Mongo: some travel note

Notes taken by Dario Varotto about MongoDB while doing mongo101P, mongo102 and mongo101JS courses by MongoDB university.

Mongo shell command:

show dbs
show collections
use <dbname>
- it's a Javascript interpreter -

commands for collection starts with **db.<collection>.**

```
find({ document specifying the search parameters}, { fields to return: true/false} )
```

_id field defaults to true and should be excluded explicitly

```
findOne( _ the syntax is the same of find _)
```

document specifying the search have all fields in logical AND,

Operators expressed as subdocuments:

to specify greater than/less, search for a subdocument with field $gt, $lt, $gte, $lte

```
{ vote:{$gte:18, $lte:30} }
```

$ne:<value> Stand for not equal (we have also $nin “not in” who need an array)
$exists: true/false
$type: <bson type (string 2)>
$regex
regex can also written with Javascript syntax:
{<fieldname>: /^regular expression$/i}
this kind of search are slower cause they can't use index except when they match the start of the string like "^pippo"

$or, $and (take a list of search terms)
$and usually is not needed, cause it's the default behaviour

Array queries

matching is polymorphic over arrays and non arrays type values:
if I check a field for a string type, if it's an array then the search search for that string in the array (on the first level, without recursion)

$all:<list> (like AND, the field (that should be an array) should contain all the elements on the list.
$in:<list> (like OR, field should be or contain at least a value on the list)

Subdocument matching:

dot notation {"email.work":"name@domain.com"}

$elemMatch (in the query) specify that I want that conditions are needed in the same subdocument

```
db.collection.find({array:{$elemMatch:{value1:1,value2:{$gt:1}}}});
```

otherwise the match could be in different subdocuments

Cursor methods

TIP: In the mongo console, to assign a cursor to a variable without it being printed do:
var cur = db.collection.find(); null;

.limit
.sort
.skip
.count
.toArray (return all the cursor elements as a list)
.each (execute the code, looping each record)

Update

```
update(<filter>, <newDoc to replace>)
```

the _id of the document doesn't change but it will remove all existing fields
this is also called *wholesale update*
To change only some part of the document I can use these operators:
(I can't use them with the previous field-name notation, that make the wholesale update)

$set:<doc with fields to update>
$inc:{fields:<inc value>} (this create the field if it doesn't exists)
$dec: "
$unset: <doc with fields to unset if to true>

Array updates:

$set:{field.3: value}	set the 4th field of the array (this is zero based)
$push:{field: value}	insert the value at the end of the array field
$pop:{field:1}	remove the last element from the array field
$pop:{field:-1}	remove the first element from the array field
$pushAll:{field: [list]}	insert all the values on the array field, at the end
$pull:{field:n}	remove all the "n" value from the array, whatever their position is
$pullAll({field:[list]}	remove all the values in the array
$addToSet{field:value}	add to the array, if it's not already there

update takes an optional third parameter: options
{upsert:true, multi:true}
upsert means we have to create the doc if it's not found by the filters (the filter, and the fields to set, but not $gt or similar)
multi states we have to change more than one record, instead of the first one

findAndModify act like update, changing only one document, but it return the changed document. We could choose to return the document before of after with the option {new:true}
the default is False (return the old changed record)

```
findAndModify(query, sort, operator, options)
```

remove(<document condition>) remove the records with many individual operations (like other operations, this is atomic only at document level), leaves the indexes.
drop() erase completely the collection

To get a report on the errors of the last statement:

```
db.runcommand( {getLastError:1} )
```

gives many useful information as a document, note "err":null if the last operation went fine

PYMONGO
sort is a little different: doesn't want a dictionary (which won't be sorted), wants a list of tuples:

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

INDEXES & PERFORMANCE

I can create an index, on an array for example, to be able to search it without doing a full scan.

```
db.students.ensureIndex({teachers:1})
```

where teacher is the field I wont to index. We said ***MultiKey*** when the indexed field is an array.
A query can use max one index!
We specify 1 or -1, depending if we wand an ascending or descending index (it doesn't affect the find).
An index can be used for the sorting, even if the order is "wrong", as **reversed,** but only if it's not a compound index (an index spanned on many fields).
I can pass a second parameter to ensureIndex, {**unique:true**} to force a constrain on unicity (_id:1 is always there as unique index).
I can drop duplicates, with a second argument:{**dropDups:true**} who keep only a document for any key, and drop the others.
I can define an index as "***sparse***" with {sparse:1} so if the key is not in the document, the document won't be indexed (it will be ignored by dropDups too), but PAY ATTENTION: if I search by a sparse key, the record without it won't be returned.
per far si che se la chiave non è presente nel documento, il documento non viene indicizzato, così da
Ensureindex with option {background:true}, create the index without locking writers (on the same DB), it's slower but happen to be useful.

If we have a **compound index**, an index using more fields (using more space to achieve better coverage), it cat be used for queries using that same fields, and also for queries using a prefix of that fields: for example, an index {name: 1, surname:1} can serve find({name: "a"}) but can't serve find({surname: "b"}).

A query like this:

```
db.sensor_readings.find({
  tstamp: {
    $gte: ISODate("2012-08-01"),
    $lte: ISODate("2012-09-01")
  },
  active: true
}).limit(3)
```

To optimize it's read time, I should use an index {active:1, tstamp:1}, so I can find active records and then search for timestamps... the index {tstamp:1, active:1} is not as much effective (even if it's useful) cause the "greater/lesser than" query is more vague.

Index commands:

db.system.indexes.find()	All indexes of current db
db.<collection>.getIndexes()	All indexes of a collection
db.<collection>.dropIndex({"nome":1})	drop an index

We can't create a compound index, if more than one key is an array (we can't have more than one multikey on the same index).

.explain() at the end of a query gets info about how that query was run (so, what index are used, how many records have been scanned and so on)

db.<collection>.stats() show some stats about the collection, like its size

in pymongo, hint have the same sintax of sort

Positional queries

Geospatial:
on a array field, with x, y coordinates: {location:[x,y]}
I can define the index with type "2d" instead of 1 or -1
and then I can use the $near operator (for a 2D model) to find the distance from another position [x,y]

es:
Suppose you have a 2D geospatial index defined on the key location in the collection places. Write a query that will find the closest three places (the closest three documents) to the location 74, 140.
db.places.find({location:{$near:[74,140]}}).limit(3)

We also have a spherical model, always with a "2d" index, but changing the query command:
geoNear:
example:

```
      db.commands({geoNear:'stores', near:[50,50], spherical:true,
maxDistance:1})
```

with maxDistance expressed in radiants (we need the earth radius to convert it to surface meters)
$within and other commands to work with zones and distances (look at the documentation)

Profiling slow queries

We have three levels of profiling queries: levels (0=off, 1=slow 10sec, 2=all)
we can set it when we start mongod like this: --profile 1 –slowms 2
or later, using
db.getProfilingStatus()
db.setProfilingLevel(<level>,<slowms>)

I can then use **db.system.profile** to look for all profiled queryes.

mongotop and **mongostat** show some stats about the use of mongo and I/O performances
For better monitoring and profile however, you should checkout the great MMS (Mongo monitoring service). Monitoring is free while there is a backup solutions which is billed monthly.

Aggregation framework

Aggregation framework (since mongo 2.2) provide convenient methods to perform queries like SQL order by, group, having

Example, to aggregate a collection named products:

```
db.products.aggregate(
        [
                {$group:{
                                _id:"$manufacturer",
                                num_products:{$sum:1}
                }
        ]
)
```

Every element on the array, the aggregation pipeline, is a step on the sequence that goes from the initial collection to the result. Pay attention that the result will be returned in a document, so we have the 16MB limit (but we could limit and skip on our code if needed).

$project	select, reshape 1:1 (add/remove fields)
$match	filter
$group	aggregation
$sort	
$skip	
$limit	
$unwind	unjoin of the embedded subarray

at the end we have a document with {result: ok}

$group

COMPOUND GROUPING

The _id key can be a document, with many subkey

```
$group:{ _id: {category:"$category", manufacturer:$manufacturer} }
```

AGGREGATION EXPRESSION

$sum

$sum:1 is used to count, I sum 1 for each record in the group
I can write $sum:"$price" to sum the value of the price field

$avg

$min

$max

$first Return the first field of the group, that should be ordered

$last

$addToSet

Return an array, with values without duplicates

$push

Like $addToSet, but including duplicates

Examples:
Aggregate everything by state, and the sum the pop values in the field population:

```
db.zips.aggregate([ {$group:{_id:"$state", population:{$sum:"$pop"} }} ])
```

Get a list of unique postal codes grouped by city:

```
db.zips.aggregate([ {$group:{_id:"$city", postal_codes:{$addToSet:"$_id"} }} ])
```

Double grouping, we can do more than one grouping operation on the aggregation pipeline.
Example:

```
db.fun.aggregate([{$group:{_id:{a:"$a", b:"$b"}, c:{$max:"$c"}}}, {$group:
{_id:"$_id.a", c:{$min:"$c"}}}])
```

```
Examples:
Get all the city starting with a number (using regex), and sum their pop
db.zips.aggregate([
    //{$project: {first_char:{$substr:["$_city",0,1]}, pop:1}},
      {$match:{city:/^[0-9]/}},
      {$group:{_id:null, totalPop:{$sum:"$pop"}}},
      //{$limit:3},
])
```

$project

Map 1:1 to change records (compute/add fields or remove others)
Receive a document describing the result
We can compute some fields with expressions
$toLower
$multiply (wants an array where every value will be multiplied)

this is useful to slim down documents before the grouping phase
If a field isn't mentioned it will be removed but _id have to be excluded explicitly.
To keep a field as is we use {fieldname: 1} or {fieldname: "$fieldname"}
Example

```
db.zips.aggregate([
 {$project:{_id:0,
            city:{$toLower:"$city"},
            pop:1,
            state:"$state", zip:"$_id"}
  }
])
```

$match

Filter the input, wants a document like the find command
A more complete example, filtering only records with state=NY, than sum their population grouped by city, and then put back the city in the city field with a project

```
db.zips.aggregate([
      {$match:{ state:"NY" }},
      {$group:{ _id:"$city",
                population:{$sum:"$pop"},
                zip_codes:{$addToSet:"$_id"}
              }
      },
      {$project:{_id:0, population:1, zip_codes:1, city:"$id"}
])
```

$sort

Sort, like the sort cursor command. Can use much memory.

$skip, $limit

They should be used after a sort to have some sense.

$unwind

Called on an array it will flat it creating many records, one for any element in the array.
It wants just the name of the key as parameter, with the $ prefix.
The reverse operation is $push

a double $unwind can be restored with two $push in sequence
example

```
{$unwind: "$to"}
```

Limitations of the aggregation framework:

The result is in a document, so it's limited to 16MB
Can use 10% of the system memory max

MapReduce

Is another way to do aggregations, other than Aggregation framework.
The "reduce" part is done in Javascript, so there is a performance limit (the interpretation, and the fact that the server is currently single-threaded).

Example:

```
m = function(){
      this.tags.forEach(function(z){ emit(z, {count:1}) });
}

r = function(key, values) {
      var total=0;
      for (var i=0; i<values.length; i++) {
                  total += values[i].count;
      }
      return {count:total};
}


res = db.collection.mapReduce(m, r, {out:{inline:1}});
db.zips.mapReduce(m, r, {out:{inline:1}, query:{state:"PA"}})
```

Application engineering

Write concern

```
db.runCommand({ getLastError:1, w:2, wtimeout:5000 })
```

w and j parameters specify:

w: if the client should wait for server ack on write operations.
w=1 when the server should answer once received the write operation
w=2 the ack arrive when the server and another node complete the write...
w='majority' when the ack arrive after the write operation reach the majority of nodes.

j=1 states that the server should wait for the write operation to journal to disk before ack.

wtimeout: is the timeout we have to wait for server write acks

Rollback

A rollback happened when a primary node (the only node who accepts write operation in mongodb) accepts some write operation, but before being able to give them to other nodes it goes offline... a new primary node will be elected and write operations continue there.
When the old primary come back online, it should do a rollback, to move the pending write operations and realign with the rest of the replicaset.
Rollback data are saved in a bson file on the (old) primary node.

Read Preferences

We can activate reading from secondary nodes, with the risk of getting stale data (eventually consistent).
Possible read preferences options:

- primary (default)
- primaryPreferred
- secondary
- secondaryPreferred
- nearest

If what you want to do is scale, sharding should be a better choice (cause it scales write operations too).

Replica sets

We can have many nodes in a replica set, that elect and act as primary node when a fault happened.
A replica set can have up to 12 members and up to 7 voters.
To trigger the election of a new primary the remaining talking nodes should be the majority of the replicaset.
It should start with at least 3 nodes to be able to vote a new primary (otherwise the remaining node remain a secondary, there is no majority with 2 disconnected nodes).
The typical election lag in case of a fail, is under 10 secs.
Mongodb replication has automatic-failover, but we can use a normal master-slave mode.
There are various type of members in a replicaset:
(http://docs.mongodb.org/manual/core/replication/#member-configuration-properties)

- Regular (primary o secondary)
- Arbiter (they can vote, but doesn't have any data)
- Delayed (the replicate with a fixed lag, for example to have a backup)
- Hidden (hidden to clients)

Replica set creation:
Run the mongod server (on the same machine I should set different ports, dbpath and logpath) and set the --replSet <replicaSetID> parameter
inside one of the DB, the DB of that member will be replicated, the same DB on other hosts will be deleted).

```
rs.initiate(<config>)
```

where <config> is a document like:

```
{_id: "rs1", members:[
                {_id:0, host:"localhost:27017", priority:0, slaveDelay:5},
                {_id:1, host:"localhost:27018"},
                {_id:2, host:"localhost:27019"},
        ]
}
```

Then to show the replicaset status:

```
rs.status()
```

To allow reading from secondary, I can use the command:

```
rs.slaveOk()
```

we than have a command to test if current node is Master

```
rs.isMaster()
```

and to get the replica set current configuration:

```
rs.conf()
```

Replica is keeped thanks to an oplog in the local db: **oplog.rs**
it's a capped collaction, we can see it using these command:

```
use local
db.oplog.rs.find().sort({$natural:1}).limit(4)
```

We can reconfigure a replica set by passing the same parameter we used for initiate (the one returned from rs.conf) to the command

```
rs.reconfig(<config>)
```

On the member list, we specify *_id* and *host* parameters, but we can use other options:

- arbiterOnly:true to keep the member an arbiter
- priority:<numero> >= zero (if zero it will never be elected primary)
- hidden:true to keep the member invisible to clients, so they can't connect
- slaveDelay (in seconds) to have a delayed backup node

- votes how many votes cast the member

db.printReplicationInfo() show details about the opLog

To connect to a replicaset with pyMongo

```
connection = pymongo.MongoClient(host="mongodb://localhost:27017",
                        replicaSet = "rs1",
                        w=1, j=True
             )
```

We can specify only one host with the replicaset name, and the other will be discovered automatically (if I am able to connect to that one).

In pymongo I can set the read preferences with MongoReplicaSetClient.

Sharding

Mongos is a server that is the intermediary between clients and mongod, addressing client to the appropriate mongod based on needed shard_key

We need

- every document in a shard collection should have a shard_key immutable
- we must have one index on the shard key or a non multi-index starting with shardkey. The shard key cannot be an array.
- If the query doen't specify the shard_key, all shards will be queried.
- We can't put unique constrain on the collection fields but in the shardkey

---

mongod parameters

for a small local test (keep the oplog small):

```
--smallfiles
--oplogSize 50
--shardsvr
```

mongod config servers parameters

When sharding, it's a best-practice, leave only mongos on port 27017, we usually should not connect to mongod so better to change their port.
Config servers are always 1 (in development) or 3 (in production)

```
--configsvr
```

mongos parameters

```
--configdb <comma separated list of config servers>
```

dariosky_run_complex_shard_mongo.py is a nice example to show how to launch

when all deaemons are started, we can show the config db to see some information of actual status.
Now we should initialize the replicaset and add shards
Start by going to mongod in the replicaset and initialize

`rs.initiate()` initiate replicaset with only that node
`rs.add("hostname:port")` add a member (hostname can't be localhost)
with `rs.conf()` I can see actual configuration

Now from mongos I create a shard:

```
sh.addShard("<setname>/<hostname:port>")
```

where setname is the replicaset name
Warning, on settings, we can't mix *localhost* and ips (or FQNS)

from mongos:

```
sh.status()
```

shows all shards with hosts of replicasets

Initially everything is in the first shard, to activate the sharding, form mongos:

```
sh.enableSharding(<dbname>)
sh.shardCollection("<dbname>.<collectionName>", {_id:1})
```

where the last, is the shardkey.
To create a shardkey, an index on the collection with the shardkey should be present
we can use explain then to see information about how the shard is queryed.

Suggestion for how choosing the shardkey:
It should have a good cardinality (many different values) and a good granularity (we shouldn't have values with too many records).
We can use compound keys.
Try to avoid monotonically increasing shard_key, or the shards should be rebalanced often
(ObjectID ARE monotonically increasing, but since Mongo 2.4 we can use hashed shardkey)

Config servers can be placed on shard servers, but it's better to avoid the first shard, who is the one will keep all unsharded collections.

Mongos can be placed on client (the application servers) or on all the machines with mongod.

DBA Operations

Find and kill slow processes

I can activate slow queries profiling with setProfilingLevel as seen
but to see processes CURRENTLY executing I should use `db.currentOp`
who return (in "inprog") a list of process executing with some details (like "secs_runnging")...
When I have the **opid** of the process, I can kill it with `db.killOp`

Compact a collection

```
db.<collection>.runcommand("compact")
```

or

```
db.runCommand({compact:"<collection>"})
```

Security

We can operate in two way:

1. trusted environment (who can access can access to everything)
2. mongodb authentication:
   via --auth ask authentication to clients
   --keyfile use the key to access intra-cluster

Traffic is not encrypted, we should use SSH (but mongo has to be recompiled)
If we connect to localhost we are allowed even in --auth mode until the first user is created (on DB admin), see below.

```
db.auth
db.addUser
```

Users are “per database”... an exception are users of db “admin” who can access everywhere
They are superuser who can execute administration operation.
Non-admin users, can be readOnly or not, since version 2.2 if they are not readOnly they can create indexes and do db-wide operations.

The **keyfile** is a normal file (between 6 and 1024 chars in base64 set, preferably readable only from the user running mongo) and should have the same content on all members of replicas.

Backup

Simple snapshot

- mongodump (dump all dbs or only one) --oplog
  (mongorestore --oplogReplay to restore)
- filesystem snapshot (like LVM or from a virtual machine.
  Journaling should be enabled and on the backup we should have the *data/journal* subfolder
- backup from secondary (terminating mongod on secondary, copying data folder and then restarting mongod)

Backup with sharding

1. turn off the balancer (`sh.stopBalancer()` to avoind sharding migrations during backup)
2. do the backup of any shard and any config server
3. `sh.startBalancer()`

Backup will contain all data in the DB when the backup started, but data changed during the backup can be or cannot be present in the backup

Example:

```
mongo –host some_mongos –eval "sh.stopBalancer()"
mongodump –host <mongos or config_server> --db config /backups/config1

for any shard (possibly in parallel):
mongodump –host <shardserver1> --oplog /backups/shard1
mongo –host some_mongos –eval "sh.startBalancer()"
```

Possibly we can check the shard status before continue.

Mongoexport

Mongoexport is a command that allow us to export subqueries even if mongod aren't running, opening directly the *data* direcory.

Various

db.getSisterDB(<dbname>)
is a nice way to connect to another DB on the same server, without doing “use”

There are the **capped collection** we saw.
There are also TTL collection who let record exit when they have a certain age, like a cache.

Binary files can be stored in mongodb with GridFS (mongofiles is an utility to work with them).

Solved example: homework on indexes usage

Suppose you have a collection with the following indexes:

```
> db.products.getIndexes()
[
        {       "key" : {
                        "_id" : 1
                },
                "name" : "_id_"
        },
        {       "key" : {
                        "sku" : 1
                },
                "unique" : true,
                "name" : "sku_1"
        },
        {       "key" : {
                        "price" : -1
                },
                "name" : "price_-1"
        },
        {       "key" : {
                        "description" : 1
                },
                "name" : "description_1"
        },
        {       "key" : {
                        "category" : 1,
                        "brand" : 1
                },
                "name" : "category_1_brand_1"
        },
        {       "key" : {
                        "reviews.author" : 1
                },
                "name" : "reviews.author_1"
        }
```

Which of the following queries can utilize an index. **Check all** that apply:

db.products.find({'brand':"GE"})

no, brand is in a compound index, at the beginning

db.products.find({'brand':"GE"}).sort({price:1})

yes, only for the sort we can use the price index reversed

db.products.find({$and:[{price:{$gt:30}},{price:{$lt:50}}]}).sort({brand:1})

yes, I can use the price index for find, not the sort one

db.products.find({brand:'GE'}).sort({category:1, brand:-1}).explain()

no, the compound index doesn't have correct sort order