Introduction to databases

Lecture 6 – Document-oriented databases:MongoDB

Gianluca Quercini

gianluca.quercini@centralesupelec.fr

Master DSBA 2020 – 2021

CentraleSupélec

What you will learn

In this lecture you will learn:

- The MongoDB general concepts.
- How to model data in MongoDB.
- How to query data in MongoDB.
- How data distribution is implemented in MongoDB.

MongoDB general concepts

MongoDB

- General-purpose database system based on the **document data model**.
- **MongoDB Community**: open-source and free edition of MongoDB.
- **MongoDB Enterprise**: needs a subscription.

- A record in MongoDB is stored in a **document**.
  - A document is an **aggregate**.
- Documents are stored in **collections**.
  - A collection is similar to a relational table.
- A MongoDB **database** is a set of collections.

MongoDB characteristics

- **Impedance mismatch** reduction.
  - Documents are **JSON objects**.
  - One-to-one mapping to objects in programming languages.
- **Flexible schema**.
  - Documents in the same collections do not have to have the same fields.
- **Rich query language**.
  - Data aggregation.
  - Text and geospatial queries.
- **High availability**.
  - Data redundancy with **replication**.
  - Automatic failover.
- **Horizontal scalability**.
  - **Sharding** distributes data across several machines.
  - Support for the creation of **zones** of data.

Data modeling

- Data modeling in relational databases is guided by **normalization**.
- In MongoDB, data modeling can but does not have to follow normalization rules.

Data modeling criteria

- Consider the application usage of data (queries, updates).
- Consider the inherent structure of the data.

Flexible schema

Consider a **collection** of documents:

- Documents do not have to have the same fields.
- The data type for a field can differ across documents.

It is possible to specify **schema validation criteria** to make sure documents have a similar structure.

Data modeling

Denormalized data

- It is possible to **embed documents** in a MongoDB document.
- Denormalized data allow applications to retrieve and manipulate related data in a **single database operation**.

```
{
    "_id":"movie:1",
    "title":"Vertigo",
    "country":"DE",
    "director":{
      "_id":"artist:3",
      first_name: "Alfred",
      "last_name":"Hitchcock"
    }
}
```

Data modeling

```
{
    "_id":"movie:1",
    "title":"Vertigo",
    "country":"DE",
    "actors": [
      {
        "_id": "artist:15",
        "first_name": "James",
        "last_name": "Stewart",
        "role": "John Ferguson"
      },
      {
        _id: "artist:16",
        first_name: "Kim",
        last_name: "Novak"
      }
    ]
}
```

Data modeling

Normalized data

- Documents can store **references** to other documents.
- References are used instead of embedded documents.
- Used to **reduce data redundancy**.

Collection `movie`

```
{
  "_id":"movie:1",
  "title":"Vertigo",
  "country":"DE",
  "director": "artist:3"
}
```

Collection `artist`

```
{
  "_id":"artist:3",
  first_name: "Alfred",
  "last_name":"Hitchcock"
}
```

Data modeling

Denormalized data

- Ability to **retrieve related data** in a **single database operation**. ☺
- **Update** related data in a **single atomic write operation**.☺
- Data redundancy. ☹

Normalized data

- Useful when embedding would result in data redundancy with no or little improvement for read operations. ☺
- Useful to represent complex **many-to-many relationships**.☺
- Splits data across different documents (need for **join operations**). ☹

Data modeling

One-to-one relationship

- **Example.** One department has only one manager (and that person can only manage one department).
- Use an **embedded document**.

```
{
  "_id": "dept:1",
  "name": "Acconting",
  budget: 50000,
  manager: {
    "_id": "emp:1",
    "first_name": "John",
    "last_name": "Smith",
    "salary": 80000
  }
}
```

Data modeling

One-to-few relationship

- **Example.** The addresses of a person.
- Use an **embedded document**.

```
{
  "_id": "pers:1",
  "first_name": "John",
  "last_name": "Smith",
  addresses: [
    {street: "123 Sesame St", "city": "New York City", "country": USA},
    {street: "3 House Avenue", "city": "New York City", "country": USA}
  ]
}
```

☞ Difficult to find all people from New York City!

Data modeling

One-to-many relationship

- **Example.** A product is composed of several hundred replacement parts.
- Use **normalized documents**.

Collection `Product`

```
{
  "_id":"product:1",
  "name":"Smoke detector",
  "manufacturer": "SmokeSafety Inc.",
  "parts": ["part:345", "part:213"]
}
```

Collection `Part`

```
{
  "_id":"part:345",
  "partno": "123-aff-456",
  "cost": 0.94
}
```

☞ The same model can represent a **many-to-many relationship**.

Data modeling

One-to-squillions relationship

- **Example.** Log messages associated to a host.
- Each host might be associated to millions of log messages.
- Use **normalized documents**.

Collection Host

```
{
  "_id":"host:1",
  "name":"host.example.com",
  "ipaddr": "192.168.3.2"
}
```

Collection LogMessage

```
{
  "_id":"msg:1",
  "message": "CPU failure"
  "host": "host:1"
}
```

☞ Storing the messages in the host document might overflow the document size limit of 16MB.

Data modeling

Two-way referencing

- **Example.** We need to track **tasks** assigned to **people**.
- The application needs to retrieve the tasks assigned to a person.
- The application needs to get the person responsible for specific tasks.
- References are stored in both documents.

Collection `Person`

```
{
  "_id":"person:1",
  "name":"John Smith",
  "tasks": ["task:1", "task:5",
            "task:7"]
}
```

Collection `Task`

```
{
  "_id":"task:1",
  "description": "Budget finalization"
  "due_date":ISODate("2021-04-01"),
  "responsible": "person:1"
}
```

☞ Reassigning a task to another person entails two updates.

Data modeling

Half-way denormalization

- **Example.** Employees and the departments where they work.
- **Fully denormalized schema**: all properties of a department are embedded in an employee document.
- **Problem.** Updating the department budget can be **expensive**.

```
{
  "_id":"emp:1",
  "name":"John Smith",
  "salary": 50000,
  "position": "secretary",
  "department": {
    "_id": "dept:1",
    "name": "Accounting",
    "budget": 12000
  }
}
```

```
{
  "_id":"emp:1",
  "name":"Jennifer Young",
  "salary": 70000,
  "position": "director",
  "department": {
    "_id": "dept:1",
    "name": "Accounting",
    "budget": 12000
  }
}
```

Data modeling

Half-way denormalization

- **Solution**. Only denormalize the fields that are queried often together with the parent document.

Collection `Employee`

```
{
  "_id":"emp:1",
  "name":"John Smith",
  "salary": 50000,
  "position": "secretary",
  "department": {
    "_id": "dept:1",
    "name": "Accounting"
  }
}
```

Collection `Department`

```
{
  "_id":"dept:1",
  " budget": 12000
}
```

Data modeling

Atomicity

- If a single **write** operation modifies a single document:
  - The operation is **atomic**. . .
  - . . . even if the operation modifies **multiple embedded documents** within a **single document**.
- If a single **write** operation modifies multiple documents:
  - The modification of each document is **atomic**.
  - The operation as a whole is **not atomic**.

Distributed transactions

- MongoDB supports **multi-document distributed transactions**.
- Used in situations that require **atomic read/write** operations on **multiple documents**.
- MongoDB encourages data models that avoid the use of distributed transactions.

Data modeling

Indexes

- MongoDB automatically creates a **unique index** on the `_id` field.
- Indexes can be manually created for any field.
- An index speeds up **read operations**.

Indexes behaviour

- Each index requires at least 8kB of extra space.
- Write operations entail an update of the index.
  - Indexes should not be created for fields with a high **write-to-read** ratio.
- Indexes consume both disk and memory space.

Data types

- `boolean`: `true` and `false`.
- `number`: 64-bit floating point numbers.
- `string`: UTF-8 strings of characters.
- `date`: {birth_date : new Date("1899-08-13")}
- `array`.

Example

- {"fruits" : ["apple", "orange", "banana"]}
- {"fruits-mixed" : ["apple", 3, "orange", 5, "banana", 1]}

- `Embedded document`.
- `Binary data`.
- `Code`.
- `ObjectId`: a 12-byte ID for documents.
- `null`: used to represent a nonexistent field.

Document ID generation

When a process p creates a document d on a host h at time t, MongoDB generates its identifier as follows:

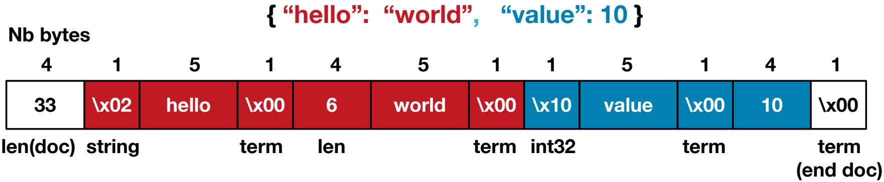
- `Timestamp`: number of seconds since the epoch (00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970) to t.
- `Hostname`: hash of the name of h.
- `PID`: identifier of p (as assigned by the operating system).
- `Counter`: to distinguish documents created by p on the same machine at the same time.

4 bytes	3 bytes	2 bytes	3 bytes
TIMESTAMP	**HOSTNAME**	**PID**	**COUNTER**

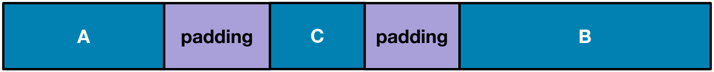
☞ To save space, specify your own document ID!

Document storage: BSON

- Documents are stored using the BSON (Binary JSON) format.
- Each document is a sequence of **bytes**.



- Documents are stored **one after the other** on disk.
- Some extra space (**padding**) is added to let a document grow.
  - Padding is proportional to the document's size.



MongoDB Server

- MongoDB is run as a **network server**.
- The server is started with the command `mongod`.
    - The default port number is 27017.
- MongoDB provides a **Javascript shell** to send commands to the server.
    - The shell is started with the command `mongo`.
- We use **MongoDB Compass** as a client for MongoDB.
- Any application can connect to a MongoDB server.
    - MongoDB provides different libraries (**drivers**) to connect applications to the MongoDB server.

Document Creation and insertion

Create database

```
use cinema;
```

Declare a document

```
artist = {"last_name": "Hitchcock",
  "first_name": "Alfred",
  "birth_date": "1899"
}
```

Insert document

```
db.artists.insert(artist)
```

Queries

db.movies.find ({ } , { })

db.movies	.find (	{ }	,	{ }	)
FROM		WHERE	,	SELECT	

```
db.artists.find()    SELECT * FROM artists
```

```
db.artists.find({"first_name": "Alfred", "last_name": "Hitchcock"})
SELECT * FROM artists WHERE first_name='Alfred' AND last_name='Hitchcock'
```

```
db.artists.find({"first_name": "Alfred", "last_name": "Hitchcock"},
                        {"birth_date": 1})
SELECT birth_date FROM artists WHERE first_name='Alfred'
                  AND last_name='Hitchcock'
```

Query examples

```
db.movies.find({"year" : {"$in" : [1910, 1958, 1992]}},
                {"title" : 1, "_id" : 0})
```

```
db.movies.find({"year" : {"$gte" : 1910, "$lte" : 1960}},
                {"title" : 1, "_id" : 0})
```

```
db.movies.find({"$or" : [{"year" : 1992},
                      {"director.last_name":"Hitchcock"}]},
                {"title" : 1, "_id" : 0})
```

```
db.movies.find().sort("title":-1)
```

Aggregation Framework

- **Aggregation Framework**: powerful tool to query MongoDB.
- A query is a **pipeline** of operations.
- The operators used in an aggregation pipeline are:
  - `$match`: equivalent to WHERE.
  - `$project`: akin to FROM (but more powerful).
  - `$group`: equivalent to GROUP BY.
  - `$unwind`: operates on the arrays.
  - `$sort`: equivalent to ORDER BY.
  - `$limit`: equivalent to LIMIT.
  - `$skip`: skips elements of a query response.

The Operator $match

- Equivalent to WHERE.
- It should be the first in the pipeline.
  - It filters out elements before applying other (potentially computationally expensive) operators.

The operator $match

```
db.movies.aggregate({$match: {"genre" : "drama"}});
```

The Operator $project

- Similar to FROM.
- It can also compute keys on the fly.

The operator $project.

```
db.movies.aggregate({$match: {"genre" : "drama"}},
                {"$project":{"director.last_name":1, "_id":0}} )
```

The operator $project (computed keys).

```
db.movies.aggregate({$match: {"genre" : "drama"}},
    {"$project":
      {"full_name" :
          {$concat: ["$director.first_name", " ",
                          "$director.last_name"]}}} )
```

The Operator $group

- Equivalent to GROUP BY.
- Used with aggregating functions.
  - `$sum`, `$avg`, `$max`, `$min` ...

The operator $group.

```
db.movies.aggregate({$match: {"genre" : "drama"}},
          {$group: {"_id" : {"country":"$country"},
                                    nb_movies : {$sum : 1}}})
```

The Operator $unwind

- Used to deconstruct an array from a document.
- Creates a document for each element of the array.
- Following example: the movie *Vertigo* has two actors.

The operator `$unwind`.

```
db.movies.aggregate(
      {$match:{"title":"Vertigo"}}, {$unwind : "$actors"})
```

```
{
  "_id":"movie:1",
  "title":"Vertigo",
  ......
  "actors":{
      "_id":"artist:15",
      ....
  }
},
```

```
{
  "_id":"movie:1",
  "title":"Vertigo",
  ......
  "actors":{
       "_id":"artist:16",
       ....
   }
 },
```

The Operator $sort

- Equivalent to ORDER BY.

The operator $sort.

```
db.movies.aggregate({$match: {"genre" : "drama"}},
          {$group: {"_id" : {"country":"$country"},
                                nb_movies : {$sum : 1}}},
          {$sort: {"nb_movies" : -1}})
```

The Operators $limit and $skip

- `$limit`. Equivalent to LIMIT.
- `$skip`. Skips the first results of a query.

The operator `$limit`.

```
db.movies.aggregate(
     {$group: {"_id" : {"genre":"$genre"},
                              nb_movies : {$sum : 1}}},
         {$sort: {"nb_movies" : -1}},
         {$limit: 10})
```

The operator `$skip`.

```
db.movies.aggregate(
     {$group: {"_id" : {"genre":"$genre"},
                              nb_movies : {$sum : 1}}},
         {$sort: {"nb_movies" : -1}},
         {$skip: 10})
```

Join operation

Collection Department

```
{
  "_id": "dept:1",
  "name": "Acconting",
  budget: 50000,
  manager: "emp:1"
}
```

Collection Employee

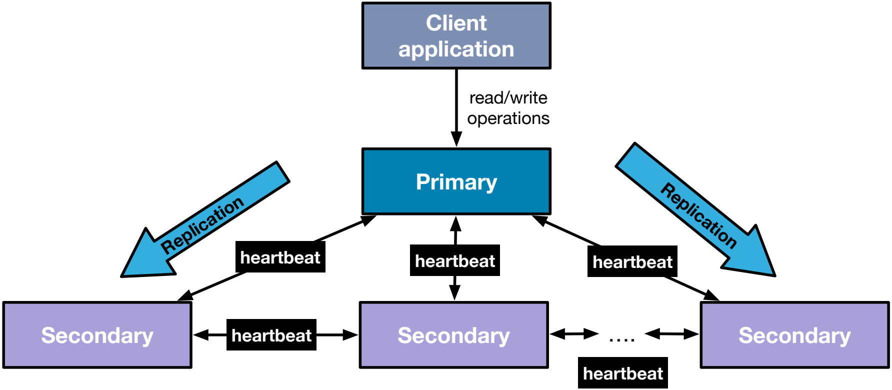
```
{
  "_id": "emp:1",
  "first_name": "John",
  "last_name": "Smith",
  "salary": 80000
}
```

```
db.Department.aggregate([
  {
    "$lookup": {
      "localField":"manager",
      "from": "Employee",
      "foreignField": "_id",
      "as" : "dept_manager"
    }
  }
]);
```

```
{ _id: 'dept:1',
  name: 'Accounting',
  budget: 50000,
  manager: 'emp:1',
  dept_manager:
   [ { _id: 'emp:1',
       first_name: 'John',
       last_name: 'Smith',
       salary: 80000 } ] }
```

Replication

- Implemented with a **replica set**.
- Master-slave architecture.
- Read/write operations only on the **primary** (master).
- Secondaries (slaves) keep a replica of the data of the primary.



Sharding

- **Autosharding.** MongoDB automatically distributes documents into **shards** that are distributed across the machines of a cluster.
- The user has to define criteria to shard (the shard key).
- Data are partitioned into **chunks**.
- A shard can contain multiple chunks.
- A process called **mongos** keeps track of which shards contain which data.

Load balancing

- **mongos** balances the load across the shards.
- If some chunk grows too much, it is split.
- The balancer process automatically migrates chunks when there is an uneven distribution.