MongoDB

Big Data computing technologies, BSC

Előd EGYED-ZSIGMOND

LIRIS/INSA de Lyon

Plan

- Introduction
- Modeling
- Getting started
- CRUD queries
- Aggregates
- Indexes
- Replication
- Sharding/Partitioning
- Conclusion

Introduction

MongoDB, is one of the most popular "NoSQL« DBMS-s

- It is a “document based” NOSQL system (such as CouchDB, Elasticsearch, …)
- relies on a semi-structured data model (JSON encoding);
- no schema (complete flexibility);
- An original (and specific) query language;
- No (or very little) transactional support.

Built to be a scalable and distributed system

- distribution by partitioning (sharding) ;
- fault tolerance through replication.

Features

- Open source solution developed by 10gen, service provider
- Written in C++
- Document-oriented DBMS => taking into account semi-structured data
- BSON format visible via JSON
- Distributed system (replication +sharding)
- Flexible schemas
- Specific query language

Lexique

- **MongoDB Cluster** : A set of processes distributed across multiple nodes; the set = routers (mongos) + config servers (mongod) + shards (mongod).
- **Router**: MongoDB mongos instances route queries and write operations to shards in a sharded cluster. It tracks what data is on which shard by caching the metadata from the config servers. The mongos uses the metadata to route operations from applications and clients to the mongod instances.
- **Shard**: A set of mongod processes, one primary, n secondaries => master-slave type data replication. A single mongod instance or replica set that stores part of a sharded cluster's total data set.
- **MongoDB database**: A database is a container for collections, where each collection stores documents in BSON format. Databases are isolated from each other, meaning data in one database is not accessible in another without explicit queries. Each MongoDB instance can host multiple databases, each identified by a unique name.
- **Collection**: n MongoDB, a collection is a grouping of documents, similar to a table in relational databases but schema-less. Collections store BSON documents and can hold diverse document structures within the same collection.

MongoDB

A MongoDB server is composed of databases.

A database contains collections.

Collections hold documents.

Each document has a unique identifier generated by MongoDB, the _id field.

Document oriented

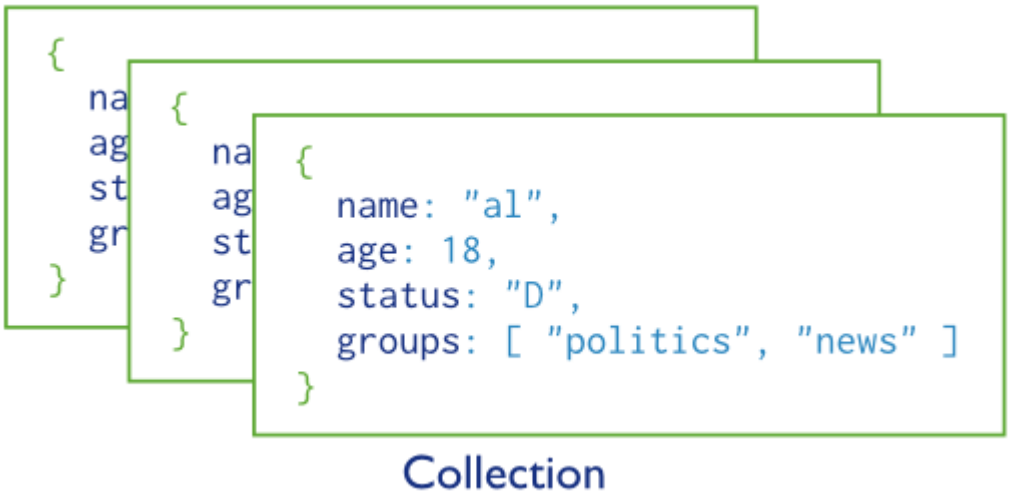
- Documents (rows) of the same nature are stored in collections (tables).
- A document is a tree, composed of keys and values.
- A value can be:
  - Scalar (int, long, string, date, binary, bool, etc.)
  - An array
  - A nested document

Basic notions

- Document

```
{
  name: "sue",                          <-- field: value
  age: 26,                              <-- field: value
  status: "A",                          <-- field: value
  groups: [ "news", "sports" ]          <-- field: value
}
```

- Collection



http://docs.mongodb.org/manual/core/crud-introduction/

MongoDB Data Model

- JavaScript Object Notation (JSON) model
- *Database* = set of named *collections* → *generalizes relation*
- *Collection* = sequence of *documents* → *generalizes tuple*
- *Document* = {attribute$_1$:value$_1$,...,attribute$_k$:value$_k$}
- *Attribute* = string (attribute$_i$≠attribute$_j$)
- *Value* = primitive value (string, number, date, ...), or a document, or an *array*
- *Array* = [value$_1$,...,value$_n$]

- Key properties: hierarchical (like XML), no schema
  - Collection docs may have different attributes

MongoDB vs Relational DBMS

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Partition	Sharding
Clustering	ReplicaSet
Joining	Linking & Embedding

Data Example

Collection inventory

```
{
      item: "ABC2",
      details: { model: "14Q3", manufacturer: "M1 Corporation" },
      stock: [ { size: "M", qty: 50 } ],
      category: "clothing”
}

{
      item: "MNO2",
      details: { model: "14Q3", manufacturer: "ABC Company" },
      stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],
      category: "clothing”
}
```

```
db.inventory.insertOne(
  {
    item: "ABC1",
    details: {model: "14Q3",manufacturer: "XYZ Company"},
    stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
    category: "clothing"
  }
)
```

Document insertion

(docs.mongodb.org)

JSON

JSON " **J**ava**S**cript **O**bject **N**otation "is a formatted exchange data readable by a human and interpreted by a machine.

Based on JavaScript, it is completely independent of programming languages

Two structures:

- A unordered collection of key/values ➔ **Object**
- An ordered collection of objects ➔ **Array**

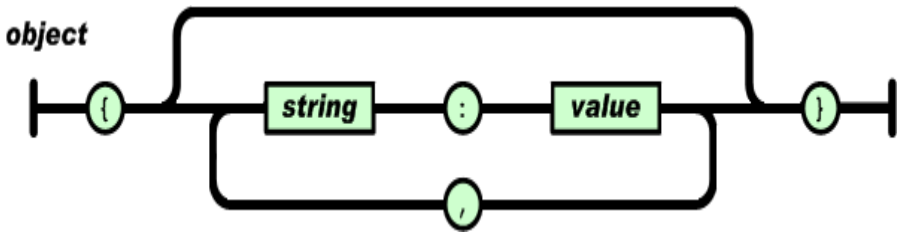
Basic constructs (recursive)

- Base values
  number, string, boolean, …
- Objects { }
  sets of label-value pairs
- Arrays [ ]
  lists of values

JSON

Object

Starts with a " { " and ends with" }" and consists of an unordered list of keys/value pairs . A key is followed by ":" and the key / value pairs are separated by " , "

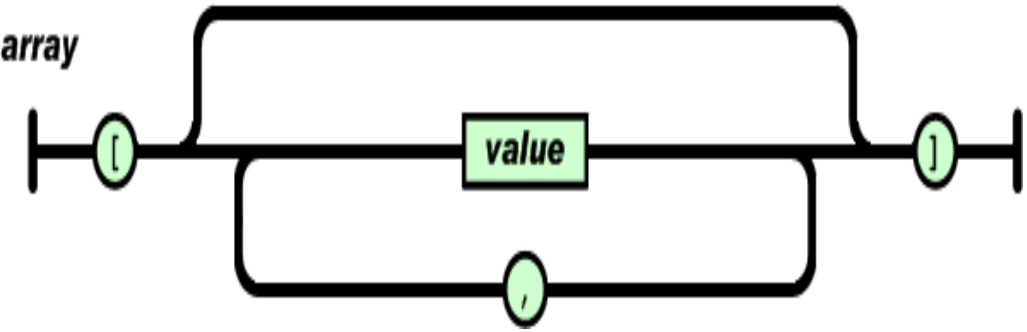



```
{ "Id": 51,
"name": "Mathematics 1 ", "resume":
"Summary of math ", "isbn":
"123654",
"category":
    {
    "id ": 2, "name": "Mathematics",
    "description": "Description of
    mathematics "
    } ,
"amount": 42,
"Photo": ""
}
```

JSON

ARRAY

ordered list of objects that begin with " [" and end with"] ". Objects are separated from each other by" , ".

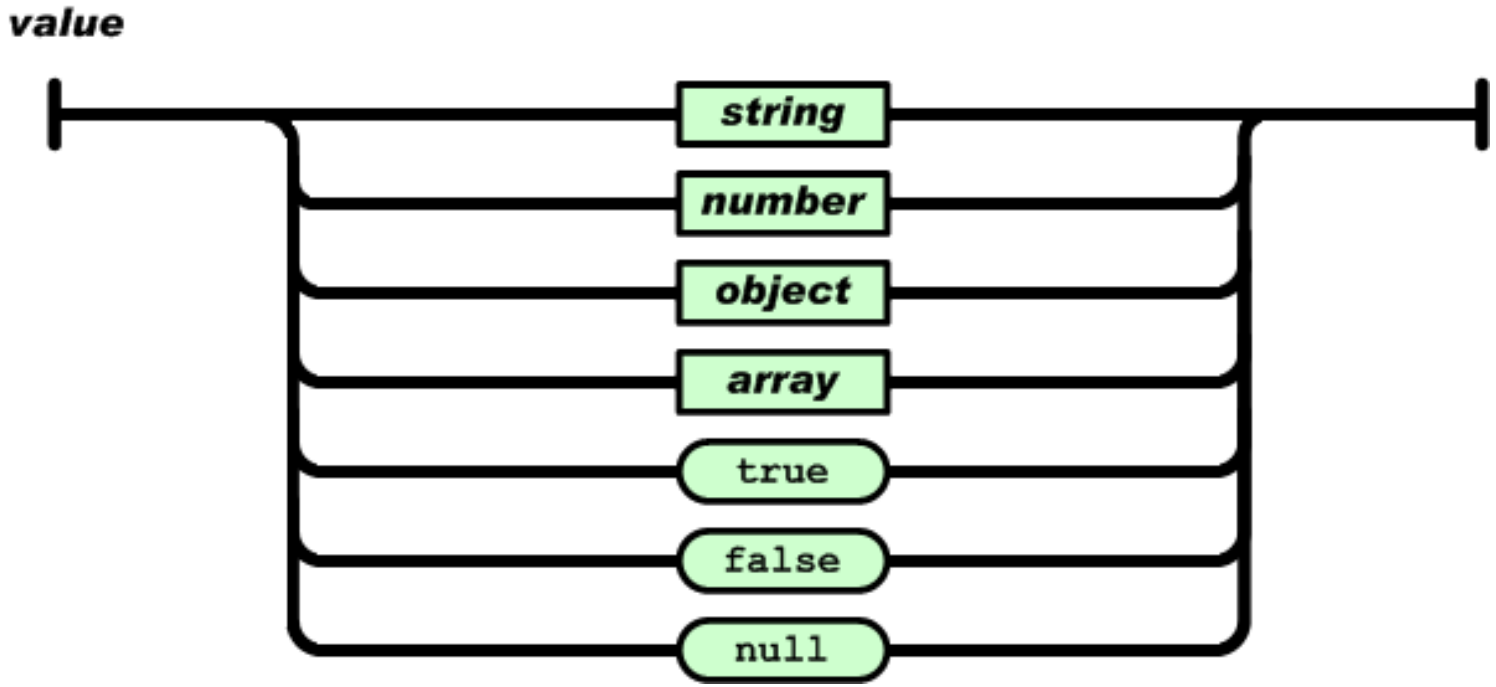



```
[
{ "Id": 51,
"name": "Mathematics 1 ",
"resume": "Resume of math",
"isbn": "123654",
"amount": 42,
"Photo": ""
} ,
{ "Id": 102,
"name": "Mathematics 1 ",
"resume": "Resume of math",
"isbn": "12365444455",
"amount": 42,
"Photo": ""
}
]
```

JSON

Value

An object can be either a string between "" or a number (integer, decimal) or boolean (true, false) or null or an object.



Much Like XML

- Plain text formats

- “Self-describing“ (human readable)

- Hierarchical (Values can contain lists of objects or values)

Not Like XML

- Lighter and faster than XML

- JSON uses typed objects. All XML values are type-less strings and must be parsed at runtime.
- Less syntax, no semantics
- Properties are immediately accessible to JavaScript code

Knocks against JSON

- Lack of namespaces
- No inherit validation (XML has DTD and templates, but there is JSONlint)
- Not extensible
- It’s basically just ***not*** XML

Why MongoDB

- Easy to implement
- Open Source, free, and active
- Drivers for all languages
- Rich but not too exotic (some relational concepts still apply)
- Fairly rich tooling
- Natively handle json data (many sources provide information in that format)

Plan

Introduction

Modeling

Getting started

CRUD queries

Aggregates

Indexes

Replication

Sharding/Partitioning

Conclusion

MongoDB Schema design

Design a schema based on the unique needs of your application!

You better know what kind of queries will be frequently submitted (Access patterns)!

In MongoDB, the maximum document size is **16 MB**.

“Limited” Nesting: no more than **100** levels of nesting in documents

MongoDB Schema design

Users

ID	First Name	Surname	City
1	Jean	Dupont	Lyon

Profession

ID	User ID	Profession
3	1	Banking
4	1	Accountant
5	1	Controller

Application must take care of **joins** and data intergity

Cars

ID	User ID	Car	Year
12	1	DS	2020
13	1	Alpine	2024

MongoDB Schema design

Users

ID	First Name	Surname	City
1	Jean	Dupont	Lyon

Profession

ID	User ID	Profession
3	1	Banking
4	1	Accountant
5	1	Controller

```
Select FirstName, Surname, Profession
from Users, Profession
where users.id = profession.userID
```

JOIN

Cars

ID	User ID	Car	Year
12	1	DS	2020
13	1	Alpine	2024

MongoDB Schema design

```
{
  First_Name : "Jean",
  Surname : "Dupont",
  City : "Lyon",

}
```

Users

ID	First Name	Surname	City
1	Jean	Dupont	Lyon

MongoDB Schema design

```
{
  First_Name : "Jean",
  Surname : "Dupont",
  City : "Lyon",
  Profession : ["Banking","Accountant", "Controller" ]

}
```

Users

ID	First Name	Surname	City
1	Jean	Dupont	Lyon

Profession

ID	Uder ID	Profession
3	1	Banking
4	1	Accountant
5	1	Controller

MongoDB Schema design

```
{
  First_Name : "Jean",
  Surname : "Dupont",
  City : "Lyon",
  Profession : ["Banking","Accountant", "Controller" ],
  Cars : [
        {
          model:"DS",
          year:2020
        },
        {
          model:"Alpine",
          year:2024
        }
  ]
}
```

Users

ID	First Name	Surname	City
1	Jean	Dupont	Lyon

Profession

ID	Uder ID	Profession
3	1	Banking
4	1	Accountant
5	1	Controller

Cars

ID	User ID	Car	Year
12	1	DS	2020
13	1	Alpine	2024

MongoDB Schema design

Denormalize !

In SQL data is normalized. Goal : do not duplicate the data

In MongoDB and in NoSQL in general, this is not the goal!

With one to many relations:

- Embed unless there's a compelling reason not to
- Avoid JOINs if it can be, **but don't be afraid if they can provide a better schema design**
- Arrays should not grow without bound
- Needing to access an object on its own is a compelling reason not to embed it

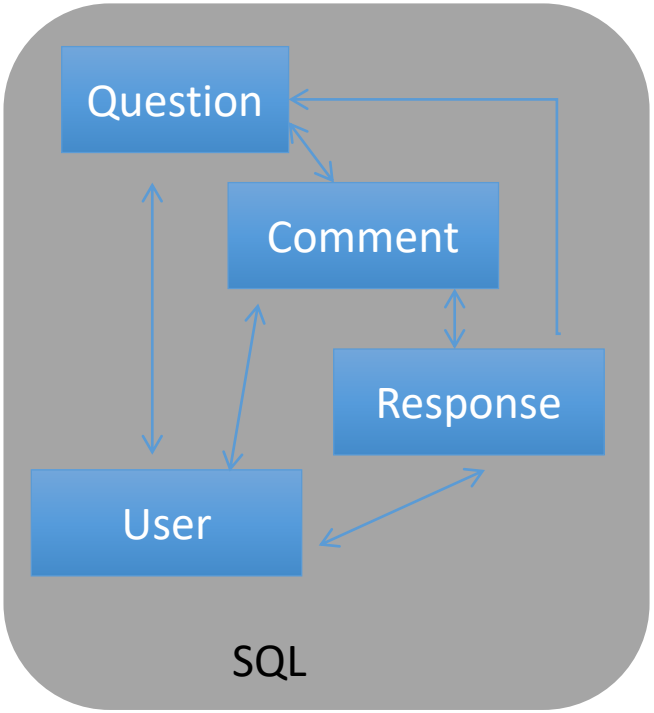
MongoDB Schema design

You can normalize if :

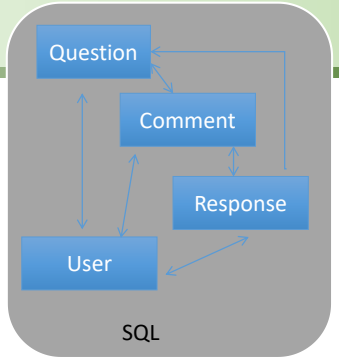
- Embedding leads to massive data duplication
- To represent complex any-to-many relationships

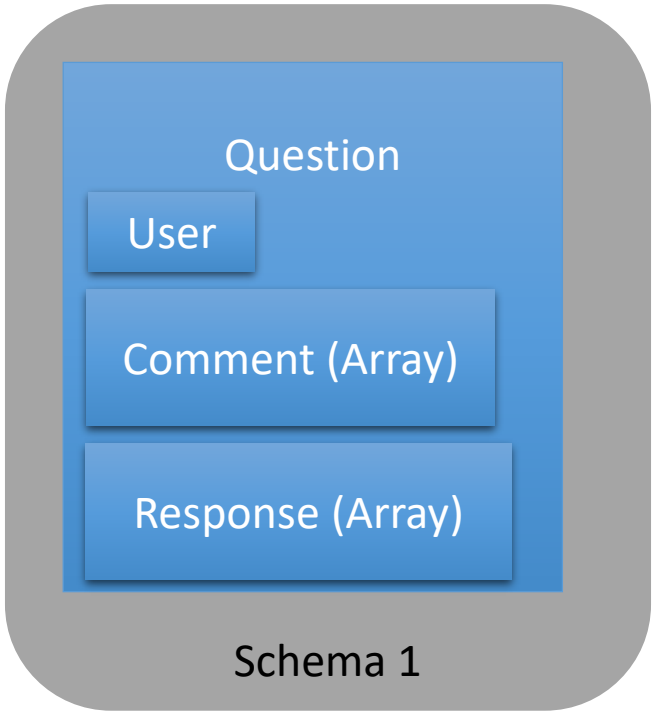
Design a schema based on the unique needs of your application!

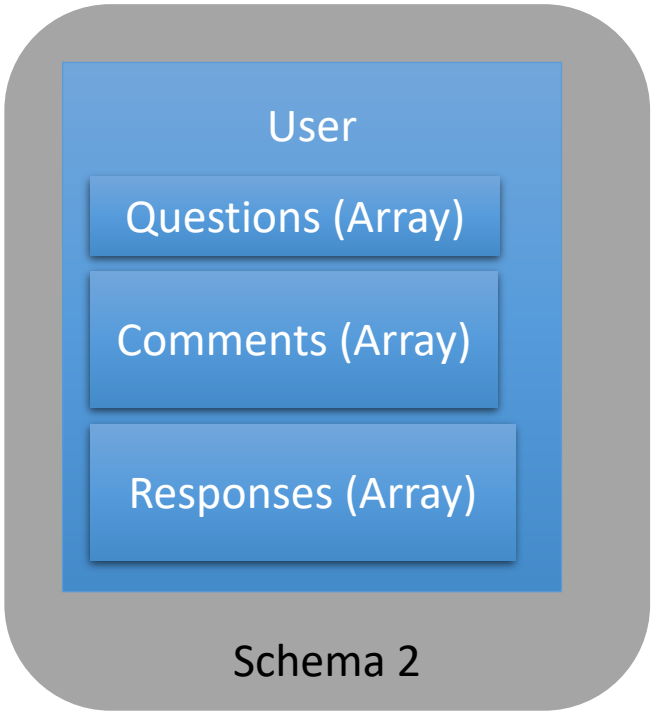
- Relational / Document



- Relational / Document







Plan

Introduction

Modeling

Getting started in practice

CRUD queries

Aggregates

Indexes

Replication

Sharding/Partitioning

Conclusion

BSON

- MongoDB uses BSON very heavily
  - Binary JSON
  - Like JSON with a binary serialization method
  - Has extensions so that it can represent data types that JSON cannot
- Used to represent documents, provide input to queries

Understanding MongoDb components

Component Set	Binaries
Server	mongod.exe
Router (Sharding service)	mongos.exe
Client	mongosh.exe
Monitoring Tools	mongostat.exe, mongotop.exe
ImportExportTools	mongodump.exe, mongorestore.exe, mongoexport.exe, mongoimport.exe
MiscellaneousTools	bsondump.exe, mongofiles.exe, mongooplog.exe, mongoperf.exe monoreplay.exe , mongoldap.exe

Practice

Starting with: bin/mongod

Some arguments:
--dbpath <path>: Data storage path.
--port <port>: Server port.
--replSet <name>: Add the server to a replica set cluster.

Command-line client: mongosh
(MongoDB shell, to be downloaded separately).

Shell

Show the current database:

db

Show the list of databases:

show dbs

Select/create a database:

use <name>

Show collections:

show collections

Interacting with MongoDB

- Multiple databases within MongoDB
  - **Switch databases**
    - use newDb
  - New databases will be stored after an insert
- **Create collection**
  - db.createCollection(“collectionName”)
  - Not necessary, collections are implicitly created on insert

Plan

Introduction

Modeling

Getting started in practice

CRUD queries

Aggregates

Indexes

Replication

Sharding/Partitioning

Conclusion

CRUD

Modification:

- Insert
- Update
- Remove

Operations apply on one collection

http://docs.mongodb.org/manual/

CRUD

Insertion

```
db.users.insertOne(          <-- collection
   {
      name: "sue",           <-- field: value
      age: 26,               <-- field: value
      status: "pending"      <-- field: value
   }                                              } document
)
```

http://docs.mongodb.org/manual/

Document

```
{
  name: "sue",
  age: 26,
  status: "A"
}
```

insert

Collection

```
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
```

users

In order to insert multiple documents, use: insertMany

The "_id" fiels is automatically added

CRUD

UpdateOne or updateMany

```
db.collection.updateOne(
   <query>,
   <update>,
   {
     upsert: <boolean>,
     multi: <boolean>,
     writeConcern: <document>
     …
   }
)
```

Mêmes contraintes que pour find

$set, $unset, …

CRUD

Update

```
db.movies.updateOne({"year": {$lt: 2000}},{$set : {old:true}})

db.movies.find({old:true})

db.movies.updateMany({"year": {$lt: 2000}},{$set : {old:true}})

db.movies.updateOne({"year": {$lt: 2000}},{$unset :
{old:true}})
```

CRUD Update example

```
db.students.insertMany( [
{
  "_id": 1,
  "student": "Skye",
  "points": 75,
  "commentsSemester1": "great at math",
  "commentsSemester2": "loses temper"
},
{
  "_id": 2,
  "student": "Elizabeth",
  "points": 60,
  "commentsSemester1": "well behaved",
  "commentsSemester2": "needs improvement"
}
] )
```

```
{
  _id: 1,
  student: 'Skye',
  points: 75,
  status: 'Modified',
  comments: [ 'great at math', 'loses temper' ],
  lastUpdate: ISODate('2025-01-07T09:10:29.779Z')
},
{
  _id: 2,
  student: 'Elizabeth',
  points: 60,
  commentsSemester1: 'well behaved',
  commentsSemester2: 'needs improvement'
}
```

```
db.students.updateOne(
{ _id: 1 },
[
{ $set: {status: "Modified",
         comments: [ "$commentsSemester1", "$commentsSemester2" ],
         lastUpdate: "$$NOW" } },
{ $unset: [ "commentsSemester1", "commentsSemester2" ] }
]
)
```

CRUD

Uptading an array

- Let'sadd an array to a document

```
db.products.insert({counter:100001, tab:['a','b','c']})
```

- You can retrieve it with the following command:

```
db.products.find({counter :100001})
```

- To append an element to the table we use the **$push** operator:

```
db.products.update({counter :100001}, {$push : {tab : 'd'}})
```

If more elements need to be added, there is the $**pushAll** operator.

CRUD

Updating an array

- With the **$pop** operator the last element can be deleted from an array:

```
db.products.update({counter:100001}, {$pop : {tab:1}})
```

- The table lost its last element
- If you want to delete the first element, you have to use the value -1:

```
db.products.update({counter :100001}, {$pop : {tab:-1}})
```

- The $**addToSet** operator adds a value to an array unless the value is already present, in which case $**addToSet** does nothing to that array.:

```
db.products.update({counter :100001}, {$addToSet : {tab : 'b'}})
```

- The "b« value was already in the array, so no modification was made.

CRUD

- **Delete a document**
- In order to delete a document you can use the **remove** command.

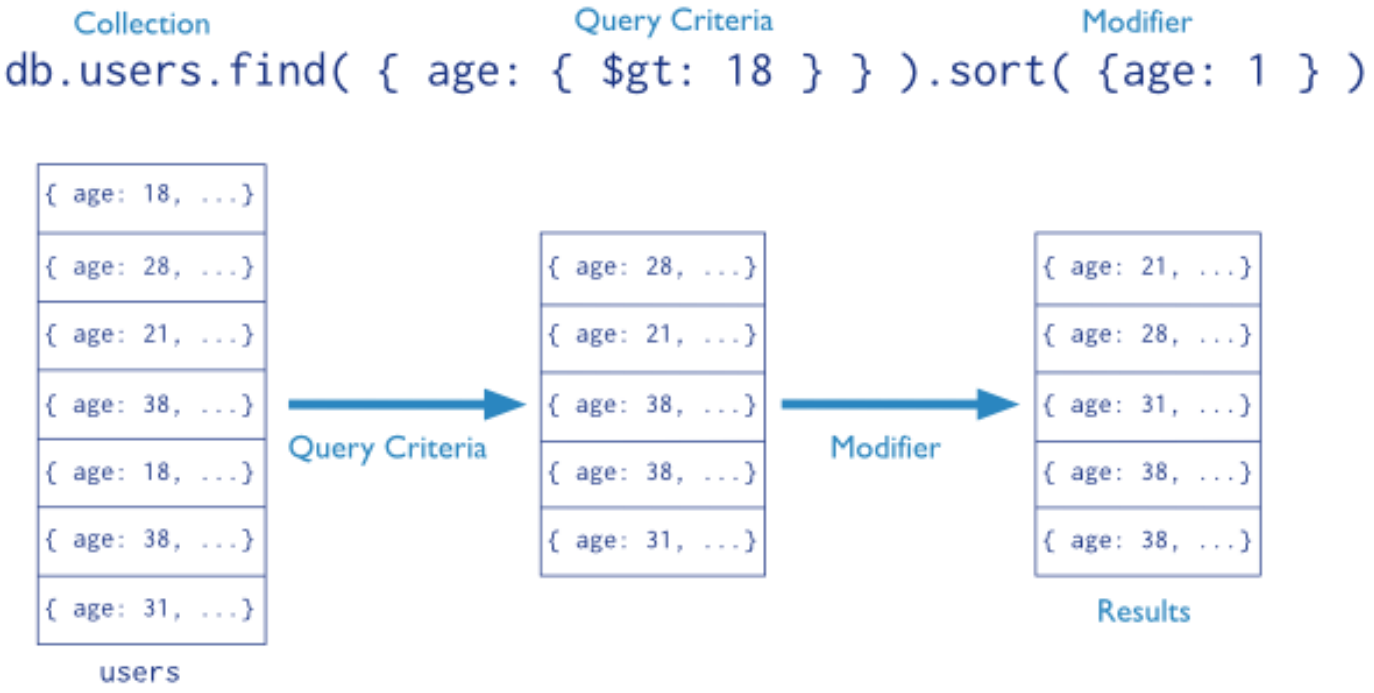
Example:

```
db.artists.remove({"last_name": "Gibson"});
```

- Deletes every document from the artists collection having "Gibson" as the value of the `last_name` field.

CRUD

Requêtes



http://docs.mongodb.org/manual/

Opérateurs

$eq	=	Values that are equal to a specified value.
$gt	>	Values that are greater than a specified value.
$gte	>=	Values that are greater than or equal to a specified value.
$lt	<	Values that are less than a specified value.
$lte	<=	Values that are less than or equal to a specified value.
$ne	!=	All values that are not equal to a specified value.
$in	$\in$	One of the values specified in an array.
$nin	$\notin$	None of the values specified in an array.
$or		
$and		
$not		
$nor		
…		

http://docs.mongodb.org/manual/reference/operator/query/

CRUD

Selection queries

```
db.users.find(                          <-- collection
   { age: { $gt: 18 } },                <-- query criteria
   { name: 1, address: 1 }              <-- projection
).limit(5)                              <-- cursor modifier
```

in SQL

```
SELECT _id, name, address   <-- projection
FROM   users                <-- table
WHERE  age > 18             <-- select criteria
LIMIT  5                    <-- cursor modifier
```

```
db.movies.find({"country": {$ne:"USA"}},{_id:0,title:1}).limit(5)
```

http://docs.mongodb.org/manual/

Example of a Simple Query

Collection **orders**

```
{
    _id: "a",
    cust_id: "abc123",
    status: "A",
    price: 25,
    items: [ { sku: "mmm", qty: 5, price: 3 },
           { sku: "nnn", qty: 5, price: 2 } ]
}
{
    _id: "b",
    cust_id: "abc124",
    status: "B",
    price: 12,
    items: [ { sku: "nnn", qty: 2, price: 2 },
           { sku: "ppp", qty: 2, price: 4 } ]
}
```

```
db.orders.find(
    { status: "A" },
    { cust_id: 1, price: 1, _id: 0 }
)
```

selection

projection

In SQL it would look like this:

```
SELECT cust_id, price
  FROM orders
WHERE status="A"
```

```
{
    cust_id: "abc123",
    price: 25
}
```

Selects/queries

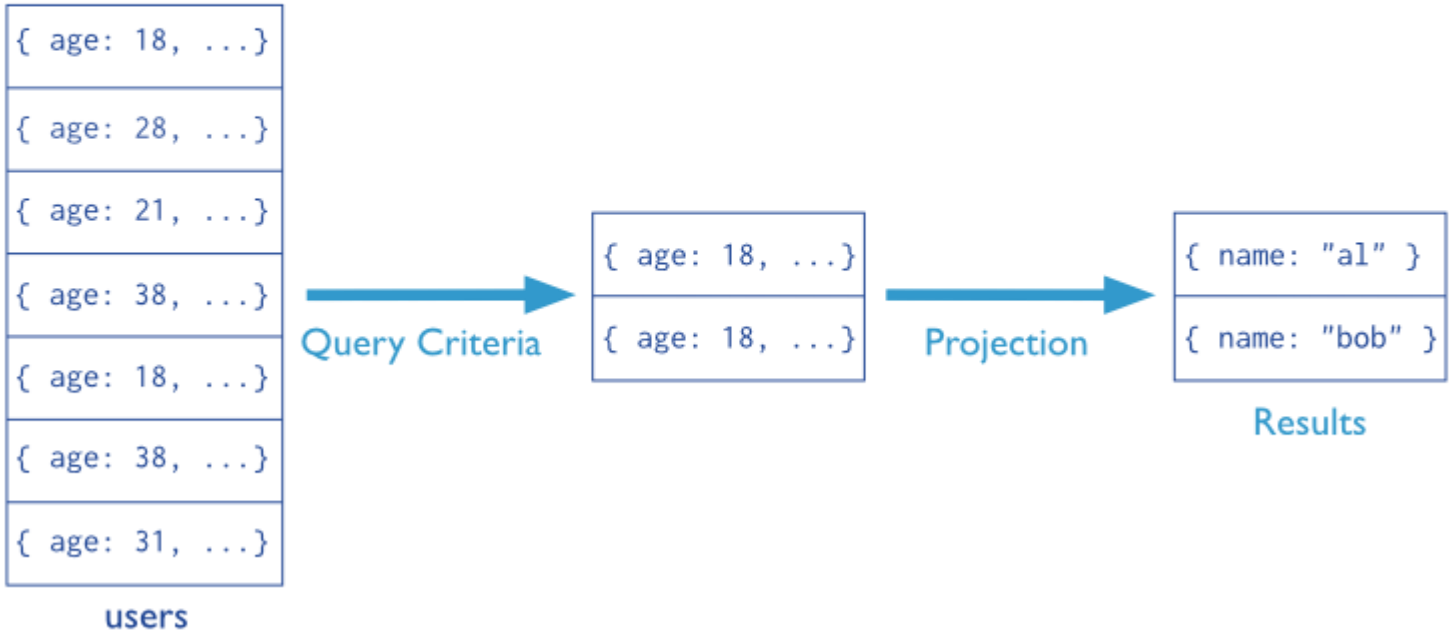
- In MongoDB, querying typically consists of providing an appropriately crafted BSON
  - SELECT * FROM collectionName
    - db.collectionName.find()
  - SELECT * FROM collectionName WHERE field = value
    - db.collectionName.find({field: value})
  - SELECT * FROM collectionName WHERE field > 5
    - db.collectionName.find({field: {$gt: 5} })
- Other functions that take a query argument have queries that are formatted this way

CRUD

Projections

Collection Query Criteria Projection

```
db.users.find( { age: 18 }, { name: 1, _id: 0 } )
```




```
db.movies.find({"country":{$ne:"USA"}},{_id:0,title:1})
```

CRUD

Exemples

Exclude the year:

```
db.movies.find({"country":"USA"},{year:0})
```

Get the title, the genre and the _id

```
db.movies.find({"country":"USA"},{title:1,genre:1})
```

Get the movies with a given role

```
db.movies.find({"actors.role":"William Munny"})
```

Get the title, the genre

```
db.movies.find({"country":"USA"},{title:1,genre:1, _id:0})
```

Sort the results

```
db.movies.find({"year": {$gt: 2000}}).sort({year:-1})
```

CRUD

Exemples

```
db.movies.find({$and : [{"year": 2003}, {genre: "romance"}] },{"title":1,"genre":1});
```

```
db.movies.find({$or : [{"year": 2003}, {genre: "romance"}] },{"title":1,"genre":1});
```

CRUD

Curseurs

```
db.collection.find()
db.movies.find({"country": "USA"})
```

Parcours un à un des résultats

```
var myCursor = db.films.find( { country: 'FR' } );

while (myCursor.hasNext()) {
   print(tojson(myCursor.next()));
}
```

```
var myCursor = db.films.find( { country: 'FR' } );

myCursor.forEach(printjson);
```

CRUD

Curseurs, méthodes

- *limit()*: pour récupérer les n premiers résultats uniquement
- *sort()*: pour trier les résultats
- *skip()*: pour "sauter" n résultats

Par exemple, pour avoir l'antépénultième film selon le nom, on pourrait faire:

```
var cur = db.movies.find();
cur.sort( {title: -1} ).limit(1).skip(2);
```

CRUD

Cursors

Transformation into an Array

```
var myCursor = db.films.find( { country: 'FR' } );
var documentArray = myCursor.toArray();
var myDocument = documentArray[1];

print(tojson(myDocument));
```

Query performance analysis

```
db.movies.find( { country: 'FR' } ).explain();
db.movies.createIndex( { country: 1 } )
```

Plan

Introduction

Modeling

Getting started in practice

CRUD queries

Aggregates

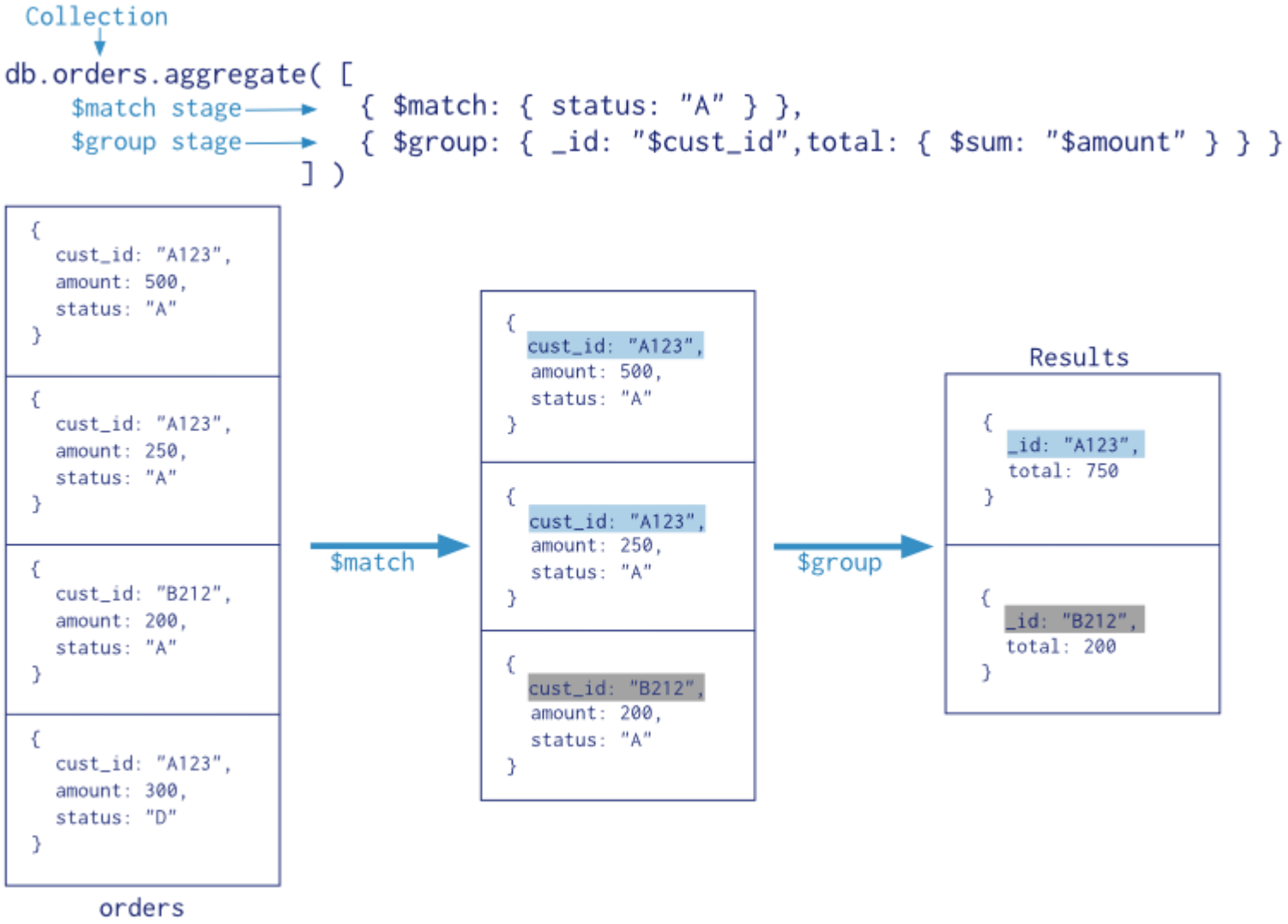
Indexes

Replication

Sharding/Partitioning

Conclusion

Agrégats



https://docs.mongodb.org

Aggregates

Movie count by genre

```
db.movies.aggregate([
  {$group:{_id:"$genre",count:{$sum:1}}}
])
```

American Movie count by genre

```
db.movies.aggregate([
{$match:{country:"USA"}},
{$group:{_id:"$genre",count:{$sum:1}}}
])
```

Date of the oldest horror movie

```
db.movies.aggregate([
{$match:{genre:"Horreur"}},
{$group:{_id:"$genre",debut:{$min:"$year"}}}
])
```

Plan

Introduction

Modeling

Getting started in practice

CRUD queries

Aggregates

Indexes

Replication

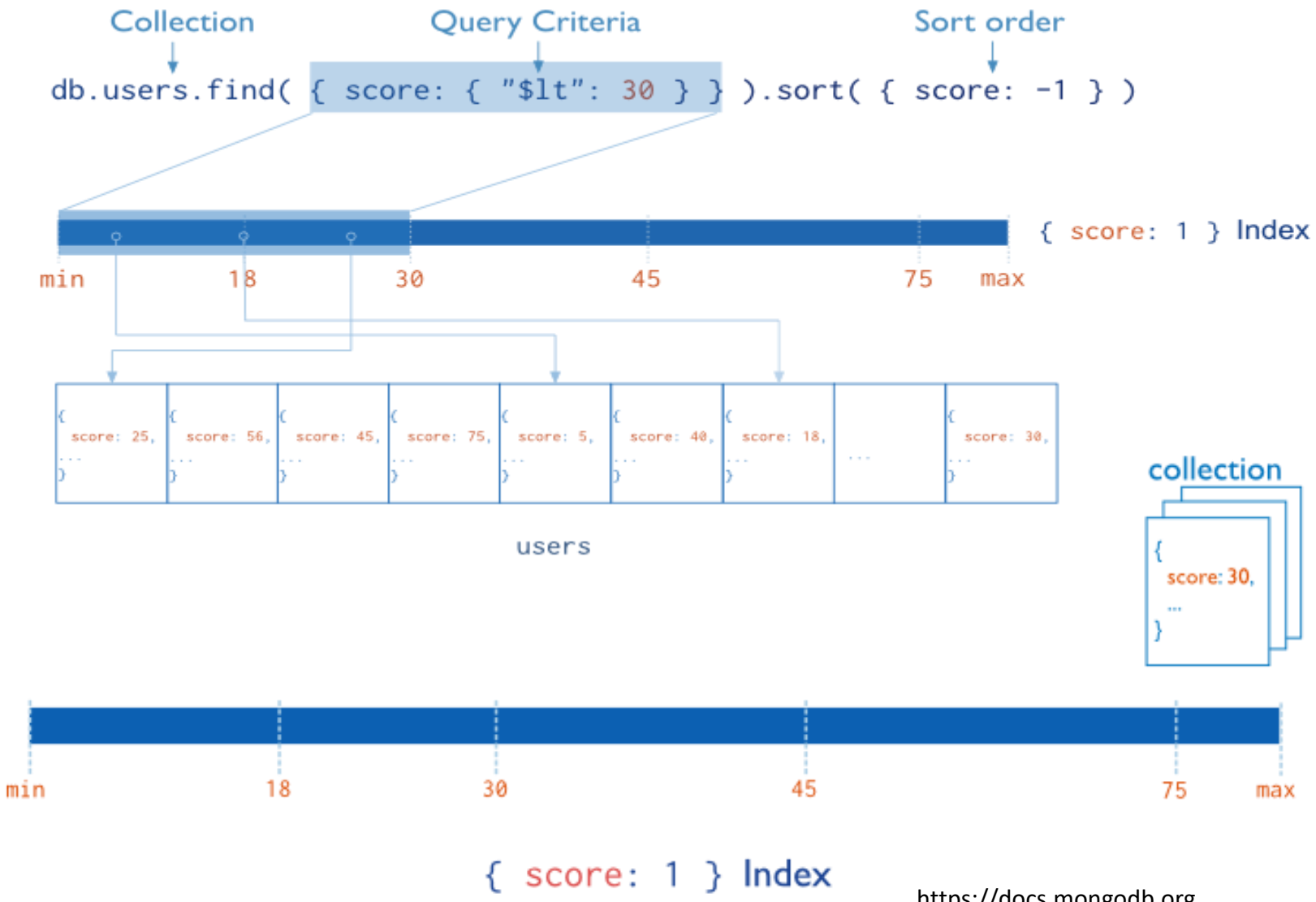
Sharding/Partitioning

Conclusion

Indexes

- Very similar to RDBMS (Relational Database Management Systems), indexing in MongoDB is done on one or more fields.
- It improves search performance.
- Indexes are stored at the collection level.
- It introduces overhead for write operations.
- The internal functioning is very similar to what is found in current relational database systems.

Indexes



https://docs.mongodb.org

Indexes

Index type

- Unique field
- Composed field
- Multi key
- Geo spatial
- Text
- Hached