

# MongoDB



# MongoDB



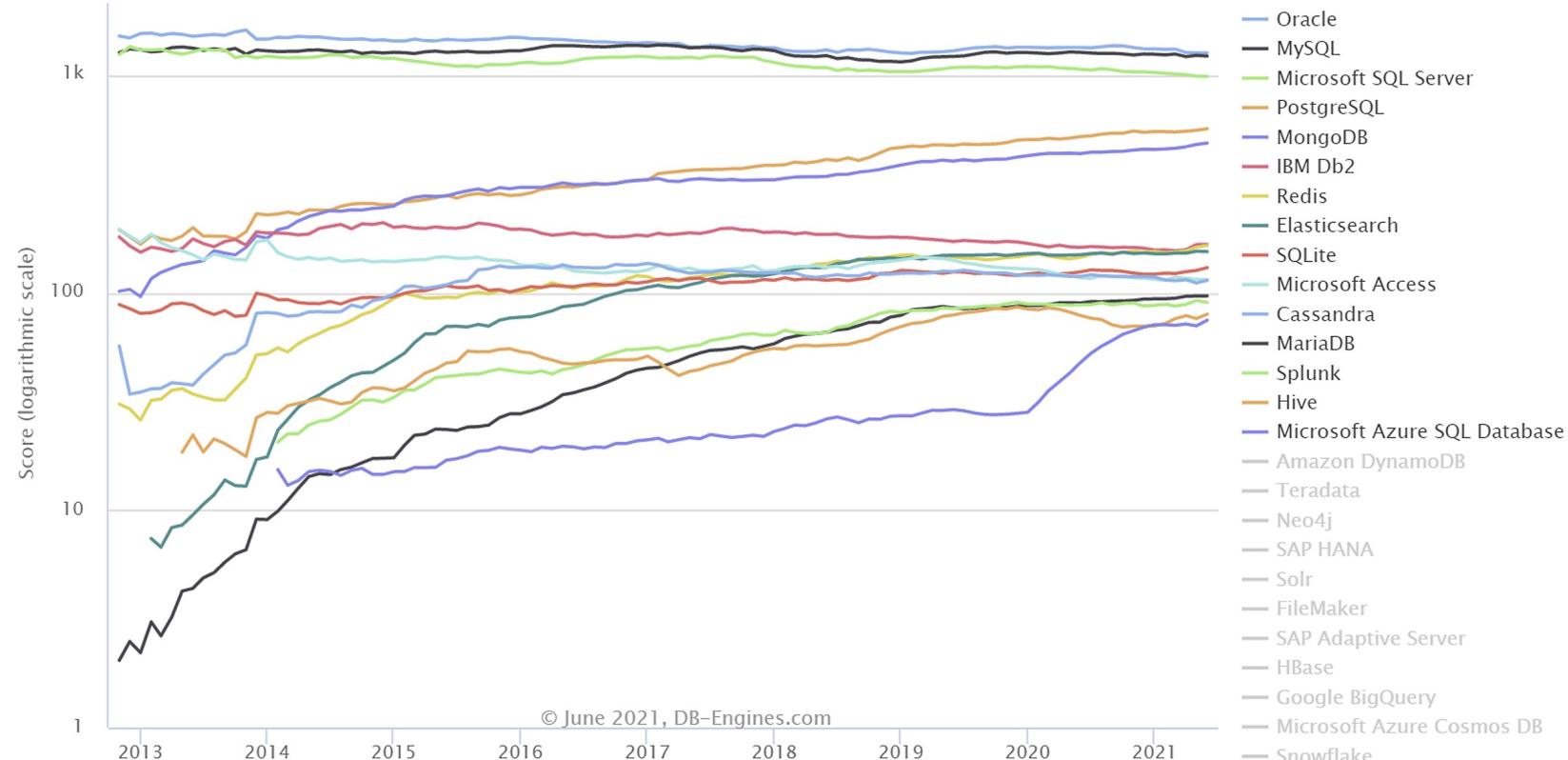
## Forte concurrence dans le secteur des bases NoSQL

Comme toutes les sociétés basant leurs modèles sur « l'Opensource », MongoDB Inc. complète son offre avec des outils spécifiques (interface d'administration Compass par exemple) ou du support ... Payants

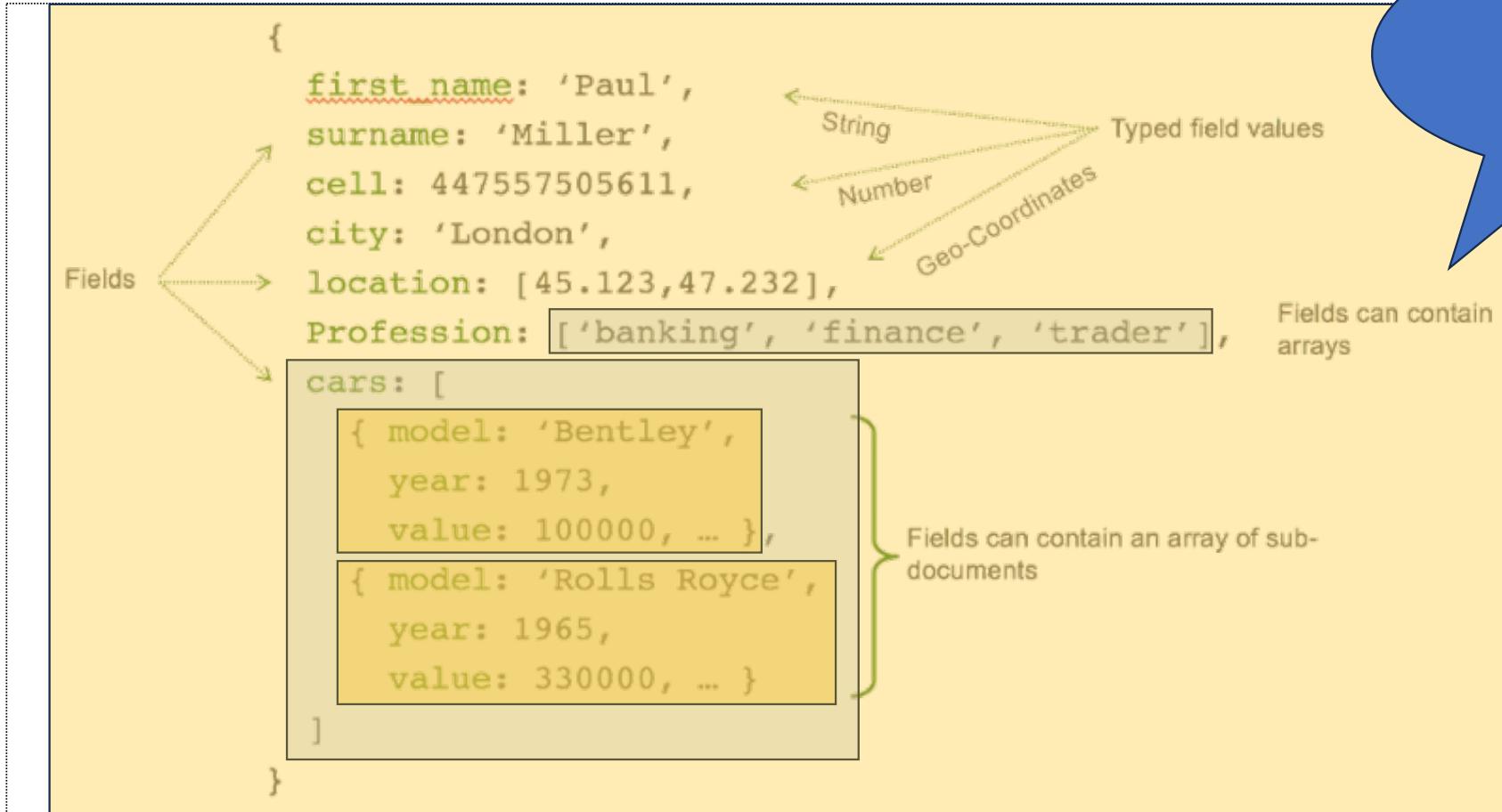
MongoDB Inc a une forte stratégie d'innovation :

- Beaucoup de versions se succèdent
- Le contour fonctionnel du produit évolue fortement (et intègre même des notions de graphes par exemple, ou de la BI, des éléments renforcés en termes de sécurité ainsi que des connexions vers des environnements Spark)
- Offre de services Cloud
- <https://www.mongodb.com/scale/mongodb-hosting-free>

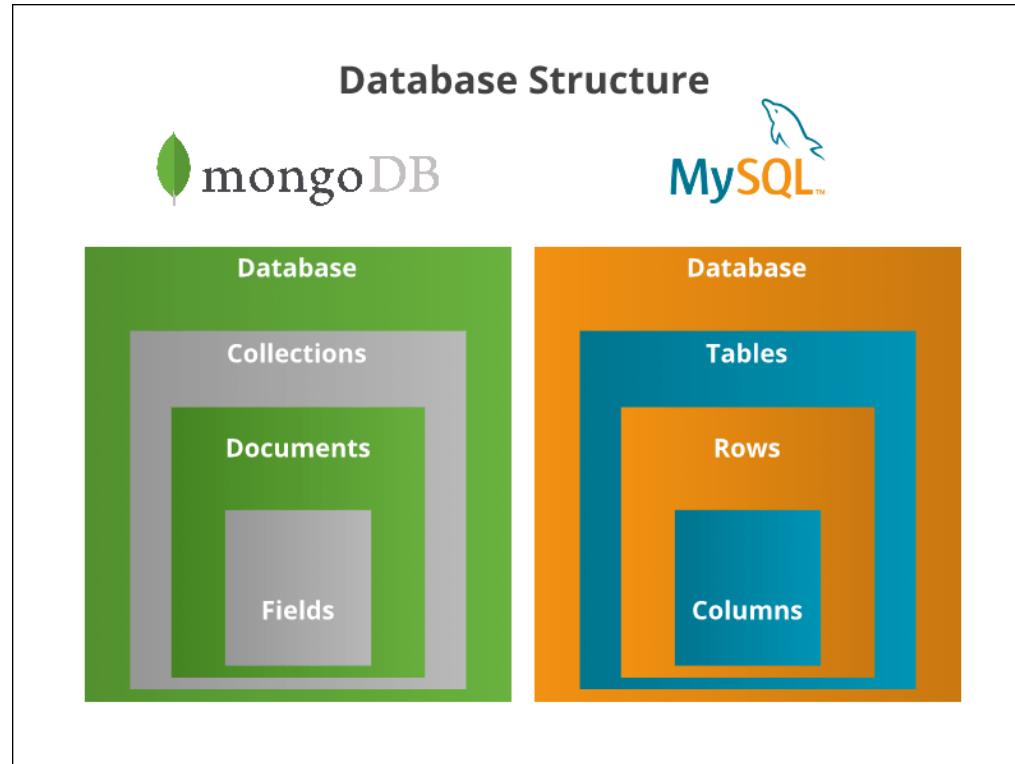
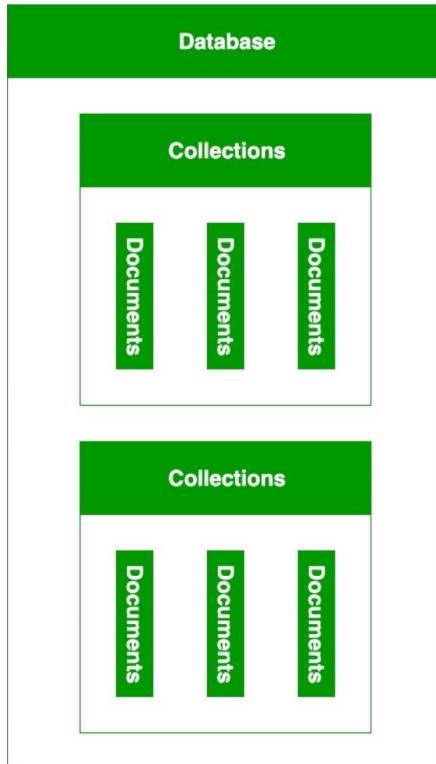
# Classement tendance (trend) des moteurs



## Structure d'un document Json



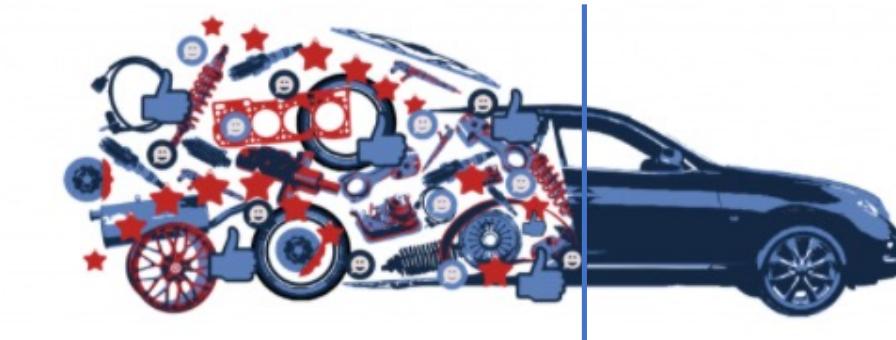
# MongoDB



# Introduction au modèle orienté document



Modèle de données Relationnel



itemid	orderid	item	amount
5	1	Chair	200.00
6	1	Table	200.00
7	1	Lamp	123.12

customerid	name	email
5	Rosalyn Rivera	rosalyn@adatum.com
6	Jayne Sargent	jayne@contoso.com
7	Dean Luong	dean@contoso.com

orderid	customerid	date	amount
1	4	11/1/17	523.12
2	3	11/15/17	32.99
3	1	11/21/17	23.99

Représentation logique sous forme de tables à 2 dimensions

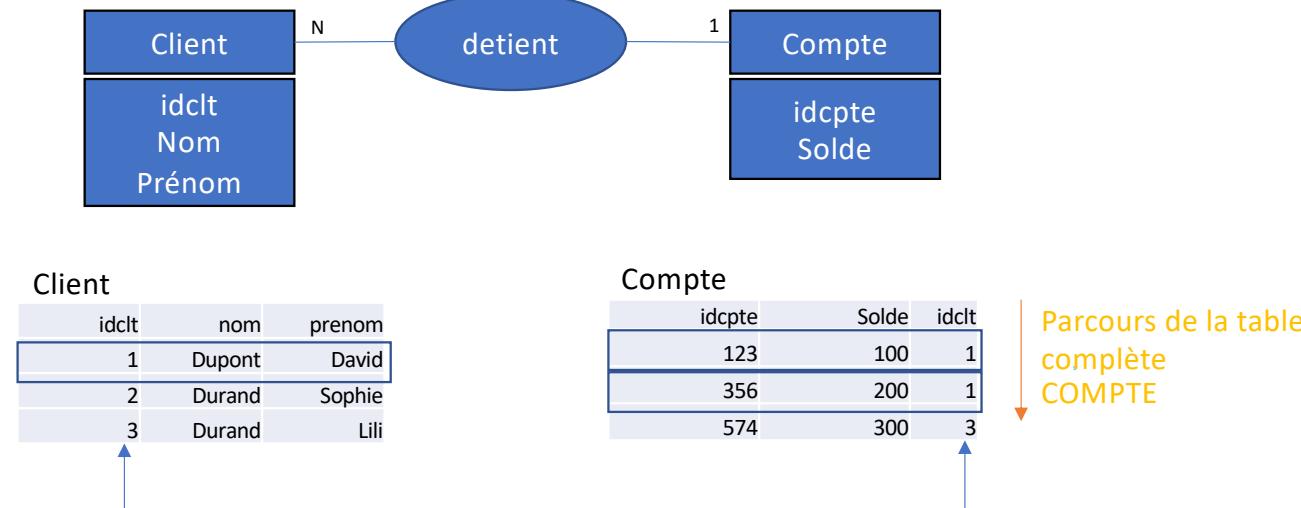
Modèle de donnée orienté Document



Représentation logique sous forme de documents

# Introduction au modèle orienté document

L'approche relationnelle modélise les données de la manière suivante :



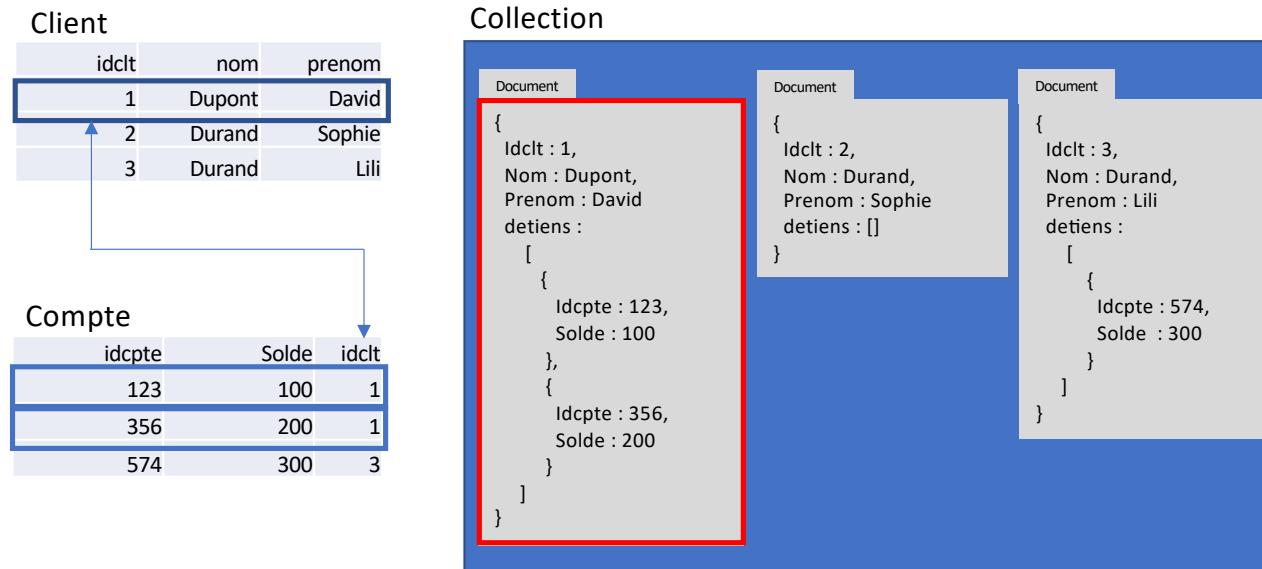
Comment obtenir le solde des comptes du Client David Dupont ?

```
Select idcpte, solde  
from compte cpt left join client clt on clt.idclt = cpt.idclt  
Where nom='Dupont' and prenom='David'
```

Le SGBDR a besoin de parcourir 2 tables pour collecter l'ensemble des données.

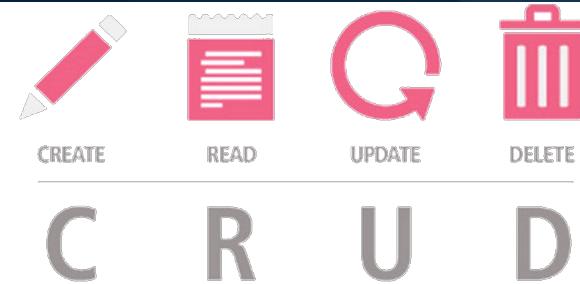
# Introduction au modèle orienté document

Avec une approche document, les données nécessaire à l'application sont regroupées dans un et un seul document.



- Plus besoin de jointures.
- Plus besoin de transactions si toutes les données à modifier sont réunies en un seul document.
- Facilement distribuable.
- Sous MongoDB, la taille d'un document ne peut excéder 16Mo.

# CRUD ?



# CRUD : Create

Comment créer un document dans une base mongo ?

```
db.products.insert( { item: "card", qty: 15 } )
```

ou

```
db.products.insertOne( { item: "card", qty: 15 } )
```



MongoDB collection products

```
{ "_id" : ObjectId("5063114bd386d8fadbd6b004"), "item" : "card", "qty" : 15 }
```



## CRUD : Create



```
db.products.insert(  
[  
    { _id: 11, item: "pencil", qty: 50, type: "no.2" },  
    { item: "pen", qty: 20 },  
    { item: "eraser", qty: 25 }  
]  
)
```



```
{ "_id" : 11, "item" : "pencil", "qty" : 50, "type" : "no.2" }  
{ "_id" : ObjectId("51e0373c6f35bd826f47e9a0"), "item" : "pen", "qty" : 20 }  
{ "_id" : ObjectId("51e0373c6f35bd826f47e9a1"), "item" : "eraser", "qty" : 25 }
```

Si une erreur survient lors de l'insertion d'un document, MongoDB continue d'insérer les documents suivants.



## CRUD : Read



```
db.collection.find({condition},{projection}) return cursor
```

La condition est exprimée sous forme de document : { "champ" : valeur , ... }

Pour trouvez tous les restaurants dans le quartier (borough) de Brooklyn:

```
db.restaurants.find( { "borough" : "Brooklyn" } )
```

Maintenant, nous cherchons parmi ces restaurants ceux qui font de la cuisine italienne.

```
db.restaurants.find( { "borough" : "Brooklyn", "cuisine" : "Italian" } )
```



## CRUD : Read



```
db.collection.find({condition},{projection}) return cursor
```

```
{  
    "_id" : 1,  
    "name" : "sue",  
    "age" : 19,  
    "type" : 1,  
    "status" : "P",  
    "favorites" : { "artist" : "Picasso", "food" : "pizza" },  
    "finished" : [ 17, 3 ]  
    "badges" : [ "blue", "black" ],  
    "points" : [ { "points" : 85, "bonus" : 20 }, { "points" : 85, "bonus" : 10 } ]  
}
```



- Quelle requête retourne le document ci-dessous :
- db.users.find( { badges: "black" },{name: 1} )
- db.users.find( { "badges.0": "black" } )
- db.users.find( { "favorites.artist": "Picasso" } )
- db.users.find( { statu: "A" , age: { \$lt: 30 } } ).skip(0).limit(5)

## CRUD : Read



Extraire les champs voulus des documents filtrés

Un deuxième paramètre (optionnel) de la fonction find permet de choisir la ou les clés à retourner dans le résultat.

```
{ "champ" : 0|1 , ... }
```

Remarques :

- Pour les documents imbriqués, même approche objet que pour le filtre ('.)
- La clé « \_id », identifiant le document, est remontée systématiquement (=> { "\_id" : 0 })
- true et false peuvent être utilisés à la place de 0 et 1.

```
db.restaurants.find( { "borough" : "Brooklyn", "cuisine" : "Italian" } ,{borough:0})
```

Affiche tout les champs de chaque document à l'exception du champs borough

```
db.restaurants.find( { "borough" : "Brooklyn", "cuisine" : "Italian" } ,{borough:1})
```

Affiche uniquement le champs borough de chaque document avec \_id



## CRUD : Read



- Opérateurs de comparaisons : \$eq, \$gt, \$gte, \$in \$nin

```
db.inventory.find( { qty: { $nin: [ 5, 15 ] } } )
```

<https://docs.mongodb.com/manual/reference/operator/query-comparison/>

- Opérateurs logiques : \$and, \$or, \$not, \$nor

```
db.inventory.find( { price: { $not: { $gt: 1.99 } } } )
```

<https://docs.mongodb.com/manual/reference/operator/query-logical/>

- Opérateur sur les éléments : \$exists, \$type

```
db.inventory.find( { price: { $type : "double" } } )
```

<https://docs.mongodb.com/manual/reference/operator/query-element/>

- Opérateur de projection

```
Db. inventory.find( { points: { $elemMatch: {point: { $gt: 70 },bonus: { $gt:90 } } } },{ "grades.$": 1 } )
```

<https://www.mongodb.com/docs/manual/reference/operator/projection/>

# CRUD : Read



```
db.addressBook.insertMany(  
[  
    { "_id" : 1, address : "2030 Martian Way", zipCode : "90698345", "contact" : true },  
    { "_id" : 2, address: "156 Lunar Place", zipCode : 43339374, "contact" : false },  
    { "_id" : 3, address : "2324 Pluto Place", zipCode: NumberLong(3921412), "contact" : false },  
    { "_id" : 4, address : "55 Saturn Ring" , zipCode : NumberInt(88602117) },  
    { "_id" : 5, address : "104 Venus Drive", zipCode : ["834847278", "1893289032"], "contact" : true },  
    { "_id" : 6, address : "222 Sun Boulevard", zipCode : null, "contact" : true }  
]  
)
```

```
db.addressBook.find(  
{ $and : [  
        {"zipCode" : { $type : "number" }},  
        {"zipCode" : { $gt : 3000000}},  
        {"zipCode" : { $lt : 50000000 } }  
    ]  
}  
)  
=> {2,3}
```

```
db.addressBook.find( { "zipCode" : { $type : "string" } })      => {1,5}  
db.addressBook.find( { "zipCode" : { $not : { $type : "null" } } }) => {1,2,3,4,5}  
db.addressBook.find( { "contact" : { $exists : true} })          => {1,2,3,5,6}  
db.addressBook.find( { "contact" : { $exists : true, $eq : true} }) => {1,5,6}
```



# CRUD : Read et les curseurs

CRUD : Read



- Un curseur (« cursor ») est un pointeur sur le résultat d'une requête.
- Une opération de lecture peut potentiellement retourner un ensemble de plusieurs centaines , milliers, ... de documents.
- Pour réduire l'impact mémoire et le trafic réseau MongoDB retourne un pointeur sur l'ensemble des documents satisfaisants la requête.
- Il est alors possible de réaliser des opérations sur cet objet.

Les méthodes « de base » :

- limit(): constraint la taille de l'ensemble résultat
- count(): modifie le curseur pour retourner le nombre de documents plutôt que les documents eux-mêmes.
- skip(): retourne un curseur qui pointe sur les documents résultat après avoir passé le nombre de documents indiqué.
- pretty(): configure le curseur pour afficher les documents de manière plus facile à lire (pour un humain)

Exemples :

- db.trips.find({"usertype" : "Customer"}).count()
- db.zips.find({"state" :"CA"}).skip(500).limit(10)

La liste complète :

<https://docs.mongodb.com/manual/reference/method/js-cursor/>

# CRUD : Update



- Pour modifier les données, MongoDB fournit la commande update
  - `Db.mycollection.updateOne({$query},{$set})` / modifie un seul document même si \$query retourne N documents
  - `Db.mycollection.updateMany({$query},{$set})`
  - `db.collection.replaceOne({$query},{replacement})`
- MongodB n'est pas transactionnel (ACID) :
  - Une commande qui met à jour plusieurs documents validera les modifications document par document.
  - Si une telle commande est interrompue, elle sera partiellement validée sur une partie des documents rendant éventuellement la base de données inconsistante.

```
db.restaurant.updateOne(  
  { "name" : "Central Perk Cafe" },  
  { $set: { "violations" : 3 } }  
)
```

```
db.restaurant.replaceOne(  
  { "name" : "Central Perk Cafe" },  
  { "name" : "Central Pork Cafe", "Borough" : "Manhattan" }  
)
```

# Delete



`db.collection.deleteOne(filter,options)`

Supprime le *premier* document correspondant au filtre **filter**. Si filtre {}, supprime le premier document retourné dans la collection. Utiliser un filtre sur ‘\_id’ pour être « précis ».



```
db.products.deleteOne( {$and : [{ item: "pencil" }, { type : "blue"}]} )
```

```
    { "acknowledged" : true, "deletedCount" : 1 }
```

`db.collection.deleteMany(filter,options)`

Supprime *tous* les documents correspondants au filtre **filter**

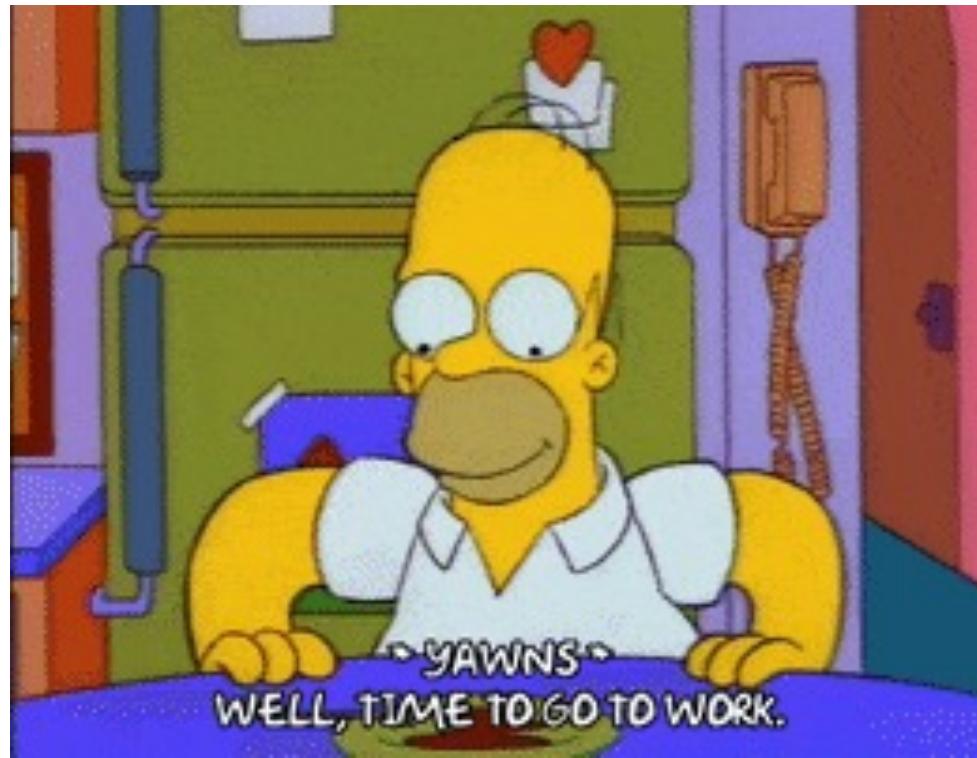


TP



Mongo\_TP01\_introduction.ipynb

Mongo\_TP02 CRUD.ipynb



## Aggregation framework



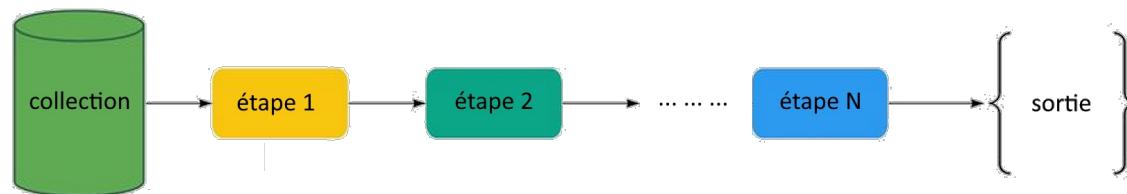
# Aggregation framework



MongoDB intègre une plateforme qui permet d'enchainer un ensemble d'opération l'une à la suite des autres.

Le pipeline d'agrégation est :

- une série ordonnée d'opérations déclaratives appelées étapes
- chaque étape réalise une transformation sur les documents en entrée
- une étape peut exclure, synthétiser ou créer des documents
- la totalité de la sortie d'une étape constitue l'entrée de l'étape suivante
- la sortie finale est un curseur, une (nouvelle) collection ou une vue

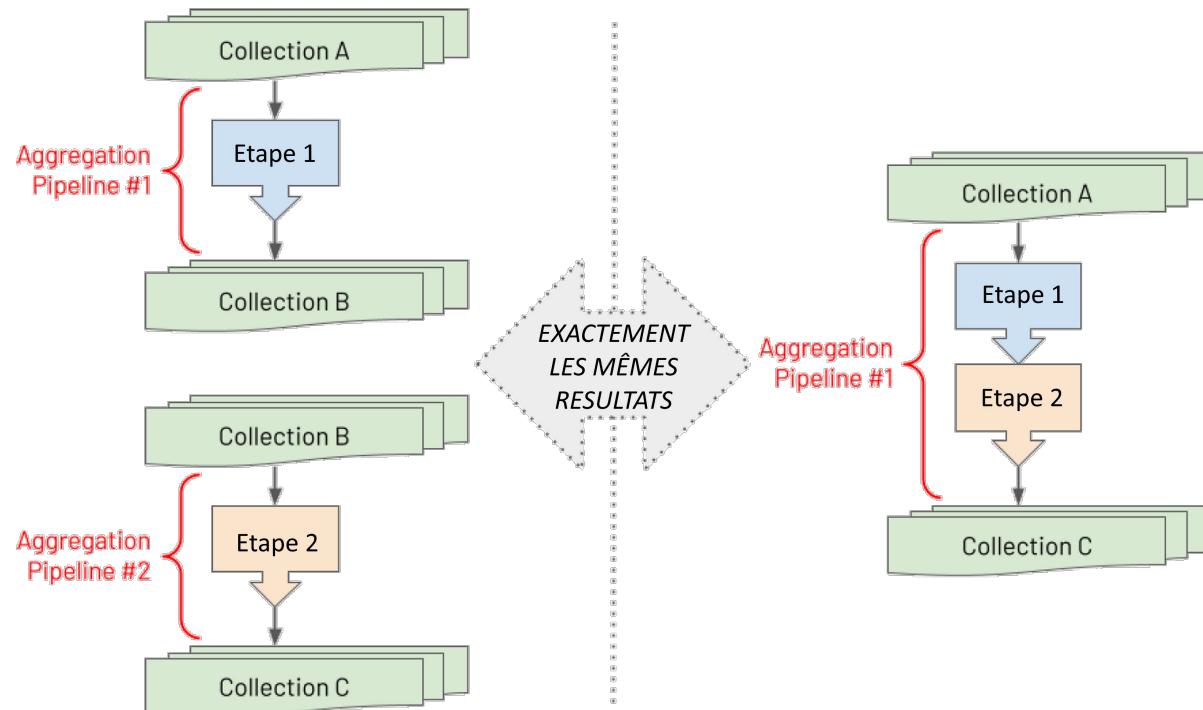


# Aggregation framework

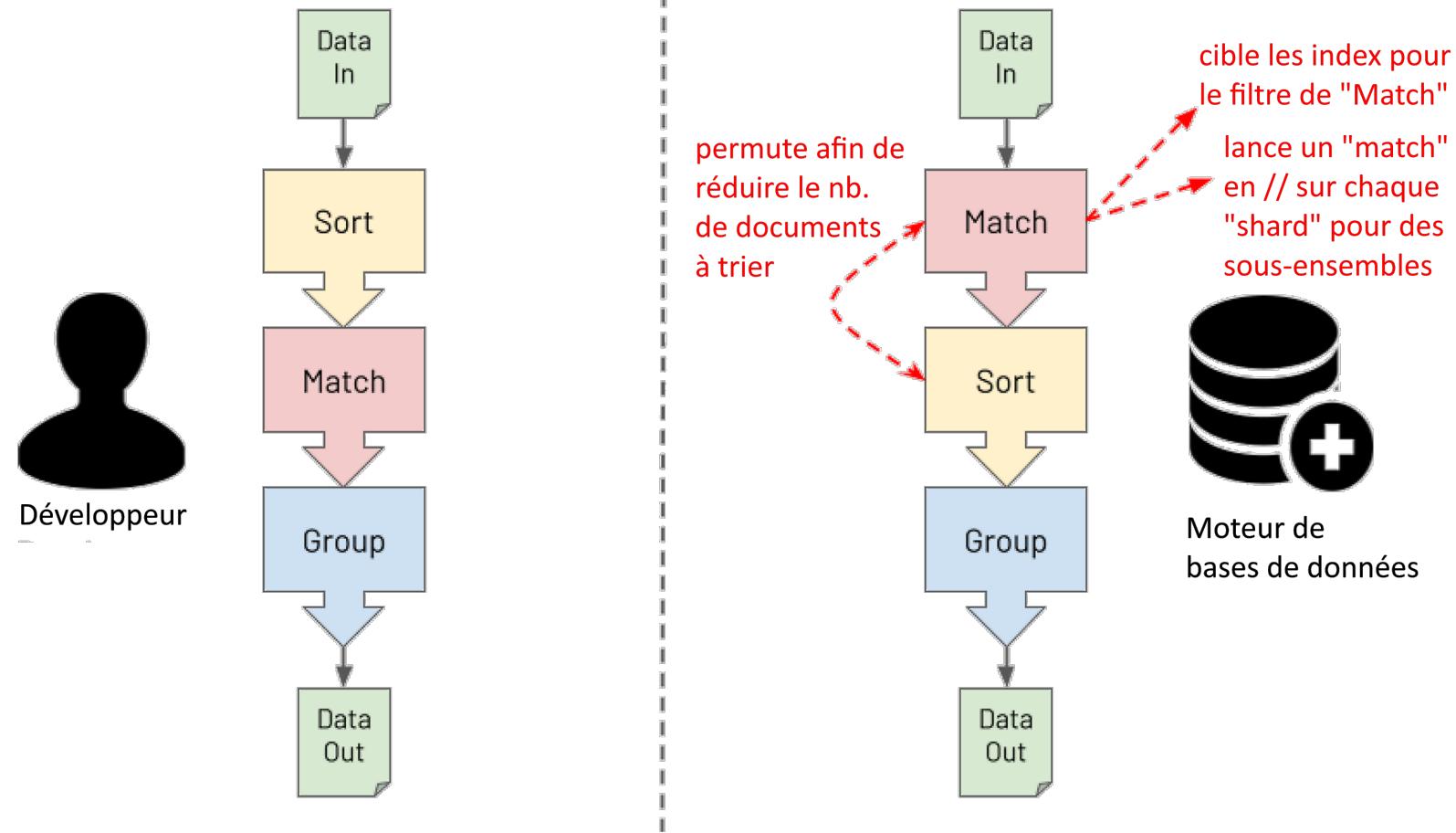


Les étages sont autonomes et sans état

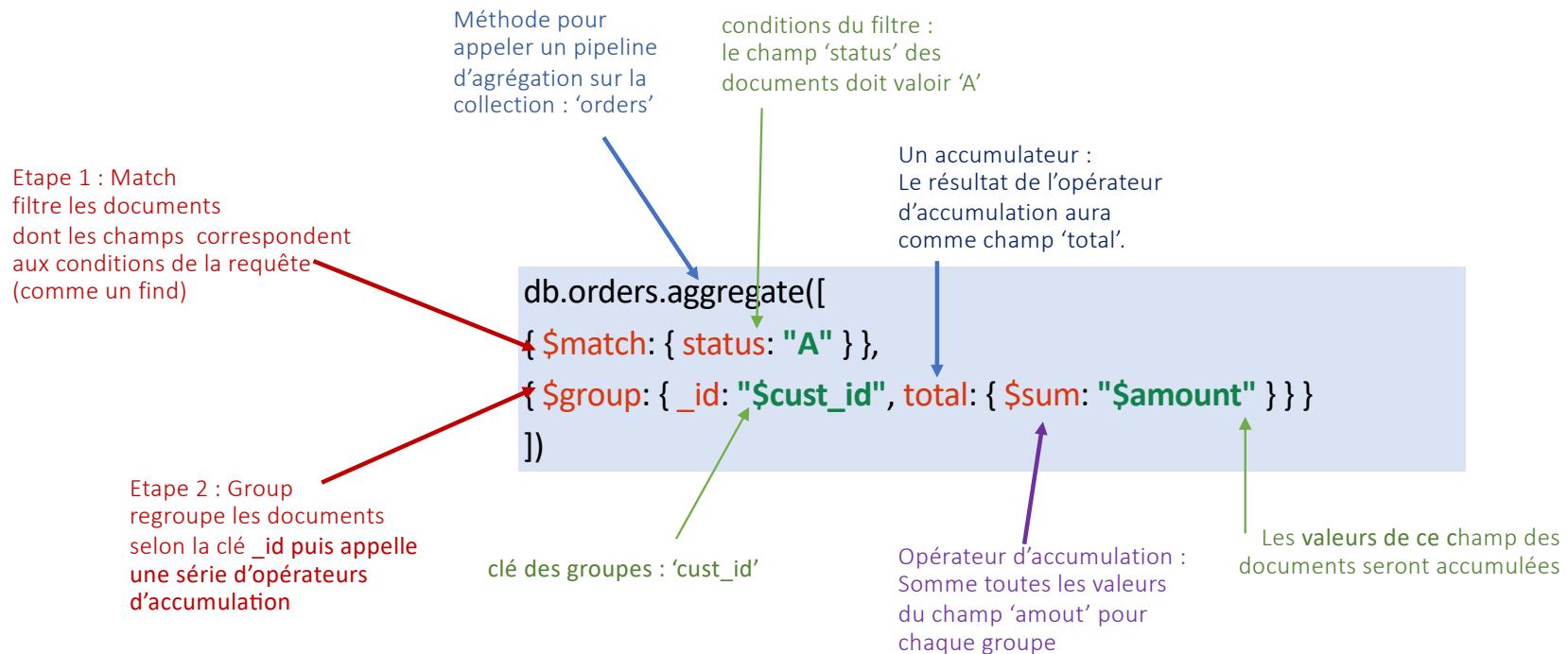
Avec l'agrégation, vous pouvez prendre un problème complexe demandant une agrégation complexe et le décomposer en série d'étapes plus simples qui peuvent être testées et validées indépendamment.



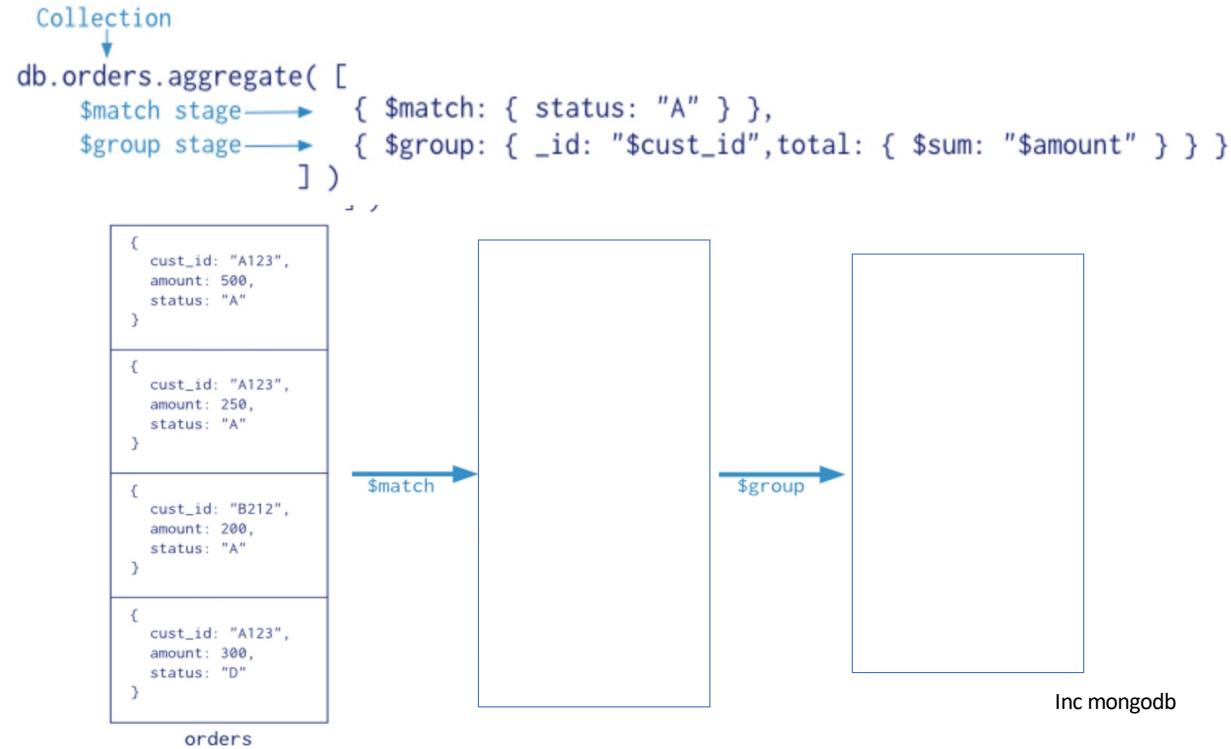
# Aggregation framework



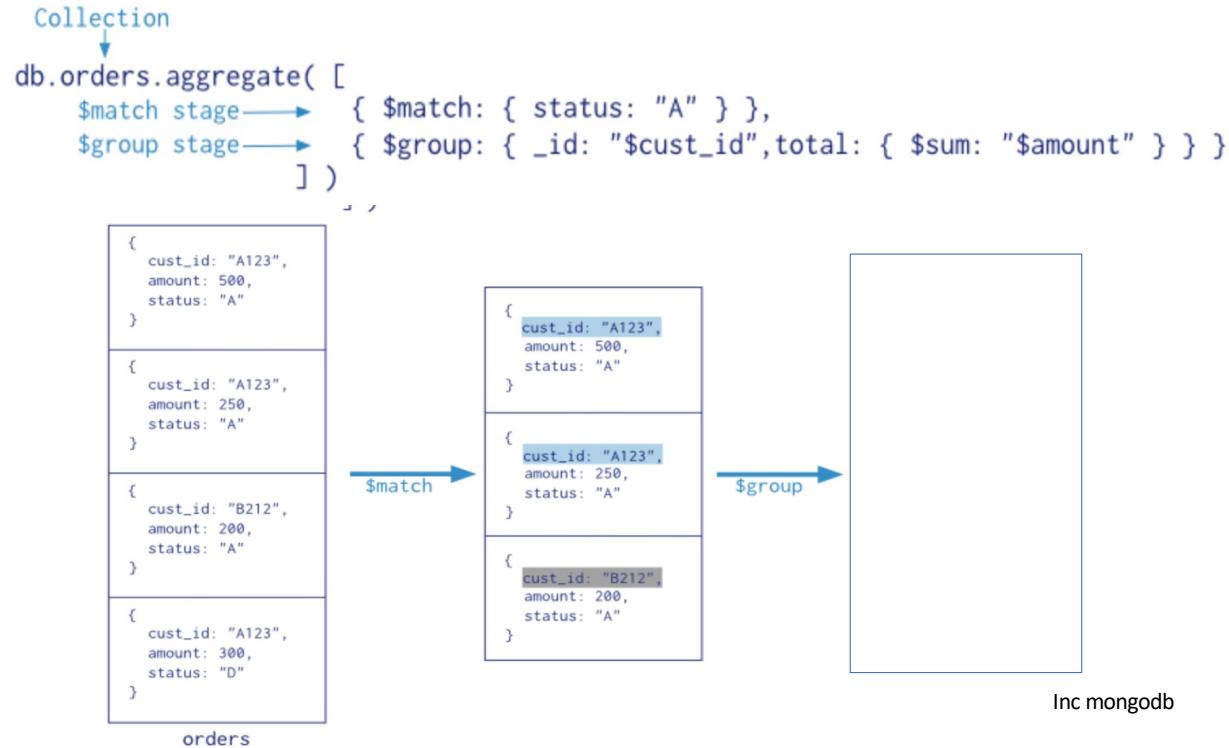
# Aggregation framework



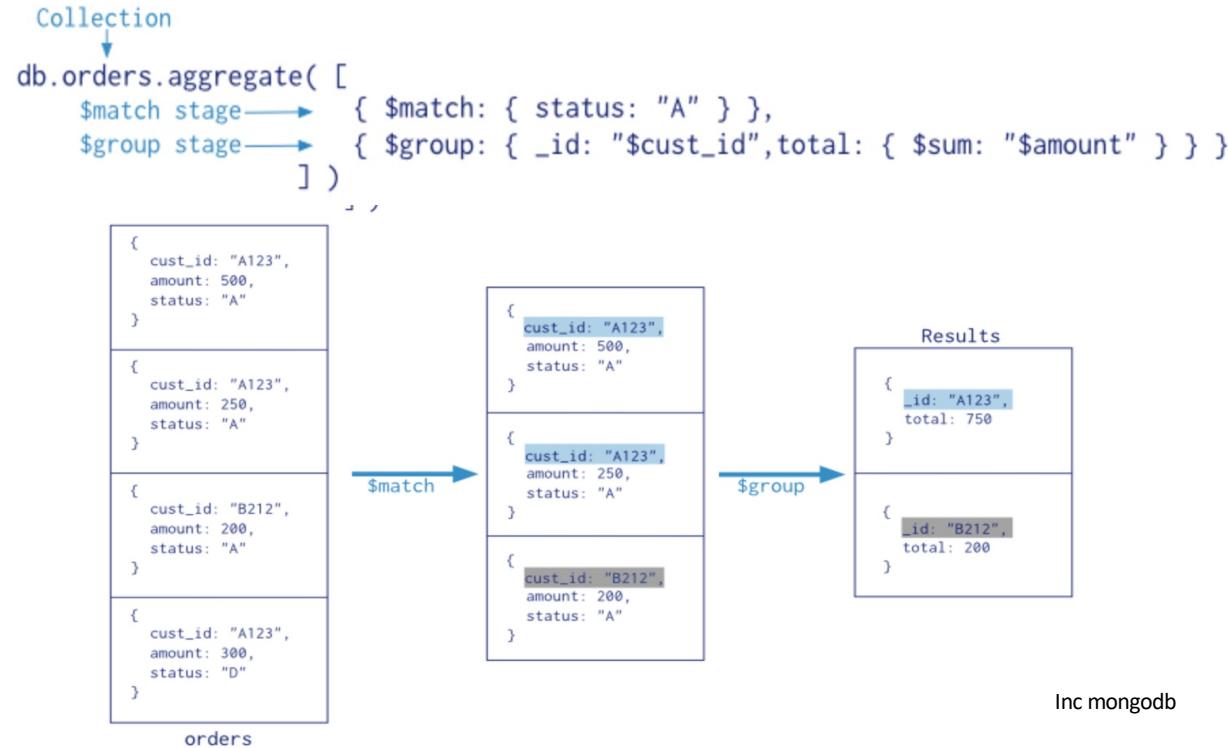
# Aggregation framework



# Aggregation framework



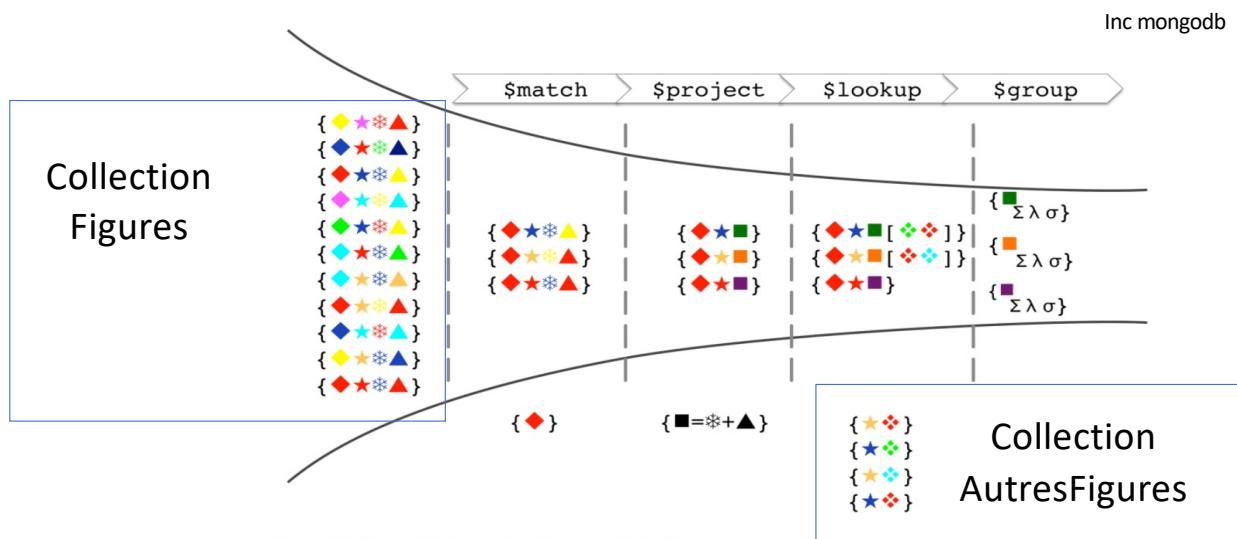
# Aggregation framework



# Aggregation framework



\$match : filtre les documents  
\$project : redéfinition des champs de projection  
\$lookup : jointure gauche  
\$group : regroupe les données par valeur  
...



## Aggregation framework



Les étapes peuvent servir à filtrer, projeter, trier, formater, ... les documents au fur et à mesure de leur avancement dans le pipeline.

La liste complète peut être retrouvée ici :

- <https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/>

Les étapes utilisent des opérateurs pour réaliser les opérations.

Ils sont au nombre de 135 et, comme les étapes peuvent être de différents types :

- Les opérateurs arithmétiques / Les opérateurs sur les tableaux / Les opérateurs booléens
- Les opérateurs sur les tailles / Les opérateurs sur les types / Les opérateurs sur les dates
- Les opérateurs sur les objets / Les opérateurs sur les chaînes / Les opérateurs trigonométriques



TP



Mongo\_TP03\_aggregation\_framework.ipynb

Mongo\_TP04\_aggregation\_framework.ipynb



## La validation des schémas



## Contraintes de domaine



« Schema-less doesn't mean schema design-less » Alex Gamias.

Pour établir des contraintes de structure sur les documents ou sur les valeurs possibles d'une clé, MongoDB implémente la spécification 4 du json schema :

<https://datatracker.ietf.org/doc/html/draft-zyp-json-schema-04>

Une approche similaire au standard XSD pour valider les documents.

Le json schema est encore à ses débuts, il n'est pas encore complet mais l'initiative est très prometteuse.

Le json schema est donc un document json qui décrit un document json.



# JSON Schema

On peut utiliser le JSON Schema pour rechercher, mettre à jour et supprimer les documents qui satisfont un ensemble de contraintes structurelles.

Des règles structurelles qu'on peut associer à une collection afin que tous les documents aient une structure minimale commune.

Pour déclarer une clé de type scalaire, nous utiliserons la syntaxe suivante :

```
macle1: {  
    bsonType: "string"  
}  
,  
macle2: {  
    bsonType: "int"  
},  
macle3: {  
    bsonType: "date"  
},
```

La liste des types supportés par MongoDB est disponible ici :

<https://docs.mongodb.com/manual/reference/operator/query/type/#available-types>

# JSON Schema



On peut définir des contraintes sur la valeur attendue suivant le type.

Par exemple, macle4 sera de type double avec une valeur comprise entre 0 et 10 et devra être un multiple de 2.

```
macle4: {  
    bsonType: "double",  
    multipleOf : 2,  
    minimum :0,  
    maximum :10  
}
```

On peut aussi utiliser les expressions régulières pour définir le patron de la valeur :

```
lastmodification : {  
    bsonType : "string",  
    description : "lastmodification est un champs non obligatoire de type string avec un format de date respectant le format indique dans pattern (e.g. 2009-03-20 / 2010.12.20 ... )"  
    pattern: "(19|20)[0-9]{2}[- /.](0[1-9]|1[012])[- /.](0[1-9]|12)[0-9]3[01])"  
},
```

La liste des contraintes disponibles sur les valeurs est disponible ici :

<https://json-schema.org/understanding-json-schema/reference/>

# JSON Schema



Pour les clés de type complexe comme un tableau, la définition est la suivante :

```
montableau: {  
    bsonType: "array",  
    description: "Mon tableau est un clé de type tableau non obligatoire et doit contenir au moins 2 valeurs quelconques mais non redondantes.",  
    minItems: 2,  
    uniqueItems: true  
}
```

On peut définir différentes contraintes sur les tableaux au niveau de :

- la taille minimale et maximale (minItems, maxItems)
- l'unicité des valeurs (uniqueItems)
- le type des valeurs associées
- le scope des valeurs attendues

Une description plus complète est disponible ici :

<https://json-schema.org/understanding-json-schema/reference/array.html>

## JSON Schema

L'exemple ci-dessous déclare la clé mytab comme un tableau de nombre dont la valeur est limitée aux valeurs 10 et 30 avec un nombre de valeur compris entre 1 et 5 :

```
mytab: {  
    bsonType: "array",  
    minItems: 1,  
    maxItems: 5,  
    items: {  
        bsonType: "double",  
        enum : [10,30]  
    }  
}
```

## JSON Schema

Pour les clés de type document /objet, la définition est la suivante :

```
mondocument: {  
    "type": "object",  
    "properties": {  
        "numero": { "type": "number" },  
        "nom_rue": { "type": "string" },  
        "type_rue": { "enum": ["Rue", "Avenue", "Boulevard"] }  
    }  
},
```

Un document est un ensemble de « clé : valeur » par conséquent il est nécessaire de préciser les propriétés / attributs du document.

Dans l' exemple ci-dessous, la clé « mondocument » est une clé dont la valeur attendue est un objet complexe/sous-document :

- numero de type number
- nom\_rue de type string
- type\_rue de type enumération ayant pour valeur Rue, Avenue ou Boulevard

## JSON Schema

Pour déclarer un document avec une clé de type tableau contenant des sous documents.

```
mondocument: {
  bsonType: "object",
  properties: {
    montableau: {
      bsonType: "array",
      items: {
        type: "object",
        properties: {
          "numero": { "type": "number" },
          "nom_rue": { "type": "string" },
          "type_rue": { "enum": ["Rue", "Avenue", "Boulevard"] }
        }
      }
    }
}
```

# JSON Schema

Le JSON Schema fournit des fonctions de compositions (anyOf, OneOf, allOf, not ) pour composer votre propre schéma dans les structures de document complexe.

L'exemple ci-dessous limite les éléments d'un tableau à des types document ou string.

```
couleurs:{  
    bsonType: "array",  
    description : "magasins est un tableau" et doit contenir une String ou un document",  
    items: {  
        oneOf: [  
            {  
                bsonType: ["object"],  
                required:["code","nom"],  
                description : "la valeur est un document et contiendra 2 clés code et nom",  
                properties: {  
                    code: {  
                        enum: ["9010", "9015", "9040"],  
                        description: "le code est un enum"  
                    },  
                    nom: {  
                        bsonType: "string",  
                        description: "le nom de la couleur est une string"  
                    }  
                }  
            },  
            {  
                bsonType: "string"  
            }  
        ]  
    }  
}
```

## JSON Schema

Il est aussi possible d'exprimer des contraintes sur la présence d'une propriété dans un document avec le mot clé required :

```
{  
    bsonType: "object",  
    required: [ "mondocument" ],  
    properties: {  
        mondocument: {  
            bsonType: "object",  
            properties: {  
                montableau: {  
                    bsonType: "array",  
                    items: {  
                        bsonType: "object",  
                        required: [ "nom_rue", "type_rue" ],  
                        properties: {  
                            numero: { "type": "number" },  
                            nom_rue: { "type": "string" },  
                            type_rue: { "enum": [ "Rue", "Avenue", "Boulevard" ] }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

Dans l'exemple ci-dessus, les documents qui valideront le JSON Schema devront avoir obligatoirement une clé « mondocument » de type document.

Cette clé « mondocument » pourra disposer de n'importe quelle clé mais si l'une d'entre elles porte le nom de « montableau » et est de type document alors elle devra contenir obligatoirement une clé « nom\_rue » et « type\_rue » en respectant le scope des valeurs.

## JSON Schema et les opérations CRUD



L'utilisation d'un JSON Schema passe par l'opérateur \$jsonSchema.

Que ce soit pour la recherche, la mise à jour ou la suppression de documents, l'opérateur \$jsonSchema intervient comme une condition additionnelle.

```
db.exemple.find(  
  {  
    $jsonSchema:{  
      bsonType: "object",  
      required: [ "qty"],  
      properties: {  
        qty: {  
          bsonType: "int",  
          minimum: 0  
        }  
      }  
    }  
  }  
)
```

Par exemple rechercher les documents  
contenant une clé « qty » de type entier

```
db.exemple.remove(  
  {  
    $jsonSchema:{  
      bsonType: "object",  
      required: [ "qty"],  
      properties: {  
        qty: {  
          bsonType: "int",  
          minimum: 0  
        }  
      }  
    }  
  })
```

Par exemple supprimer les documents  
contenant une clé « qty » de type entier

## JSON Schema et les opérations CRUD

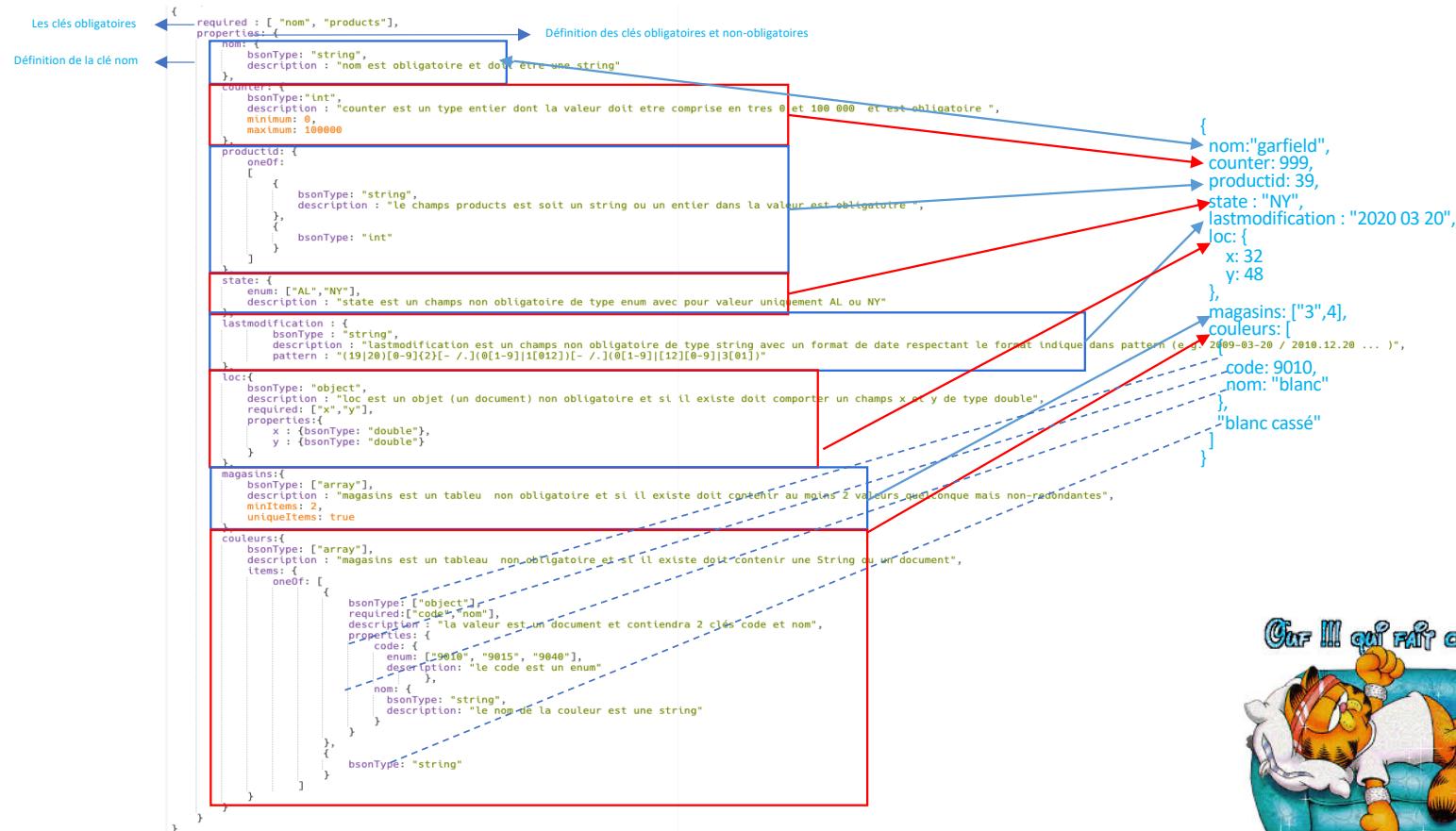


On peut bien entendu coupler les contraintes structurelles avec des conditions sur les clés :

```
db.exemple.find(  
  { $and:  
    [  
      {  
        $jsonSchema:{  
          bsonType: "object",  
          required: [ "qty"],  
          properties: {  
            qty: {  
              bsonType: "int",  
              minimum: 0  
            }  
          }  
        },  
        {instock: true}  
      ]  
    }  
  )
```

Par exemple rechercher les documents contenant une clé « qty » de type entier avec une valeur positive et dont la clé « instock » est à true.

## JSON Schema : exemple



# Les applications d'un JSON Schema



Les contraintes de domaine peuvent être associées :

- à la collection sur les opérations d'insertion et d'update.

```
db.createCollection( <collection>, { validator: { $jsonSchema: <schema> } } )
```

```
db.collection.find( { $jsonSchema: <schema> } )
db.collection.updateMany( { $jsonSchema: <schema> }, <update> )
db.collection.remove( { $jsonSchema: <schema> } )
db.collection.deleteMany( { $nor: [ { $jsonSchema: myschema } ] } )
```

```
db.collection.aggregate( [ { $match: { $jsonSchema: <schema> } } ] )
```



# TP Json Schema

Mongo\_TP04\_json\_schema.ipynb



Les doigts dans le nez !!!

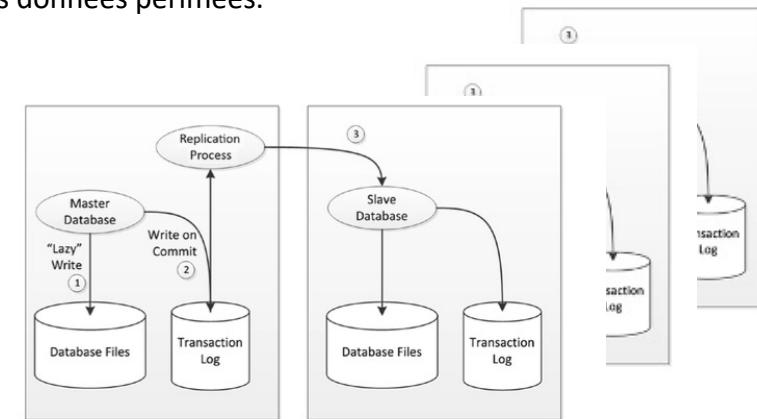


# La réPLICATION avec MongoDB

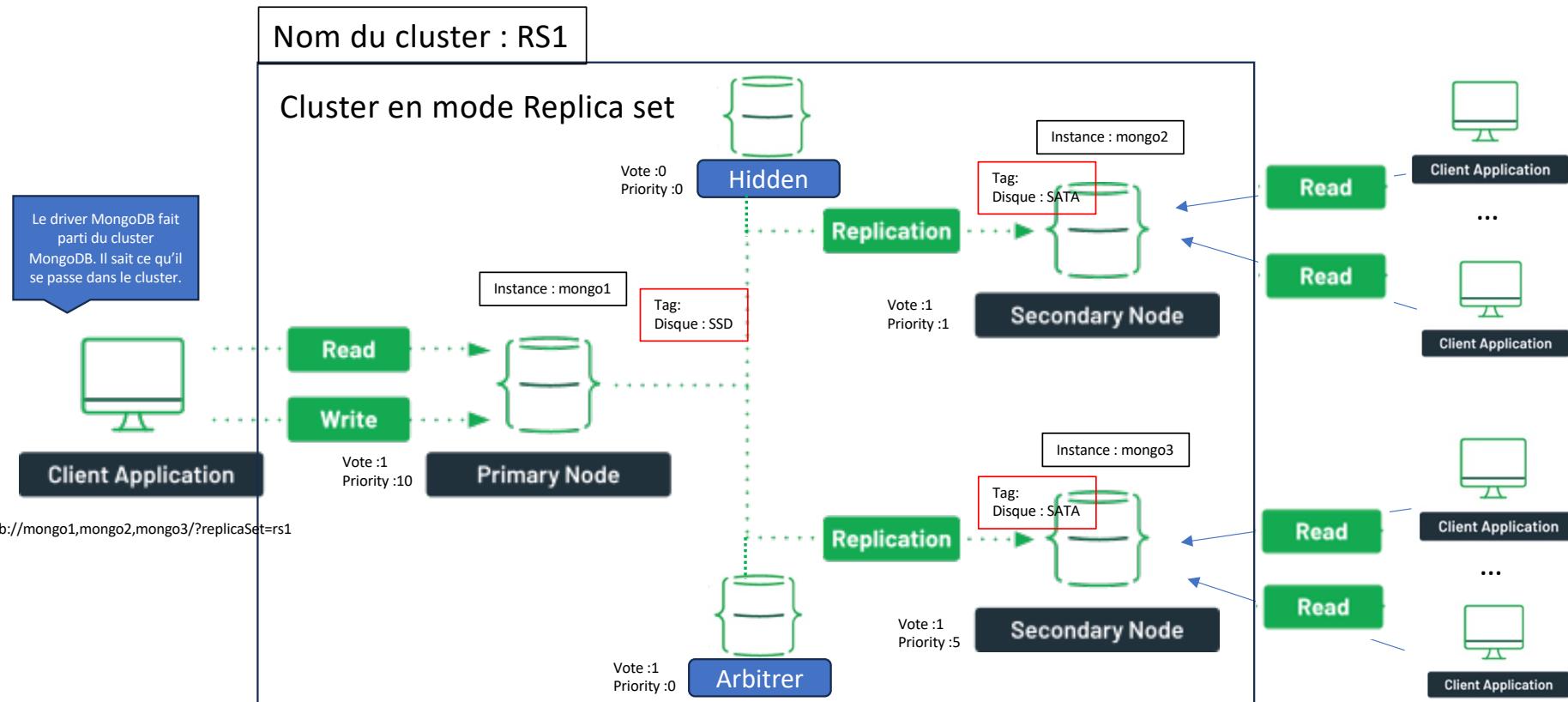


# La réPLICATION avec MongoDB

- La réPLICATION est très utilisée pour les applications où les données sont très souvent accédées en lecture.
- A titre d'indication, le ratio écriture / lecture est en général de 20 / 80 sur une base de données.
- Avec le début web 2.0, la première approche consistait à distribuer les lectures sur plusieurs réplicas.
- Une solution améliorant la disponibilité.
- Les données sont répliquées via les journaux de transactions mais un compromis doit être fait entre performance et cohérence :
  - Si la réPLICATION est synchrone, la transaction est validée uniquement lorsque celle-ci est validée sur le primaire ET le secondaire (Performances dégradées).
  - Si la réPLICATION est asynchrone, il est possible de lire sur le secondaire des données périmées.



# La réPLICATION avec MongoDB



# La réPLICATION avec MongoDB

- Pour configurer un cluster en mode réPLICATION, il sera nécessaire de déclarer les membres de votre cluster.
- Comment le primaire est élu ?
  - L'élection se fait avec un mécanisme de vote à la majorité(quorum)
  - Uniquement les membres ayant une carte de vote (vote:1) et une priorité (priority > 0) peuvent voter.
  - Pour définir des membres à privilégier comme primaire il suffit d'associer une priorité plus élevée que les autres membres.
  - Un membre avec une priorité de 0 ne peut pas devenir primaire mais il contiendra une copie des données.
  - Les membres avec un vote à 0 ne contribueront pas à l'élection du primaire.
- Les rôles pour les membres :
  - Primary : priority > 0 et vote =1
  - Secondary : priority > 0 et vote =1
  - Hidden: idem 'Secondary' mais non visible (vote=0 et priority=0). Il contient des données et peut être utilisé pour les backups.
  - Arbitre : utilisé lors du mécanisme d'élection du primaire (vote=1 et priority=0), il ne contient pas de données.

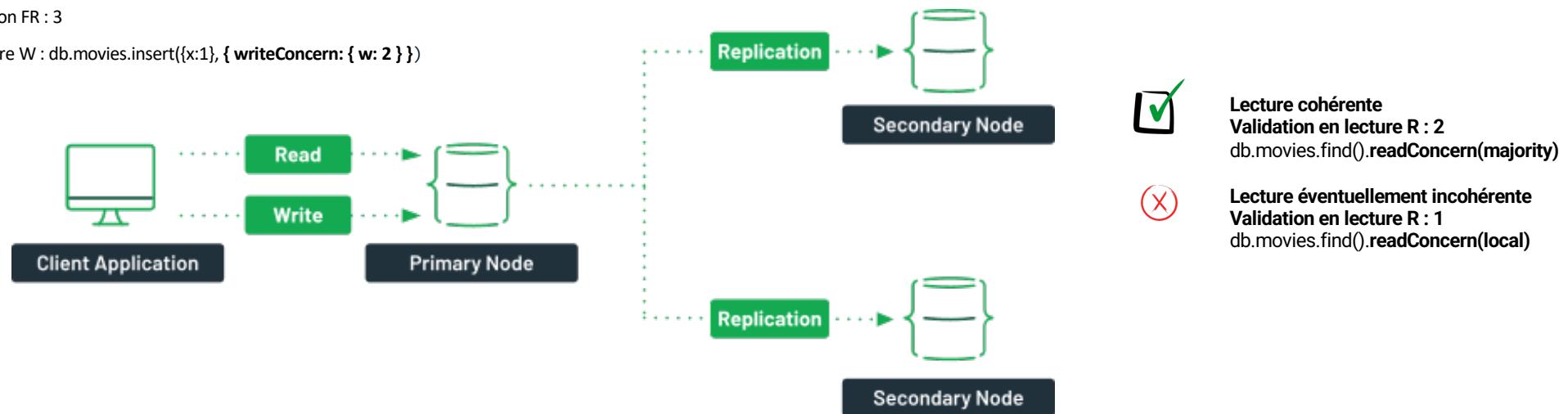
# La réPLICATION avec MongoDB

La cohérence des données lues est établie selon le niveau de cohérence demandé en écriture :

$$W + R > FR$$

Facteur de réPLICATION FR : 3

Validation en écriture W : db.movies.insert({x:1}, { writeConcern: { w: 2 } })



TP



# Le sharding

Les données sont fragmentées horizontalement en partition.

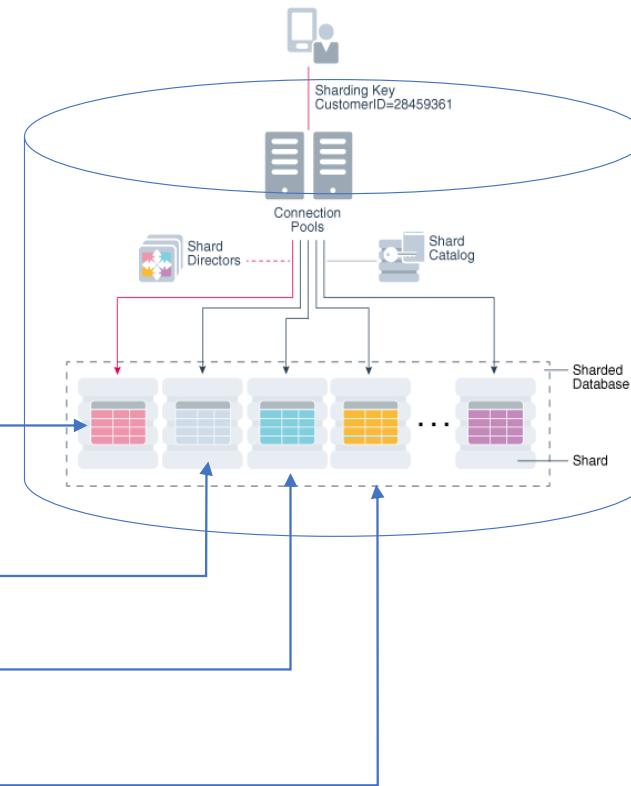
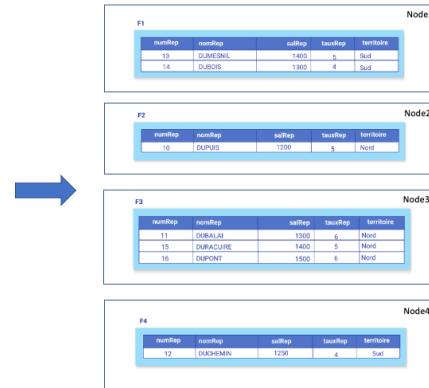
Les partitions sont distribuées sur plusieurs machines.

Les requêtes sont réparties sur un sous-ensemble de machines.

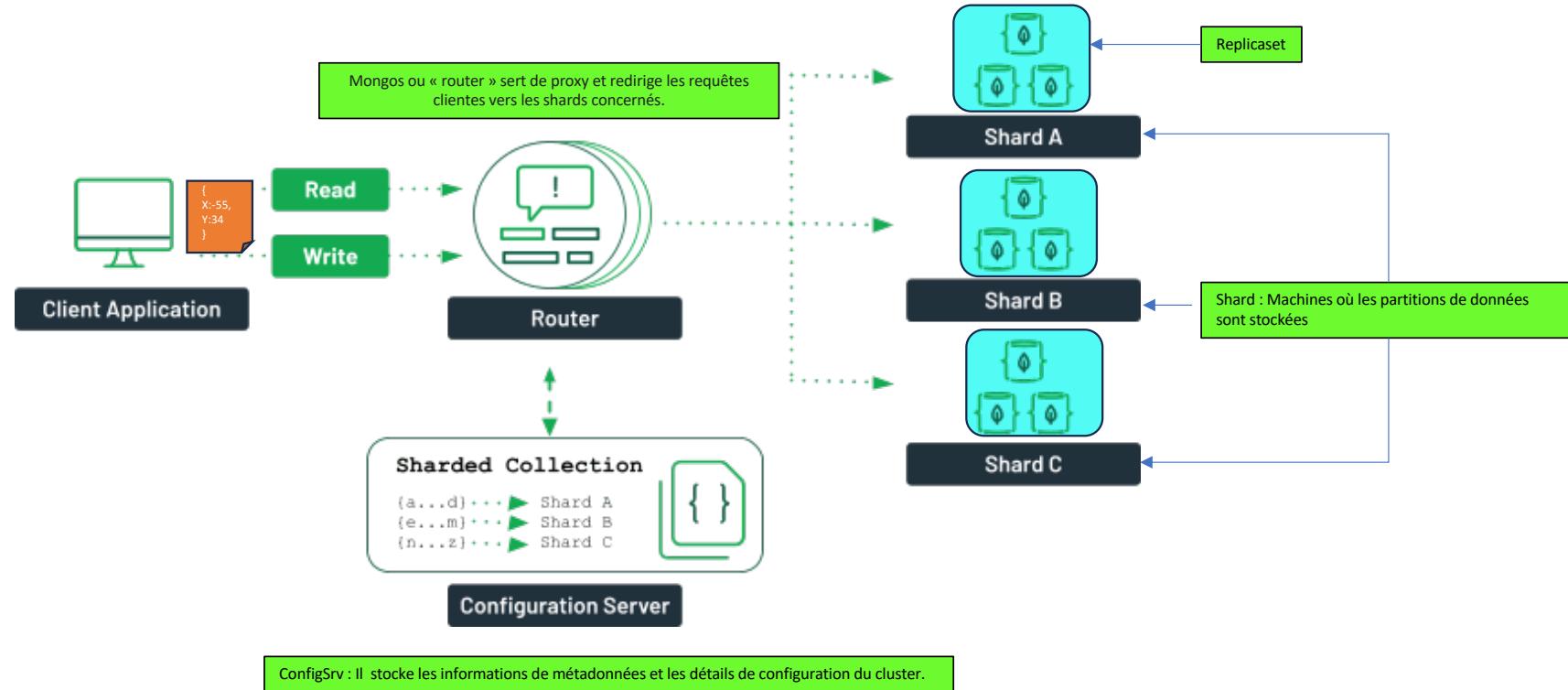
## Gain en performance

- Des écritures et lectures distribuées
- Des lectures parallélisées

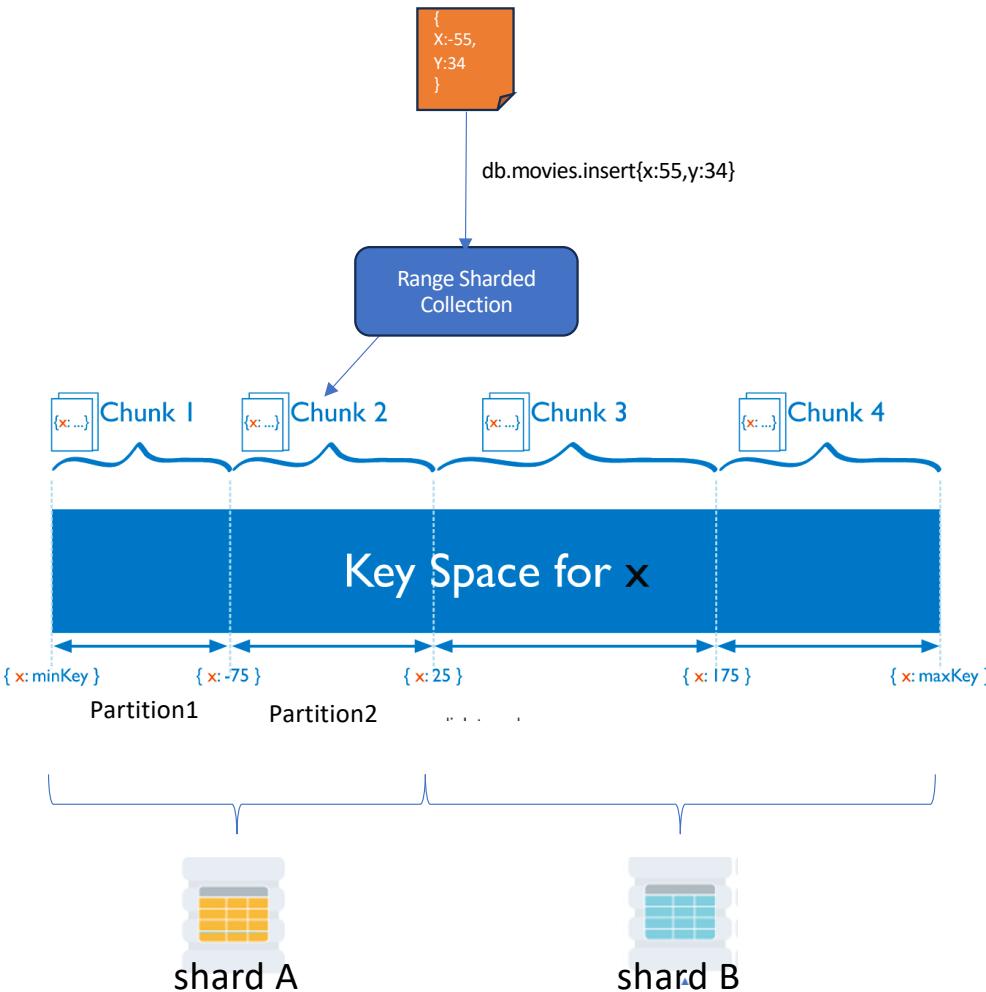
REPRÉSENTANT					
numRep	nomRep	salleRep	tauxRep	territoire	
10	DUBALAI	1200	5	Nord	
11	DUBALAI	1300	6	Sud	
12	DUCHEMIN	1250	4	Sud	
13	DUCHEMIN	1400	5	Sud	
14	DUBOIS	1300	4	Sud	
15	DURACURE	1400	5	Nord	
16	DUPONT	1500	6	Nord	



# Le sharding : Les composants



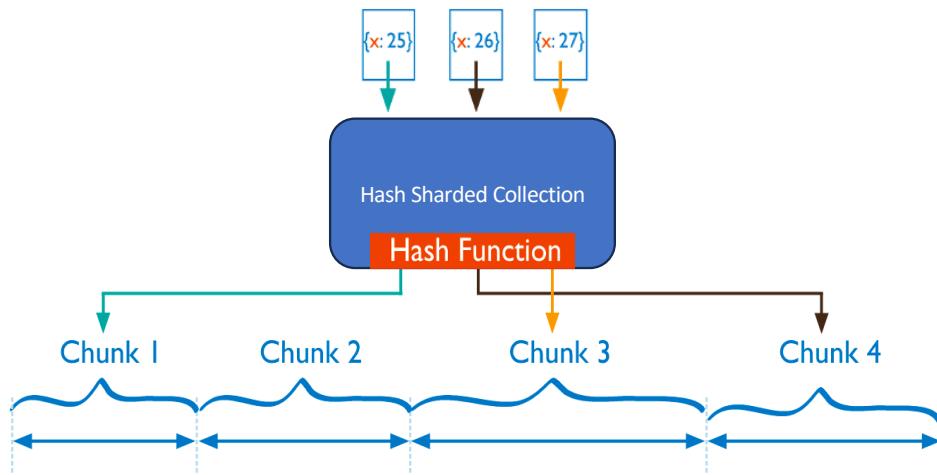
# Le sharding : Distribution par intervalle (Range)



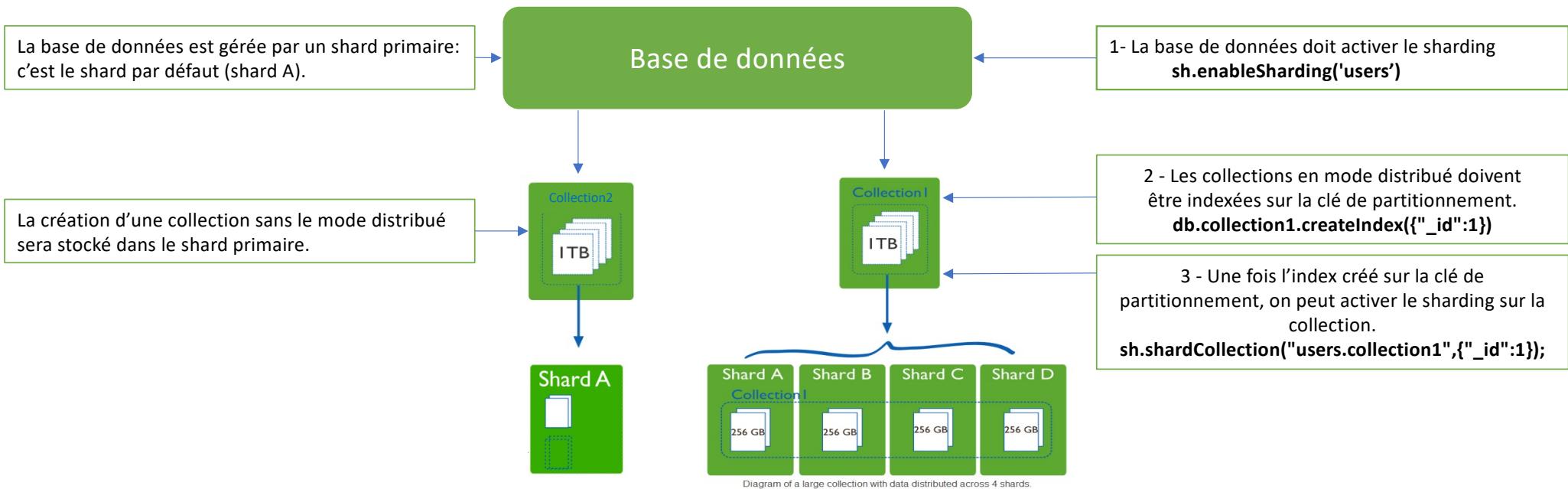
- ❑ Une collection distribuée par intervalle attribue l'intervalle d'un document selon une clé ou d'un sous ensemble de clés.
- ❑ La où l'ensemble des clés utilisées pour la distribution est appelé clé de partitionnement (shard key).
- ❑ On appellera chunk / partition l'ensemble des documents associé à un intervalle.
- ❑ L'attribution d'une partition à un document est déterminée par la valeur de la clé de partitionnement.
- ❑ La Shard Key est un champ obligatoire pour tous les documents à insérer dans la collection distribuée.
- ❑ Chaque chunk est stocké physiquement dans un nœud/shard.
- ❑ Exemple:
  - Une collection partitionnée sur la clé X avec une logique par intervalle inserera le document avec la clé X=55 au chunk 2.

# Le sharding : Distribution par Hash

- Une collection distribuée par Hash attribue un document selon une clé ou d'un sous ensemble de clés.
- La où l'ensemble des clés utilisées pour la distribution est appelé clé de partitionnement.
- On appellera chunk / partition l'ensemble des documents associé à un intervalle.
- Chaque chunk est stocké physiquement dans un nœud/shard.
- L'attribution d'une partition à un document est déterminée par la valeur de la clé de partitionnement.
- Cette approche utilise une fonction de hachage pour transformer la valeur de la clé de shard en une autre valeur, qui est ensuite utilisée pour déterminer le shard sur lequel un document donné doit être stocké.

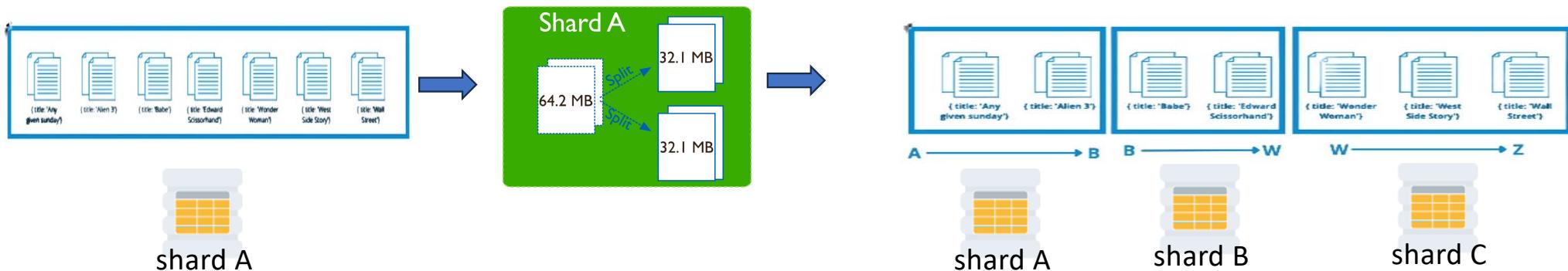


# Le sharding : Son implémentation



# Le sharding : La répartition des chunks

- ❑ Lors de la création d'une clé de sharding, tous les documents sont rassemblés dans un même chunk.
- ❑ Les chunks peuvent avoir une taille maximum de 1Mo à 1024Mo.
- ❑ Les documents sont répartis en chunks de taille équivalente en fonction de la shard key et de la taille maximale de chaque chunk.
- ❑ Le choix de la shard key est critique pour l'extensibilité.
- ❑ Par exemple, le choix de sharding par la première lettre du titre limitera le sharding à 26 shards. En revanche, en choisissant une shard key hashée à partir du titre, on s'affranchit de cette limite et on laisse mongoDB définir le nombre de shards sans limite.



# Le sharding : la clé de partitionnement

- Recommandations pour la clé de partitionnement
- La cardinalité
  - Plus la cardinalité de la clé est élevée et mieux mongoDB pourra partitionner la collection en chunk pour les distribuer sur les shards. On favorisera une extensibilité horizontale en ajoutant des shards.
  - Utiliser un shard key sur le mois d'une date limite le nombre de chunk à 12 et par conséquent l'extensibilité de notre cluster à 12 shards.
- La fréquence des opérations
  - Si les opérations de lecture et d'écriture sont fréquentes pour certaines valeurs de la shard key, il est important que ces valeurs soient distribuées de manière équilibrée entre les différents shards.
  - Éviter les hotspots: Les hotspots se produisent lorsque certaines valeurs de la shard key sont beaucoup plus fréquentes que d'autres, entraînant une charge inégale entre les shards.
  - Optimisation des requêtes : Si certaines valeurs de la shard key sont utilisées plus fréquemment dans les requêtes.
- Unicité de la shard key
  - Dans le modèle relationnel, la shard key est souvent défini comme une super clé primaire (un sous ensemble de la clé primaire) afin d'assurer une meilleure répartition des données.
  - Même si cela est une bonne pratique et une obligation dans le modèle relationnel, MongoDB nous permet de définir n'importe quelle clé comme shard key.
- La clé est immuable
  - une fois qu'une shard key a été définie, celle-ci ne doit pas être modifiée.
  - Modifier la shard key d'une collection entraînerait des déplacements de données conséquents afin de respecter la nouvelle répartition des données entre les shards, ce qui est très coûteux.

# TP: chap6\_MongoDB\_Sharding\_TP02

