	`db.ships.find({type:{$exists:true}})`
`$regex`	Perl-style pattern matching	`db.ships.find({name:{$regex:’^USS\\sE’}})`
`$type`	search by type of an element	`db.ships.find({name : {$type:2}})`

BSON Types

String	2
Array	4
Binary Data	5
Date	9

http://www.w3resource.com/mongodb/mongodb-type-operators.php

Updating Documents

`db.ships.update({name : 'USS Prometheus'}, {name : 'USS Something'})`	Replaces the whole document

`db.ships.update({name : 'USS Something'},`
`{$set : {operator : 'Starfleet', class : 'Prometheus'}})`
	sets / changes certain attributes of a given document

`db.ships.update({name : 'USS Something'},`
`{$unset : {operator : 1}})`
	removes an attribute from a given document

Removing Documents

`db.ships.remove({name : 'USS Prometheus'})`	removes the document
`db.ships.remove({name:{$regex:’^USS\\sE’}})`	removes using operator

Each individual document removal is atomic with respect to a concurrent reader or writer. No client will see a document half removed.

G+ Community Page:
https://plus.google.com/u/0/communities/115421122548465808444

Working with Indexes

Creating an index	`db.ships.ensureIndex({name : 1})`
Dropping an index	`db.ships.dropIndex({name : 1})`
Creating a compound index	`db.ships.ensureIndex({name : 1, operator : 1, class : 0})`
Dropping a compound index	`db.ships.dropIndex({name : 1, operator : 1, class : 0})`
Creating a unique compound index	`db.ships.ensureIndex({name : 1, operator : 1, class : 0}, {unique : true})`

Indexes – Hints & Stats

`db.ships.find ({'name':'USS Defiant'}).explain()`	Explains index usage
`db.ships.stats()`	Index statistics
`db.ships.totalIndexSize()`	Index size

Top & Stats System Commands

`./mongotop`	Shows time spent per operations per collection
`./mongostat`	Shows snapshot on the *MongoDB* system

Σ AGGREGATION FRAMEWORK

Pipeline Stages

$project	Change the set of documents by modifying keys and values. This is a 1:1 mapping.
$match	This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage. This can be used for example if aggregation should only happen on a subset of the data.
$group	This does the actual aggregation and as we are grouping by one or more keys this can have a reducing effect on the amount of documents.
$sort	Sorting the documents one way or the other for the next stage. It should be noted that this might use a lot of memory. Thus if possible one should always try to reduce the amount of documents first.
$skip	With this it is possible to skip forward in the list of documents for a given amount of documents. This allows for example starting only from the 10th document. Typically this will be used together with “$sort” and especially together with “$limit”.
$limit	This limits the amount of documents to look at by the given number starting from the current position.
$unwind	This is used to unwind document that are using arrays. When using an array the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.

Comparison with SQL

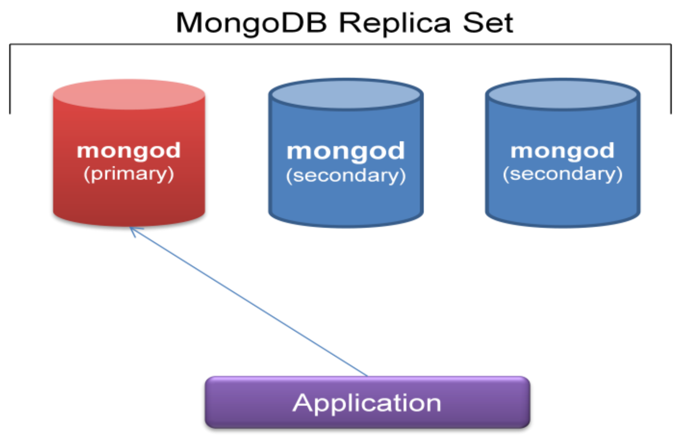
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM	$sum
COUNT	$sum
JOIN	$unwind

Aggregation Examples

`db.ships.aggregate([{$group : {_id : "$operator", num_ships : {$sum : 1}}}])`	Counts the number of ships per operator, would be in SQL: `SELECT operator, count(*) FROM ships GROUP BY operator;`
`db.ships.aggregate([{$project : {_id : 0, operator : {$toLower : "$operator"}, crew : {"$multiply" : ["$crew",10]}}}])`	Combination of $project-stage and $group-stage.

Aggregation Expressions

$sum	Summing up values	`db.ships.aggregate([{$group : {_id : "$operator", num_ships : {$sum : "$crew"}}}])`
$avg	Calculating average values	`db.ships.aggregate([{$group : {_id : "$operator", num_ships : {$avg : "$crew"}}}])`
$min / $max	Finding min/max values	`db.ships.aggregate([{$group : {_id : "$operator", num_ships : {$min : "$crew"}}}])`
$push	Pushing values to a result array	`db.ships.aggregate([{$group : {_id : "$operator", classes : {$push: "$class"}}}])`
$addToSet	Pushing values to a result array without duplicates	`db.ships.aggregate([{$group : {_id : "$operator", classes : {$addToSet : "$class"}}}])`
$first / $last	Getting the first / last document	`db.ships.aggregate([{$group : {_id : "$operator", last_class : {$last : "$class"}}}])`

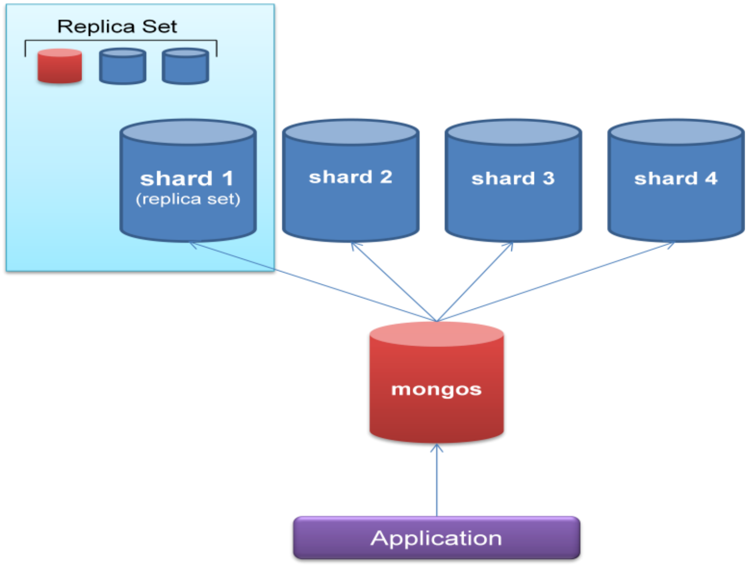


Replica Sets

Type	Allowed to vote?	Can become Primary?	Description
Regular	Yes	Yes	This is the most typical kind of node. It can act as a primary or secondary node
Arbiter	Yes	No	Arbiter nodes are only there for voting purposes. They can be used to ensure that there is a certain amount of nodes in a replica set even though there are not that many physical servers.
Delayed	Yes	No	Often used as a disaster recovery node. The data stored here is usually a few hours behind the real working data.
Hidden	No	No	Often used for analytics in the replica set.

Sharding

- Every document has to define a shard-key.
- The value of the shard-key is immutable.
- The shard-key must be part of an index and it must be the first field in that index.
- There can be no unique index unless the shard-key is part of it and is then the first field.
- Reads done without specifying the shard-key will lead to requests to all the different shards.
- The shard-key must offer sufficient cardinality to be able to utilize all shards.



Durability of Writes

- **w** – This tells the driver to wait for the write to be acknowledged. It also ensures no indexes are violated. Nevertheless the data can still be lost as it is not necessarily already persisted to disc.
- j – This stands for journal-mode. It tells the driver to wait until the journal has been committed to disk. Once this has happened it is quite sure that the write will be persistent unless there are any disc-failures.

w=0 j=0	This is "fire and forget".
w=1 j=0	Waits for an acknowledgement that the write was received and no indexes have been violated. Data can still be lost.
w=1 j=1	The most save configuration by waiting for the write to the journal to be completed.
w=0 j=1	Basically the same as above.

In the context of replica sets the value for the w-parameter now means the amount of nodes that have acknowledged a write. There is a useful short notation to ensure write was done to a majority of nodes by using w='majority'. For the journal-parameter the value of one is still the best that can be done. It means the data is written to the journal of the primary node.