Nicolas Travers
Conservatoire National des Arts et Métiers

Introduction

- Hu*mongo*us (monstrous / enormous)
- NoSQL: Documents Oriented
  - JSon
  - Serialized format: BSon objects
  - Implemented in C++
  - Keys indexing (BTree + 2DSphere)
  - Replication (*Replica Set*) + Distribution (*Sharding*)
  - Storage: *GridFS*
- Used by:
  - *Doodle, SAP, sourceforge, NY times, bit.ly, github, foursquare, EA games, grooveshark*
  - License AGPL (Apache)

Basic commands

- **Connection to database**
  > use **myDB** ;
- **Collections** of documents
  - Creation: > db.createCollection('***users***');
  - Usage: > db.**users**. <commande> ;
    - Functions : **find**(), save(), delete(), update(), **aggregate**(), distinct(), mapReduce()…
    - Equivalent to "*FROM*" in SQL
- **Documents**
  - JSon :

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

  - Saving : > db.users.save (↓); //no quotations
  - **No** schema

Queries: *find* (1/2)

- Document oriented queries
- Command: > db.users.**find**(<filter> , <projection>);
- **Filter**:
  - "JSon" pattern contained into targeted documents
  - Key/Value format
  - Can contain operations, arrays, nesting
    - operation: **$**op (no quotes)
  - Equivalent to "*WHERE*" in SQL
- **Projection**:
  - Key/Value which must be present
  - Equivalent to "*SELECT*" in SQL *(only attributs)*
- Example:

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

Queries: *find* (2/2)

- Exact matching

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

- Nesting match

  > 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 matching

  > db.users.find({ ”job” : “MI6”}); //dans la liste

  > db.users.find({ ”job.1” : “MI6”}); //2° place de la liste

  > db.users.find({ ”job” : [“MI6”]}); //recherche exacte

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

Queries: Distinct - Count

- Distinct values from a key

  > db.users.distinct(“name”);

  > db.users.distinct(“address.city”);

- Count on a collection of documents

  > db.users.count();

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

Queries: pipeline (1/3)

- **aggregate() :** *Ordered sequence* of operators
  pipeline aggregate
- Command:
  > db.users.**aggregate([{$op1 : {}}, {$op2 : {}}, …])**;
- **Operators**:
  - $match : simple matching //equivalent to: where
  - $project : Keys projection //equivalent to: select
  - $sort : sort collection on keys //equivalent to: order by
  - $unwind : normalization to 1NF
  - $group : aggregate + function //equivalent to: group by + fn
  - $lookup : left outer join (since 3.2)
  - $out : result storage (depuis 3.2)
  - $geoNear : compute geolocalisation distance (lat/long)
  - $redact : conditional pruning (nested documents)
  - + $sample, $limit, $skip,

Queries: pipeline (2/3)

- ***Pipeline*** : each operator output is the input of the following

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

- **$unwind**
  - Create a document for each instance of an array

```
> 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"
}
```

Queries: pipeline (3/3)

- **$group :** key (**_id**) + aggregate function ($sum / $avg / …)
  No grouping value: *null* (count all instances)
  > db.users.aggregate([{$group : {"_id" : **null**, "res": {$sum : 1}}}]);
  Grouping by value: *$key*
  > db.users.aggregate([{$group:{"_id" : "**$age**", "res": {$sum : 1}}}]);
  Average: *$key*
  > db.users.aggregate([{$group:{"_id":"$address.city", "moy": {$avg: "**$age**"}}}]);
- **Sequence example**

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


Indexing

- **Default sharding**
  - Clustered BTree on « _id » (sort)
  - Change the sort:
    - Only one unique index
      > db.users.createIndex ({ "login" : 1 }, { "unique" : true }) ;
- **Hashed** Sharding
      > db.users.createIndex({ "_id" : "hashed" }) ;
  - Consistent hashing: DHT
- Secondary indexes: **BTree**
      > db.users.createIndex({"age":1}) ;
  - More efficient matches
  - No combination of indexes

Updates

- No transactions
- Updates on the entire document
  - $set – update value (or create key)
  - $unset – delete key
  - $inc – Increment
  - $push – Push in an array
  - $pushAll – push several values
  - $pull – delete from an array
  - $pullAll – delete several values

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


MongoDB utils

- In “bin” folder
  - *mongo* : shell for commands
  - *mongod* : server process (daemon)

- **Robomongo**
  - *UI for queries*
  - https://robomongo.org