5 Getting acquainted with MongoDB – Part 1

5.1 Source

This text is fully based on https://docs.mongodb.com/manual/introduction/

5.2 Introduction to MongoDB

MongoDB is a document database designed for ease of development and scaling.

5.2.1 Document Database

A record in MongoDB is a document, which is a data structure composed of **field** and **value** pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

The advantages of using documents are:

- Documents (i.e. objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism.

5.2.2 Collections

MongoDB is a document-oriented database system made up of the hierarchy of **database > collection > document > field > key:value**. Here is an approximate comparison between MongoDB and SQL data models.

MongoDB	**SQL**
database	database
collection	table
document	Row or record
field	-
{key:value}	-

In MongoDB, databases hold collections of documents and a collection stores documents. Collections are analogous to tables in relational databases. Unlike a table, however, a collection does not require its documents to have the same schema.

In MongoDB, documents stored in a collection must have a unique _id field that acts as a primary key.

MongoDB stores data records as BSON documents. BSON is a binary representation of JSON documents, though it contains more data types than JSON.

5.2.3 Getting started

Follow the instructions in the document ***How To Install MongoDb + Studio 3T***

Create a new connection and name it e.g. DBProgramming. Start the IntelliShell.

5.3 MongoDB CRUD Operations

5.3.1 Insert documents (mongoDB, MongoDB CRUD Operations, 2018)

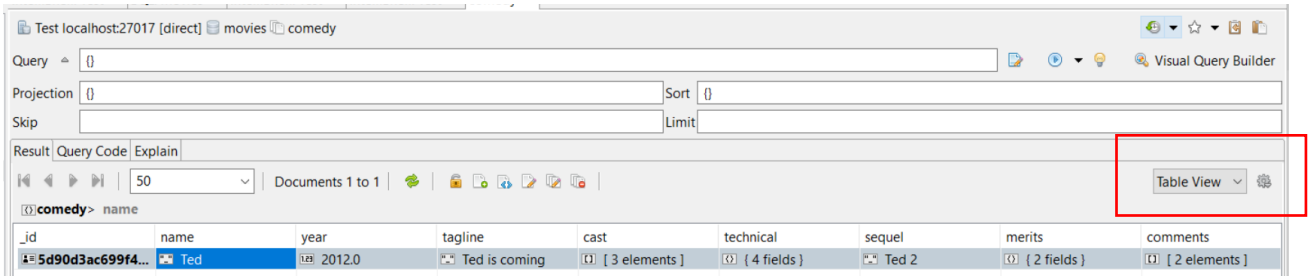
```
/* To begin creating a new database, first execute the use command to switch to a new database. At this moment, you won't see any new database yet */
```

```
use movies
```

```
/* Select the following command and press Ctrl + Enter or click F9 (to execute the selected command) */

/* The command saves some data to the database.

/* Afterwards the new database "movies" will be created */

/* In the process a new collection called "comedy" will also be created in the database. */
```

```
db.comedy.insert({name: 'Ted', year: 2012, tagline: 'Ted is coming',
    cast: ['Mark Walhlberg', 'Mila Kunis', 'Seth MacFarlane'],
    technical: {runningTime: 106, language: 'English', prizes: 13, nominations: 27},
    merits: {budget: 50, boxOffice: 535},
    comments:[{by:'Steve', text:'Loved the movie'}, {by:'Dave', text:'Really funny!'}]})
```

```
/* Right click on the Connection at the left. Choose Refresh All */

/* Right click on movies > Collections > comedy at the left. Choose Open Collection Tab => a new tab is opened. Take a look at it */

/* There are multiple View Modes. You can change the View Mode using the buttons at the upper right. The default mode is Table View */
```



```
/* New in version 3.2: insertOne: inserts a single document into a collection */
db.comedy.insertOne({name: 'We\'re the Millers', year: 2013, tagline: 'If anyone asks',
    cast: ['Jennifer Aniston', 'Luis Guzmán', 'Ed Helms', 'Kathryn Hahn'],
    technical: {runningTime: 110, language: 'English'},
    sequel: 'Ted 2',
    merits: {budget: 37, boxOffice: 270},
    comments:[{by:'Taylor', text:'First class movie!'}, {by:'Rob', text:'I like it'}]})

/* New in version 3.2: insertMany: inserts multiple documents into a collection
*/
db.comedy.insertMany([{name: 'The Hangover', year: 2009, tagline: 'Some guys just can\'t
handle Vegas',
    cast: ['Bradley Cooper', 'Ed Helms', 'Zach Galifianakis'],
    technical: {runningTime: 100, language: 'English'},
    sequel: 'The Hangover Part II',
    merits: {budget: 35, boxOffice: 467.5},
    comments:[{by:'Alex', text:'Dude, it rocked'}, {by:'Steve', text:'The best movie
ever!'}]},
    {name: "The Hangover Part II", year: 2011, tagline:'Bangkok has them now',
    cast: ['Bradley Cooper', 'Ed Helms', 'Zach Galifianakis'],
    technical: {runningTime: 102, language: 'English'},
    sequel: 'The Hangover Part III',
    merits: {budget: 80, boxOffice: 581},
    comments:[{by:'Anne', text:'Liked the first part better'}, {by:'Robin', text:'Over
the top'}]}])

/* Insert a new comedy
    - name: 'Ted 2'
    - year: 2015
    - tagline: 'Ted is coming, again'
    - cast: ['Mark Walhlberg', 'Seth MacFarlane', 'Amanda Seyfried', 'Morgan
Freeman']
    - technical: {runningTime: 115, language: 'English'},
    - merits: {budget: 85}
    - comments:[{by:'Anne', text:'Funny'}, {by:'Kate', text:'I still love Ted'},
{by:'Leo', text:'Nice movie'}]*/
db.comedy.insert({name: 'Ted 2', year: 2015, tagline: 'Ted is coming, again',
    cast: ['Mark Walhlberg', 'Seth MacFarlane', 'Amanda Seyfried', 'Morgan Freeman'],
    technical: {runningTime: 115, language: 'English'},
```

```
    merits: {budget: 85},
    comments:[{by:'Anne', text:'Funny'}, {by:'Kate', text:'I still love Ted'}, {by:'Leo', text:'Nice movie'}]})
```

5.3.2 Query documents (mongoDB, Query Documents, 2018)

```
/* To read data from a collection */
db.comedy.find()

/* Conditional operators */
/* How do you do an 'equal to' query? Just match the value for the queried key */
db.comedy.find({year: 2012})

/* Find all movies released in the year 2013 */
db.comedy.find({year: 2013})

/* Use these special forms for greater than and less than comparisons in queries, since they have to be represented in the query document:
    - db.collection.find({ "field" : { $gt: value } } );    // greater than  : field > value
    - db.collection.find({ "field" : { $lt: value } } );   // less than  :  field < value
    - db.collection.find({ "field" : { $gte: value } } );  // greater than or equal to : field >= value
    - db.collection.find({ "field" : { $lte: value } } );  // less than or equal to : field <= value
*/
db.comedy.find({year: {$lt: 2012}})
db.comedy.find({year: {$gt: 2010}})

/* Find all movies from the year 2011 until now */
db.comedy.find({year: {$gte: 2011}})

/* To search an object inside an object, just use the regular JavaScript dot notation of the target object as the key and quote it. The '' around 'merits.budget' are mandatory */
/* Find all movies with a budget over 50 million dollar */
db.comedy.find({'merits.budget': {$gt: 50}})

/* Find all  movies of which the runningTime is longer than 105 minutes */
db.comedy.find({'technical.runningTime': {$gt: 105}})
```

```
/* Find all movies of which the language is English */
db.comedy.find({'technical.language': 'English'})

/* You can also combine these operators to specify ranges */
db.comedy.find({year: {$gt: 2010, $lt: 2013}})

/* Find all movies with runningTime between 100 and 110 */
db.comedy.find({'technical.runningTime': {$gt: 100, $lt: 110}})

/* Until now all fields of the documents are shown in the result */
/* What if you want to get only some particular fields in the result? */
db.comedy.find({year: {$lt:2012}}, {name:true})

/* In the above example, we excluded most fields by not specifying them in the
second parameter of the find() function.
To get more than one exclusive field in the results you might do something like
this */
db.comedy.find({year: {$lt:2012}}, {name:true, year:true})

/* Give name and boxOffice of all movies with boxOffice over 500 million dollar
*/
db.comedy.find({'merits.boxOffice': {$gt:500}}, {name:true, 'merits.boxOffice':true})

/* What if you want to get almost all except for some fields in the result? */
db.comedy.find({year: {$lt:2012}}, {name:false})

/* Give name and boxOffice of all movies with boxOffice over 500. Get rid of _id
*/
db.comedy.find({'merits.boxOffice': {$gt:500}}, {name:true, 'merits.boxOffice':true,
_id:false})
```

(you might get an error message, which you can ignore)

```
/* A quoted number is a string, and is not the same as the actual number */
db.comedy.find({year:2012})
/* is totally different from */
db.comedy.find({year: '2012'})

/* Use $ne for "not equals" */
```

```
db.comedy.find({year: {$ne: 2011}})
```

/* The $in operator is analogous to the SQL IN modifier, allowing you to specify an array of possible matches. */

```
db.comedy.find({year: {$in: [2010,2011,2012]}});
```

/* Find all comedies with a budget of 50, 60, 70 or 80 */

```
db.comedy.find({'merits.budget': {$in: [50,60,70,80]}});
```

/* The $nin operator is similar to $in except that it selects objects for which the specified field does not have any value in the specified array. */

```
db.comedy.find({year: {$nin: [2010,2011,2012]}});
```

/* Find all comedies that have a runningTime other than 100 or 105 */

```
db.comedy.find({'technical.runningTime': {$nin: [100,105]}});
```

/* The $or operator lets you use boolean or in a query. You give $or an array of expressions, any of which can satisfy the query. */

/* The $or operator retrieves matches for each or clause individually and eliminates duplicates when returning results. */

```
db.comedy.find({$or: [{year: 2012}, {name: 'The Hangover'}]});
```

/* Find all comedies with the name Ted or The Hangover from the year 2012 */

```
db.comedy.find({year: 2012, $or: [{name: 'Ted'}, {name: 'The Hangover'}]});
```

/* Find all comedies with a boxOffice over 500 from the year 2010 or 2011 */

```
db.comedy.find({'merits.boxOffice': {$gt:500}, $or: [{year: 2010}, {year: 2012}]});
```

/* The $nor operator lets you use a boolean nor expression to do queries. You give $nor a list of expressions, none of which can satisfy the query. */

/* Find all comedies except for Ted or The Hangover */

```
db.comedy.find({$nor: [{name: 'Ted'}, {name: 'The Hangover'}]});
```

/* Find all comedies not released in the years 2010 or 2011 */

```
db.comedy.find({$nor: [{year: 2010}, {year: 2011}]});
```

/* The $and operator lets you use boolean and in a query.

You give $and an array of expressions, all of which must match to satisfy the query. */

```
db.comedy.find({$and:[{year: {$gt: 2010}}, {year:{$lt: 2012}}]})
```

```
/* Find all movies for which the boxOffice exceeded over 500 million dollar and
the budget was lower than or equal to 50 million dollar */
```

```
db.comedy.find({$and:[{'merits.budget': {$lte: 50}}, {'merits.boxOffice':{$gt: 500}}]})
```

```
/* You can also query an array */
```

```
db.comedy.find({cast:'Bradley Cooper'})
```

```
/* When the key is an array, you can look for an object inside the array */
```

```
db.comedy.find({'comments.by':'Steve'})
```

```
/* Find the movies that have comments by Rob or Alex */
```

```
db.comedy.find({$or: [{'comments.by':'Rob'}, {'comments.by':'Alex'}]})
```

```
/* The $size operator matches any array with the specified number of elements. */
```

```
db.comedy.find({comments: {$size:2}})
```

```
/* Find the movies with 4 actors */
```

```
db.comedy.find({cast: {$size:4}})
```

```
/* You cannot use $size to find a range of sizes (for example: arrays with more
than 1 element). If you need to query for a range, create an extra size field that
you increment when you add elements. */

/* You can even use regular expressions in your queries */

/* i means it's case-insensitive */
```

```
db.comedy.find({name:{$regex: /bill|ted/i}})
```

```
/* An example with a syntax a bit shorter */
```

```
db.comedy.find({name: /The hangover.*/i})
```

```
/* Another way of writing the same */
```

```
db.comedy.find({name: {$regex: 'The hangover.*', $options: 'i'}})
```

```
/* Find all movies for which the comments contain the word love */
```

```
db.comedy.find({'comments.text': /love.*/i})
```

```
/* If you wish to specify both a regex and another operator for the same field,
you need to use the $regex clause.  */
```

```
db.comedy.find({name: {$regex: /The hangover.*/i, $nin: ['The Hangover Part II']}});
```

```
/* The $not meta operator can be used to negate the check performed by a standard operator. */

db.comedy.find({name: {$not: /The hangover.*/i}});

/* The following doesn't work! --> "errmsg" : "$not needs a regex or a document" */

db.comedy.find({year: {$not: 2012}});

/* Find all comedies that were not released in 2012 */

db.comedy.find({year: {$ne: 2012}});

/* MongoDB queries support JavaScript expressions in combination with the where operator! */

db.comedy.find({$where: 'this.year > 2009 && this.name !== "Ted"'})

/* Try to retrieve the same result as in the previous query, but not using JavaScript */

db.comedy.find({year:{$gt:2009}, name:{$ne:'Ted'}})

/* Note that the flexibility of JavaScript expressions comes at a price. It is slower than native MongoDB operators. */

/* Find all comedies that were released in 2011 or later with Ed Helms as part of the cast */

db.comedy.find({cast:'Ed Helms', $where: 'this.year >= 2011'})

/* Find the movies that were commented by Steve and have a budget of 50 million dollar or more */

db.comedy.find({'comments.by':'Steve', $where: 'this.merits.budget >= 50'})

/* Find the movies with 3 comments or more */

db.comedy.find({$where: 'this.comments.length >= 3'})

/* The $all operator is similar to $in, but instead of matching any value in the specified array all values in the array must be matched. */

/* An array can have more elements than those specified by the $all criteria. $all specifies a minimum set of elements that must be matched. */

db.comedy.find ({'cast': {$all: ['Bradley Cooper', 'Ed Helms']}})

db.comedy.find ({'cast': {$all: ['Bradley Cooper', 'Mila Kunis']}})

/* Find all the movies commented by Anne and Robin */

db.comedy.find ({'comments.by': {$all: ['Anne', 'Robin']}})
```

/* $exists checks for existence (or lack thereof) of a field. */

```
db.comedy.find ({tagline: {$exists : true}})
db.comedy.find ({'merits.boxOffice': {$exists : true}})
```

/* Find all movies that have a sequel */

```
db.comedy.find ({sequel: {$exists : true}})
```

/* Find all movies that have a sequel of the Hangover */

```
db.comedy.find ({sequel: {$exists : true}, sequel:/The Hangover.*/i})
```

/* Find the movies which have a field prizes and which won more than 10 prizes */

```
db.comedy.find ({$and:[{'technical.prices': {$exists: true}}, {'technical.prizes': {$gt: 10}}]});
```

/* Any good database system should have a count() method which returns the number of records that will be returned for a query.

MongoDB too has a count() method which you can call on a collection to get the count of the results. */

/* This will return the total number of documents in the collection comedy */

```
db.comedy.count()
```

/* This will return the total number movies with the value of year more than 2009 */

```
db.comedy.count({year: {$gt:2009}})
```

/* Find the number of movies witch are commented by Steve */

```
db.comedy.count({'comments.by': 'Steve'})
```

/* To limit the collection to just two */

```
db.comedy.find().limit(2)
```

/* The skip() expression allows one to specify at which object

the database should begin returning results. */

```
db.comedy.find().skip(1).limit(2)
```

/* In the shell, a limit of 0 is equivalent to setting no limit at all. */

/* sort() is analogous to the ORDER BY statement in SQL - it requests that items be returned in a particular order.

```
You can pass sort() a key pattern which indicates the desired order for the result.
*/
```

```
db.comedy.find().sort({name : 1})
```

```
/* Sort the movies chronologically */
```

```
db.comedy.find().sort({year : 1})
```

```
/* Sort the movies reverse chronologically */
```

```
db.comedy.find().sort({year : -1})
```

5.3.3 Update documents

```
/* Suppose you would want to add a new field to the movie Ted.

What comes to your mind instantly might look something like this (do not execute
this) */
```

```
db.comedy.update({name:'Ted'}, {director:'Seth MacFarlane'})
```

```
/* HOLD IT! That's about to do something disastrous.

It will overwrite the whole document with {director:'Seth MacFarlane'}, instead
of updating it by appending a new field.

The right way to do is: */
```

```
db.comedy.update({name:'Ted'}, {'$set':{director:'Seth MacFarlane'}})
```

```
/* The above command will reorder the fields, but worry not, the data is safe and
intact.

The db.collection.update() method modifies existing documents in a collection.

The db.collection.update() method can accept query criteria to determine which
documents to update.

By default, the db.collection.update() method updates a single document.

Operations performed by an update are atomic within a single document. */
```

```
/* With the multi option, update() can update all documents in a collection that
match a query. Otherwise the update method will affect only 1 document. Which
document will be affected, is undefined. */
```

```
db.comedy.update({name: /The hangover.*/i}, {'$set': {director:'Todd Phillips'}}, {multi:
true})
```

```
/* Add a new field voiceOver: 'Patrick Stewart' to both Ted movies */
```

```
db.comedy.update({name: /Ted.*/i}, {'$set': {voiceOver:'Patrick Stewart'}}, {multi:
true})
```

```
/* If the update() method includes upsert: true and no documents match the query
portion of the update operation, then the update operation creates a new document.
```

If there are matching documents, then the update operation with the upsert: true modifies the matching document or documents.

By specifying upsert: true, applications can indicate, in a single operation, that if no matching documents are found for the update, an insert should be performed. */

```
db.comedy.update({name: 'The Hangover Part III'}, {'$set': {year: 2013}}, {upsert: true})
```

/* If you want to update at most a single document that match a specified filter even though multiple documents may match the specified filter. New in version 3.2 */

```
db.comedy.updateOne({name: /The hangover.*/i}, {'$set': {distributedBy:'Warner Bros Pictures'}})
```

/* If you want to update all documents that match a specified filter. New in version 3.2 */

```
db.comedy.updateMany({name: /The hangover.*/i}, {'$set': {distributedBy:'Warner Bros Pictures'}})
```

/* Update all Ted documents so the distribution company is Universal Pictures */

```
db.comedy.updateMany({name: /Ted.*/i}, {'$set': {distributedBy:'Universal Pictures'}})
```

/* Write operations – if they affect multiple documents – are not isolated transactions, instead they can affect a few documents, then yield and allow other readers or writers to operate and then pick up again to affect some more documents.

However, an individual document manipulation is always atomic with respect to any concurrent readers or writers.

So no reader or writer in the system will see the document half updated. */

/* How do you update a value which is an array? */

```
db.comedy.update({name: 'Ted'}, {'$push': {cast: 'Joel McHale'}})
```

/* Add the actor 'Giovanni Ribisi' to the cast of Ted */

```
db.comedy.update({name: 'Ted'}, {'$push': {cast: 'Giovanni Ribisi'}})
```

/* If you need to remove something from the cast array, you can do it this way */

```
db.comedy.update({name: 'Ted'}, {'$pull': {cast: 'Giovanni Ribisi'}})
```

/* You can also use $pop to remove the first element */

```
db.comedy.update({name: 'Ted'}, {'$pop': {cast: -1}})
```

/* You can also use $pop to remove the last element */

```
db.comedy.update({name: 'Ted'}, {'$pop': {cast: 1}})
```

/* How can you delete a field from a document? */

```
db.comedy.update({name:'Ted'}, {$unset: {cast:1}})
```

/* Remove the voiceOver field from both Ted movies */

```
db.comedy.updateMany({name: /Ted.*/i}, {$unset: {voiceOver: 1}})
```

/* In case you want to delete a field from all the documents of a collection */

/* The false parameter is for upsert option, true is for multiple option.

We set multiple option to true because we want to delete them all from the collection. */

```
db.comedy.update({},{$unset: {cast:1}}, false, true)
```

/* How can you delete a document from a collection? */

```
db.comedy.remove({name: /Ted*/i})
```

/* The above command deletes a single document or all documents that match a specified filter. */

/* If you want to delete at most a single document that match a specified filter even though multiple documents may match the specified filter. New in version 3.2 */

```
db.comedy.deleteOne({name: /The Hangover*/i})
```

/* If you want to delete all documents that match a specified filter. New in version 3.2 */

```
db.comedy.deleteMany({name: /The Hangover*/i})
```

/* How do you empty a collection of its documents? */

```
db.comedy.remove({})
```

/* Note, the above command does not delete the collection, it just empties the collection like the SQL truncate command. */

/* How do you drop a collection? */

```
db.comedy.drop()
```

/* How do you drop a database? */

```
use movies
db.dropDatabase()
```

5.4 Exercises

5.4.1 Exercise 1

First create the collection smartphones

1 Give all smartphones with a price between 400 and 700 euro
2 Give all smartphones with Android as operating system
3 Give name, price and reviews of the most expensive smartphone
4 Add to the Apple iPhone the field ‘wifi’ with the following values: 802.11b, 802.11g, 802.11n
5 Give all smartphones with a phonebook and a clock, but no call list or calculator in the organizer

5.4.2 Exercise 2

First create the collection photoframes

1 Give all photoframes with a price lower than 50 euro or with a screensize smaller then 20
2 Give name, price and screensize of the second cheapest photoframe
3 Give all photoframes with 2 or more reviews
4 Give all photoframes that can handle memorycards of type SDHC, MMC, xD, but can’t handle memorycards of type MS or CF
5 Add the remark ‘Not available’ to all the photoframes with a contrastratio different from 500:1, 600:1 or 700:1

5.4.3 Exercise 3

First create the collection laptops

1 Give all laptops with a price between 500 and 1000 euro or with a hard disk of 850 GB
2 Change Windows 10 to Windows 11
3 Give name, price and operating system of the 2 cheapest laptops
4 Give all laptops with at least 1 review with a score between 4 and 4.2
5 Give all laptops with 2 USB 3_0 ports and that support at least 2 languages including Dutch

5.4.4 Exercise 4

First create the collection irons

1 Give all irons of the brand Philips or Tefal that cost 30 euros or less, that are red, have a power of 2000 or more and have a cord storage space and spray function.
2 Give the model, price and brand of the most expensive iron that is not red, weighing less than 2 kilos, that has at least 2 plus points, a maximum height of 20 centimeters and that does not have cord storage space as extra