Models and Systems for Big Data

DOCUMENT DATABASE - MONGODB

Nacéra Seghouani

Computer Science Department, CentraleSupélec
Laboratoire de Recherche en Informatique, LRI LaHDAK group
nacera.bennacer@centralesupelec

2019-2020

Introduction

- ☞ JSON (JavaScript Object Notation) is used as a data model in NoSQL document-based databases such as MongoDB, CouchDB, CouchBase, RethinkDB
- ☞ JSON: a format to exchange structured data in a distributed environment, used in Ajax Web applications and in Rest architecture based Web services.
- ☞ JSON: simple and intuitive, it hasn't schema specification and query language (generally Javascript)
- ☞ A JSON document is a set of key-value pairs (attribute, value).
  - ✎ Simple values

    ```
    "title": "The Social network",
    "year": 2010,
    "oscar": false
    ```

  - ✎ Complex values

    ```
    "artist": {"last_name": "Fincher", "first_name": "David"},
    "artists": [{"first_name": "Jesse", "last_name":"Eisenberg"},
                {"first_name": "Rooney",  "last_name": "Mara"}]
    ```

JSON Data Model

- Documents are characterized by their structure which can be more or less complex.
- A document has a tree structure: nodes contain simple values (strings, integers, reals, booleans) or complex values (arrays of simple values or embedded documents). Simple values are the tree leaves.

```
{
"title": "The Social network",
"summary": "On a fall night in 2003, Harvard undergrad and programming  ...",
"year": 2010,
"director": {"last_name": "Fincher", "first_name": "David"},
"actors":  [
            {"first_name": "Jesse", "last_name":"Eisenberg"},
            {"first_name": "Rooney",  "last_name": "Mara"}
           ]
}
```

JSON Data Model

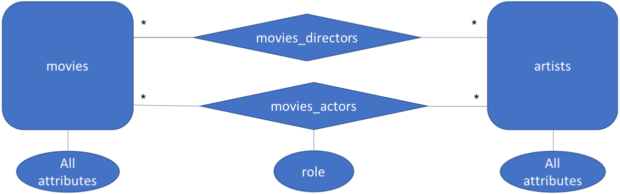
☞ Powerful modeling: attributes with no atomic values, embedded/nested documents

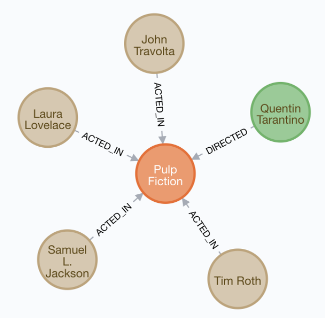
```
[
{"title": "The Social network",
"summary": "On a fall night in 2003, Harvard undergrad and programming  ...",
"year": 2010,
"director": {"last_name": "Fincher", "first_name": "David"},
"actors":  [{"first_name": "Jesse", "last_name":"Eisenberg"},
{"first_name": "Rooney",  "last_name": "Mara"}] }
{"title": "Pulp fiction",
"year": "1994",
"genre": ["Action", "Crime", "Comedy"]
"country": "USA"
"director": {"last_name":"Tarantino","first_name":"Quentin","birth_date": "1963"},
"actors":[{"last_name":"Travolta","first_name":"John","role":"Vincent Vega"},
{"last_name":"Willis","first_name":"Bruce","role":"Butch Coolidge" },
{"last_name":"Jackson","first_name": "Samuel L.","role":"Jules Winnfield"},
   ...]
}
...]
```

JSON Data Model

- ☞ Documents in the same collection are not described using the same attributes
- ☞ Hierarchy leads to a non symmetric representation, the access to the root is privileged, and a certain perspective is imposed
- ☞ No information scattered among different documents likewise no need to join operations as in relational databases
- ☞ No autonomous entities, redundancy and perhaps inconsistencies.

JSON Data Model Versus Graph & Relational Model





Document Database MongoDB

- MongoDB (from hu**mongo**us) is a free and open-source platform database.
- A powerful, flexible, and scalable NoSQL general-purpose database used in platforms such as: *Craigslist, eBay, Foursquare, SourceForge.net, Viacom, and New York Times*.
- A document-oriented database
  - a row is a document, with embedded documents and arrays, complex hierarchical relationships in a single row
  - a collection is a group of documents, as a table with a dynamic schema
  - free schema: documents within a collection can have any number of different attributes, not of fixed types or sizes
- Flexible structure (presence, absence, variations), complex and free imbrication.
- Few reference to other documents.
- Transaction in MongoDB . . . and data distribution -> see later

Data Types

- Each document has a unique _*id* within a collection

```
[
{_id: 1,
name: "sue",
age: 19,
gender : 2,
favorites: { artist: "Picasso", food: "pizza"},
badges: [ "blue", "black" ]
}
{_id: 2,
name: "john",
age: 21
}
]
```

- Data types:

```
regular expression {"x" : /foobar/i}
array {"x" : ["a", "b", "c"]} {"things" : ["pie", 3.14]}
embedded document {"name":"John Doe","address":
{"street":"123 Park Street","city":"AnyTown","state":"NY"} }
javascript code {"x" : function() { /* ... */ }}
```

Basic Operations

- MongoDB comes with a simple but powerful JavaScript shell
  - insert a blog in `blog` collection (created if it doesn't exist) located in current database `db`. `\_id` field is added to each document
  - query the collection

```
> post = {
"title": "My Blog Post",
"content": "Here's my  post",
"date" : new Date()}

> db.blog.insert(post)

> db.blog.find()
  { "_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
  "title" : "My Blog Post",
  "content" : "Here's my blog post.",
  "date" : ISODate("2017-04-24T21:12:09.982Z")
  }
```

Basic Operations

- ☞ modify the variable post, add a "comments" attribute

```
> post.comments = []

> db.blog.update({title : "My Blog Post"}, post)

> db.blog.find()
 {
        "_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
        "title" : "My Blog Post",
        "content" : "Here’s my blog post.",
        "date" : ISODate("2012-08-24T21:12:09.982Z"),
        "comments" : [ ]
}
```

- ☞ remove documents with the specified attributes values from a collection

```
> db.blog.remove({title : "My Blog Post"})
```

Querying

- ☞ Selection: the first parameter of `find()` specify a set of conditions `{field_i: val_i, field_j.field_k: val_{jk}, ... }` on different attributes as conjunctive conditions

```
db.users.find()
// all documents in users collection

db.restaurants.find( { "borough": "Manhattan"})
//documents with  attribute "borough" and value "Manhattan"

db.users.find({"username" : "joe", "age" : 27})
 //documents where "username" : "joe" and "age" : 27

db.users.find({"username" : "joe", "address.city" : "Paris"})
db.restaurants.find({"cuisine":"Italian", "address.zipcode":"10075"})
//condition with embedded document
```

Querying

☞ Projection: the second parameter of `find()`
`{field_i: 1, field_j: 0, ... }` 1 if projected, 0 otherwise.

☞ *_id* key is returned by default.

```
db.users.find({}, {"username" : 1, "email" : 1})

db.users.find({}, {"username" : 1, "email" : 1, "_id":0})

db.restaurants.find({"cuisine":"Italian"}{"address.zipcode": 1})
```

Querying

- Selection operators on a single attribute $\{field_i : \{\$operator : val_i\}, \ldots\}\}$
- $*ne* can be used with any type

```
db.users.find({"age":{"$gte" : 18, "$lte" : 30}})
//users who are between 18 and 30

start = new Date("01/01/2007")
db.users.find({"registered" : {"$lt" : start}})

db.users.find({"username" : {"$ne" : "joe"}})

db.restaurants.find( { "borough": "Manhattan", "grades.score": { $gt: 30 } })

db.restaurants.find( { "grades.grade": { $in: [ "A", "B" ] } })

db.raffle.find({"ticket_no" : {"$nin" : [725, 542, 390]} })
```

Querying

- Selection operators on multiple attributes
  $\{\$operator : [condition_i, condition_j, ...]\}$
- Two ways to do an `or` query
  - `$in` used to query for a variety of values for a single field.
  - `$or` used to query for any of the given values across multiple fields.

```
db.raffle.find({"$and" : [{"ticket_no" : 725}, {"winner" : true}] })

db.restaurants.find({$or:[{"cuisine": "Italian"}, {"address.zipcode": "10075"}] })
```

Querying arrays

- ☞ Querying for an array, for an element `i` of an array.

  `{field: [val_1, val_2, ...] }; {field: val}; {field.i: val}`

- ☞ Querying for arrays of a given size `$size`

```
db.food.insert({"fruit" : ["apple", "banana", "peach"]})

db.food.find({"fruit" : "banana"})//at least 1 of array elements matchs

db.food.find({"fruit":["apple","banana","peach"]})
//exact array match (including order)

db.food.find({$and : [{fruit:"apple"}, {fruit:"banana"}]})
//at least 2 of array elements are   "apple" and "banana"

db.food.find({"fruit.2" : "peach"}) //element with index 2 is  "peach"

db.food.find({"fruit" : {"$size" : 3}})
```

Querying arrays

- Querying for arrays that contain at least one element that matches all the specified query conditions (useful if more than one condition).

```
 {field: {$elemMatch : {condition1},{condition2}, ...} }
```

```
db.scores.find({ results: { $elemMatch: { $gte: 80, $lt: 85 } } })
```

- `$elemMatch` won’t match non-array elements.

```
>db.test.find()
{"x" : 5}{"x" : 15}{"x" : 25}{"x" : [5, 25]}
>db.test.find({"x" : {"$gt" : 10, "$lt" : 20}})
{"x" : 15}{"x" : [5, 25]}
>db.test.find({{"x" : {"$elemMatch" : {"$gt" : 10, "$lt" : 20}}})
// no results
```

Querying Arrays

☞ Embedded documents in Arrays

```
>db.blog.find()
{
 "content" : "...",
 "comments" : [
        {"author" : "joe", "score" : 3,"comment" : "nice post"},
        {"author" : "mary","score" : 6, "comment" :"terrible post"}
        ]
        }

>db.blog.find({"comments" : {"author" : "joe", "score" : {"$gte" : 5}}})
// no embedded document

>db.blog.find({"comments.author": "joe", "comments.score": {"$gte" : 5}})
// two embedded documents

>db.blog.find({"comments":{"$elemMatch":{"author":"joe","score":{"$gte" : 5}}}})
// no embedded document
```

Querying

- ✎ `$slice` operator can be used to return a subset of elements for an array attribute, also by taking an offset and the number of elements to return

```
db.blog.posts.find({}, {"comments" : {"$slice" : 10}}) \\the first 10

db.blog.posts.find({}, {"comments" : {"$slice" : -10}}) \\the last 10

db.blog.posts.find({}, {"comments" : {"$slice" : [23, 10]}})
//skip the first 23 elements and return the 24th through 33th
```

Cursors

```
> for(i=0; i<100; i++) db.collection.insert({x : i});

> var cursor = db.collection.find();
> while (cursor.hasNext()) {obj = cursor.next();}// do stuff
```

✎ Client-side cursors control the output of a query, limit the number of results, skip over some results

✎ When *find* is called, the client doesn't query the database immediately (only when the cursor start fetching),

✎ The cursor fetches the first 100 results or first 4 MB of results at once (by default) so that the next calls to *next*() or *hasNext*() will not have to make trips to the server.

✎ After the client has run through the first set of results, it will again ask the server for more results with a *getMore*() request (transparent).

Cursors

```
> for(i=0; i<100; i++) db.collection.insert({x : i});

> var cursor = db.collection.find();
> while (cursor.hasNext()) {obj = cursor.next();}// do stuff
```

- ✎ On the server side, a cursor takes up memory and resources. Once a cursor runs out of results or the client sends a message telling it to die, the database can free the resources.
- ✎ A cursor automatically dies when it finishes iterating through the matching results, or after 10 minutes of inactivity.
- ✎ Many drivers have implemented a function called immortal, which tells the database not to time out the cursor. If you turn off a cursor's timeout, you must iterate through all of its results or kill it to make sure it gets closed.

Cursors

- The most common query options are limiting the number of results returned, skipping a number of results, and sorting. All these options must be added before a query is sent to the database server.

```
> db.c.find().limit(3);
//only 3 matching documents are returned

> db.c.find().sort({username : 1, age : -1}).limit(3);
 //1 (ascending) or -1 (descending)

> db.c.find().sort({username : 1, age : -1}).skip(3);
//skip the first 3 matching documents
//If there are fewer than 3 documents, no results.

> var page1 = db.foo.find({}).limit(100)

> var page2 = db.foo.find({}).skip(100).limit(100)

> var page3 = db.foo.find({}).skip(200).limit(100) ...
```

Aggregation

- MongoDB provides aggregation capabilities using `aggregate`:
  `{$group: { _id: groupingFields, field1: {function : exp }, ... } }`

```
$group : {"_id" : "$day"}
$group : {"_id" : "$grade"}
$group : {"_id" : {"state" : "$state", "city" : "$city"}}
```

- Several aggregation operators as `$sum`

```
>db.restaurants.aggregate([{ $group: { "_id": "$borough", "count": {$sum:1}} }])

{ "_id" : "Staten Island", "count" : 969 }
{ "_id" : "Brooklyn", "count" : 6086 }
{ "_id" : "Manhattan", "count" : 10259 }
{ "_id" : "Queens", "count" : 5656 }
{ "_id" : "Bronx", "count" : 2338 }
{ "_id" : "Missing", "count" : 51 }
```

- `$match` filters documents so that you can run an aggregation on a subset of documents.

```
>db.restaurants.aggregate([
{ $match: { "borough": "Queens", "cuisine": "Brazilian" } },
{ $group: { "_id": "$address.zipcode" , "count": { $sum: 1 } } }  ])
```

Aggregation

- `$project$` and `$sort$` aggregated documents, to retrieve the five most prolific authors.

```
>db.articles.aggregate(
{$project : {"author" : 1}},
{$group : {"_id": "$author", "count": {"$sum" : 1}}},
{$sort : {"count" : -1}},
{"$limit" : 5})
//groups the authors by name, increments "count"
// for each document an author appears in.
```

- `$unwind` turns each attribute of an array into a separate document.

```
>db.blog.aggregate(
{$project : {"comments" : 1}},
{$unwind : "$comments"},
{$match : {"comments.author" : "Mark"}})
```