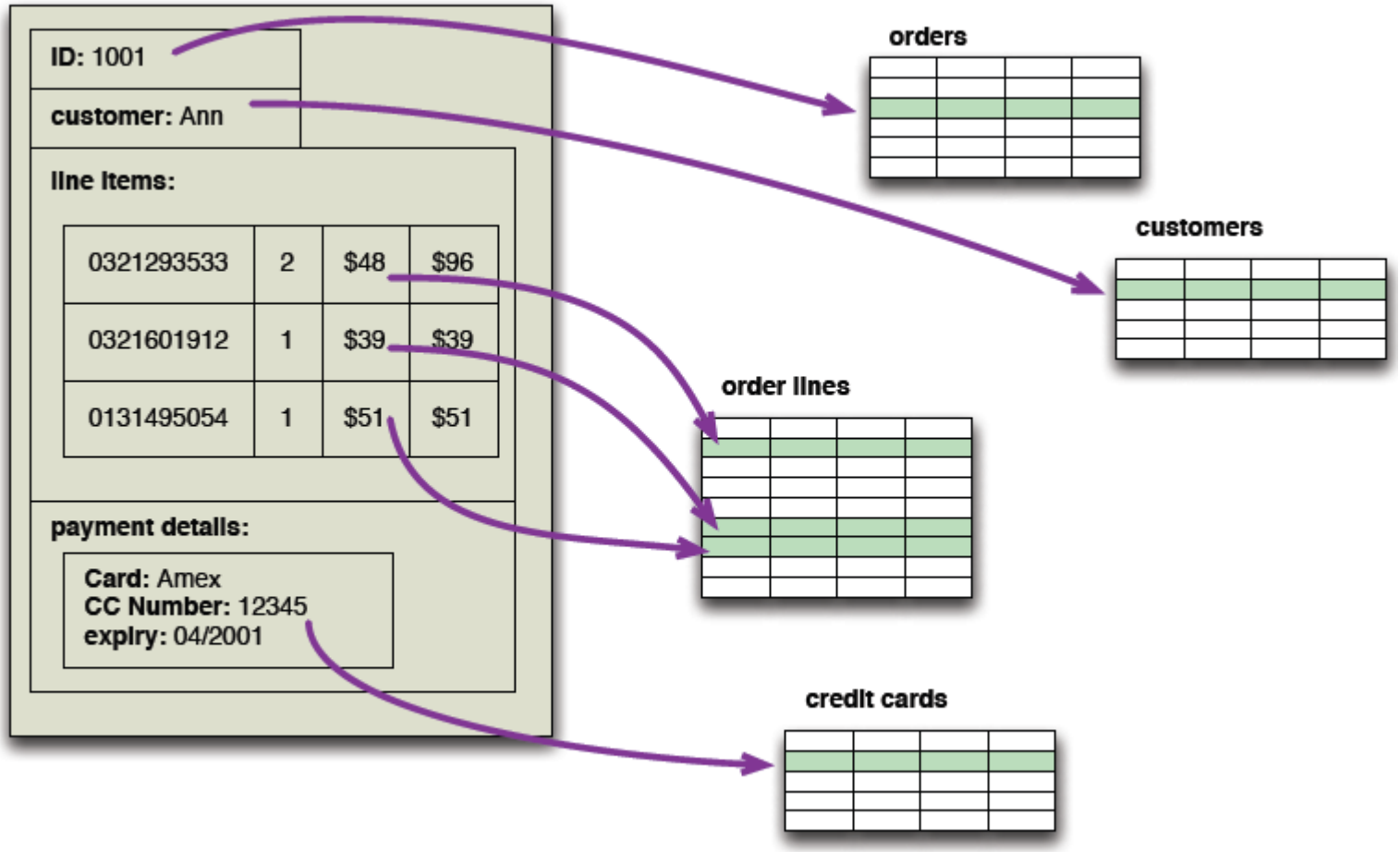
MongoDB

Mbengue Mohamadou

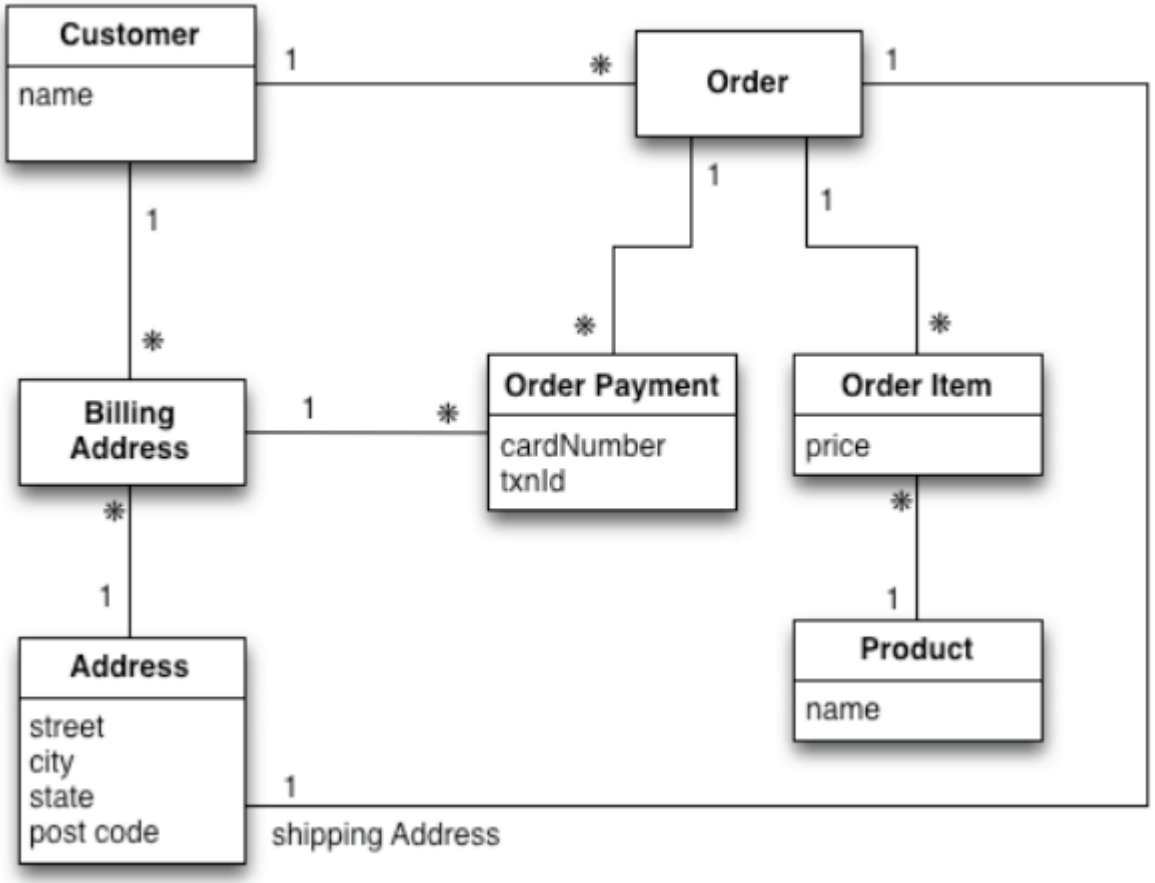
Vue d'ensemble de la formation

- **Leçon 1 : Introduction au NoSQL**
- Leçon 2 : Opération CRUD
- Leçon 3 : Les requêtes
- Leçon 4 : Design et Data Modéle
- Leçon 5 : Performance
- Leçon 6 : Agrégation Framework
- Leçon 7 : Administration
- Leçon 8 : Driver Java

Impedance mismatch



Domain driven data models



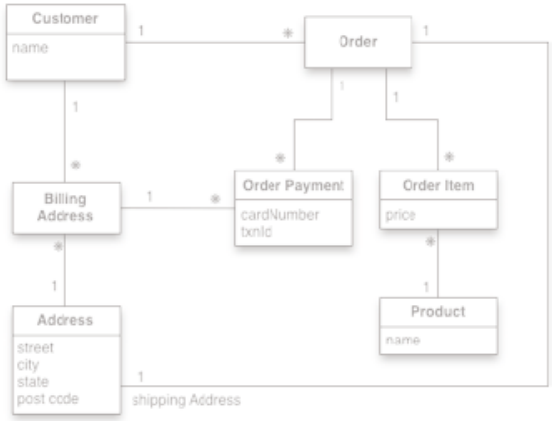
RDBMS data

Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77



Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

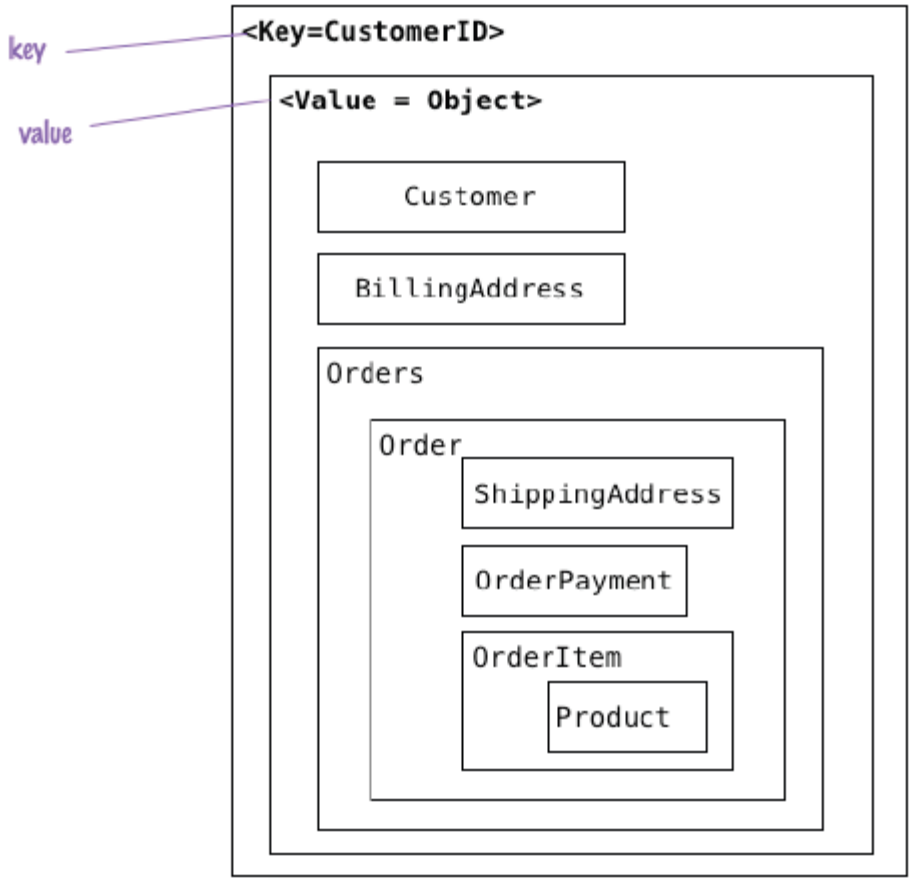
Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Le monde Relationnel

Caractéristique:

- Données structurées (Table/Schéma)
- Données Normalisées (Formes Normales)
- Standard (SQL)
- Transactionnel (ACID)
- Requête Complexe (Jointure)
- Des contraintes(Intégrité des données)
- …

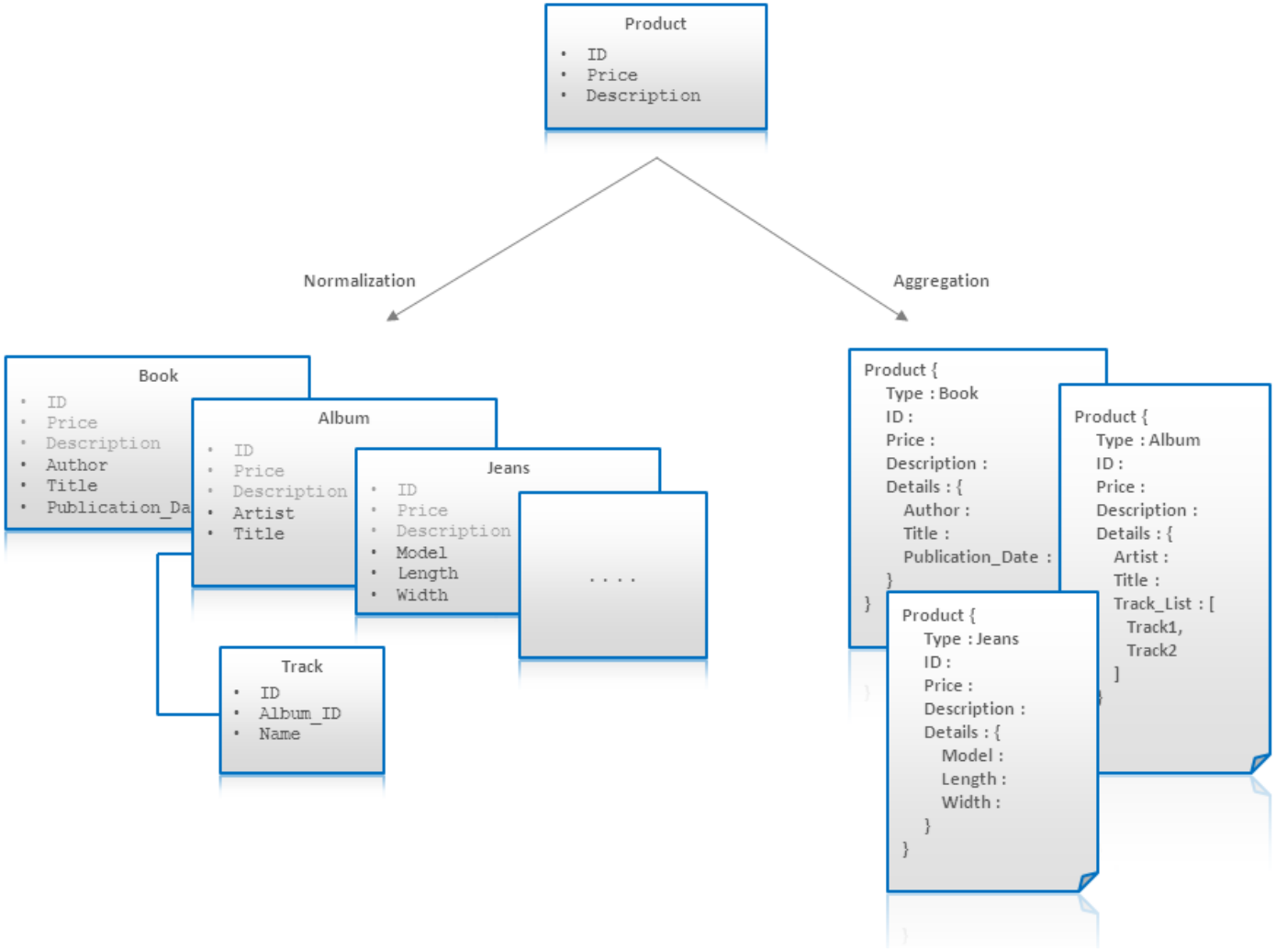
Aggregate model



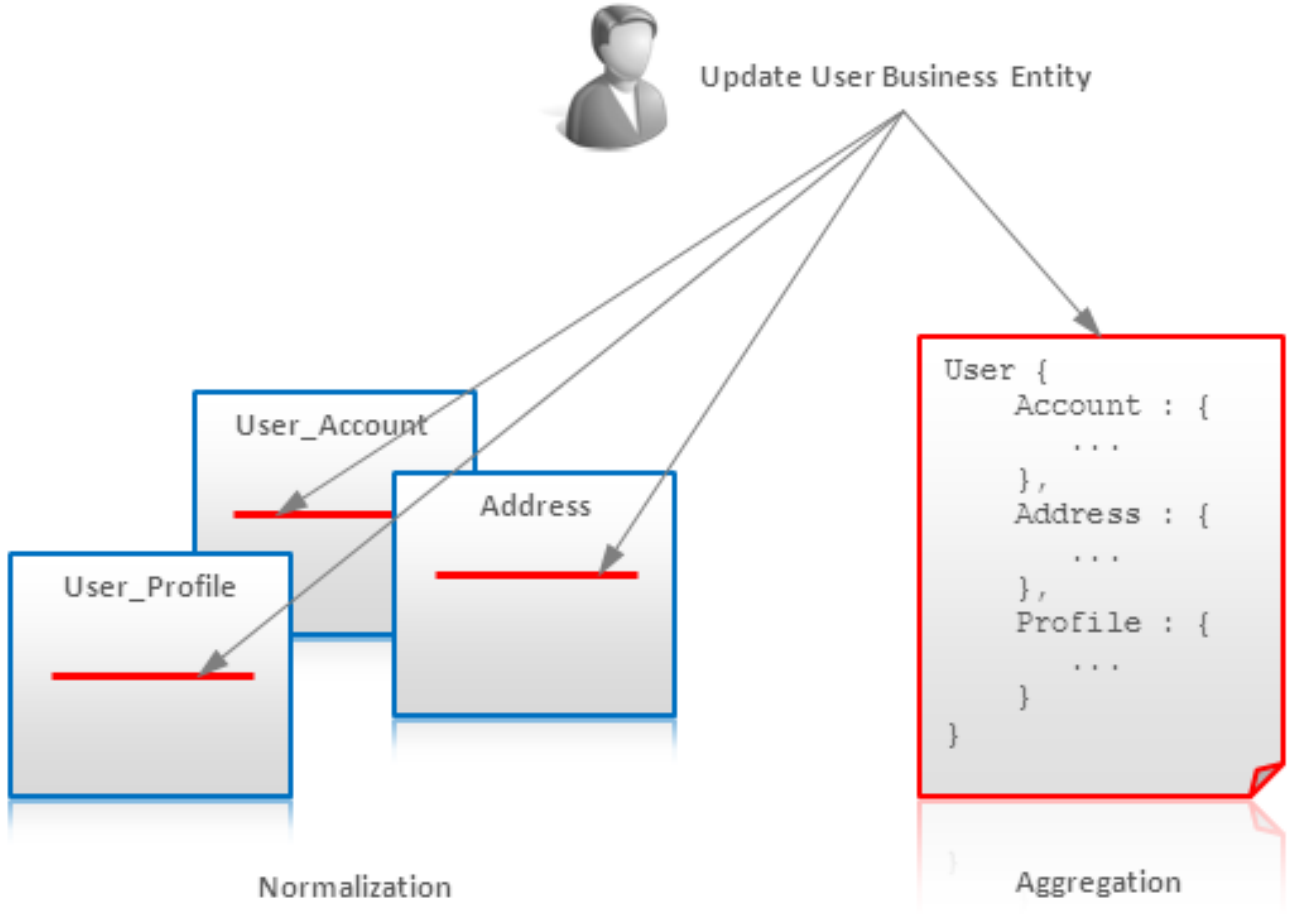
Aggregate Data

```
    // in customers
{
  "customer": {
    "id": 1,
    "name": "Martin",
    "billingAddress": [{"city": "Chicago"}],
    "orders": [
        {
          "id":99,
          "orderItems":[
            {
              "productId":27,
              "price": 32.45,
              "productName": "NoSQL Distilled"
            }
          ],
          "shippingAddress":[{"city":"Chicago"}]
          "orderPayment":[
            {
              "ccinfo":"1000-1000-1000-1000",
              "txnId":"abelif879rft",
              "billingAddress": {"city": "Chicago"}
            }
          ],
        }
    ]
  }}
```

Normalisation vs Agrégation

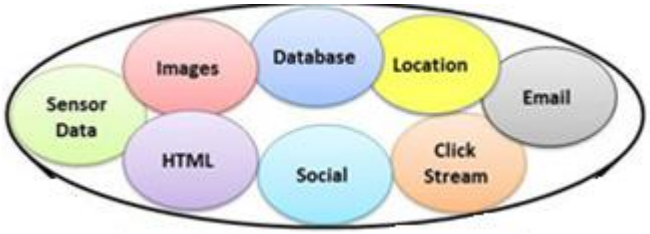


Normalisation vs Agrégation



Le monde aujourd'hui

Volumétrie



Variété de données

Le monde aujourd'hui

RDBMS et Clusters

Big Data


Variety
Velocity
Volume
Structured & Unstructured
Batch
Structured
Big Data
Streaming Data
Zettabytes
Terabytes

Les évolutions...

- Nouvelles Données :
  - Web 2.0 : Facebook, Twitter, news, blogs,
  - Flux : capteurs, GPS, …
- Nouveaux Traitements :
  - Moteurs de recherche , Extraction, analyse, ...
  - Recommandation, filtrage collaboratif, …
- Nouvelles Infrastructures :
  - Cluster, réseaux mobiles, microprocesseurs multi-coeurs, …

- → très gros volumes, données pas ou faiblement structurées
- → transformation, agrégation, indexation
- → distribution, parallélisation, redondance

Big Data

Le NoSQL

Les bases NoSQL (comprendre Not Only SQL) répondant aux problématiques:

- de hautes disponibilités,
- grandes performances en lecture et/ou écriture
- le traitement de grands volumes de données.

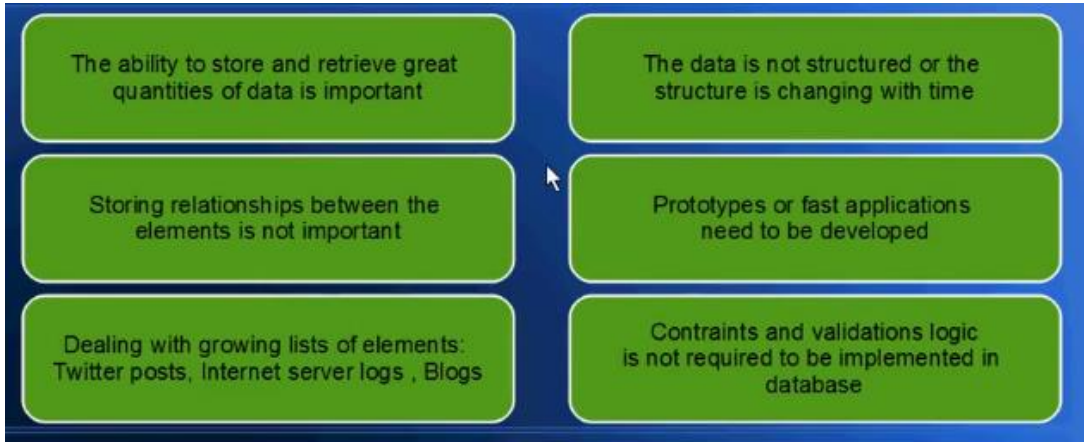
Le NoSQL

Un moteur NoSQL c'est…

- Une structure de données
- Une absence de contraintes
- Une méthode de modélisation
- Un schéma de données… flexible
- Une relation avec un modèle objet
- Du requêtage
- Une absence de transactions et des compromis

Le NoSQL

Alors, SQL versus NoSQL ?



← OUI

Non →

Théorème CAP

Théorème CAP (proposé par Brewer, 2000 et démontré par Gilbert et Lynch 2002)

- Dans un environnement distribué, il n'est pas possible de respecter simultanément :
  - **C :** Cohérence
  - **A :** Disponibilité
  - **P :** Résistance au morcellement

- On peut, en revanche, respecter deux contraintes.

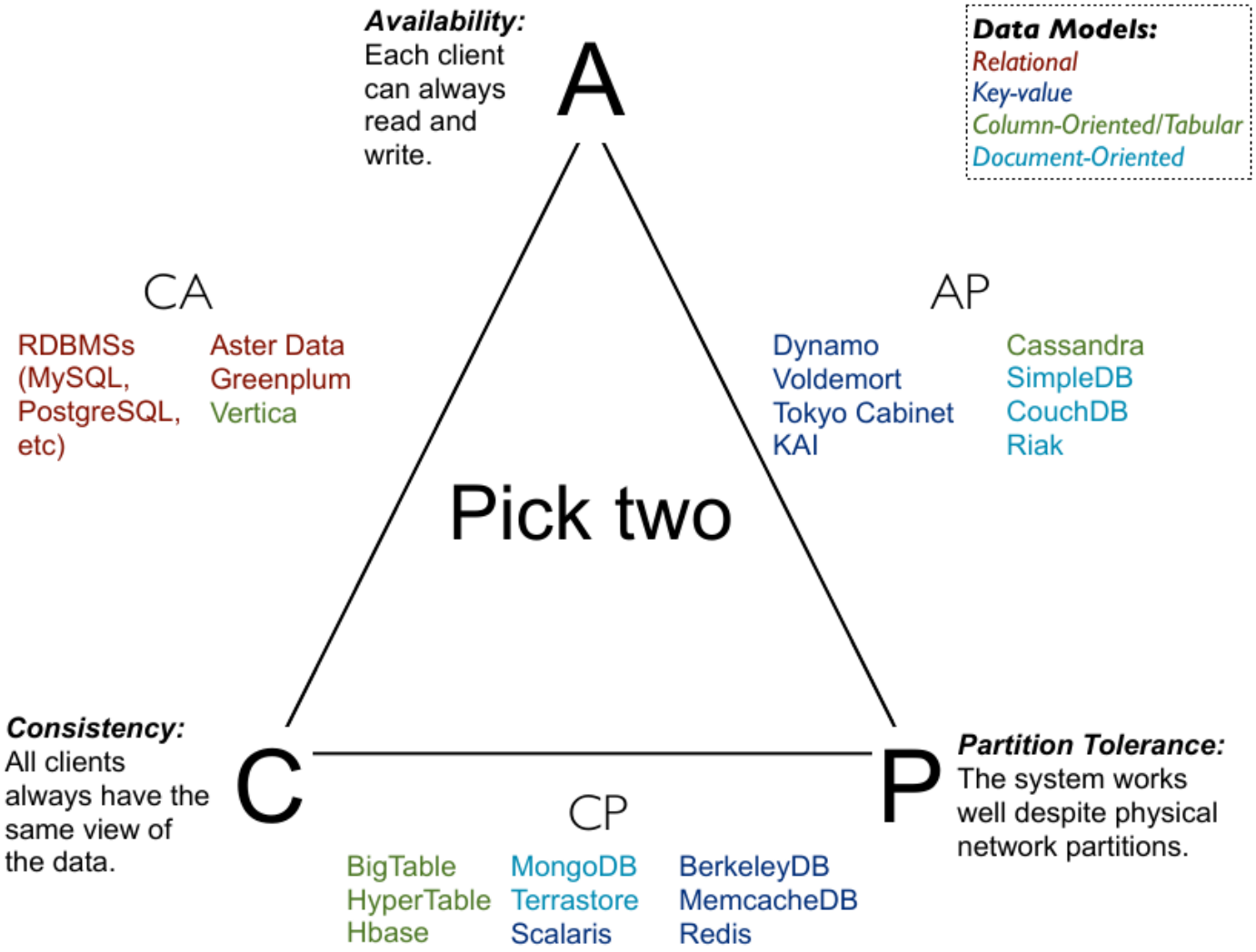
Illustration : CAP

Soit A et B deux utilisateurs du système, soit N1 et N2 deux nœuds du système.

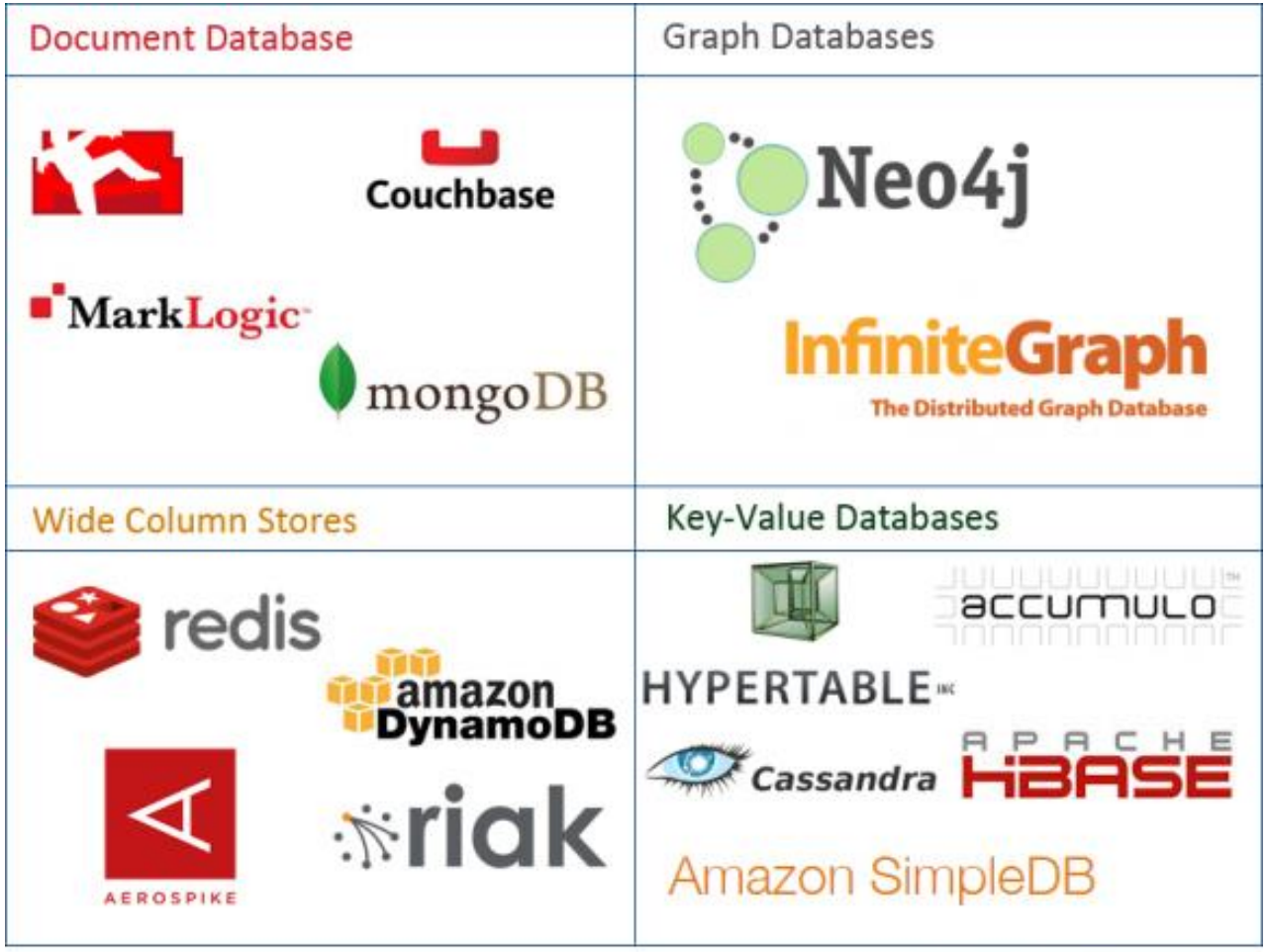
- **Si** A modifie une valeur sur N1, **alors** pour que B voie cette valeur sur N2 il faut attendre que N1 et N2 soient synchronisés.
- **Si** N1 et N2 doivent toujours servir des valeurs cohérentes, **alors** il y a un temps incompressible entre le début de l'écriture, la synchronisation et la lecture suivante.

Sur un système très chargé et très vaste, ce temps incompressible va considérablement influencer la disponibilité et la résistance au morcellement.

Guide visuel des solutions NoSQL



Les solutions NoSQL



Les solutions NoSQL

- **Clé/valeur**. à la manière des tableaux associatifs des langages de programmation. A une clé correspond une valeur.

- **Orienté colonne** : à une clé correspond un ensemble de colonne, chacune ayant une valeur.

- **Orienté document** : à une clé correspond des ensembles champs/valeurs qui peuvent être hiérarchisés

- **Orienté graphe** : les données sont modélisées sous forme de nœuds qui ont des liaisons entre eux.

MongoDB

MongoDB est une base de données orientée document sponsorisée par **10gen**.

- Un « document » est une entrée dans une base de donnée.

- Du JSON que Mongo stock en binaire(BSON)

MongoDB

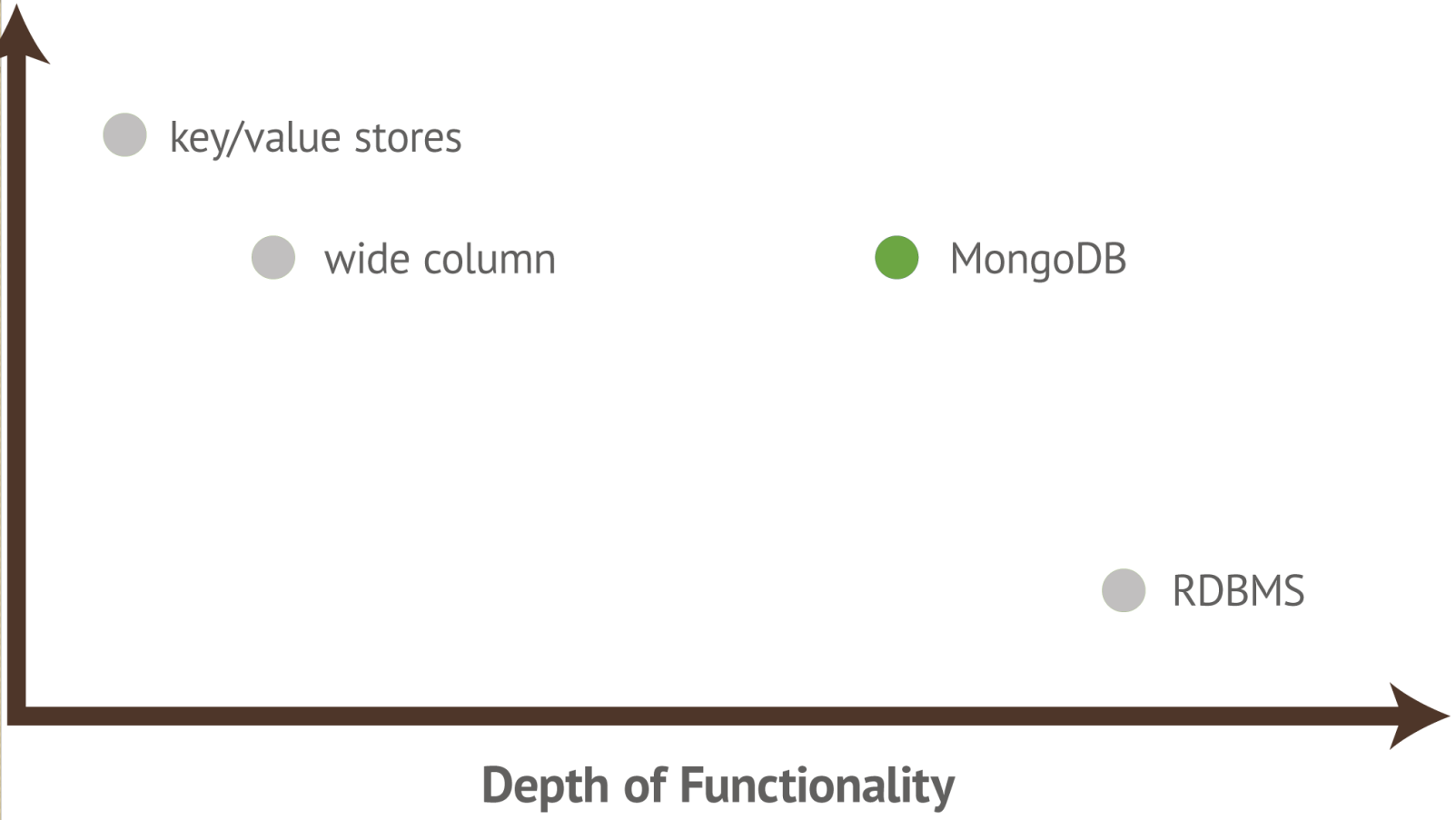
- Base orientée document signifie :
  - que les objets stockés sont représentés sous la forme d'un document BSON
  - permet facilement de se mapper sur les objets que l'on manipule dans nos programmes.

- MongoDB est **schemaless**.
  - aucun schéma de document n'est nécessaire pour pouvoir stocker les données.

MongoDB

- MongoDB est un SGBD :
  - orienté documents
  - libre
  - scalable : réplication, auto-sharding
  - flexible : pas de schéma de données, full-text index
  - écrit en C++.

Base de donnée opérationnelle



Modèle de donnée Document

Relationnel - Tables

PERSON

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rome

NO RELATION

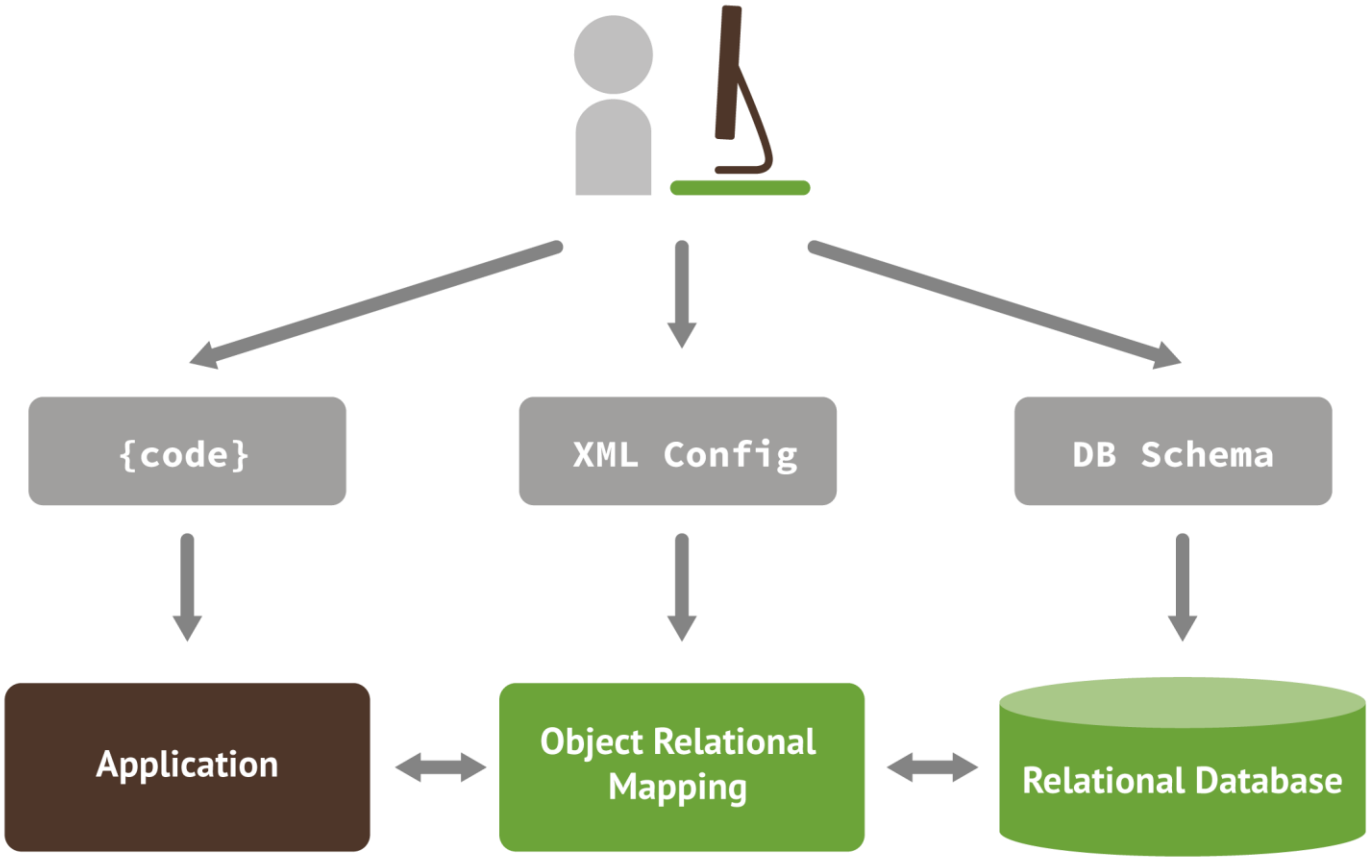
CAR

Car_ID	Model	Year	Value	Pers_ID
101	Bently	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

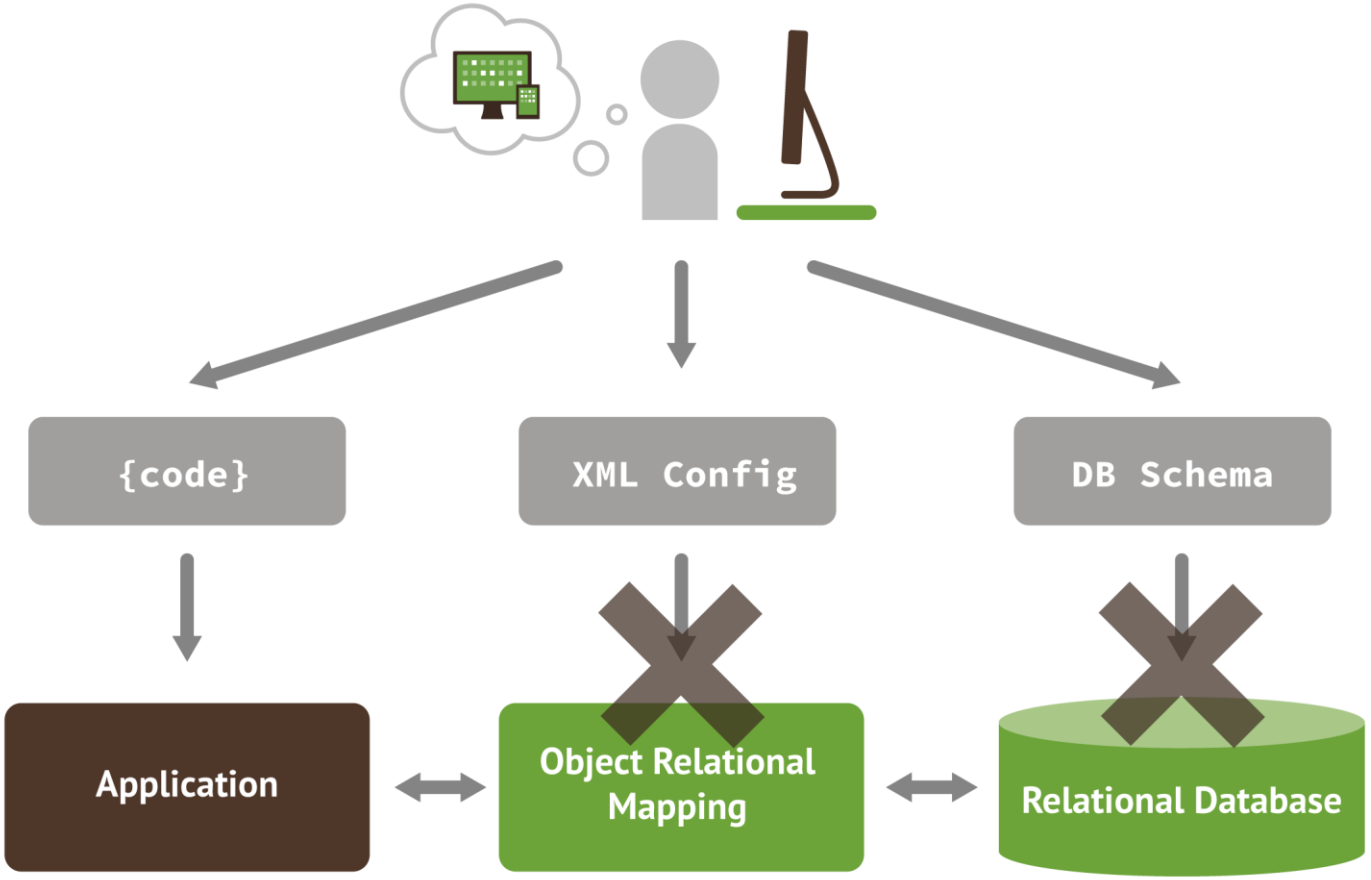
Document - Collections

```
{ first_name: 'Paul',
  surname: 'Miller',
  city: 'London',
  location: {
        type: "Point",
        coordinates :
                [-0.128,
51.507]
        },
  cars: [
    { model: 'Bentley',
      year: 1973,
      value: 100000, … },
    { model: 'Rolls Royce',
      year: 1965,
      value: 330000, … }
  ]
}
```

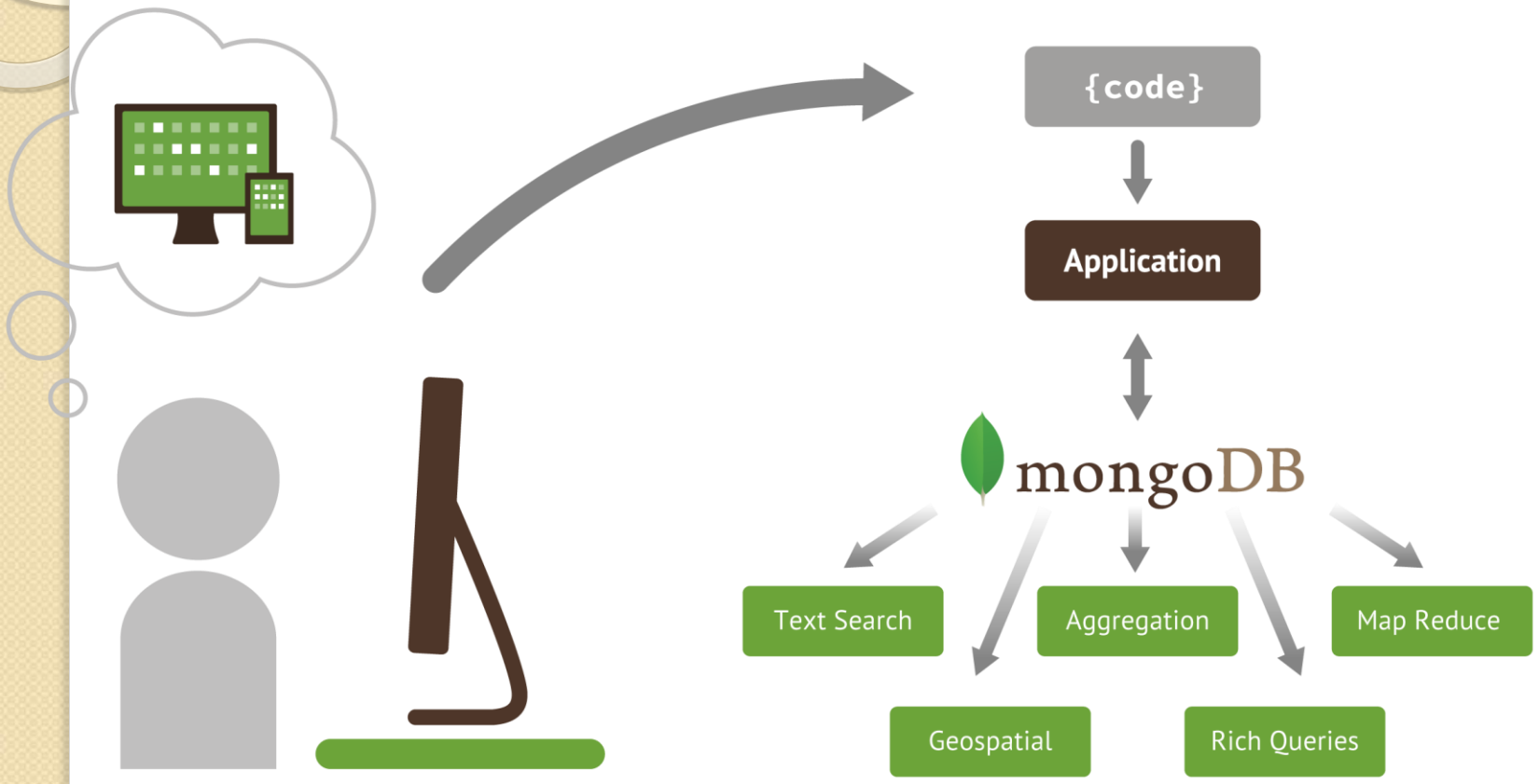
Développement traditionnel



Développement simplifié



Développement MongoDB



Shell and Drivers

Drivers

Drivers pour la majorité des langages de programmation et frameworks


Java


Ruby

JavaScript


Perl


Python


Haskell

Shell

Ligne de commande pour intéragir directement avec la base de données.

```
> db.collection.insert({company:"10gen", product:"MongoDB"})
>
> db.collection.findOne()
{
            "_id" : ObjectId("5106c1c2fc629bfe52792e86"),
            "company"    : "10gen"
            "product"    : "MongoDB"
}
```

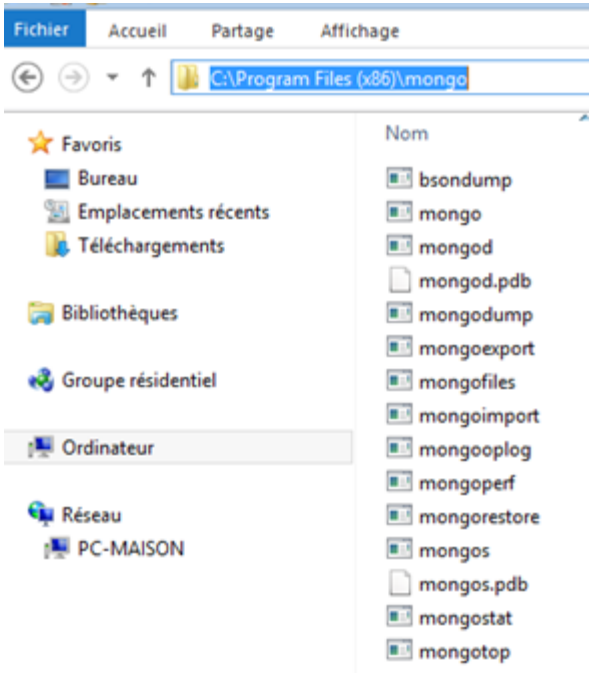
Vocabulaire

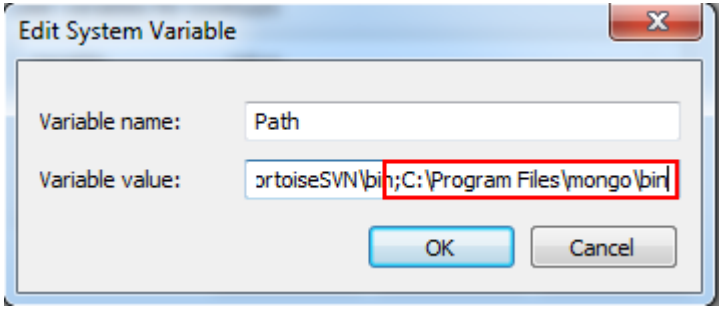
RDBMS		MongoDB
Database	→	Database
Table	→	Collection
Row	→	Document
Index	→	Index
Join	→	Embedded Document
Foreign Key	→	Reference

Installation de MongoDB

- **Télécharger**
  - http://www.mongodb.org/downloads
- **Décompressez/Renommez**
  - Déplacer dans par exemple:

    C:\Programs Files\mongo
- **Création du dossier de données**
  - Dans par exemple : **c:\data**
- **Configuration des variables d'environnement**
  - **Path: C:\Programs Files\mongo\bin**

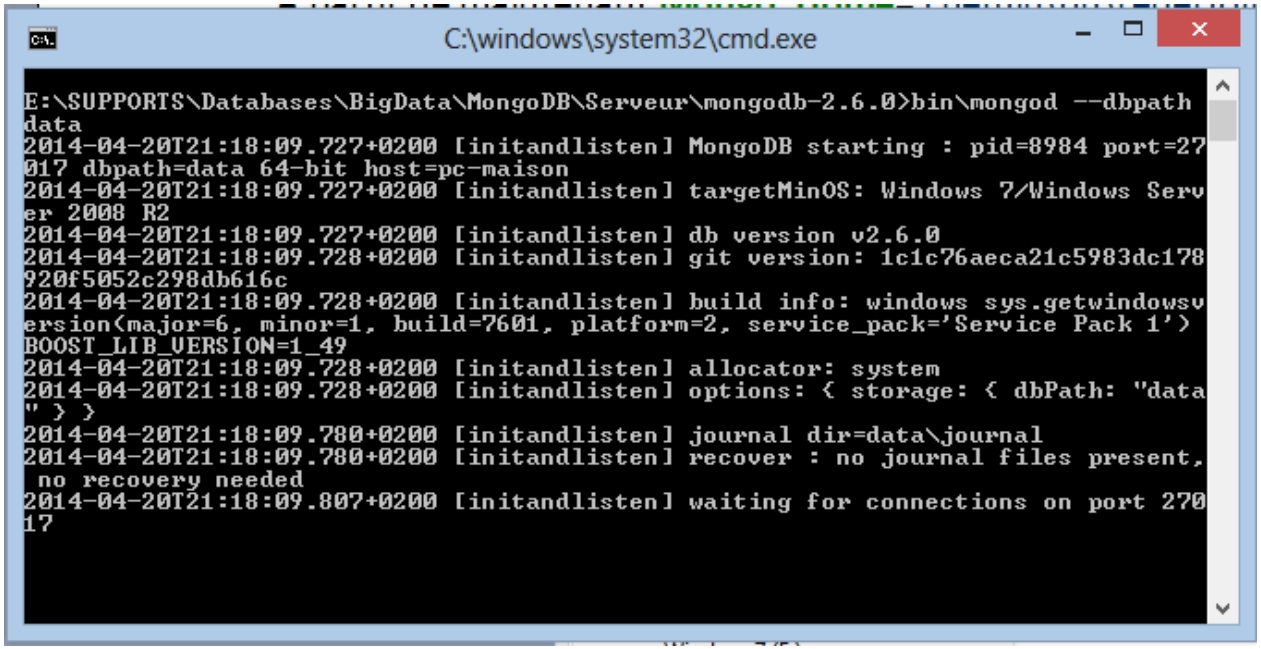




Démarrez une instance MongoDB

- Ouvrir un shell (ligne de commande DOS) et tapez:

  mongod --dbpath c:\data



Connexion

Pour utiliser MongoDB, entrez simplement ***mongo*** dans un terminal.

>mongo

MongoDB shell version: 2.6

connecting to: test

Nous sommes connectés à la base « test » .

Si nous souhaitons nous connecter à une autre base – peoples – nous utiliserons :

> use peoples

switched to db peoples

Commandes de base

Syntaxe	**show dbs** Lister les bases de données
Syntaxe	**use peoples** Création ou Connexion à une base
Syntaxe	**db.alpha.insert**({nom:"Jean"}) création de collection "alpha" et : insertion de document {nom:"Jean"}
Syntaxe	**db** Base sur laquelle votre session est en cours

Commandes de base

Syntaxe	**db.dropDatabase()** Supprimer une database
Syntaxe	**db.createCollection(name,option)** Création de collections (Tables)
Syntaxe	**db.nomCollection.drop()** Supprimer une collection
Syntaxe	**Crtl+C** Quitter.

Leçon 1 : FIN

Question ?

Leçon 1

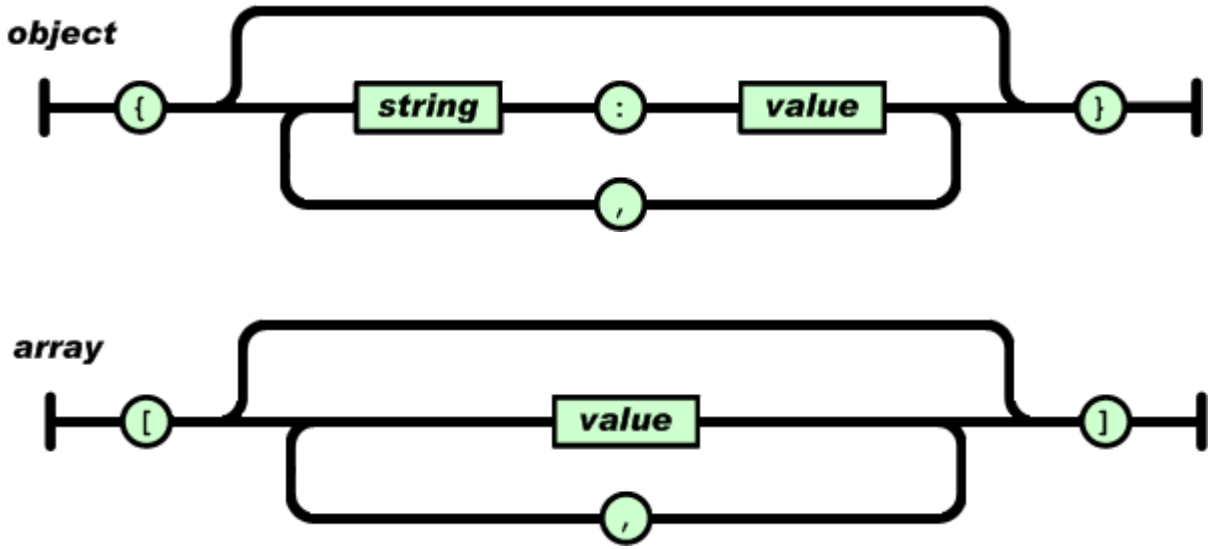
TP | **Prise en main MongoDB**

Vue d'ensemble de la formation

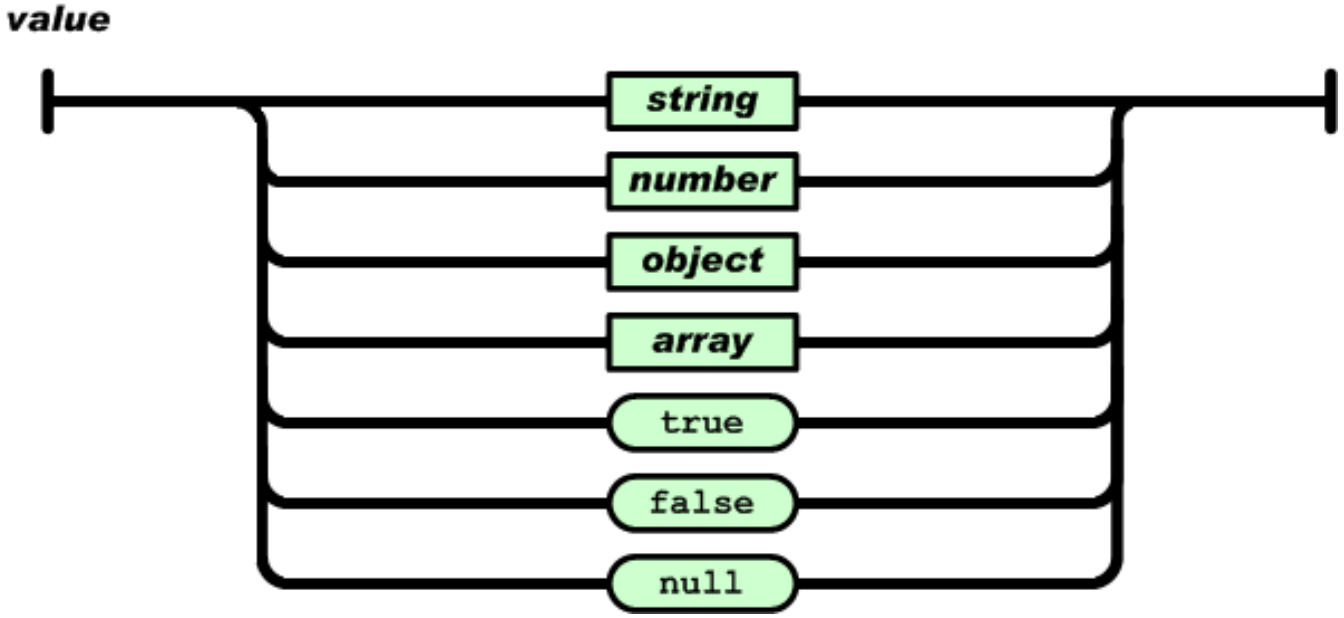
- Leçon 1 : Introduction au NoSQL
- **Leçon 2 : Opération CRUD**
- Leçon 3 : Les requêtes
- Leçon 4 : Design et Data Modéle
- Leçon 5 : Performance
- Leçon 6 : Agrégation Framework
- Leçon 7 : Administration
- Leçon 8 : Driver Java

Document JSON-Format

- Official Site: http://json.org/
- Un document **json** est de la forme: **{ }** (object) ou **[ ]** (tableau).



Document JSON-value



Document JSON-Exemple-I

- object:

```
{ "firstName":"John" , "lastName":"Doe" }
```

- array:

```
[ "Text goes here", 29, true, null ]
```

- Array with objects :

```
[
 { "name": "Dagny Taggart", "age": 39 },
 { "name": "Francisco D'Anconia", "age": 40 },
 { "name": "Hank Rearden", "age": 46 }
]
```

Document JSON-Exemple-2

Object with nested array

```
{
 "first": "John",
 "last": "Doe",
 "age": 39,
 "sex": "M",
 "salary": 70000,
 "registered": true,
 "interests": [ "Reading", "Mountain Biking", "Hacking" ]
}
```

Object with nested object

```
{
 "first": "John",
 "last": "Doe",
 "age": 39,
 "sex": "M",
 "salary": 70000,
 "registered": true,
 "favorites": {
  "color": "Blue",
  "sport": "Soccer",
  "food": "Spaghetti"
 }
}
```

Document JSON-Exemple-3

Object with nested arrays and objects

```
{
 "first": "John",
 "last": "Doe",
 "age": 39,
 "sex": "M",
 "salary": 70000,
 "registered": true,
 "interests": [ "Reading", "Mountain Biking", "Hacking" ],
 "favorites": {
  "color": "Blue",
  "sport": "Soccer",
  "food": "Spaghetti"
 },
 "skills": [
  {
   "category": "PHP",
   "tests": [
    { "name": "One", "score": 90 },
    { "name": "Two", "score": 96 }
   ]
  },
  {
   "category": "CouchDB",
   "tests": [
    { "name": "One", "score": 32 },
    { "name": "Two", "score": 84 }
   ]
  },
  {
   "category": "Node.js",
   "tests": [
    { "name": "One", "score": 97 },
    { "name": "Two", "score": 93 }
   ]
  }
 ]
}
```

Opération CRUD

Les opération de CRUD MongoDB sont disponibles sous formes de fonctions / méthodes (API) d'un langage de programmation et non comme un langage séparé (i.e SQL).

Vocabulaire:

	MongoDB	SQL
CREATE	Insert	Insert
READ	Find	Select
UPDATE	Update	Update
DELETE	Remove	Delete

Langage MongoDB

- **Mise-à-jour**
  - db.collection.insert()
  - db.collection.update()
  - db.collection.save()
  - db.collection.remove()

- **Interrogation**
  - db.collection.find()
  - db.collection.findOne()

Insert, Find & Count

- **db.collection.insert({…})**
- The collection unique ID field is called "**_id**" and can be provided. If not provided an ObjectID will be generated based on the time, machine, process-id and process dependent counter.
- "_id" does not have to be a scalar value – it can be a document, e.g. _id : {a: 1, b:'ronald'}
- **db.collection.find || findOne ({…}, {field1 : true, …}).pretty()** //no argument will find all docs
- **db.collection.count({…})**

Création d'un document

- Trois façons d'insérer un document dans mongoDB:
  - Insérer un document avec **insert**
  - Insérer un document avec **update**
  - Insérer un document avec **save**

Insérer un document avec **insert**

Syntaxe	db.collection.insert(document) db.collection.insert(documents)

- **document** est un tableau clefs / valeurs
- **documents** est une liste de tableaux clefs / valeurs
- **collection** est le nom de la collection à laquelle on souhaite ajouter le(s) document(s).

  Rem: Si la collection n'existait pas au préalable, elle est créée (c'est de cette manière qu'on crée les collections)

Insérer un document avec **insert**

Syntaxe	**db.collection.insert(document)** **db.collection.insert(documents)**

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

```
INSERT INTO users                  <-- table
            ( name, age, status )  <-- columns
VALUES      ( "sue", 26, "A" )     <-- values/row
```

Insérer un document avec **save**

Syntaxe	db.collection.save(document)

- Cas: Le document **contient** _id ➔ Remplace le document(de la base) par le nouveau document.

```
> db.personne.save({_id:4, prenom:"Jean", nom:"Paul"})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

- Cas : Le document **ne contient pas** _id ➔ Il fait une insertion.

```
> db.personne.save({prenom:"Luc",nom:"Larue"})
WriteResult({ "nInserted" : 1 })
```

MAJ d'un document avec **update**

Syntaxe	db.collection.update(criteria,donnée_maj) db.collection.update(criteria,donnée_maj,multi) db.collection.update(criteria,donnée_maj,upsert,multi)

- **criteria** est de la même forme que pour find
- **donnée_maj** est un tableau clefs / valeurs définissant des opérations sur les champs.
- **option**:
  - *multi* (booléen) :
    - false (défaut) : maj d'une occurrence trouvée (laquelle ?)
    - true : maj toutes les occurrence
  - *upsert* (booléen) :
    - false (défaut) : update
    - true : update or insert if not exists

MAJ d'un document avec **update**

Syntaxe	db.collection.update(criteria,donnée_maj,option)

```
db.users.update(                    ← collection
   { age: { $gt: 18 } },            ← update criteria
   { $set: { status: "A" } },       ← update action
   { multi: true }                  ← update option
)
```

```
UPDATE users          ← table
SET    status = 'A'   ← update action
WHERE  age > 18       ← update criteria
```

MAJ d'un tableau avec **update**

- db.collection.update({ myQuery }, {myField:"newValue", … })

  ➔ replaces the existing document

- db.collection.update({ myQuery }, {$set : {myField:"newValue"}})

  ➔ Create or update myField

- db.collection.update({ myQuery }, {$inc : {age: 1}})
- db.collection.update({ myQuery }, {$unset : {myField: 1}})
- db.collection.update({ myQuery }, {$set : {myField:"newValue"}}, {upsert: true})

  ➔ Create or update document specified by { myQuery } with myField

Arrays Update

- db.collection.update({ myQuery }, {$set : {"myArray.2":"x"}}) -> Set 3rd position of Array
- db.collection.update({ myQuery }, {$push : {myArray:"y"}})
- db.collection.update({ myQuery }, {$addToSet : {myArray:"y"}}) //will only add if does not exist yet
- db.collection.update({ myQuery }, {$pop : {myArray: 1}}) // pop right-most
- db.collection.update({ myQuery }, {$pop : {myArray: -1}}) // pop left-most
- db.collection.update({ myQuery }, {$pushAll : {myArray: ["a","b","c"]}})
- db.collection.update({ myQuery }, {$pull : {myArray:"c"}}) // remove value "c"
- db.collection.update({ myQuery }, {$pullAll : {myArray: ["a","b","c"]}})

Supprimer un document avec **remove**

Syntaxe	db.collection.remove() db.collection.remove(<query>)

- **sans paramètre ie {}**, tous les documents sont supprimés (**attention, donc**)
- **query** est de la même forme que pour find, elle désigne les documents qui seront supprimés.

Supprimer un document avec **remove**

Syntaxe	db.collection.remove(<query>)

```
db.users.remove(          ← collection
   { status: "D" }        ← remove criteria
)
```

```
DELETE FROM users         ← table
WHERE  status = 'D'       ← delete criteria
```

Lecture de document avec **find**

Syntaxe	db.collection.find() db.collection.find(<criteria>) db.collection.find(<criteria>,<projection>)

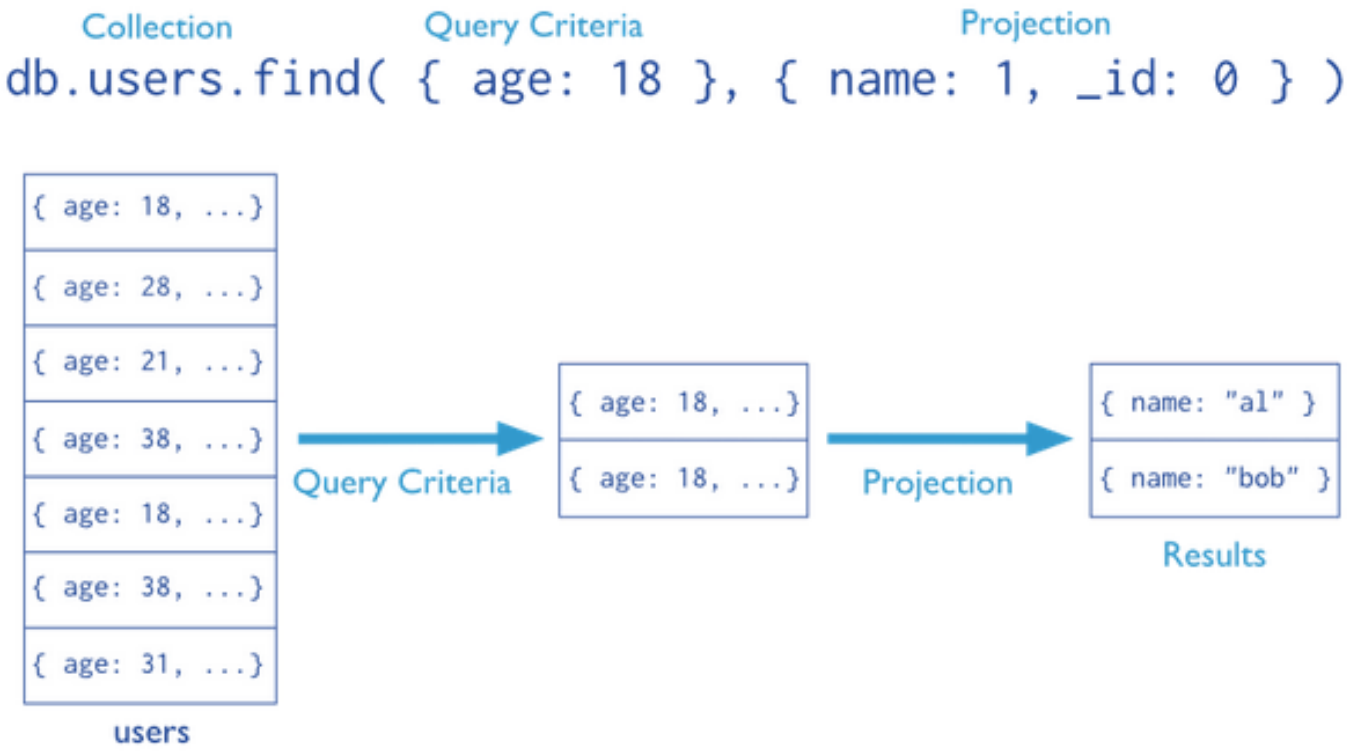
- **sans paramètre**, tous les documents sont renvoyés
- **criteria** est un tableau clefs / valeurs spécifiant des opérateurs sur les champs des documents recherchés
- **projection** est un tableau permettant de limiter les champs que l'on souhaite consulter dans les documents recherchés (cette option sera traitée ultérieurement)

- Exemple: db.etudiants.find()

  db.etudiants.find({ 'prenom':'Camille' })

Lecture de document avec **find**

Syntaxe	db.collection.find(<criteria>,<projection>)



Lecture de document avec **find**

Syntaxe	db.collection.findOne() db.collection.findOne(<criteria>) db.collection.findOne(<criteria>,<projection>)

- La fonction findOne fait la même chose que find mais sans s'embarrasser d'un curseur, lorsqu'on souhaite récupérer un document unique (par son identifiant, par exemple).

- Si la requête désigne plusieurs documents, le premier trouvé est renvoyé.

Curseurs

- myCursor = db.collection.find(); null;
  - append null as not to print out the cursor immediately
- myCursor.hasNext() myCursor.next()
- myCursor.skip(2).limit(5).sort({name : -1}); null; ☐ modifies the query executed on the server

```
>var cursor = db.articles.find ( { 'author' : 'Tug Grall' } )

>cursor.hasNext()
true

>cursor.next()
 {   '_id' :     ObjectId(...),
     'text': 'Article content…',
     'date' : ISODate(...),
     'title' : 'Intro to MongoDB',
     'author' : 'Dan Roberts',
     'tags' : [ 'mongodb', 'database', 'nosql' ]
 }
```

Leçon 2 : FIN

Question ?

Leçon 2

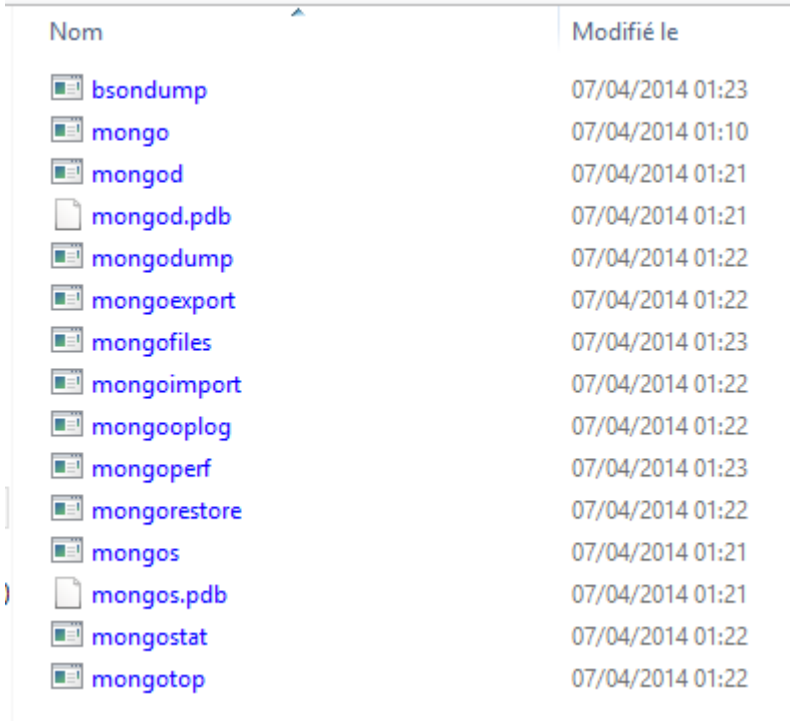
TP	Opérations CRUD

Vue d'ensemble de la formation

- Leçon 1 : Introduction au NoSQL
- Leçon 2 : Opération CRUD
- **Leçon 3 : Les requêtes**
- Leçon 4 : Les requêtes
- Leçon 5 : Design et Data Modéle
- Leçon 6 : Performance
- Leçon 7 : Agrégation Framework
- Leçon 8 : Administration

Quelques commandes pratiques

Sous le répertoire *bin* se trouvent plusieurs exécutables.



Nom	Modifié le
bsondump	07/04/2014 01:23
mongo	07/04/2014 01:10
mongod	07/04/2014 01:21
mongod.pdb	07/04/2014 01:21
mongodump	07/04/2014 01:22
mongoexport	07/04/2014 01:22
mongofiles	07/04/2014 01:23
mongoimport	07/04/2014 01:22
mongooplog	07/04/2014 01:22
mongoperf	07/04/2014 01:23
mongorestore	07/04/2014 01:22
mongos	07/04/2014 01:21
mongos.pdb	07/04/2014 01:21
mongostat	07/04/2014 01:22
mongotop	07/04/2014 01:22

mongodump

Syntaxe	mongodump --db databaseCible --out databaseTarget

Permet de faire un dump(Sauvegarde) de la base de données :

```
mongodb-2.6.0\bin>mongodump --db test --out ./data/mongodump
connected to: 127.0.0.1
2014-05-04T13:19:12.105+0200 DATABASE: test    …29467 documents.
```

- **A Noter :** Sous le répertoire **/data/dumpmongo** nous trouverons un nouveau dossier « test » qui contient deux fichiers : system.indexes.bson et products.bson

mongorestore

Syntaxe	mongorestore --db databaseTarget –drop databaseOrig

- Permet d'installer une base de données à partir d'un dump dans un répertoire local ou distant en utilisant le paramètre –host « l'ip remote »

mongorestore --db test --drop ./data/mongodump/test

connected to: 127.0.0.1

2014-05-04T13:19:12.105+0200 DATABASE: test …29467 documents.

mongoimport

Syntaxe	mongoimport --db test --c collection < nomFile

Exemple:

```
>mongoimport -d students -c grades < grades.js
connected to: 127.0.0.1
2014-04-26T14:46:35.460+0200 check 9 800
2014-04-26T14:46:35.461+0200 imported 800 objects
```

Les opérateurs

$each, $slice, $sort, $inc, $push

$inc, $rename, $setOnInsert, $set, $unset, $max, $min

$, $addToSet, $pop, $pullAll, $pull, $pushAll, $push

$each, $slice, $sort,$size

$gt, $gte, $in, $lt, $lte, $ne, $nin

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

Opérateurs de Comparaison

Syntaxe	{myField: {$gt: 100, $lt: 10}}

- **$gt, $gte, $in, $lt, $lte, $ne**

Exemple:

```
> db.grades.find({score:{$gt:95}})
{ "_id" : 1, "student_id" : 1, "type" : "quiz", "score" : 96.76851542258362 }
{ "_id" : 2, "student_id" : 2, "type" : "homework", "score" : 97.75889721343528 }
...
Type "it" for more


> db.grades.find({score:{$gt:85,$lt:95},type:"quiz"})
{ "_id" : 1, "student_id" : 20, "type" : "quiz", "score" : 92.76554684090782 }
{ "_id" : 2, "student_id" : 22, "type" : "quiz", "score" : 86.0800081592629 }
...
```

Opérateur sur les éléments

Syntaxe	{myField: {$exists: true}} , {myField: {$type: 2}}

- $exists: checks if particular key exists in document
- $type: 2=String as defined in BSON spec

Exemple:

```
> db.people.find({profession:{$exists:true}})
{ "_id" : 1, "name" : "Smith", "age" : 30, "profession" : "hacker" }
{ "_id" : 2, "name" : "Jones", "age" : 35, "profession" : "boulanger" }

> db.people.find({name:{$type:2}})
{ "_id" : ObjectId("535bb427a8f64ced71fb8b74"), "name" : "Bob" }
{ "_id" : ObjectId("535bb442a8f64ced71fb8b75"), "name" : "Charlie" }
{ "_id" : ObjectId("535bb44ca8f64ced71fb8b76"), "name" : "Dave" }
```

Opérateurs Evalué

Utilisé pour requêter la base de données

Syntaxe	$mod, $regex, $search

Exemple:

```
> db.people.find({name:{$regex:"e$"}})
{ "_id" : ObjectId("535bb442a8f64ced71fb8b75"), "name" : "Charlie" }
{ "_id" : ObjectId("535bb44ca8f64ced71fb8b76"), "name" : "Dave" }

> db.people.find({name:{$regex:"^C"}})
{ "_id" : ObjectId("535bb442a8f64ced71fb8b75"), "name" : "Charlie" }
```

Opérateurs: Logique

Syntaxe	{$or: [{…}, {…}, …]}

- **$and, $not, $ne**
- $not - negates result of other operation or regular expression query
- tags: {$ne:"gardening"} // works on keys pointing to single values or arrays – inefficient – can't use indexes

```
> db.people.find({$or:[{name:{$regex:"e$"}},{age:{$exists:true}}]})
{ "_id" : ObjectId("535bb442a8f64ced71fb8b75"), "name" : "Charlie" }
{ "_id" : ObjectId("535bb44ca8f64ced71fb8b76"), "name" : "Dave" }
```

```
> db.people.find({$and:[{name:{$gt:"C"}},{name:{$regex:"a"}}]})
{ "_id" : ObjectId("535bb442a8f64ced71fb8b75"), "name" : "Charlie" }
{ "_id" : ObjectId("535bb44ca8f64ced71fb8b76"), "name" : "Dave" }
{ "_id" : ObjectId("535bb456a8f64ced71fb8b77"), "name" : "Edgar" }
```

Requête sur une liste (embedded)

```
> db.accounts.find().pretty()
{
        "_id" : ObjectId("535bd635a8f64ced71fb8b7c"),
        "name" : "George",
        "favorites" : [
                "ice cream",
                "pretzels"
        ]
}
```

```
> db.accounts.find({favorites:"beer",name:{$gt:"H"}})
{ "_id" : ObjectId("535bd667a8f64ced71fb8b7d"), "name" : "Howard", "favorites" : [ "pretzels", "beer" ] }
```

Opérateur sur une liste

Syntaxe	$in, $all

```
> db.accounts.find().pretty()
{
        "_id" : ObjectId("535bd635a8f64ced71fb8b7c"),
        "name" : "George",
        "favorites" : [
                "ice cream",
                "pretzels"
        ]
}
```

```
> db.accounts.find({favorites  : { $all : ["pretzels","beer"]} })
{ "_id" : 1, "name" : "Howard", "favorites" : [ "pretzels", "beer" ] }
{ "_id" : 2, "name" : "Irving", "favorites" : [ "pretzels", "beer", "cheese" ] }
```

```
> db.accounts.find({favorites  : { $in : ["pretzels","beer"]} })
{ "_id" 3, "name" : "George", "favorites" : [ "ice cream", "pretzels" ] }
{ "_id" : 4, "name" : "Howard", "favorites" : [ "pretzels", "beer" ] }
```

Requête avec la notation «.»

```
> db.users.findOne()
{
        "_id" : ObjectId("535be5b1a8f64ced71fb8b81"),
        "name" : "richard",
        "email" : {
                "work" : "richard@yahoo.fr",
                "personnal" : "richard@gmail.com"
        }
}
```

```
> db.users.find({"email.work"  : "richard@yahoo.fr"})
{ "_id" : 1, "name" : "richard", "email" : { "work" : "richard@yahoo.fr",
                                             "personnal" : "richard@gmail.com" } }
```

Opérateur sur une liste

Syntaxe	$push, $pop, $pull, $pushAll, $pullAll, $addToSet

```
> db.arrays.update({_id:0 },{$set:{"a.2":5}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

```
> db.arrays.update({_id:0 },{$push : {a:6}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

```
> db.arrays.update({_id:0 },{$pop : {a:1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Leçon 3 : FIN

Question ?

Leçon 3

TP	Opérateurs

Vue d'ensemble de la formation

- Leçon 1 : Introduction au NoSQL
- Leçon 2 : Opération CRUD
- Leçon 3 : Les requêtes
- **Leçon 4 : Design et Data Modèle**
- Leçon 5 : Performance
- Leçon 6 : Agrégation Framework
- Leçon 7 : Administration
- Leçon 8 : Driver Java

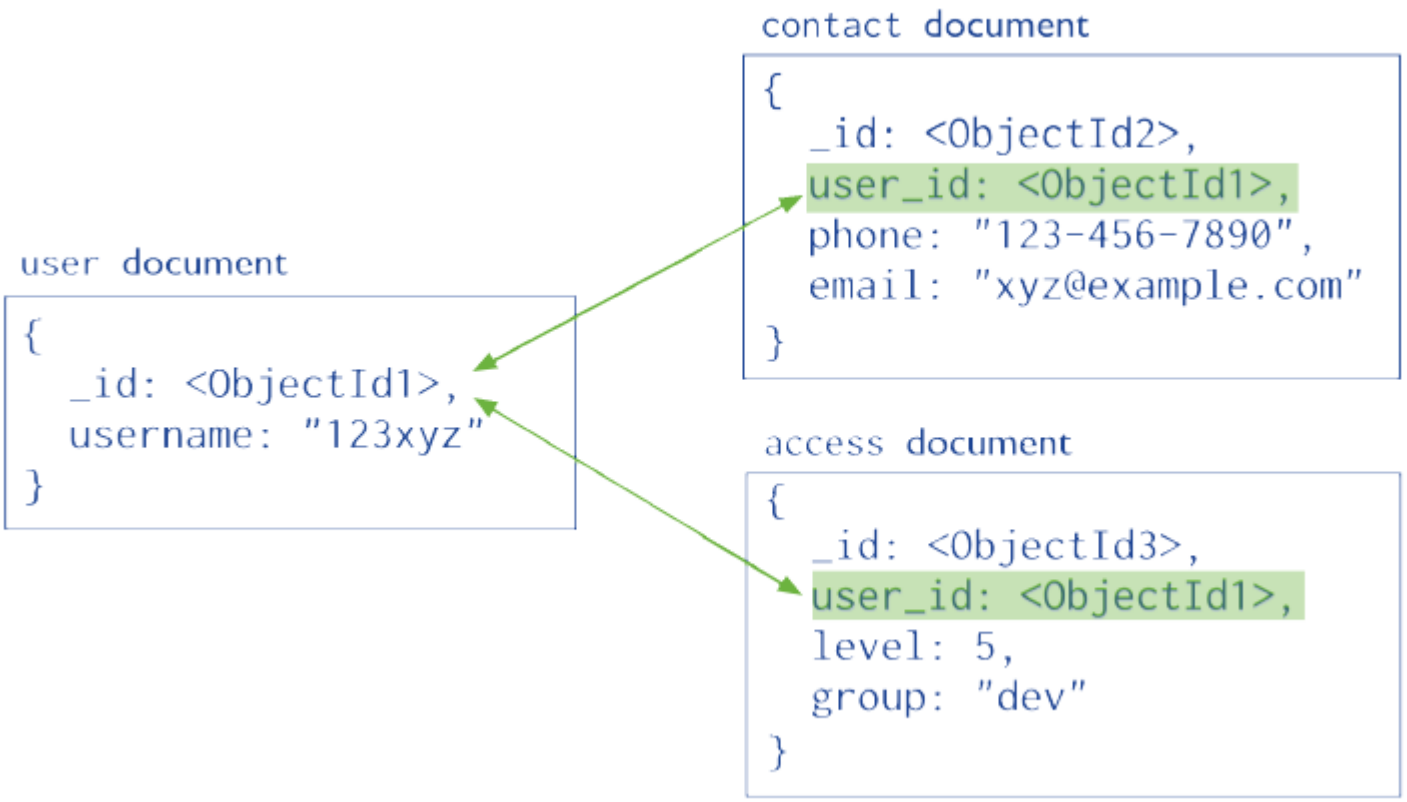
Vocabulaire

RDBMS		MongoDB
Database	→	Database
Table	→	Collection
Row	→	Document
Index	→	Index
Join	→	Embedded Document
Foreign Key	→	Reference

There are some patterns

- **Embedding**
- **Linking(par reference)**

Linking


contact document
{
_id: <ObjectId2>,
user_id: <ObjectId1>,
phone: "123-456-7890",
email: "xyz@example.com"
}
user document
{
_id: <ObjectId1>,
username: "123xyz"
}
access document
{
_id: <ObjectId3>,
user_id: <ObjectId1>,
level: 5,
group: "dev"
}

Embedding & Linking

One to One Pattern

```
{
    _id: "joe",
    name: "Joe Bookreader"
}

{
    patron_id: "joe",
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA",
    zip: "12345"
}
```

Référence

Embedded

```
{
    _id: "joe",
    name: "Joe Bookreader",
    address: {
                street: "123 Fake Street",
                city: "Faketon",
                state: "MA",
                zip: "12345"
             }
}
```

One to many Pattern

3 solutions possibles

- Embedding
- One To Few
- One To Many

One to many vs Embedding

```
{
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher: {
                name: "O'Reilly Media",
                founded: 1980,
                location: "CA"
              }
}

{
    title: "50 Tips and Tricks for MongoDB Developer",
    author: "Kristina Chodorow",
    published_date: ISODate("2011-05-06"),
    pages: 68,
    language: "English",
    publisher: {
                name: "O'Reilly Media",
                founded: 1980,
                location: "CA"
              }
}
```

One to Few

```
{
    name: "O'Reilly Media",
    founded: 1980,
    location: "CA",
    books: [12346789, 234567890, ...]
}

{
     _id: 123456789,
     title: "MongoDB: The Definitive Guide",
     author: [ "Kristina Chodorow", "Mike Dirolf" ],
     published_date: ISODate("2010-09-24"),
     pages: 216,
     language: "English"
}

{
    _id: 234567890,
    title: "50 Tips and Tricks for MongoDB Developer",
    author: "Kristina Chodorow",
    published_date: ISODate("2011-05-06"),
    pages: 68,
    language: "English"
}
```

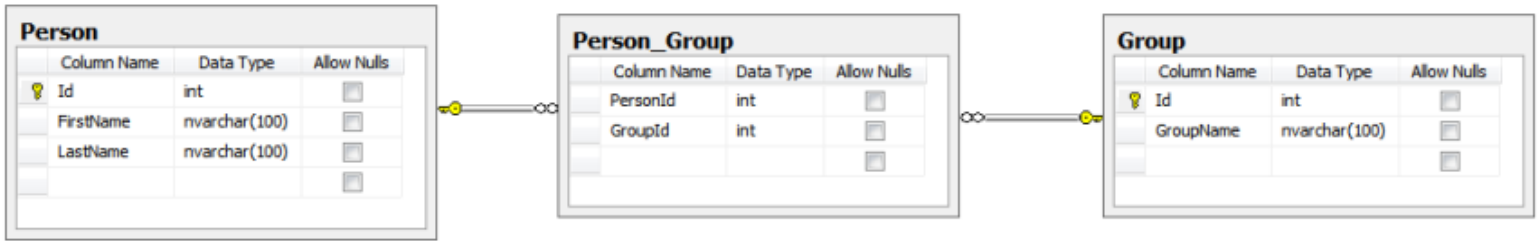
One to many

```
{
    _id: "oreilly",
    name: "O'Reilly Media",
    founded: 1980,
    location: "CA"
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher_id: "oreilly"
}

{
    _id: 234567890,
    title: "50 Tips and Tricks for MongoDB Developer",
    author: "Kristina Chodorow",
    published_date: ISODate("2011-05-06"),
    pages: 68,
    language: "English",
    publisher_id: "oreilly"
}
```

Linking vs. Embedding

- Embedding représente une pré-jointure de données.
- Des opérations au niveau du document sont faciles à gérer pour le serveur.
- Embarquer lorsque les objets «Many» apparaissent toujours avec leurs parents.
- Relier lorsque vous avez besoin de plus de flexibilité.

Many to many relationship



Many to many relationship

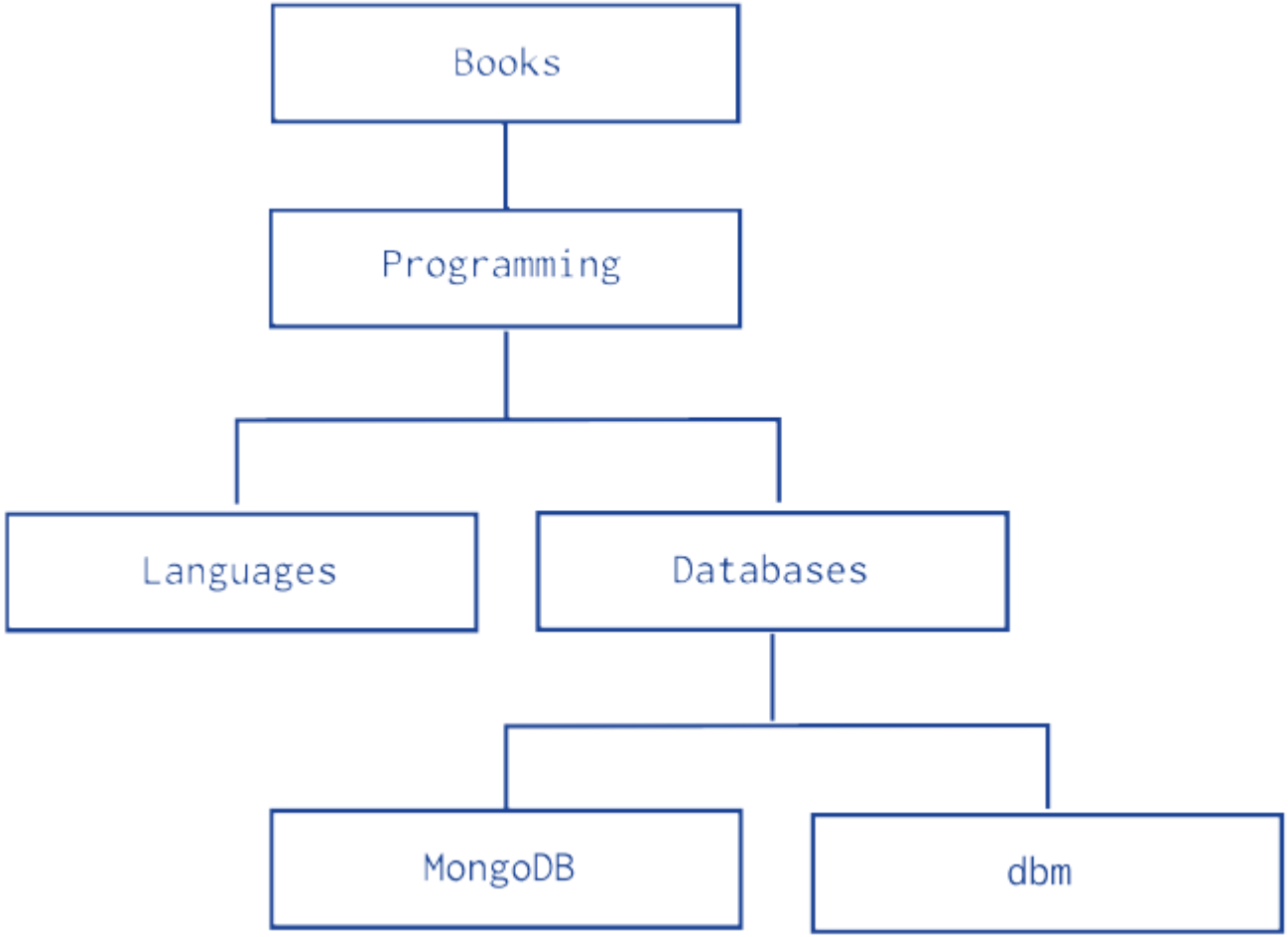

```
{
  "_id": "4e54ed9f48dc5922c0094a43",
  "firstName": "Joe",
  "lastName": "Mongo",
  "groups": [
    "4e54ed9f48dc5922c0094a42",
    "4e54ed9f48dc5922c0094a41"
  ]
}

{
  "_id": "4e54ed9f48dc5922c0094a40",
  "firstName": "Sally",
  "lastName": "Mongo",
  "groups": [
    "4e54ed9f48dc5922c0094a42"
  ]
}
```

```
{
  "_id":"4e54ed9f48dc5922c0094a42",
  "groupName": "mongoDB User",
  "persons": [
    "4e54ed9f48dc5922c0094a43",
    "4e54ed9f48dc5922c0094a40"
  ]
}

{
  "_id": "4e54ed9f48dc5922c0094a41",
  "groupName": "mongoDB Administrator",
  "persons": [
    "4e54ed9f48dc5922c0094a43"
  ]
}
```

Many to many

```
// Get all persons in the "mongoDB User" group
db.person.find({"groups": "4e54ed9f48dc5922c0094a42"});

// Get all persons in the "mongoDB Administrator" group
db.person.find({"groups": "4e54ed9f48dc5922c0094a41"});

// Get all groups for "Joe Mongo"
db.groups.find({"persons": "4e54ed9f48dc5922c0094a43"});

// Get all groups for "Sally Mongo"
db.groups.find({"persons": "4e54ed9f48dc5922c0094a40"});
```

Modèle hiérarchique



Modèle hiérarchique

Solution 1 : Lien Parent

```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )

db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

Pour récupérer le parent

```
db.categories.findOne( { _id: "MongoDB" } ).parent
```

Pour récupérer les enfants

```
db.categories.find( { parent: "Databases" } )
```

Modèle hiérarchique

Solution II : Liste enfants

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )
db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

Pour récupérer le parent

```
db.categories.find( { children: "MongoDB" } )
```

Pour récupérer les enfants

```
db.categories.findOne( { _id: "Databases" } ).children
```

Modèle hiérarchique

Solution III Liste ancêtres

```
db.categories.insert( { _id: "MongoDB", ancestors: [ "Books", "Programming", "Databases" ], parent: "Databases" } )

db.categories.insert( { _id: "dbm", ancestors: [ "Books", "Programming", "Databases" ], parent: "Databases" } )

db.categories.insert( { _id: "Databases", ancestors: [ "Books", "Programming" ], parent: "Programming" } )

db.categories.insert( { _id: "Languages", ancestors: [ "Books", "Programming" ], parent: "Programming" } )

db.categories.insert( { _id: "Programming", ancestors: [ "Books" ], parent: "Books" } )

db.categories.insert( { _id: "Books", ancestors: [ ], parent: null } )
```

Pour récupérer les ancêtres

```
db.categories.findOne( { _id: "MongoDB" } ).ancestors
```

Pour récupérer les descendants

```
db.categories.find( { ancestors: "Programming" } )
```

Leçon 4 : FIN

Question ?

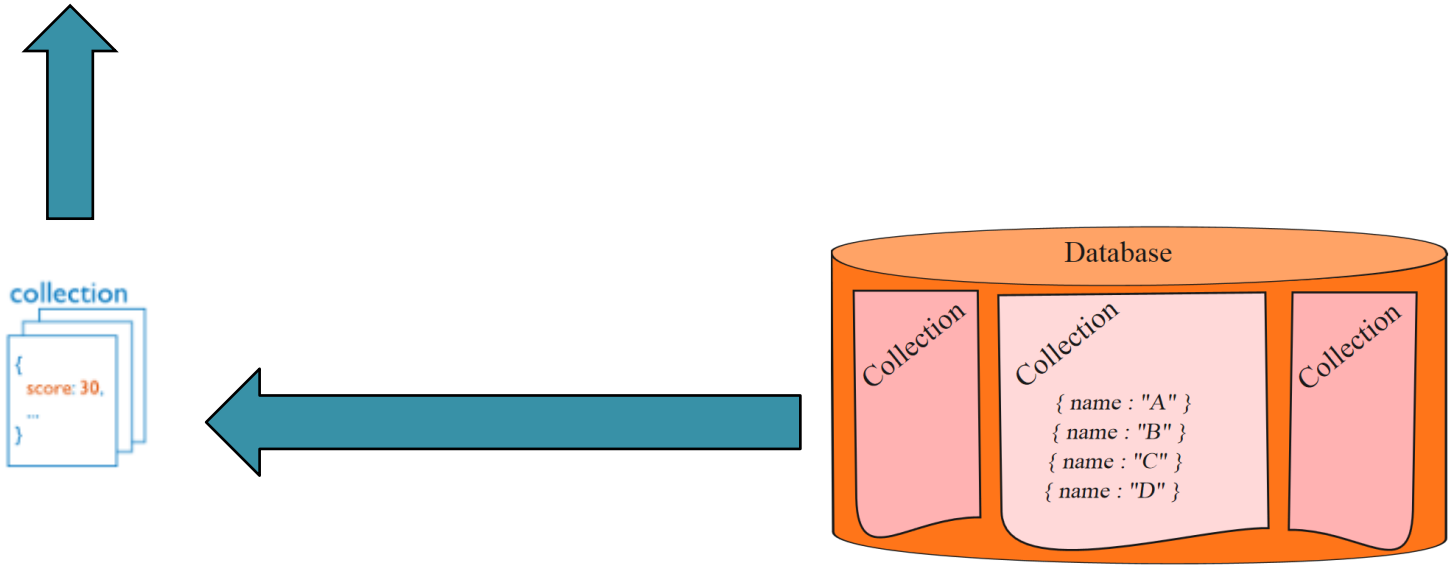
Vue d'ensemble de la formation

- Leçon 1 : Introduction au NoSQL
- Leçon 2 : Opération CRUD
- Leçon 3 : Les requêtes
- Leçon 4 : Design et Data Modéle
- **Leçon 5 : Performance**
- Leçon 6 : Agrégation Framework
- Leçon 7 : Administration
- Leçon 8 : Driver Java
- Leçon 9 : Spring Data

Index

- Que fait la base de données MongoDB quand nous l'interrogeons?
  - MongoDB doit analyser chaque document
  - Processus inefficace lorsqu'on a un grand volume de données

db.users.find({ score: { "$lt" : 30} })



Definition of Index

- **Definition:** Les index sont des structures de données particulières qui stockent une petite partie des données d'une collection dans un format simple et facile.

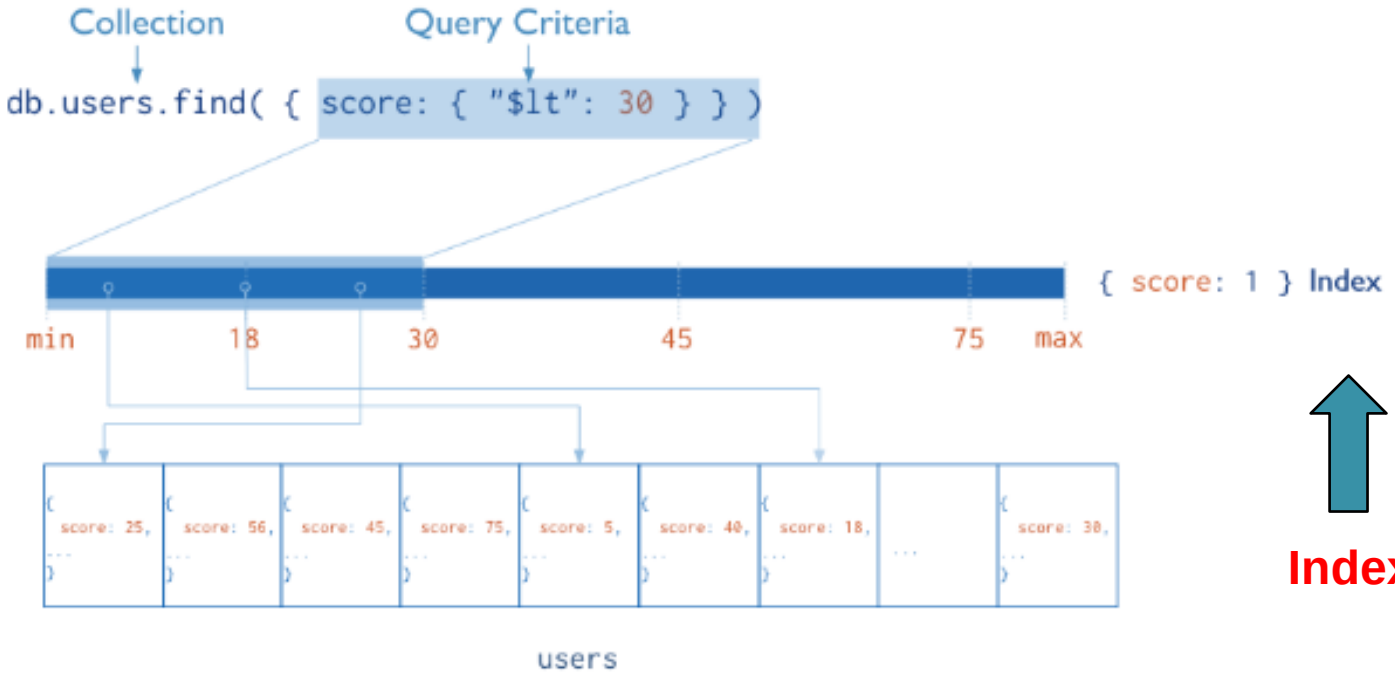


Diagram of a query that uses an index to select

Index in MongoDB

Operations

- **Creation index**
  - **db.users.ensureIndex({ score: 1 })**
- **Show existing indexes**
  - **db.users.getIndexes()**
- **Drop index**
  - db.users.dropIndex({score: 1})
- **Explain—Explain**
  - **db.users.find().explain()**
  - Returns a document that describes the process and indexes
- **Hint**
  - **db.users.find().hint({score: 1})**
  - Overide MongoDB’s default index selection

Index Types

- **Single Field Indexes (Index champ Unique)**
- **Compound Field Indexes(Index a champ composé)**
- **Multikey Indexes (Index à clef multiple)**

Création des Indexes

- Single Field Index
- Compound Field Indexes
- Multikey Indexes

```
db.zips.ensureIndex({pop: -1})
db.zips.ensureIndex({state: 1, city: 1})
db.zips.ensureIndex({loc: -1})
```

Index Types

- **Single Field Indexes**
  - db.users.ensureIndex({ score: I })

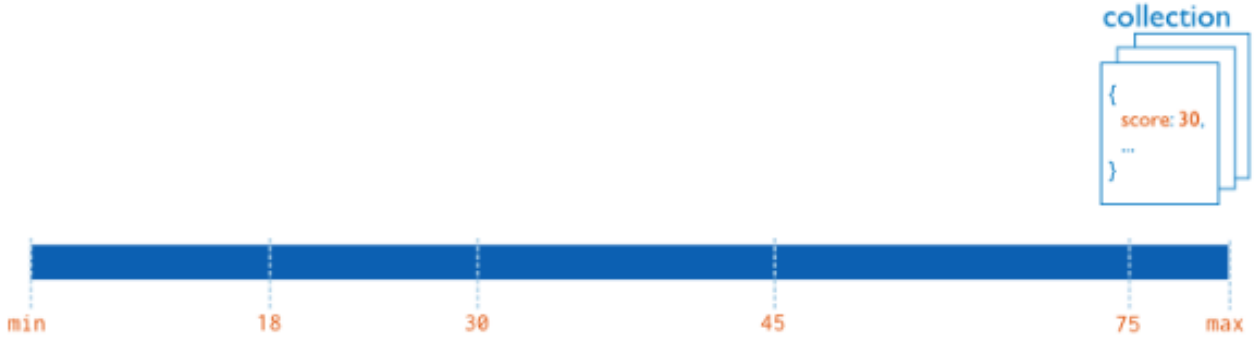


Diagram of an index on the score field (ascending).

Single Field Indexes

- db.collection.ensureIndex({student_id:1})

- 1=ascending,
- -1=descending
  - important for sorting not so much for searching

- Sorting can also use a reverse index if the sort criteria are exactly the reverse of an (simple or compound) index

Index Types

- **Compound Field Indexes**
  - db.users.ensureIndex({ userid:1, score: -1 })

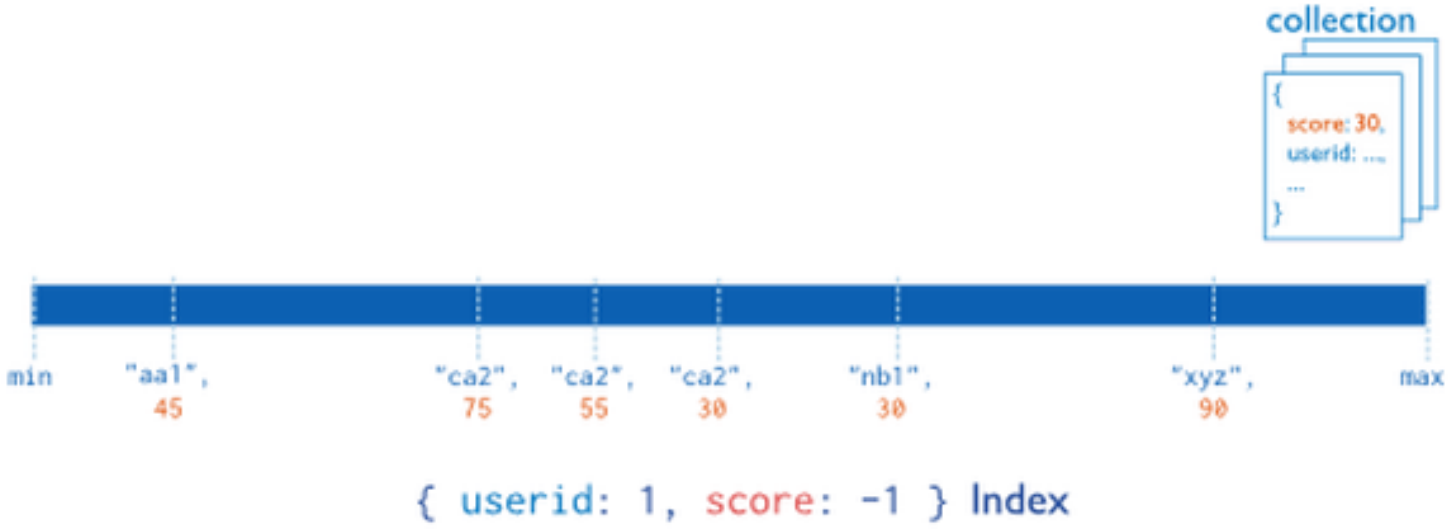


Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

Compound Field Indexes

- General rule: A Query where one term demands an exact match and another specifies a range requires a compound index where the range key comes second.

Index generality

- *Unique Index*: db.collection.ensureIndex({student_id:1}, {unique: true})
  - // dropDups: true ➔ dangerous
- By default index creation is done in the foreground which is fast but blocking all other writers to the same DB.
- Background index creation {background: true} will be slow but it will not block the writers
- We want indexes to be in memory. Find out the index size: db.col.stats() or db.col.totalIndexSize()
- db.system.indexes.find() ➔ finds all indexes of the current db
- db.collection.getIndexes() ➔ all indexes of collection
- db.collection.dropIndex({student_id:1})

Index Types

- **Multikey Indexes**
  - db.users.ensureIndex({ addr.zip:1})

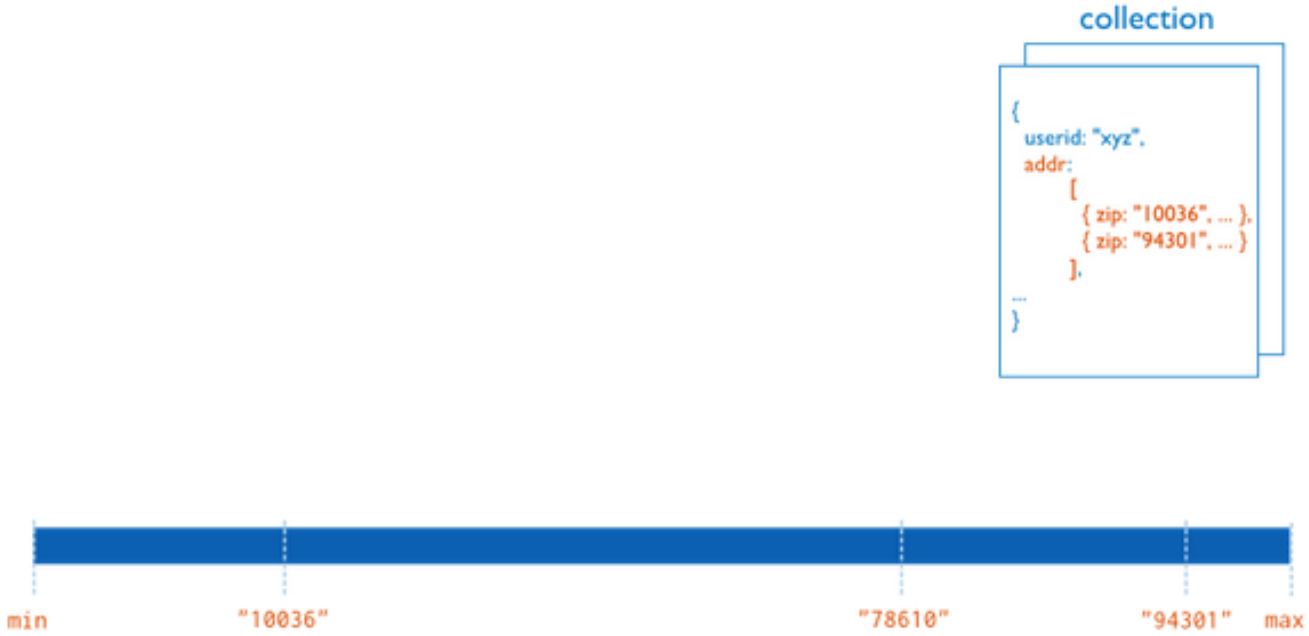


Diagram of a multikey index on the `addr.zip` field. The `addr` field contains an array of address documents. The address documents contain the `zip` field.

Multikey Indexes

- A multi key index is an index on an array field of a document, e.g. a student document has array of teacher-ids. One can add a multi key index on the teachers-array, which indexes all of the values in the array for all the documents.
- Multi key indices are one of the reason that linking works so well in MongoDB
- It is not possible to have a compound index with two array (multi key) fields

Sparse Index

- Missing index key in documents map to null
  - unique key not possible because multiple nulls are not allowed
- Sparse indexes only index documents that have a key set for the key being indexed {unique: true, sparse: true}
- On a sorted find the non-indexed documents will not be found when the sparse index is used for the sort

Explain & Hint

- db.collection.find({…}).explain()
- db.collection.find({…}).explain(true) //shows all possible plans
- db.collection.find({…}).hint({a:1, b:1}) // use specified index
- db.collection.find({…}).hint({$natural:1}) // use no index

- In Java:
  - .find(query).hint("IndexName") OR
  - .find(query).hint(new BasicDocument(a, 1).append(b, 1))

Explain vs sans index

```
> db.zips.dropIndexes()
{
        "nIndexesWas" : 4,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
> db.zips.find({city: 'NASHVILLE', state: 'TN'}).explain()
{
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 19,
        "nscannedObjects" : 29467,
        "nscanned" : 29467,
        "nscannedObjectsAllPlans" : 29467,
        "nscannedAllPlans" : 29467,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 33,
        "indexBounds" : {

        },
        "server" : "g:27017"
}
```

Explain vs avec index

```
> db.zips.find({city: 'NASHVILLE', state: 'TN'}).explain()
{
        "cursor" : "BtreeCursor state_1_city_1",
        "isMultiKey" : false,
        "n" : 19,
        "nscannedObjects" : 19,
        "nscanned" : 19,
        "nscannedObjectsAllPlans" : 19,
        "nscannedAllPlans" : 19,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 0,
        "indexBounds" : {
                "state" : [
                        [
                                "TN",
                                "TN"
                        ]
                ],
                "city" : [
                        [
                                "NASHVILLE",
                                "NASHVILLE"
                        ]
                ]
        },
        "server" : "g:27017"
}
```

Efficiency of indexes

- $gt, $lt, $ne, $nin, $not($exists) might be inefficient even if an index is used because still many index items (indexed documents) need to be scanned
  - may be a good idea to use a hint to use a diff. index
- $regex can only use an index if it is stemmed on the left side, e.g. /^abc/

Profiling slow queries

- MongoDB logs slow queries (>100ms) by default into the logfile
- Use pofiler:
- mongod --profile 1 --slowms 10 // logs all queries taking longer than 10ms to system.profile collection
- Levels: 0=off (default) 1=log slow queries 2=log all queries (general debugging feature for dev.)
- Mongo shell: db.getProfilingLevel() db.getProfilingStatus() db.setProfilingLevel(level, slowms)
- mongod --notablescan option: Set notablescan = true on your dev or test machine to find operations that require a table scan

mongotop & mongostat

- mongotop 3 // runs every 3 seconds showing you in which collection how much time (read, write, total) is spent

- mongostat // shows inserts, queries, updates, deletes, … per second

➔ idx miss % = index which could not be accessed in memory

Leçon 5 : FIN

Question ?

Leçon 5

TP	Performances

Vue d'ensemble de la formation

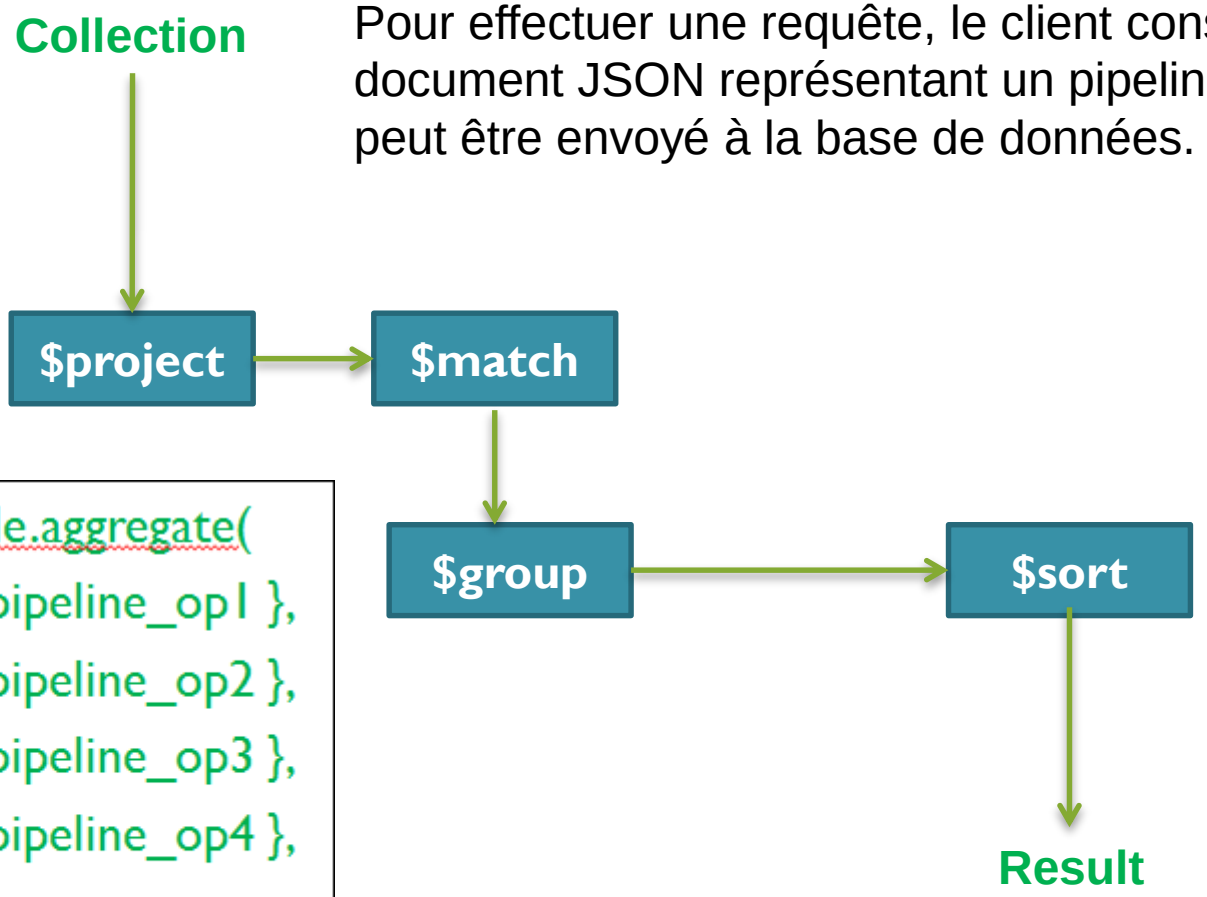
- Leçon 1 : Introduction au NoSQL
- Leçon 2 : Opération CRUD
- Leçon 3 : Les requêtes
- Leçon 4 : Design et Data Modéle
- Leçon 5 : Performance
- **Leçon 6 : Agrégation Framework**
- Leçon 7 : Administration
- Leçon 8 : Driver Java

Framework d'agrégation

Le framework d'agrégation étend les capacités de requetes et de traitement des données.

Avec les nouveaux opérateurs, les développeurs pourront trier et agréger par groupes les données sur lesquelles portent les requêtes et leur appliquer diverses opérations.

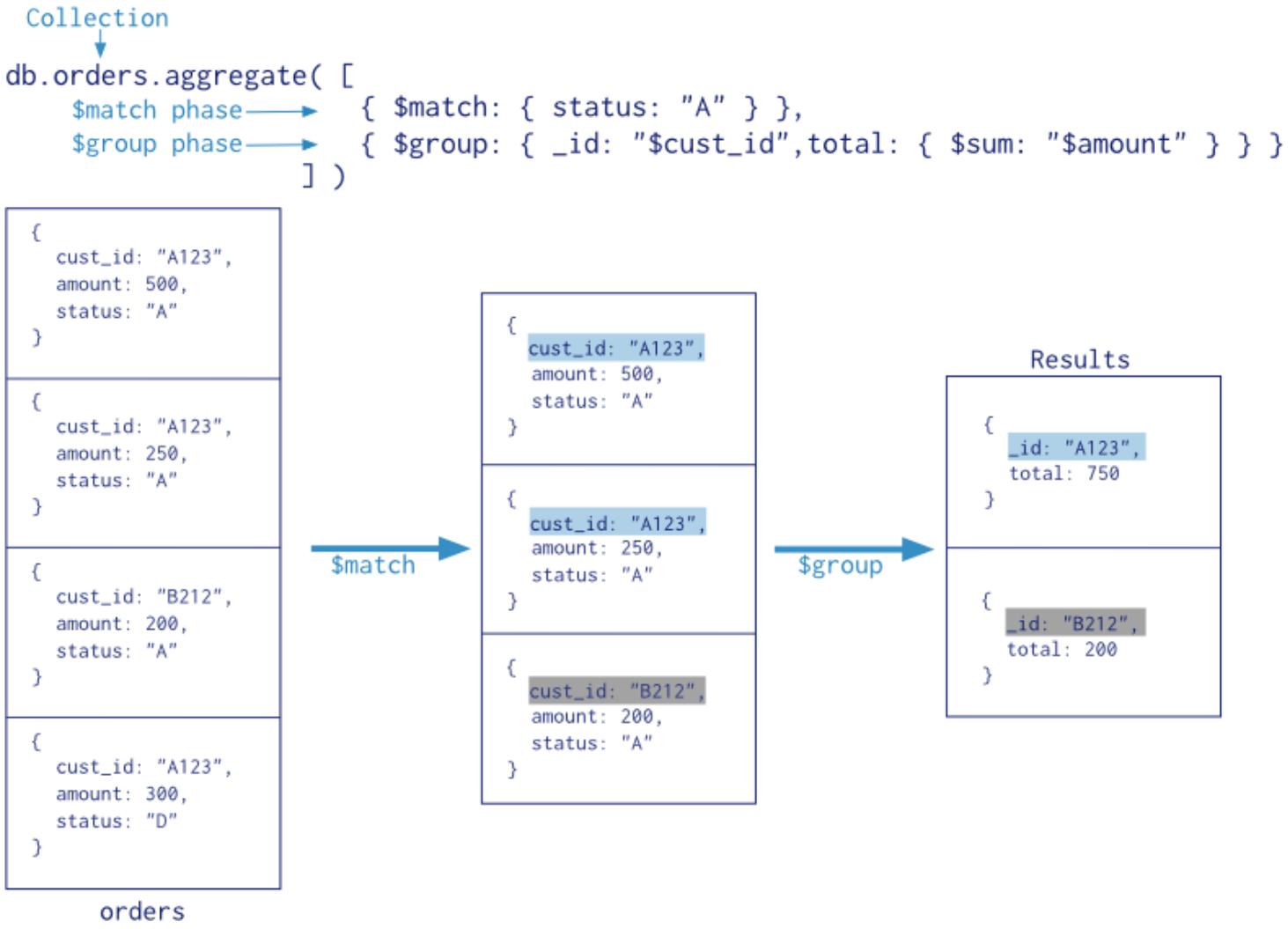
Pipeline



Pour effectuer une requête, le client construit un document JSON représentant un pipeline qui peut être envoyé à la base de données.

```
db.article.aggregate(
        { $pipeline_op1 },
        { $pipeline_op2 },
        { $pipeline_op3 },
        { $pipeline_op4 },
        ...
);
```

Exemple I


Collection
db.orders.aggregate([
$match phase
{ $match: { status: "A" } },
$group phase
{ $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
])
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

Exemple II

```
db.zips.aggregate([
    {$match: {state: {$in: ["CA", "NY"]}}},
    {$group: {_id: {city: "$city", state: "$state"}, pop: {$sum: "$pop"}}},
    {$match: {pop: {$gt: 25000}}},
    {$group: {_id: null, avg: {$avg: "$pop"}}}
])
```

Pipeline Opérations

- **$match** : requête, équivalent du find
- **$project** : met en forme les résultats, notamment en enlevant/ajoutant des champs
- **$unwind** : permet de faire du streaming de tableau, chaque élément du tableau sera traité comme un document
- **$group** : agrège les données
- **$sort** : fonctionnalités de trie
- **$limit** : limite le nombre de documents renvoyés
- **$skip** : exclue certains documents du résultat

Cardinalité

Opérateur	Cardinalité
$match	N : 1
$project	1 : 1
$unwind	1 : N
$group	N : 1
$sort	1 : 1
$limit	N : 1
$skip	N : 1

$match

- Utilisation équivalente au find({…})

```
{
        "_id" : "35004",
        "city" : "ACMAR",
        "loc" : [
                -86.51557,
                33.584132
        ],
        "pop" : 6055,
        "state" : "AL"
}
```

db.zips.aggregate([{$match: { state:"NY" } }])

db.zips.aggregate([{$match: { pop: { $gt:1000} } }])

$sort, $limit, $skip

- **$sort** : Tri les documents
  - Memory intensive; Can't use index (at least after grouping)

```
db.zips.aggregate([ {$match: {  state:"NY"  } }, {$sort:  { population:-1 } } ])
```

- **$limit** : limite le nombre de documents
- **$skip** : exclue certains documents du résultat
  - Makes only sense when you do a sort first
  - First skip – then limit (order of the stages in the pipeline matter)

```
db.zips.aggregate([
   {$match:  { state:"NY"  } },
   {$sort:  { population:-1  } },
   {$skip: 10},
   {$limit: 5}
])
```

$project

- **$project** : permet de remanier un document en ajoutant, supprimer, renommer … des champs.

```
db.products.aggregate([
    {$project:
     {
         title:1, /* champs à inclure, s'il existe */
         author:1, /* champs à inclure, s'il existe */
         'details.tag':1
     }
    }
])
```

```
db.products.aggregate([
    {$project:
     {
         title:0, /* champs à exclure */
         author:0 /* champs à exclure */
     }
    }
])
```

$project

- Remove keys - If you don't mention a key, it is not included, except for _id, which must be explicitly suppressed {$project: {_id: 0, …
- Add keys (also possible to create new subdocuments)
- Keep keys: {$project: {myKey: 1, …
- Rename keys / Use functions: $toUpper, $toLower, $add, $multiply

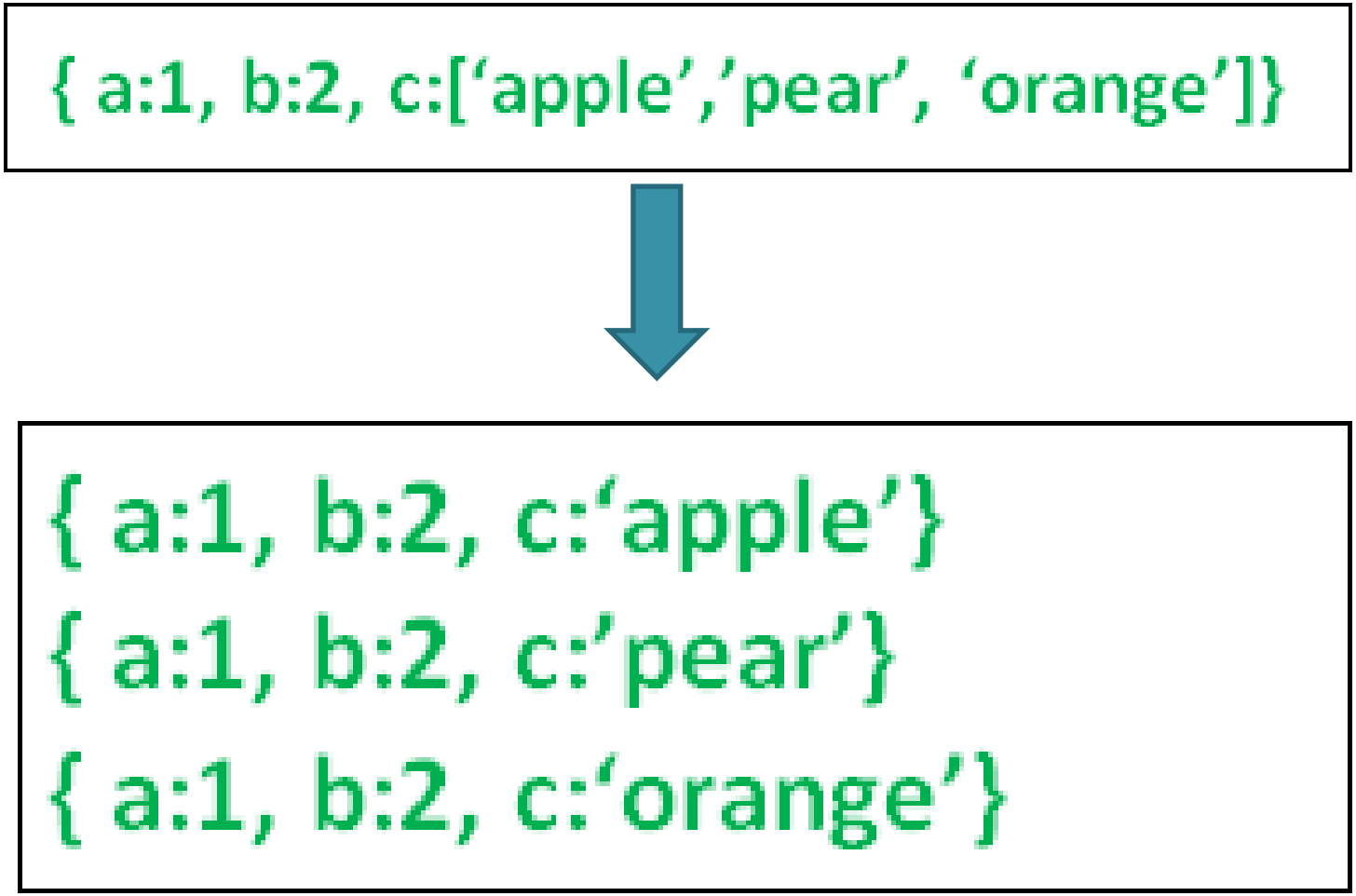
$project

Autre Exemple

```
{
    "_id" : 1,
    "name" : "iPad 16GB Wifi",
    "manufacturer" : "Apple",
    "category" : "Tablets",
    "price" : 499
}
```

```
db.products.aggregate([
    {$project:
     {
         _id:0,
         'maker': {$toLower:"$manufacturer"},
         'details': {'category': "$category",
                     'price' : {"$multiply":["$price",10]}
                    },
         'item':'$name'
     }
    }
])
```

$unwind


{ a:1, b:2, c:['apple','pear', 'orange']}
{ a:1, b:2, c:'apple'}
{ a:1, b:2, c:'pear'}
{ a:1, b:2, c:'orange'}

$group

```
> db.products.aggregate([
...    {$group:
...    {
...      _id: { "manufacturer":"$manufacturer",   "category" : "$category"},
...        num_products:{$sum:1}
...    }
...    }
... ])
```

Opération sur les groupes

- $sum: Add one to a key
  - mySum: {$sum: 1}) pour compter le nombre d'éléments
  - sum_prices:{$sum:"$price"}) : somme de variables
- $avg, $min, $max: Average, Minimum or maximum value of a key
- Create arrays: $push, $addToSet
  - categories: {$addToSet:"$category"}
- Only useful after a sort: $first, $last
  - {$group:{_id:"$_id.state", population:{$first:"$population"}}}

Limitations of the aggregation framework

- A result document can only be 16GB
- One can only use up to 10% of the memory on a machine
- In sharded environment: After first $group / $sort the next phase have to be performed on the mongos router
- Alternative to aggregation framework: map-reduce

MongoDB

SQL Statement	Mongo Statement
`SELECT COUNT(*) FROM users`	`db.users.aggregate([ { $group: {_id:null, count:{$sum:1}} } ])`
`SELECT SUM(price) FROM orders`	`db.users.aggregate([ { $group: {_id:null, total:{$sum:"$price"} } } ])`
`SELECT cust_id,SUM(price) FROM orders GROUP BY cust_id`	`db.users.aggregate([ { $group: {_id:"$cust_id",total:{$sum:"$price"}} } ])`
`SELECT cust_id,SUM(price) FROM orders WHERE active=true GROUP BY cust_id`	`db.users.aggregate([ { $match:{active:true} }, { $group:{_id:"$cust_id",total:{$sum:"$price"}} } ])`

SQL Comparison:
http://docs.mongodb.org/manual/reference/sql-aggregation-comparison/

Leçon 6 : FIN

Question ?

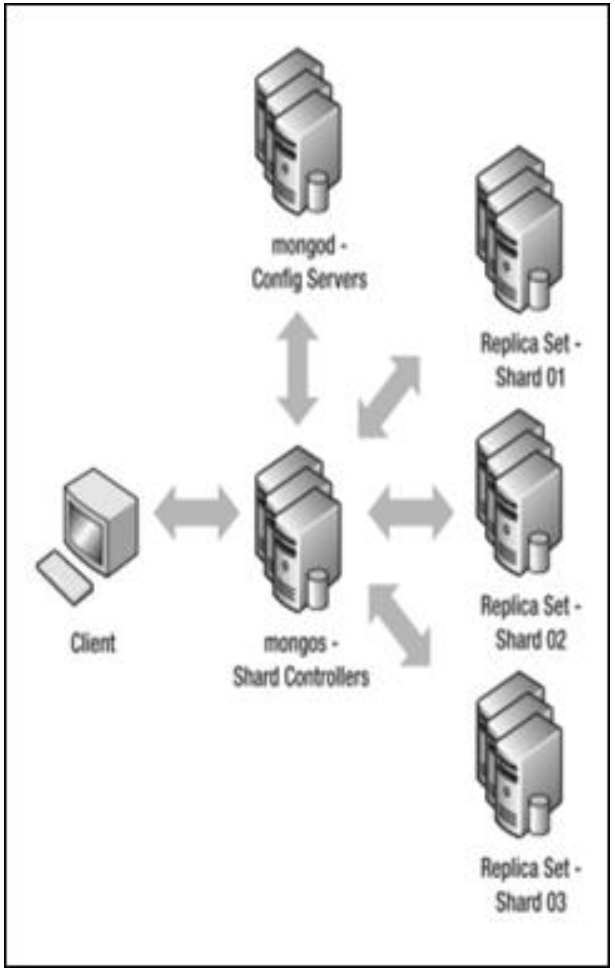
Leçon 6

TP	Agrégation

Vue d'ensemble de la formation

- Leçon 1 : Introduction au NoSQL
- Leçon 2 : Opération CRUD
- Leçon 3 : Les requêtes
- Leçon 4 : Design et Data Modéle
- Leçon 5 : Performance
- Leçon 6 : Agrégation Framework
- **Leçon 7 : Administration**
- Leçon 8 : Driver Java

Replication & Sharding



MongoDB réplication

- Qu'est ce qu'est une réplication?
- But de la replication/redondance
- Tolerance de panne
  - Disponibilité
    - augmenter la capacité de lecture

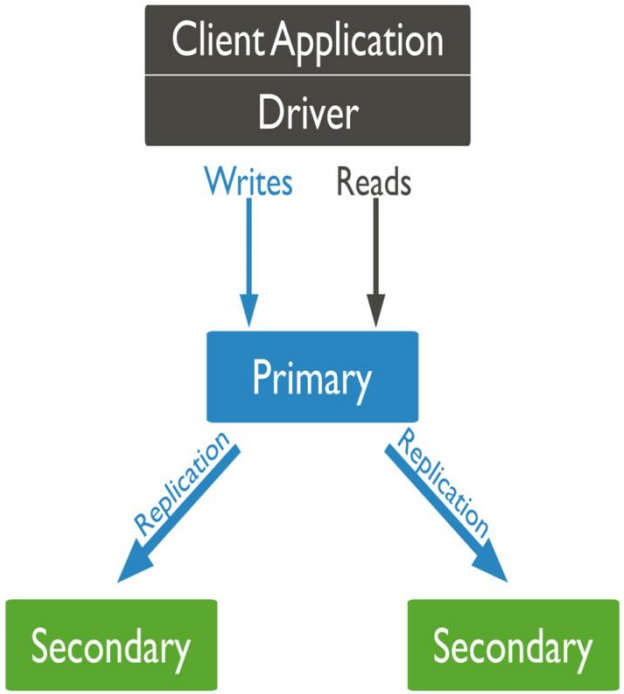
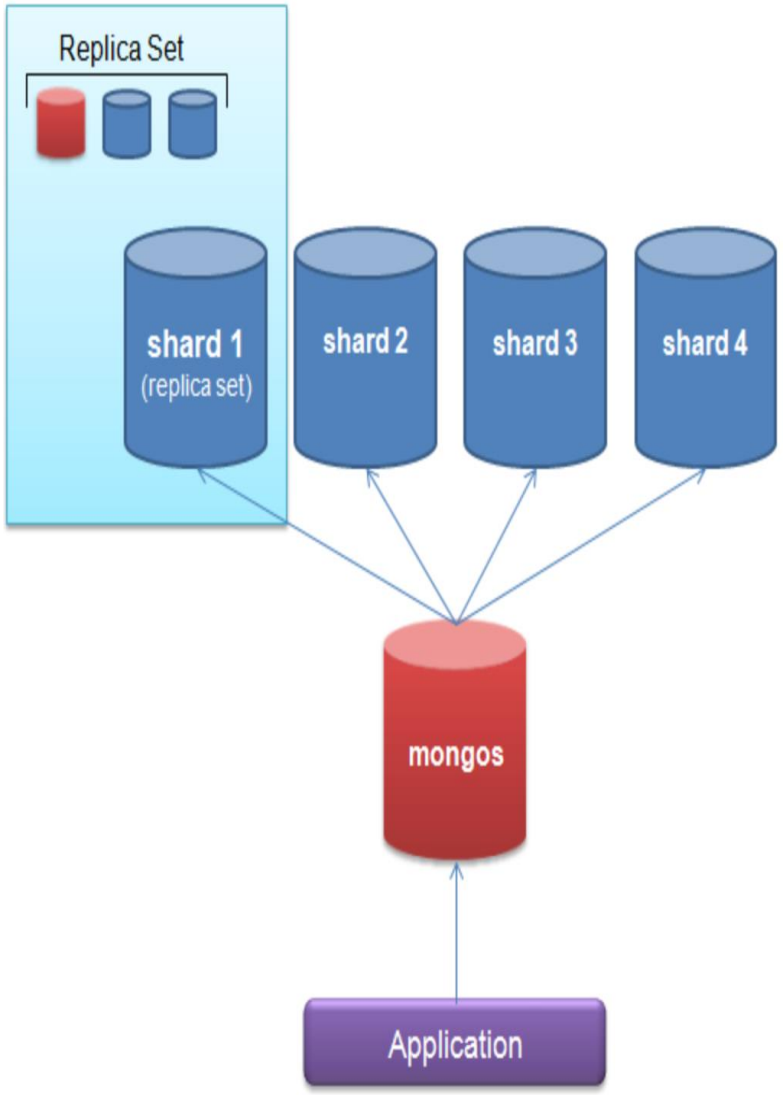


Figure 1: Diagram of default routing of reads and writes to the primary.

Sharding

- Qu'est ce que c'est ?
- But du sharding
  - Horizontal scaling out
- Query Routers
  - mongos
- Shard keys
  - Range based sharding
  - Cardinality
  - Avoid hotspotting



Réplication

- Qu'apporte la réplication ?
  - Redondance
  - Simplification de tâches (backups, ...)
  - Augmentation de la capacité de lecture
- Un replica set est un cluster d'instances MongoDB.
- Stratégie maître / esclaves
- Il doit TOUJOURS y avoir un unique maître.
- Les clients effectuent les écritures sur l'instance ... ?

Réplication

- La réplication du maître vers les esclaves est asynchrone.
- Quels sont les avantages et inconvénients ?
  - Synchrone : Bloquant / Coûteux / Forte cohérence
  - Asynchrone : Non bloquant / Rafraîchissement des données obligatoires.

Replica Set Members

- Primary
  - Read, Write operations
- Secondary
  - Asynchronous Replication
  - Can be primary
- Arbiter
  - Voting
  - Can't be primary
- Delayed Secondary
  - Can't be primary

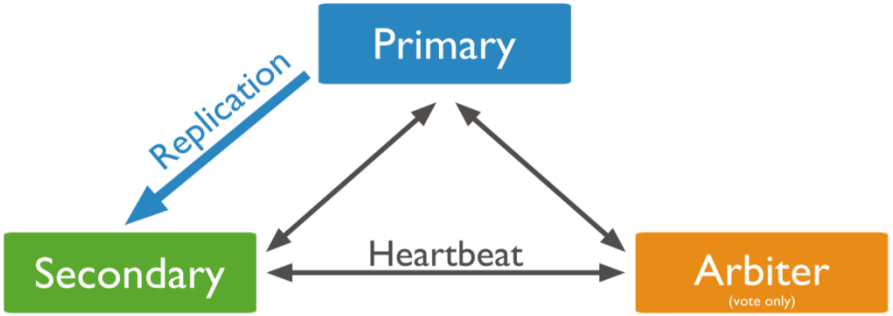


Figure 3: Diagram of a replica set that consists of a primary, a secondary, and an arbiter.

Réplication

- Tolérance aux pannes
- Un replica set est tolérant aux pannes.
- Si le nœud primaire tombe, les nœuds secondaires peuvent élire un nouveau nœud primaire.
- Comment rendre l'élection automatique ?

Réplication

- Comment rendre l'élection automatique ?
  - Détection de la mort du noeud primaire (ping / heartbeat)
  - Lancement d'une élection
  - Le nœud ayant reçu une majorité de vote devient le nœud primaire, grâce à une priorité .

Replication dans MongoDB

- Automatic Failover(prise en charge automatique par une autre machine)
  - Heartbeats
  - Elections
- The Standard Replica Set Deployment
- Déployer un nombre impair de membres Rollback
- Replication is done via a capped collection database
- Secondaries ask the primary for any items since a certain timestamp

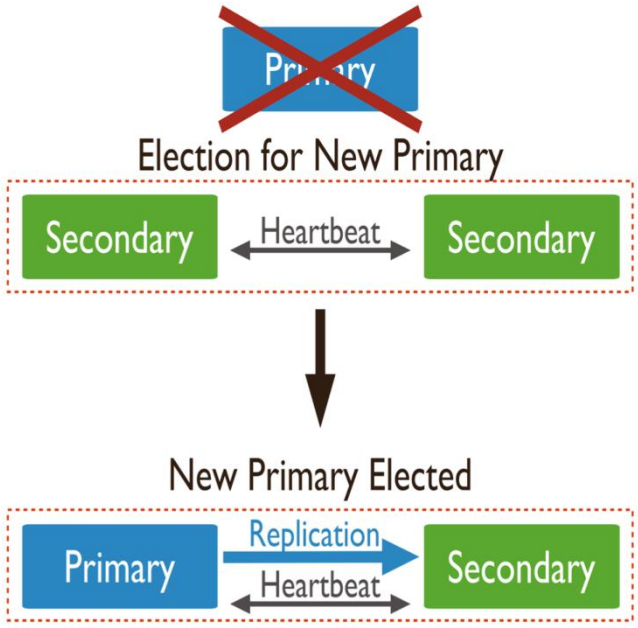


Figure 21: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

Réplication - Consistance

- Que se passe t'il si un nœud primaire accepte une écriture et tombe en panne avant la réplication de l'écriture ?
- On perd la donnée, et le replica set devient inconsistent.
- Peut arrivé lors d'une partition du réseau, avec un lag, par exemple.
- Comment corriger cela ?

Réplication - Consistance

- Idée inspirée des SGBDR : Le rollback
- Le noeud primaire écrit en local les opérations demandées lorsqu'il accepte une écriture.
- Lors de son retour, soit il relance les opérations, soit il les annule (les rollback).

Réplication - Préoccupations

- Lors de la mise en place d'un replica set , deux paramètres sont à prendre en compte
  - - Write Concern : Message envoyé pour vérifier la validité d'une opération.
  - - Read Preferences : Favoriser les lectures sur les nœuds secondaires.

Réplication - Write Concern

- Qualité de chaque opération d'écriture et décrit le montant de préoccupation d'une application pour l'écriture.
- Plus la préoccupation augmente, plus les performances augmentent, plus la cohérence diminue.

Réplication - Type de Write Concern

- **Erreurs ignorés** : Opérations non acquittées. Pas de notification d'erreurs (réseau, ...)
- **Sans acquittement** : Opérations non acquittées. Au courant des erreurs réseaux.
- **Acquittement** : Opérations acquittées. Ne résiste pas au failover.
- **Journalisé** : Opérations valides si acquittées et écrites dans le journal.
- **Acquittement du réplica** : Tous les noeuds secondaires acquittent les opérations.

Réplication - Read Preferences

- Par défaut, les opérations de lecture sont envoyées au nœud primaire.
- Les lectures sur le nœud primaire garantissent d'obtenir toujours les données les plus fraîches.
- Les lectures sur les nœuds secondaires améliorent le débit de lecture en distribuant les lectures.

Réplication - Read Preferences

- Penser à modifier cela lorsque :
  - Opérations n'affectant pas le front-end (backup, reporting, ...).
  - Application distribuée géographiquement. On envoie le client sur le noeud secondaire le plus proche.

Réplication - Types de Read Preferences

- Les différents type de read preferences sont :
  - **primary** : Toujours utiliser le nœud primaire. Exception si pas de noeud primaire.
  - **primaryPreferred** : Toujours utiliser le nœud primaire. On utilise les nœuds secondaires si pas de nœud primaire.
  - **secondary** : Toujours les nœuds secondaires. Exception si pas de nœuds secondaires.
  - **secondaryPreferred** : Toujours les nœuds secondaires. On utilise le nœud primaire si pas de nœuds secondaires.
  - **nearest** : On prend le nœud le plus proche, selon le choix fait par l'utilisateur

Réplication

Replica Set Members

- Primary
  - Read, Write operations
- Secondary
  - Asynchronous Replication
  - Can be primary
- Arbiter
  - Voting
  - Can't be primary
- Delayed Secondary
  - Can't be primary

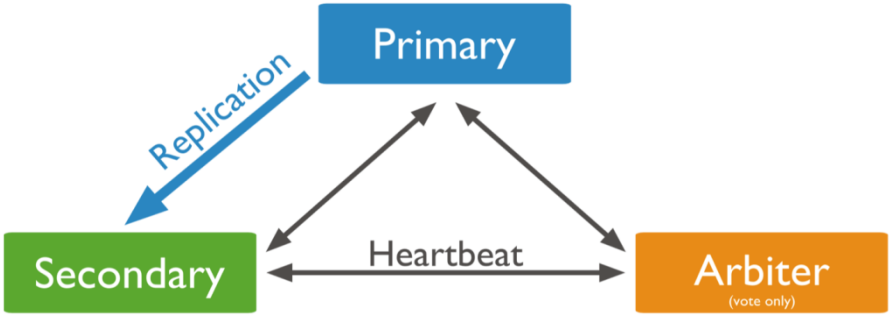


Figure 3: Diagram of a replica set that consists of a primary, a secondary, and an arbiter.

Replication dans MongoDB

- Automatic Failover(prise en charge automatique par une autre machine)
  - Heartbeats
  - Elections
- The Standard Replica Set Deployment
- Déployer un nombre impair de membres Rollback
- Replication is done via a capped collection database
- Secondaries ask the primary for any items since a certain timestamp

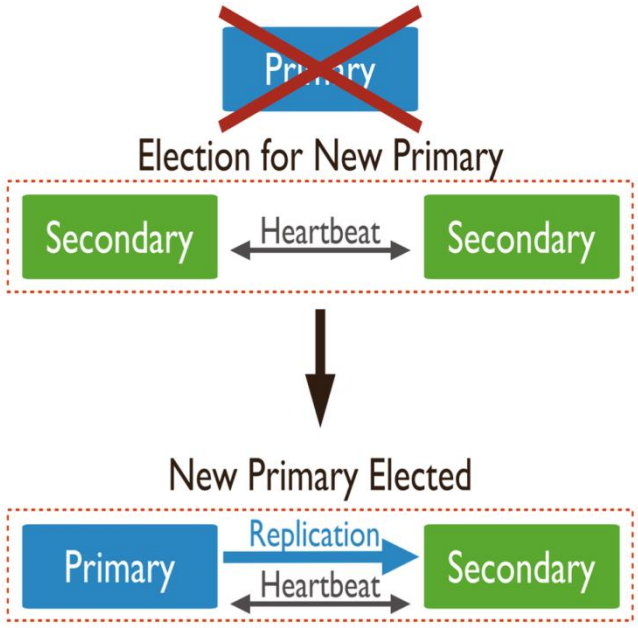


Figure 21: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

Replica Set Members

- You can only write to the primary node which replicates asynchronous to secondary nodes
- If you only read from the primary you will have strong consistency (default behaviour)
- You can allow your reads to go to secondaries
  - you might read stale data and have eventual consistency
- If the primary goes down the secondaries elect a new primary and the Java driver automatically connects to the new primary

Replication in MongoDB

- Arbiter nodes exist for voting purposes, e.g. if you have an even number of regular (= primary & secondary) nodes it can ensure a majority in an election
  - Allows you to have only two regular nodes.
- Delayed nodes are disaster recovery nodes: Can be set to be whatever time behind the regular nodes. Can't participate in elections
- Hidden node (e.g. for reporting) can't become the primary but can participate in elections

Replication Configuration

- Start a replication set: mongod -replSet m101 --logpath "1.log" --dbpath /data/rs1 -fork
- Register replica set nodes in the mongo shell:
- config = { _id: "m101", members:[

  { _id : 0, host : "localhost:27017", priority:0, slaveDelay:5},

  { _id : 1, host : "localhost:27018"},

  { _id : 2, host : "localhost:27019"}]

  }

Replication Configuration

- **rs.initiate(config)** ← Initializes the replica set // can't be executed on a node which can't become primary
- **rs.status()** ← Gives you the status information about the replica set
- **rs.isMaster()** ← Tells you if you are the primary node
- **rs.slaveOk()** ← If issued on a secondary node it allows you to read from this secondary node
- **rs.stepDown()** ← Forces primary node to step down as a primary node

Failover and Rollback

Scenario:

- When the primary dies a secondary which becomes elected as a new primary which does not have the latest entries from the old primaries oplog
- When the former primary node comes back up as a secondary node it will request the oplog data from the new primary and roll back the writes the current primary does not have and write them to a rollback file which can be applied manually
- If the oplog of the new primary has looped during the time the old primary was down the entire dataset will be copied from the new primary
- The risk of losing data due to a rollback can be avoided by waiting till the majority of the nodes have the data ➔ set the write concern w=majority

Connecting from the Java Driver

Provide a seed list to the MongoClient instance

```
new MongoClient(Arrays.asList((
    new ServerAddress("localhost", 27017),
    new ServerAddress("localhost", 27018),
    new ServerAddress("localhost", 27019),
)));
```

➔ Will work even if the primary is not part of the seed list. The Java Client starts a background thread which pings all nodes from the seed list and all discovered nodes to find out which one is the primary

Write Concerns

Client writes to a primary:

1. Primary writes into RAM (collection and oplog)
2. The writes are asynchronously journaled (Gives recoverability in case of a crash)
3. The writes are written into the data directory
4. Secondaries are replicating writes from the primary's oplog
   - The insert method sent from the Client does not expect a response
   - The client sends a second command "getLastError"

Write Concerns

- w=0 ➔ Unaknowledged; Fast writes – no "getlastError" command
- w=1 ➔ Will wait for the primary to write into RAM
- w=[n] ➔ Will wait for the primary and n-1 to write into RAM
- w=majority ➔ Wait for majority of replica set to write to RAM
- j=true ➔ Will wait for the primary to write into the journal
- fsync=true ➔ Will wait for the primary to write into the data directory
- wtimeout=[milliseconds] ➔ Indicate how long you are willing to wait

Write Concerns

- Java Driver
  - Default: w=1 and wtimeout=0 (=infinit)
  - client|db|collection.setWriteConcern(WriteConcern.JOURNALED)
  - collection.insert(doc,WriteConcern.JOURNALED)
  - WriteConcern.JOURNALED = new WriteConcern([w=]1,[wtimeout=]0, [fsync=]false,[j=]true)

Read Preferences

- Primary ➔All reads are send to the primary (default to guarantee strict consistency)
- Secondary ➔Send reads to randomly selected secondaries, but not to the primary
- Secondary Preferred ➔Send reads to secondaries or to the primary if all secondaries are down
- Primary Preferred ➔Sends reads to primary or to a secondary if primary is down
- Nearest ➔Send reads to secondary or primary

Read Preferences

- If you read from secondaries you might get stale reads. The might be OK if different applications do the writes and the reads
- For any read preference (except Primary) the driver will look at ping times and will only send reads to nodes which are within the latency window (15ms) of the fastest
- client|db|collection.setReadPreference(ReadPreference.primaryPreferred())
- collection.find().setReadPreference(ReadPreference.nearest());

Sharding

- Enables horizontal scalability
- Shards are typically itself replica sets
- mongos is the sharding router which distributes data to the individual shards
- The application (and also the mongo shell) connects to mongos instead of mongod
- There can be multiple mongos - mongos typically runs on the same server as the application

Sharding

- If a mongos goes down the application will connect to a different one – similar to replica sets
- Shard key determines to which shard a document goes
- Sharding is at database level – but you can define if you want to shared or not shard a specific collection
- Config servers (which are mongod) keep track of where the shards are – in production you typically use 3 of them

Building a sharded environment

Set up two shards each a replica set of three mongod nodes

- Set up shard as a replication set:
- mongod --replSet s0 --logpath "s0-r0.log" --dbpath /data/shard0/rs0 --port 37017 --fork –shardsvr
- Set up config server:
- mongod --logpath "cfg-a.log" --dbpath /data/config/config-a --port 57040 --fork –configsvr
- Set up mongos router with information about the config servers:
- mongos --logpath "mongos-1.log" --configdb localhost:57040,localhost:57041,localhost:57042 –fork
- mongos now listens on the default mongod port

Building a sharded environment

- On the mongo shell – tell the config servers (via the mongos) about the shards:
- db.adminCommand({ addshard : "s0/"+"localhost:37017" });
- … add further shards
- db.adminCommand({enableSharding: "test"})
  - enable sharding on test DB
- db.adminCommand({shardCollection: "test.grades", key: {student_id:1}});
  - shard collection "grades" with the shard key "student_id"
- sh.help()
  - Will display all the shard commands available in the mongo shell, e.g. sh.status()

Sharding implications

- Needs an index on first element of the shard key (can be compound but not multi-key)
- Each document needs the shard key
- The shard key is immutable
- On an update you need to specify the shard key or specify multi: true
- A find with no shard key will go to all shards
- The key used in most queries should be the shard key
- You can't have a unique index unless it is part of/starts with the shard key
- Write concerns are still important in a sharded setup

Sharding key

- Sufficient cardinality (variety of values)
- Avoid monotonically increasing keys to avoid hotspotting in writing (e.g. order_id, order_date)
- Compound sharding key is possible

Vue d'ensemble de la formation

- Leçon 1 : Introduction au NoSQL
- Leçon 2 : Opération CRUD
- Leçon 3 : Les requêtes
- Leçon 4 : Design et Data Modéle
- Leçon 5 : Performance
- Leçon 6 : Agrégation Framework
- Leçon 7 : Administration
- **Leçon 8 : Driver Java**

Java Driver

MongoDB offre un bon support et une intégration avec de nombreux langages.

Le **driver Java** propose une API pour:

- réaliser la transformation des objets en documents,
- la connexion à la base,
- les CRUD,
- les requêtes,
- etc.

Transformation Objet-BSON

Afin de pouvoir insérer dans la base,
la **transformation Objet-BSON** doit être faite.

Nous avons plusieurs possibilités.

- Réaliser la transformation à la main

- Utiliser une libraire
  - Morphia
  - Mungbean
  - Daybreak

Connexion à la base MongoDB

```
MongoClient mongoClient1 = new MongoClient();
// ou
MongoClient mongoClient2 = new MongoClient( "localhost" );
// ou
MongoClient mongoClient3 = new MongoClient( "localhost" , 27017 );
// ou, to connect to a replica set, with auto-discovery of the primary,
// supply a seed list of members
MongoClient mongoClient4 = new MongoClient(Arrays.asList
                                    (new ServerAddress("localhost", 27017),
                                     new ServerAddress("localhost", 27018),
                                     new ServerAddress("localhost", 27019)));
```

CRUD-Insertion

```
{
    "name" : "MongoDB",
    "type" : "database",
    "count" : 1,
    "info" : {
                x : 203,
                y : 102
              }
}
```

```
BasicDBObject doc = new BasicDBObject("name", "MongoDB").
        append("type", "database").
        append("count", 1).
        append("info", new BasicDBObject("x", 203).append("y", 102));

coll.insert(doc);
```

CRUD-Read

- Retrouver des documents

```
// Retrouver un ensemble de documents
DBCursor cursor = coll.find();
int i = 1;
while (cursor.hasNext()) {
    System.out.println("Inserted Document: " + i);
    System.out.println(cursor.next());
    i++;
}
```

CRUD-Update

```
//éléments à maj
DBObject upDocument = new BasicDBObject();
upDocument.put("count", 2);

// critere de selection des documents à màj
DBObject criteriaDocument = new BasicDBObject();
criteriaDocument.put("name", "MongoDB");

// update un seul document par défault
coll.update(criteriaDocument, upDocument);
```

update : remplace le premier document qui vérifie le critère de sélection *criteriaDocument*, par le document *upDocument*.

CRUD-Update-multi

```
BasicDBObject updatedDocument = new BasicDBObject();
updatedDocument.append("$set",
        new BasicDBObject().append("count", 3));

// critere de selection des documents à màj
DBObject criteriaDocument = new BasicDBObject();
criteriaDocument.put("name", "MongoDB");

// update tous les documents selectionnés
coll.update(criteriaDocument, updatedDocument, false, true);
```

$set : permet de màj des valeurs directement dans le document.

multi: true (4° param) le changement s'applique à l'ensemble des documents sélectionnés.

CRUD-Delete

```
DBObject myDoc = coll.findOne();
coll.remove(myDoc);
```

Leçon 8 : FIN

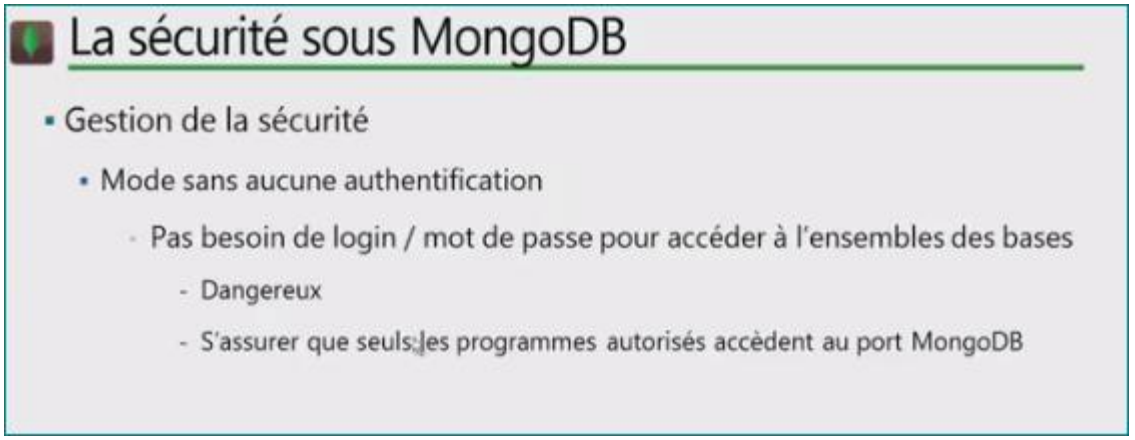
Question ?

Leçon 8

TP	JavaMongo

Sauvegarde et restauration

- Sauvegarde à froid (le plus simple)
  - Les étapes
    - Arrêter le service mongodb
    - Copier tous les fichiers de --dbpath
      - <nom_base>.<numero_fichier>
      - <nom_base>.ns
    - Démarrer le service mongodb


La sécurité sous MongoDB
Gestion de la sécurité
Mode sans aucune authentification
Pas besoin de login / mot de passe pour accéder à l'ensembles des bases
Dangereux
S'assurer que seuls les programmes autorisés accèdent au port MongoDB