# Complex Systems Theory and Practice

Lecture 8

Piotr Syga

2021-12-03

# Documents

## Structure

```
{ field1:  value,
field2:  value,
...
fieldN: value}
```

_id field is added automatically with `ObjectId` type.

# Documents

## Arrays and embedded documents

Fields may use non-atomic types as arrays, e.g., `color: ["white", "red"]`. You may address the field by its position `color.2` or reference all positions (e.g., at updating) with `color.[]`.
Matching: `["white", "red"]` vs `$all: ["white", "red"]`
Non-atomic fields may also be a document, e.g., `phone: {home: "XXX", cell: "YYY"}`. We reference the values hierarchically with dot notation `phone.cell`.
Naturally, one may create arrays of embedded documents and embedded documents that include arrays.

Default size limit of a single document (with embeddings) is 16MB, yet can be modified.

# Basic operations

- adding a single document: `db.collection.insertOne()`
- adding an array of documents: `db.collection.insertMany()`
- queries: `db.collection.find()`
- updates: `db.collection.update()`, `db.collection.findAndModify()`
- adding fields: `db.collection.update()` with `$inc:{newField: value}`
- counting documents (with condition): `db.collection.count()`, `db.collection.find({speed:{$lt:240}}).count()`
- deleting documents: `db.collection.remove({condition})`,`db.collection.remove()`
- dropping a collection or a database: `db.collection.drop()`/`db.dropDatabase()`

# Finds

## SQL

```
SELECT * FROM studens WHERE name='Doe'
```

```
db.studens.find({,,name'':  ,,Doe''}).pretty()
```

Some of the conditions:

- comparison: $eq, $ne, $gt, $gte, $lt, $lte, $in, $nin
- logical conjunctions: $and, $not, $or, $nor
- array operators: $all, $elemMatch, $size
- evaluations, e.g., $regex, $mod
- selecting only fixed fields: `.find({condition},{field1:  0, field2:  true, field3:1, field:false})`

# Finds

**SQL**

```
SELECT * FROM studens WHERE name='Doe'
```

```
db.studens.find({,,name'':  ,,Doe''}).pretty()
```

**SQL**

```
SELECT * FROM studens
WHERE course=''CS'' AND (semester=1 or semester=2);
```

```
db.studens.find({ course :  ,,CS'' , $or :  [{ semester :  1
}, { semester :  2 }]}).pretty()
```

# Update

## Schema

`db.collection.update(`*query*`,` *modifications*`,`*options*`)`

**Important:** `upsert` and `multi` options are false by default

## Example: Changing field value

```
db.studenci.update({semestr:3},{$set:{semestr:4}})
```

# Update

## Example: overwriting a document

```
db.studens.save({"_id": ObjectId(6083341721031b0c459c2),
,,name'':''John'', ,,surname'': ,,Smith'', ,,comment'':''foreign
exchange student''})
```

## Example: Modifying all the fields

```
db.studens.update({ name: ,,John'', surname:''Smith'' },
{name: ,,John'', surname:''Smith'', ID:123456789, semester:1,
faculty: ,,W11'', major:''BDA''})
```

# Removing documents

## Deleting with ID

Similarly to SQL, the safest way to delete a document is by using ID (a primary key) – `db.students.find({surname: {$regex:'Smi'}},{_id:1})` and then `db.students.deleteOne({ _id: XXX})`

## Direct deleting

Alternatively, one can delete the result of the selection `db.students.deleteOne({surname: 'Smith'})`

- just as with inserts (`*.insertOne()`, `*.insertMany()`) one can delete one (first match) or more documents (`*.deleteOne()`, `*.deleteMany()`)
- note that even though the effect of `remove()` and `deleteMany()` is similat there are important differences – default number of deleted elements, the returned type (object vs document), both commands may not be viable in different programming languages (`remove()` is deprecated in many of them)

# Aggregation

- simple aggregation is possible e.g., `db.collection.estimatedDocumentCount()`, `db.collection.count()`
- MongoDB utilizes aggregation pipelines, that are successors of Map/Reduce
- the aggregation map `db.collection.aggregate([])` consists of 2 main stages – matching and grouping
- `$match` filters out the documents that are passed to the grouping stage, e.g., `{ $match: { name: "John"} }`
- `$group` does two things: groups the documents based on the value of the grouping field and performs the aggregate function, e.g., `{ $group: { _id: "$major", averageMarks: { $avg: "$grade"} } }`
- additional stage – `$setWindowFields` may be performed additionally to add partitions, sorts, and document ranges

# Aggregation

Selected aggregate functions for grouping operator (accumulator operators) include:

- known from SQL: $avg, $count, $max, $min, $sum and two variants of std: $stdDevPop, $stdDevSamp
- document list and set operations: $first, $last, $mergeObjects, $push, $addToSet
- custom $accumulator function

# Aggregation

## $accumulator definition

{ $accumulator: {
init: <code>,
initArgs: <array expression>,
accumulate: <code>,
accumulateArgs: <array expression>,
merge: <code>,
finalize: <code>,
lang: <string>
} }

# Aggregation

## $accumulator example – data

```
db.books.insertMany([ { '_id' : 8751, 'title' : 'The Banquet', 'author' :
'Dante', 'copies' : 2 }, { '_id' : 8752, 'title' : 'Divine Comedy', 'author' :
'Dante', 'copies' : 1 }, { '_id' : 8645, 'title' : 'Eclogues', 'author' :
'Dante', 'copies' : 2 }, { '_id' : 7000, 'title' : 'The Odyssey', 'author' :
'Homer', 'copies' : 10 }, { '_id' : 7020, 'title' : 'Iliad', 'author' :
'Homer', 'copies' : 10 } ])
```

# Aggregation

```
db.books.aggregate([{$group : { '_id : "$author", avgCopies:{
$accumulator: init: function() { //Set the initial state
return { count: 0, sum: 0 } },
accumulate: function(state, numCopies) { // Define how to update the state
return {count: state.count + 1, sum: state.sum + numCopies } },
accumulateArgs: ["$copies"], // Argument required by the accumulate function
merge: function(state1, state2) { // When the operator performs a merge,
return { //add the fields from the two states
count: state1.count + state2.count,
sum: state1.sum + state2.sum } },
finalize: function(state) { //After collecting the results from all documents,
return (state.sum / state.count) // calculate the average
},
lang: "js"} }}}])
```

# Aggregation

## Additional aggregation stages

Aside from 2 main stages and `$setWindowFields` the aggregation pipeline allows additional stages, some that are worth checking inluce:

- $project
- $lookup
- $unwind
- $sort
- $limit
- $skip

# Indexes

- similarly to SQL databases, Mongo supports indexes
- indexes are created at the collection level (cf. table's indexes)
- there's always at least 1 index – _id
- index creation `db.collection.createIndex( <key and index type specification>, <options> )`
- possible option include `unique`, `name`, `sparse`, `expireAfterSeconds`
- the index may be created on a single or multiple fields, the default name is equal to concatenation of all the used fields
- one can determine the order of the index by putting a positive or a negative key, e.g., `db.students.createIndex({surname:-1})`
- the order of fields in multifield indexes is important, moreover a separate index is created for each level of indexed embedded documents or arrays
- partial vs sparse index

# Indexes

Limitation:

- there can be only 1 index of a given name
- if the unique option is set to true, the index will prohibit to enter duplicate values
- a single collection is limited to 64 indexes (including those for embedded documents and arrays)
- there can be no more than 32 fields in a compound index
- the limit on the memory usage of `CreateIndexes` is 200MB, overhead has to be done in an additional command

# Transactions

- MongoDB uses atomic operations on a single document, transactions are required for atomicity of multidocument queries
- transactions may be used on replicas and sharded sets
- one can create collections inside a transactions (both implicitly as a side effect of evocation and explicitly)
- indexes may be created inside a transactions but only on non-existent or empty collections
- race condition may require to use write concern majority
- the transaction is started by creating a session object `mySession = db.getMongo().startSession()` and then starting transaction `mySession.startTransaction()`
- the transaction is ended with `mySession.commitTransaction()`

# Replication

- replicas are a copies of the data kept on a separate process (and possibly server)
- the main goal is to provide redundancy (for majority votes and backups) and high availability
- replicas consist of a primary database copy and a set (at least two) secondary copies that allow data restoration – the votes are based on majority for inconsistent databases
- in the case of lacking replicas one could establish arbiter who cannot become a primary, however takes part in all the votes

## Creating a replica set

We create the set in command line `mongod -port <port> -dbpath /var/lib/mongodb -replSet <name>`.
Starting the set and adding new instances are performed in mongo shell `rs.initiate()`, `rs.add(<hostname:port>)`