MongoDB

- leading noSQL database
- *document oriented*
- Main concepts: **collection** (group of documents) and **document**
- **IMPORTANT** Collections do not enforce schema (documents inside collection can have dif. schema)
- Document: set of key-value pairs

RDBMS	MongoDB
database	database
Table	Collection
Tuple/row	Document
Column	Field
Table join	Embedded documents
Primary key	Primary key (default _id by database)

Document example:

```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by: 'tutorials point',
   url: 'http://www.tutorialspoint.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100,
   comments: [
      {
         user:'user1',
         message: 'My first comment',
         dateCreated: new Date(2011,1,20,2,15),
         like: 0
      },
      {
         user:'user2',
         message: 'My second comments',
         dateCreated: new Date(2011,1,25,7,45),
         like: 5
      }
   ]
}
```

When to use MongoDB ?

//TODO

1. Running database: `mongod.exe --dbpath "d:\set up\mongodb\data"`
2. Running db client: `mongo.exe`

Best Practices:

1. Design your schema according to user requirements.
2. Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
3. Duplicate the data (but limited) because disk space is cheap as compare to compute time.
4. Do joins while write, not on read.
5. Optimize your schema for most frequent use cases.
6. Do complex aggregation in the schema.

TODOS

// data types
// INDEXING
// Replication support: Replica set, primary node, secondary nodes Sharding - similar to replication but more than one primary node, faster response Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

COMMANDS

1. Create new database (if doesn't exist): `use dbName`
2. Check currently selected database: `db`
3. Show all databases: `show dbs`
4. Insert document: `db.COLLECTION_NAME.insert({"name":"HELLO WORLD"})`
   Insert multiple docs: `db.COLLECTION_NAME.insert[{...}, {...}]`
5. Drop database (first you need to select db): `db.dropDatabase()`
6. Create Collection: `db.createCollection(name, options)` Create Collection options: *capped*, *autoIndexId*, *size*, *max*
7. Drop collection: `db.COLLECTION_NAME.drop()`
8. **save** vs. **insert** - if you don't pass *_id* , save work's the same like insert. With *_id* passed it overrides existing document
9. Quering DB (pretty is optional): `db.COLLECTION_NAME.find().pretty()` Find only one: `db.collection_name.findOne()`
10. Update document: `db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)` By default: only one document is updated. Use `{multi: true}` for multiple
11. Remove document: `db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)`
    Only one flag : `db.COLLECTION_NAME.remove(DELLETION_CRITTERIA, 1)`
    Remove all documents: `db.COLLECTION_NAME.remove()`
12. **Projection** (Selecting only necessary fields from document)
    `db.COLLECTION_NAME.find({},{KEY_NAME:1})` : **1** to show, **0** to hide
13. Limit: `db.COLLECTION_NAME.find().limit(NUMBER)`

14. Skip (number of docs from the beginning): `db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)`
15. Sort: `db.COLLECTION_NAME.find().sort({KEY_NAME:1})` : **1** ascending, **-1** descending
16. Aggregate example: `db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}])`

DB QUERYING

Operation	**Syntax**	**Example**	**RDBMS Equivalent**
Equality	{key:value}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{key:{$lt:value}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{key: {$lte:value}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{key: {$gt:value}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{key: {$gte:value}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{key: {$ne:value}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes !=

** AND (OR) SYNTAX **

```
>db.mycol.find(
   {
      $and: [
         {key1: value1}, {key2:value2}
      ]
   }
).pretty()
```

Examples:

- Example 1 - ***starts with* example** - Find all users with the names that start with **Mi**
  `db.users.find({name: /^Mi/})`
  NOTE queries can take regex!

- Example 2 - **QUERY EXAMPLE**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
   {"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
```

- Example 3: **UPDATE EXAMPLE**

```
>db.mycol.update({'title':'MongoDB Overview'},
   {$set:{'title':'New MongoDB Tutorial'}},{multi:true})
```

- Example 4:

```
var ages = [20, 21];
db.users.find({age: {$in: ages}})
```

Aggregation

The aggregation pipeline is a framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into aggregated results. `db.collection.aggregate( [ { <stage> }, ... ] )`

For example, the following is the same as **find()**: `db.users.aggregate([{$match:{name: /^Mi/}}])` **There can be multi-stages that document enters sequentially (similar to data streams in JS)

example: sum

example: **$lookup** performs something like left join in sql

```
db.orders.aggregate([{
    $lookup: {
        from: "users",
        localField: "user_id",
        foreignField: "_id",
        as: "userInfo"
    }
}]).pretty()
```

It returns:

```
{
        "_id" : ObjectId("5aada355274d0c2736116d52"),
        "orderedAt" : ISODate("2018-03-17T23:23:01.915Z"),
        "price" : 150,
        "user_id" : ObjectId("5aada210274d0c2736116d4d"),
        "userInfo" : [
                {
                        "_id" : ObjectId("5aada210274d0c2736116d4d"),
                        "name" : "Petko",
                        "salary" : 1000,
                        "city" : "Krusevac"
                }
        ]
}
```

There is orders collection - how to sum all orders that are above 150?

```
db.orders.aggregate([{
    $match: {
        price: {
            $gte: 150
        }
    }
}, {
    $group: {
        _id: "test", #or it can be $someField
        total: {
            $sum: "$price"
        }
    }
}])
```

Relationships between documents

- two approaches: **Embedded** and **Referenced**
- Relationships can be: 1:N, N:1, 1:1 or N:N
- Embedded example: `{_id: ObjectId(...), name: "Milos", addresses: [{city: "BGD", street: "Juzni Bulevar"}]}`
- Referenced example: `{_id: ObjectId(...), name: "Milos", addresses: [ObjectId(...), ObjectId(...)]}`
- Querying in referenced model:

  ```
  >var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
  >var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
  ```

- **DBRefs vs Manual References**
  Manual Reference (^^ example). Imagine there were different collections like home_addresses, work_addresses etc.
  We don't know where to look!

- **DBRef example**

```
{
   "_id":ObjectId("53402597d852426020000002"),
   "address": {
     "$ref": "address_home",
     "$id": ObjectId("534009e4d852427820000002"),
     "$db": "tutorialspoint"  // if collection is from different db
   },
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin"
}
```