1 M101J - MongoDB for Java Developers

1.1 General

- mongoimport -d myDB -c xxx [--drop] < xxx.json // imports a JSON file into a collection

1.2 Week 2 – CRUD

1.2.1 Insert, Find & Count

- db.[collection].insert({…})
- The collection unique ID field is called "_id" and can be provided. If not provided an ObjectID will be generated based on the time, machine, process-id and process dependent counter.
- "_id" does not have to be a scalar value – it can be a document, e.g. _id : {a:1, b:'ronald'}
- db.[collection].find || findOne ({…}, {field1 : true, …}).pretty() //no argument will find all docs
- db.[collection].count({…})

1.2.1.1 Operators

- Ranges: {myField: {$gt: 100, $lt: 10}} $gte, $lte → Can be applied to numbers and strings (ASCII)
- Regex: {myField: {$regex: "a$"}}
- Set operators: {myField: {$in: ["one", "two", …]}} $nin
- Boolean:
  - {$or: [{…}, {…}, …]} $and
  - $not - negates result of other operation or regular expression query
  - tags: {$ne: "gardening"} // works on keys pointing to single values or arrays – inefficient – can't use indexes
  - {myField: {$exists: true}} //checks if particular key exists in document
- {myField: {$type: 2}} // 2=String as defined in BSON spec
- http://docs.mongodb.org/manual/reference/operator/

1.2.1.2 Array-Operators

- {myArrayField: "test"} → Will find any documents where the array contains the value "test"
- {myArrayField.0 : "test"} → Value at particular position within array
- {myArrayField: {$all: ["one", "two", …]}} → array contains all given values in any order
- {myArrayField: {$size: 3}} → array with three elements

1.2.1.3 Nested Documents

- {"myField.mySubfield" : "test"} → Dot-Notation needs to be put in ""
- {"myArrayField.0.mySubfield" : "test"} → stipulate zeroth element of array
- {"myArrayField.mySubfield" : "test"} → search in any of the array elements
- {myArrayField : {$elemMatch: { mySubfield : "test", mySubfield2 : "test2"}}} → restrict multiple conditions to same subdocument of array field

1.2.1.4 Cursors

- myCursor = db.[collection].find(); null; → append null as not to print out the cursor immediately
- myCursor.hasNext() myCursor.next()
- myCursor.skip(2).limit(5).sort({name : -1}); null; → modifies the query executed on the server

1.2.2 Updates

- db.[collection].update({ myQuery }, {myField: "newValue", … }) → replaces the existing document
- db.[collection].update({ myQuery }, {$set : {myField: "newValue"}}) → Create or update myField
- db.[collection].update({ myQuery }, {$inc : {age: 1}})

- db.[collection].update({ myQuery }, {$unset : {myField: 1}})
- db.[collection].update({ myQuery }, {$set : {myField: “newValue”}}, {upsert: true}) → Create or update document specified by { myQuery } with myField

Arrays

- db.[collection].update({ myQuery }, {$set : {“myArray.2”: “x”}}) -> Set 3rd position of Array
- db.[collection].update({ myQuery }, {$push : {myArray: “y”}})
- db.[collection].update({ myQuery }, {$addToSet : {myArray: “y”}}) //will only add if does not exist yet
- db.[collection].update({ myQuery }, {$pop : {myArray: 1}}) // pop right-most
- db.[collection].update({ myQuery }, {$pop : {myArray: -1}}) // pop left-most
- db.[collection].update({ myQuery }, {$pushAll : {myArray: [“a”, “b”, “c”]}})
- db.[collection].update({ myQuery }, {$pull : {myArray: “c”}}) // remove value “c”
- db.[collection].update({ myQuery }, {$pullAll : {myArray: [“a”, “b”, “c”]}})

Multi-Update

- db.[collection].update({}, {$set: {title: “Dr.”}}, {multi: true}) // {} matches all documents

1.2.3 Deletes

- db.[collection].remove({…}) // no argument will remove all documents from collection – not isolated
- db.[collection].drop() // faster but drops collection including indexes

1.2.4 Get last error

- db.runCommand({getLastError: 1}) // get info/error of last operation on this connection to mongod

1.2.5 JavaDriver

```
MongoClient mongo = new MongoClient();
DB db  = client.getDB(“test”);
DBCollection collection = db.getCollection(“test”);

DBObject document = new BasicDBObject(“key”, “value”).append(“key2”, “value2”);
collection.insert(document);

DBObject query = new BasicDBObject(“key”, “value”).append(“y” new BasicDBObject(“$gt”, 0).append(“$lt”, 99));
query = new QueryBuilder().start(“key”).is(“value”).and(“y”).greaterThan(0).lessThan(99).get();
document = collection.findOne()
collection.count(query);

DBObject fieldSelector = new BasicDBObject(“key”: true).append(“_id”, false);
DBCursor cursor = collection.find(query, fieldSelector).sort(sortDocument).skip(2).limit(5);
cursor.hasNext(); document = cursor.next();
finally { cursor.close();}

collection.update(query, new BasicDBObject(“$set”, new BasicDBObject(“lala”, 1)));
collection.remove();

collection.findAndModify(query, fields, sortCriteria, removeDocumentFlag, updateDocument, returnNewFlag,
upsertFlag); // is atomic
```

1.3 Week 3 – Schema Design

- Which data is used together; which data is read; which data is written all the time → *Make the schema matching to the data access patterns of your application*
- Mongo has no joins / no foreign key constraints → *but can embed Documents (Pre-Join)*
- MongoDB does not support transactions → *but atomic operations on ONE document, so instead of transactions, you have three options*
    - *Restructure data to live in one document*
    - *Implement transaction in Software*
    - *Tolerate a little bit of inconsistency*

1.3.1 One-to-one relations

- EITHER two collections were one document point to the document in the other collection by _id
- OR embed one document in the other
- Decision driven by
    - Frequency of access; Are the documents read together – do you want to pull everything into memory
    - Are the documents written together
    - Document max size: 16 MB
    - Do you need atomicity

1.3.2 One-to-many relations

- Two collections – linking from “many collection” to _id of “one collection”
- If it is really one-to-few: possible to have “one collection” and embed “few document”

1.3.3 Many-to-many relations

- It often really is a few-to-few relation, e.g. authors-books
- EITHER have two collections and add an array of book-ids in author document or vice versa – depends on access pattern
- (OR embed book in author document but this might lead to inconsistency as one book might be duplicated – also not a good idea if you need to store one item before the other exits, e.g. student-teacher)

1.3.4 GridFS - Blobs

- > 16MB
- Files collection and Chunks collection. MongoDB spits Blob into chunks of 16MB and stores them in the Chunks collection. Each chunk has a file_id pointing to the _id of its file document.

```
GridFS videos = new GridFS(db, “videos”);
GridFSInputFile video = videos.createFile(inputStream, “video.mp4”);
BasicDBObject metadata = …
video.setMetaData(metadata);
video.save();
…
GridFSDBFile myVideo = videos.findOne(new BasicDBObject(“filename”, “video.mp4”)));
myVideo.writeTo(outputStream)
```

1.4 Week 4 – Performance

1.4.1 Index creation and deletion

- db.[collection].ensureIndex({student_id:1})
- 1=ascending, -1=descending→ important for sorting not so much for searching
- → Sorting can also use a reverse index if the sort criteria are exactly the reverse of an (simple or compound) index
- *Compound Index*:
    - db.[collection].ensureIndex({student_id:1,class:-1})
    - General rule: A Query where one term demands an exact match and another specifies a range requires a compound index where the range key comes second
- *Unique Index*: db.[collection].ensureIndex({student_id:1}, {unique: true}) // dropDups: true → dangerous
- By default index creation is done in the foreground which is fast but blocking all other writers to the same DB. Background index creation {background: true} will be slow but it will not block the writers
- We want indexes to be in memory. Find out the index size: db.[col].stats() or db.[col].totalIndexSize()
- db.system.indexes.find() → finds all indexes of the current db
- db.[collection].getIndexes() → all indexes of collection
- db.[collection].dropIndex({student_id:1})

1.4.2 Multi key indices

- A multi key index is an index on an array field of a document, e.g. a student document has array of teacher-ids. One can add a multi key index on the teachers-array, which indexes all of the values in the array for all the documents.
- Multi key indices are one of the reason that linking works so well in MongoDB
- It is not possible to have a compound index with two array (multi key) fields

1.4.3 Sparse Index

- Missing index key in documents map to null → unique key not possible because multiple nulls are not allowed
- Sparse indexes only index documents that have a key set for the key being indexed {unique: true, sparse: true}
- On a sorted find the non-indexed documents will not be found when the sparse index is used for the sort

1.4.4 Explain & Hint

- db.[collection].find({…}).explain()
- db.[collection].find({…}).explain(true) //shows all possible plans
- db.[collection].find({…}).hint({a:1, b:1}) // use specified index
- db.[collection].find({…}).hint({$natural:1}) // use no index
- In Java:
    - .find(query).hint(“IndexName”) OR
    - .find(query).hint(new BasicDocument(a, 1).append(b, 1))

1.4.5 Efficiency of indexes

- $gt, $lt, $ne, $nin, $not($exists) might be inefficient even if an index is used because still many index items (indexed documents) need to be scanned → may be a good idea to use a hint to use a diff. index
- $regex can only use an index if it is stemmed on the left side, e.g. /^abc/

1.4.6 Geospatial indexes

- .ensureIndex({location: ‘2d’, type: 1}) // Compound index on location (uses 2d-index) and ascending type

1.4.7 Profiling slow queries

- MongoDB logs slow queries (>100ms) by default into the logfile
- Use pofiler:
- mongod --profile 1 --slowms 10 // logs all queries taking longer than 10ms to system.profile collection
- Levels: 0=off (default) 1=log slow queries 2=log all queries (general debugging feature for dev.)
- Mongo shell: db.getProfilingLevel() db.getProfilingStatus() db.setProfilingLevel(level, slowms)
- mongod --notablescan option: Set notablescan = true on your dev or test machine to find operations that require a table scan

1.4.8 mongotop & mongostat

- mongotop 3 // runs every 3 seconds showing you in which collection how much time (read, write, total) is spent
- mongostat // shows inserts, queries, updates, deletes, … per second
- → idx miss % = index which could not be accessed in memory

1.5 Week 5 – Aggregation

- group by like-functionallity
- db.[col].aggregate([

```
  {$group: {
    _id: ‘$manufacturer’,  → set the field ‘_id’ to the field you want to group by
    count: {$sum:1}        → define field ‘count’
  }}
])
```

1.5.1 Aggregation Pipeline

- Each document in the array parameter to the aggregate function is a stage in the pipeline
- E.g. Collection → project stage → match stage → group stage → sort stage → result
- Stages:
  - $project: Select relevant fields and reshape document (in: 1 / out: 1)
  - $match: Filters documents; (in: n / out: n-x)
  - $group: Aggregates; Reduces the number of documents (in: n / out: n-x)
  - $sort: Sorts the documents (in: 1 / out: 1)
  - $skip: Skips documents (in: n / out: n-x)
  - $limit: Limits returned documents (in: n / out: n-x)
  - $unwind: Explodes arrays - Produces a document for each value in an array-key-field with everything else repeated (in: n / out: n+x)
- Each stage can exist more than once in a pipeline

Example:

```
db.zips.aggregate([
{$match: {state: {$in: ["CA", "NY"]}}},
{$group: {_id: {city: "$city", state: "$state"}, pop: {$sum: "$pop"}}},
{$match: {pop: {$gt: 25000}}},
{$group: {_id: null, avg: {$avg: "$pop"}}}
])
```

1.5.2 Compound grouping

- Use a compound id: _id: {myManufacturer: “$manufacturer”, myCategory: “$category”}, …

1.5.3 Group stage

- $sum: Add one to a key (→ mySum: {$sum:1}) or sum up keys (→ sum_prices:{$sum:”$price”})
- $avg, $min, $max: Average, Minimum or maximum value of a key
- Create arrays: $push, $addToSet → categories: {$addToSet: “$category”}
- Only useful after a sort: $first, $last → {$group:{_id:"$_id.state", population:{$first:"$population"}}}

1.5.4 Project stage

- Remove keys - If you don't mention a key, it is not included, except for _id, which must be explicitly suppressed {$project: {_id: 0, …
- Add keys (also possible to create new subdocuments)
- Keep keys: {$project: {myKey: 1, …
- Rename keys / Use functions: $toUpper, $toLower, $add, $multiply

1.5.5 Match stage

- {$match: {pop: {$gt: 100000}}}

1.5.6 Sort stage

- Memory intensive; Can’t use index (at least after grouping)
- {$sort: {population: -1}}

1.5.7 Skip and limit stage

- Makes only sense when you do a sort first
- First skip – then limit (order of the stages in the pipeline matter)

1.5.8 Unwind stage

- {$unwind : “$tags”}

1.5.9 Limitations of the aggregation framework

- A result document can only be 16GB
- One can only use up to 10% of the memory on a machine
- In sharded environment: After first $group / $sort the next phase have to be performed on the mongos router
- Alternative to aggregation framework: map-reduce

SQL Comparison: http://docs.mongodb.org/manual/reference/sql-aggregation-comparison/

1.6 Week 6 - Replication

- You can only write to the primary node which replicates asynchronous to secondary nodes
- If you only read from the primary you will have strong consistency (default behaviour)
- You can allow your reads to go to secondaries → you might read stale data and have eventual consistency
- If the primary goes down the secondaries elect a new primary and the Java driver automatically connects to the new primary
- Arbiter nodes exist for voting purposes, e.g. if you have an even number of regular (= primary & secondary) nodes it can ensure a majority in an election → Allows you to have only two regular nodes.

- Delayed nodes are disaster recovery nodes: Can be set to be whatever time behind the regular nodes. Can't participate in elections
- Hidden node (e.g. for reporting) can't become the primary but can participate in elections
- Start a replication set: mongod -replSet m101 --logpath "1.log" --dbpath /data/rs1 -fork
- Register replica set nodes in the mongo shell:

```
config = { _id: "m101", members:[
        { _id : 0, host : "localhost:27017", priority:0, slaveDelay:5},
        { _id : 1, host : "localhost:27018"},
        { _id : 2, host : "localhost:27019"} ]
}
```

rs.initiate(config) ← Initializes the replica set // can't be executed on a node which can't become primary
rs.status() ← Gives you the status information about the replica set
rs.isMaster() ← Tells you if you are the primary node
rs.slaveOk() ← If issued on a secondary node it allows you to read from this secondary node
rs.stepDown() ← Forces primary node to step down as a primary node

- Replication is done via a capped collection called oplog.rs in the "local" database
- Secondaries ask the primary for any items since a certain timestamp

1.6.1 Failover and Rollback

Szenario:

- When the primary dies a secondary which becomes elected as a new primary which does not have the latest entries from the old primaries oplog
- When the former primary node comes back up as a secondary node it will request the oplog data from the new primary and roll back the writes the current primary does not have and write them to a rollback file which can be applied manually
- If the oplog of the new primary has looped during the time the old primary was down the entire dataset will be copied from the new primary
- The risk of losing data due to a rollback can be avoided by waiting till the majority of the nodes have the data → set the write concern w=majority

1.6.2 Connecting from the Java Driver

Provide a seed list to the MongoClient instance

```
new MongoClient(Arrays.asList((
        new ServerAddress("localhost", 27017),
        new ServerAddress("localhost", 27018),
        new ServerAddress("localhost", 27019),
)));
```

→ Will work even if the primary is not part of the seed list. The Java Client starts a background thread which pings all nodes from the seed list and all discovered nodes to find out which one is the primary

1.6.3 Write Concerns

Client writes to a primary:

1. Primary writes into RAM (collection and oplog)
2. The writes are asynchronously journaled (Gives recoverability in case of a crash)
3. The writes are written into the data directory
4. Secondaries are replicating writes from the primary's oplog

- The insert method sent from the Client does not expect a response
- The client sends a second command "getLastError"

- w=0 → Unaknowledged; Fast writes – no “getlastError” command
    - w=1 → Will wait for the primary to write into RAM
    - w=[n] → Will wait for the primary and n-1 to write into RAM
    - w=majority → Wait for majority of replica set to write to RAM
    - j=true → Will wait for the primary to write into the journal
    - fsync=true → Will wait for the primary to write into the data directory
    - wtimeout=[milliseconds] → Indicate how long you are willing to wait
- Java Driver
    - Default: w=1 and wtimeout=0 (=infinit)
    - client|db|collection.setWriteConcern(WriteConcern.JOURNALED)
    - collection.insert(doc, WriteConcern.JOURNALED)
    - WriteConcern.JOURNALED = new WriteConcern([w=]1,[wtimeout=]0, [fsync=]false,[j=]true)

1.6.4 Read Preferences

- Primary →All reads are send to the primary (default to guarantee strict consistency)
- Secondary → Send reads to randomly selected secondaries, but not to the primary
- Secondary Preferred → Send reads to secondaries or to the primary if all secondaries are down
- Primary Preferred → Sends reads to primary or to a secondary if primary is down
- Nearest →Send reads to secondary or primary
- If you read from secondaries you might get stale reads. The might be OK if different applications do the writes and the reads
- For any read preference (except Primary) the driver will look at ping times and will only send reads to nodes which are within the latency window (15ms) of the fastest
- client|db|collection.setReadPreference(ReadPreference.primaryPreferred())
- collection.find().setReadPreference(ReadPreference.nearest());

1.7 Week 6 – Sharding

- Enables horizontal scalability
- Shards are typically itself replica sets
- mongos is the sharding router which distributes data to the individual shards
- The application (and also the mongo shell) connects to mongos instead of mongod
- There can be multiple mongos - mongos typically runs on the same server as the application
- If a mongos goes down the application will connect to a different one – similar to replica sets
- Shard key determines to which shard a document goes
- Sharding is at database level – but you can define if you want to shared or not shard a specific collection
- Config servers (which are mongod) keep track of where the shards are – in production you typically use 3 of them

1.7.1 Building a sharded environment

Set up two shards each a replica set of three mongod nodes

- Set up shard as a replication set:
- mongod --replSet s0 --logpath "s0-r0.log" --dbpath /data/shard0/rs0 --port 37017 --fork –shardsvr
- Set up config server:
- mongod --logpath "cfg-a.log" --dbpath /data/config/config-a --port 57040 --fork –configsvr
- Set up mongos router with information about the config servers:
- mongos --logpath "mongos-1.log" --configdb localhost:57040,localhost:57041,localhost:57042 –fork
- mongos now listens on the default mongod port
- On the mongo shell – tell the config servers (via the mongos) about the shards:
- db.adminCommand({ addshard : "s0/"+"localhost:37017" });

- … add further shards
- db.adminCommand({enableSharding: "test"}) → enable sharding on test DB
- db.adminCommand({shardCollection: "test.grades", key: {student_id:1}}); → shard collection “grades” with the shard key “student_id”
- sh.help() → Will display all the shard commands available in the mongo shell, e.g. sh.status()

1.7.2 Sharding implications

- Needs an index on first element of the shard key (can be compound but not multi-key)
- Each document needs the shard key
- The shard key is immutable
- On an update you need to specify the shard key or specify multi: true
- A find with no shard key will go to all shards
- The key used in most queries should be the shard key
- You can’t have a unique index unless it is part of/starts with the shard key
- Write concerns are still important in a sharded setup

1.7.3 Sharding key

- Sufficient cardinality (variety of values)
- Avoid monotonically increasing keys to avoid hotspotting in writing (e.g. order_id, order_date)
- Compound sharding key is possible