mongoDB

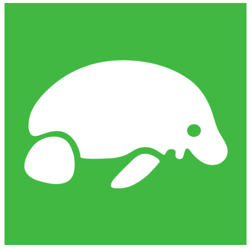
Advanced topics in NoSQL databases
ESILV - A4 IBO

Introduction mongoDB

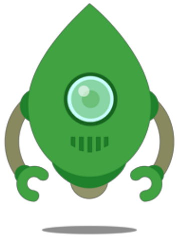
- Distributed NoSQL database
  - Large data management (Hu**mongo**us)
  - Document-oriented NoSQL
    - JSON documents
    - (BSON Objects serialization)
- Key points:
  - Simple to integrate for developers
  - Rich query language
    - MQL: Mongo Query Language
  - Several optimization features
    - Attributes indexing (Shard/Btree/RTree)
  - Smart data allocation
    - Sharding (GridFS) [+ tagging / clustering]
  - Several deployement
    - Private/Public Cloud, Local (On Premise)

Interactions: JavaScript

- JS objects
  - Attributes + functions
  - *db*: database
  - *sh*: sharding
  - *rs*: replica set
- MQL
  - "JSON" = pattern
- MapReduce
  - Functions (untyped)



Studio3t



Robo3t

Starting Commands

- **Select database (shell)**
  - Command: → use **myBD** ;
- **Collections** of documents
  - Create: → db.createCollection(‘***users***’);
  - Manipulate: → db.**users**. <command> ;
    - Commands : find(), save(), delete(), update(), aggregate(), distinct(), mapReduce()…
    - Meaning in SQL: *FROM*
- **Documents**
  - JSON:

```
{
        "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
        "name": "James Bond", "login": "james", "age": 50,
        "address”: {"street": "30 Wellington Square", "city": "London”},
        "job" : ["agent", "MI6"]
}
```

  - Insert: → db.users.save (   ); //no quotes!

MQL: find queries[1] (1/2)

- Document oriented queries
- Command: → db.users.**find**(<filter> , <projection>);
- **Filter**:
  - "JSON" pattern that must be *matched* on the document
  - "Key/Value" format
  - Can embed exact matching, operations, nesting, arrays
    - Operations: **$**op (no quotes)
  - Meaning in SQL: *WHERE*
- **Projection**:
  - "Key/Value" pairs that are given in output
  - Meaning in SQL: *SELECT (no aggregates)*
- Example:

  → db.users.find({"login" : "james"} , {"name" : 1, "age" : 1});

[1] - find queries are simple queries for your report

MQL: find queries (2/2)

- Exact match:

  → db.users.find({ “login” : "james" } , {“name” : 1, “age” : 1});

- With nesting "." for the path to the given key

  → db.users.find({ **“address.city”** : "London" });

- Operations

  → db.users.find({ “age” : **{ $gt : 40 }** });

  //$gt, $gte, $lt, $lte, $ne, $in, $nin, $or, $and, $exists, $type, $size, $cond…

- Regular expressions[1]

  → db.users.find({ ”name” : {**$regex** : “james”, **$options** : “i" }});

- Arrays

  → db.users.find({ ”job” : “MI6”}); //Check in the list

  → db.users.find({ ”job.1” : “MI6”}); //2nd position in the list

  → db.users.find({ ”job” : [“MI6”]}); //List exact match

[1] - regex : https://docs.mongodb.com/manual/reference/operator/query/regex/

MQL: Distinct - Count

- List of distinct values from a key

```
→ db.users.distinct( “name” );
→ db.users.distinct( “address.city” );
```

- Number of documents

```
→ db.users.count();
→ db.users.find( { “age” : 50 }).count();
```

MQL: pipeline[2] (1/3)

- **aggregate() :** Ordered sequence of operators ***aggregation pipeline***
- Command:

  → db.users.**aggregate([{$op1 : {}}, {$op2 : {}}, …]);**
- **Operators**:
  - $match : simple queries //meaning in SQL: WHERE
  - $project : projections //meaning in SQL: SELECT
  - $sort : sort result set //meaning in SQL: ORDER BY
  - $unwind : normalization in 1NF
  - $group : group by value + aggregate function // meaning in SQL: GROUP BY + fn
  - $lookup : left outer join (since 3.2 – work locally)
  - $out : store the output in a collection (since 3.2)
  - $geoNear : sort by nearest points (lat/long)
  - …

[2] – Pipeline queries (aggregate) are complex queries. For long pipelines, it can be hard queries

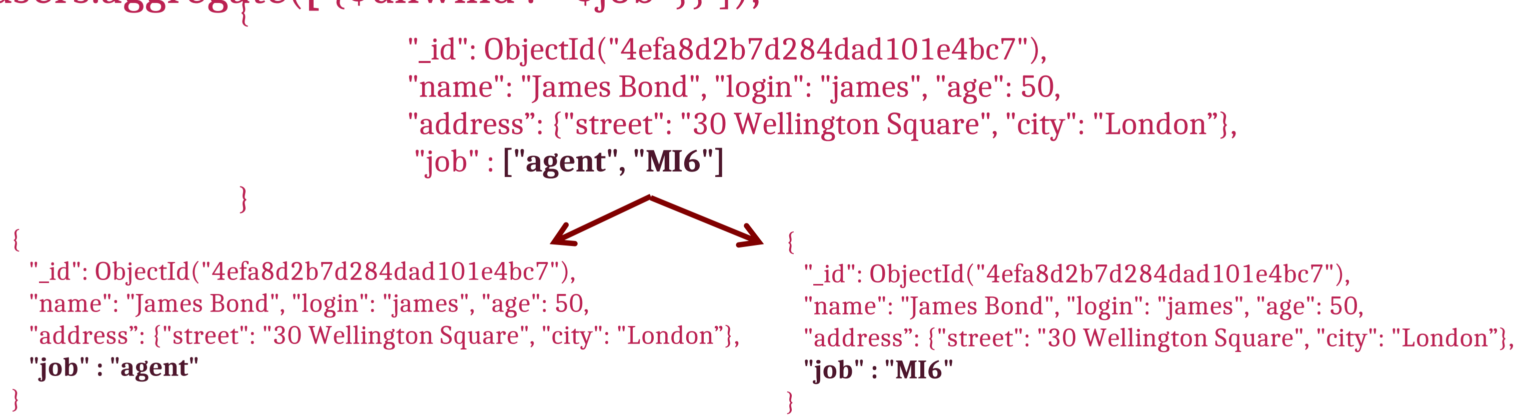
MQL: pipeline (2/3)

- ***Pipeline*** : Each step (operator) gives its output to the following operator (input)

```
> db.users.aggregate([{$match : {"address.city" :  "London"}},
                                        {$project : {"login" : 1, "age" : 1}},
                                        {$sort : {"age" : 1, "login" : -1}}
         ]);
```

- **$unwind**
  - Unnest an array and produce a new document for each item in the list

```
> db.users.aggregate([ {$unwind :  "$job"}} ]);
```



```
{
"_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
"name": "James Bond", "login": "james", "age": 50,
"address": {"street": "30 Wellington Square", "city": "London"},
 "job" : ["agent", "MI6"]
}
```

```
{
  "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
  "name": "James Bond", "login": "james", "age": 50,
  "address": {"street": "30 Wellington Square", "city": "London"},
  "job" : "agent"
}
```

```
{
  "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
  "name": "James Bond", "login": "james", "age": 50,
  "address": {"street": "30 Wellington Square", "city": "London"},
  "job" : "MI6"
}
```

MQL: pipeline (3/3)

- **$group :** key (**_id**) + aggregate ($sum / $avg / …)

  No grouping key: *only one output*

  → db.users.aggregate([{$group : {"_id" : "**age**", "res": {$sum : 1}}}]);

  Group by value: ***$**key*

  → db.users.aggregate([{$group:{"_id" : "**$age**", "res": {$sum : 1}}}]);

  Apply an aggregate function: ***$**key*

  → db.users.aggregate([{$group:{"_id":"$address.city", "avg": {$avg: "**$age**"}}}]);

- Sequence **example**

```
> db.users.aggregate([
        {$match:    {"address.city" : "London"} },
        {$unwind : "$job" },
        {$group :   {"_id" : "$job", "val": {$avg: "$age"}} },
        {$match :   {"val" : {$gt : 30}} },
        {$sort :      { "val" : -1} } ]);
```

MQL: updates

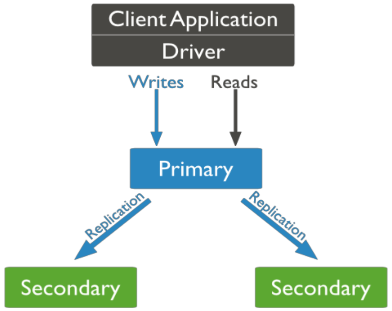
```
> db.users.update (
        { "_id" : ObjectId("4efa8d2b7d284dad101e4bc7") },
        { "$inc" : { "age" : 1 } } );
```

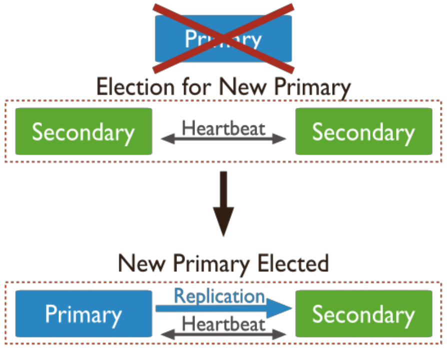
- Atomic updates (one document)
  - $set – modify value
  - $unset – remove a key/value
  - $inc – Increment
  - $push – Add inside an array
  - $pushAll – Add several values
  - $pull – Remove a value in an array
  - $pullAll – Remove several values

UpdateAll – all documents are updated

Replica Set

- Data replication
  - Asynchronus
    - Primary server: writes
    - Secondary servers: reads
  - Fault tolerance
    - New primary server election
    - Needs an arbiter
- Consistency vs availability
  - Reads applied on the primary (by default)
  - Updates via oPlog (log file)





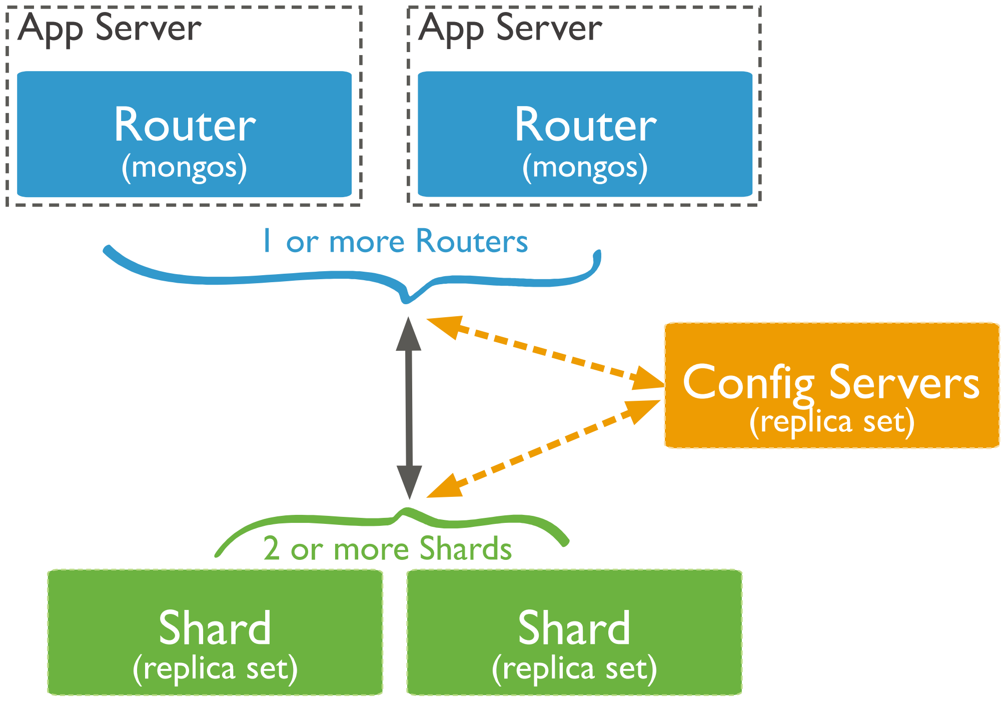
Sharding

- Data distribution on a cluster
- Data balancing
  - Sharding key choice: begining
- 1 Shard = 1 Replica Set
- Min config:
  - *Config Servers (x3)*
  - *Mongos* (router x3)
- Partitioning key (sharding)
  - Ranged-based (GridFS : sort)

    → sh.shardCollection("myBD.users", "login");
  - Hash-based (md5 sur clé)

    → sh.shardCollection("myBD.users", {"_id":"hashed"})
  - By zones/tags (+ sharding)

Sharding & Replica Sets



Indexing

- Create a **BTree**

  → db.users.createIndex({"age":1}) ;

  - All queries on « age » become more efficient
    - Even for Map/Reduce with « queryParam »
    - Execution plan « .explain() »
  - No combination of indexes

Indexing: 2DSphere

Apply geolocalized queries

→ db.users.ensureIndex({ "address.location" : "2dsphere" });

- Localization format

```
"location":{"type": "Point", "coordinates" : [51.489220, -0.162866]}
```

- Spherical queries

```
var near = {$near: {$geometry:{"type":"Point","coordinates":[51.489220,-0.162866]},
                                        $maxDistance : 10000}};
db.users.find( {"address.location":near}, {"name":1, "_id":0});
```

- $geoNear operator (*aggregate*)

```
var geoNear = {"near":{"type":"Point", "coordinates" : [51.489220,-0.162866]},
            "maxDistance":10000, "distanceField" : "outputDistance", "spherical":true};
db.users.aggregate([ {"$geoNear": geoNear} ]);
```

- « polygonales » queries

```
var polygon = {$geoWithin : {$geometry : {"type" : "Polygon", "coordinates" : [ [P1], [P2], [P3], [P1] }}}
```

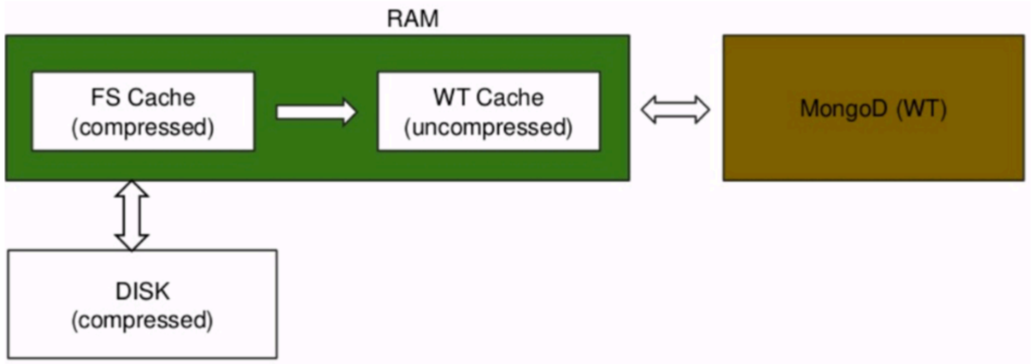
http://docs.mongodb.org/manual/tutorial/query-a-2d-index/

Storage Engine

- **Wired Tiger**
  - By default since 3.2
  - Created for *BerkeleyDB*, used for Oracle NoSQL
  - Concurrency purpose: multi-version
- MMAPV1
  - Original engine
  - Mapping in main memory
  - Lock on collections (long writes)
- Encrypted storage
  - Since 3.2
  - Based on Wired Tiger
  - KMIP (Key Management Interoperability Protocol)
- In Memory
  - Since 3.2.6
  - Based on Wired Tiger
  - Must fit in main memory (error : *WT_CACHE_FULL*)

Wired Tiger – Cache system

- 50% RAM – 1GB (or 256MB)
- Data are compressed in main memory



Drivers / API

- Drivers: http://docs.mongodb.org/ecosystem/drivers/
  - Python, Ruby, Java, Javascript (Node.js), C++, C#, PHP, Perl, Scala…
  - Syntax: http://docs.mongodb.org/ecosystem/drivers/syntax-table/

Driver Java

Connection

```
    ServerAddress server = new ServerAddress ("localhost",
    27017);
    Mongo mongo = new Mongo(server);
```

Environment (DB + collection)

```
    DB db = mongoClient.getDB( "maBD" );
    DBCollection users = db.getCollection("users");
```

Authentification

```
    db.authenticate(login, passwd.toCharArray());
```

Document inserting (DBObject)

```
    DBObject doc = new BasicDBObject("name", "MongoDB")
     .append("type", "database")
     .append("count", 1)
     .append("info", new BasicDBObject("x",
              203).append("y", 102));
// or doc = (DBObject) JSON.parse(jsonText) ;
    users.insert(doc);
```

Querying (pattern query + cursor)

```
    BasicDBObject query = new
              BasicDBObject("name", "James Bond");
    DBCursor cursor = coll.find(query);
    try {
      while(cursor.hasNext()) {
           System.out.println(cursor.next());
        }
    } finally {
        cursor.close();
    }
```

Map/Reduce :

```
    MapReduceCommand cmd =
              new MapReduceCommand("users", map,
                        reduce, "outputColl",
              MapReduceCommand.
              OutputType.REPLACE, query);
    MapReduceOutput out = users.mapReduce(cmd);
    for (DBObject o : out.results()) {
              System.out.println(o.toString());
    }
    //outputType : INLINE, REPLACE, MERGE, REDUCE
```

http://api.mongodb.org/java/current/index.html?_ga=1.177444515.761372013.1398850293

Driver C#

References/Libraries

```
MongoDB.Bson.dll
        using MongoDB.Bson;
MongoDB.Driver.dll
        using MongoDB.Driver;
```

Connection

```
var connectionString = "mongodb://localhost";
var client = new MongoClient(connectionString);
var server = client.GetServer();
```

Database & Collection

```
var db = server.GetDatabase("maBD");
var coll = db.GetCollection<User>("users");
```

Objet « User » to define

```
public class User{
   public ObjectId Id { get; set; }
   public string name { get; set; }
   public string login { get; set; }
   public int age { get; set; }
   public Address address { get; set; }
}
```

Get a document :

```
var query=Query<User>.EQ(e=>e.login,"james");
var entity = coll.FindOne(query);
```

Package for output results: LINQ

```
using MongoDB.Driver.Linq;
```

Queries:

```
var query = coll.AsQueryable<User>()
                          .Where(e => e.age > 40)
                          .OrderBy(c => c.name);
```

Answer:

```
foreach (var user in query)
{
   // traitement
}
```

MapReduce :

```
var mr = coll.MapReduce(map, reduce);
foreach (var document in mr.GetResults()) {
   Console.WriteLine(document.ToJson());
}
```

http://www.nuget.org/packages/mongocsharpdriver/

Driver Python

Importation

```
>>> import pymongo
```

Connexion

```
>>> from pymongo import MongoClient
>>> client = MongoClient()
>>> client = MongoClient('localhost', 27017)
```

Database & collection

```
>>> db = client.maBD
>>> coll = db.users
```

GET one document

```
coll.find_one ( { "login" : "james"} )
```

Query

```
>>> for c in coll.find ( {"age": 40} ) :
…              pprint.pprint (c)
```

MapReduce

```
>>> from bson.code import Code
>>> map = Code("function () {"
... " this.tags.forEach(function(z) {"
... " emit(z, 1);"
... " });"
... "}")
>>> reduce = Code("function (key, values) {"
... " var total = 0;"
... " for (var i = 0; i < values.length; i++) {"
... " total += values[i];}"
... " return total; } ")
>>> result = coll.map_reduce(map, reduce,
              "myresults")
>>> for doc in myresults.find():
...      print doc
```