

MongoDB



MongoDB



Forte concurrence dans le secteur des bases NoSQL

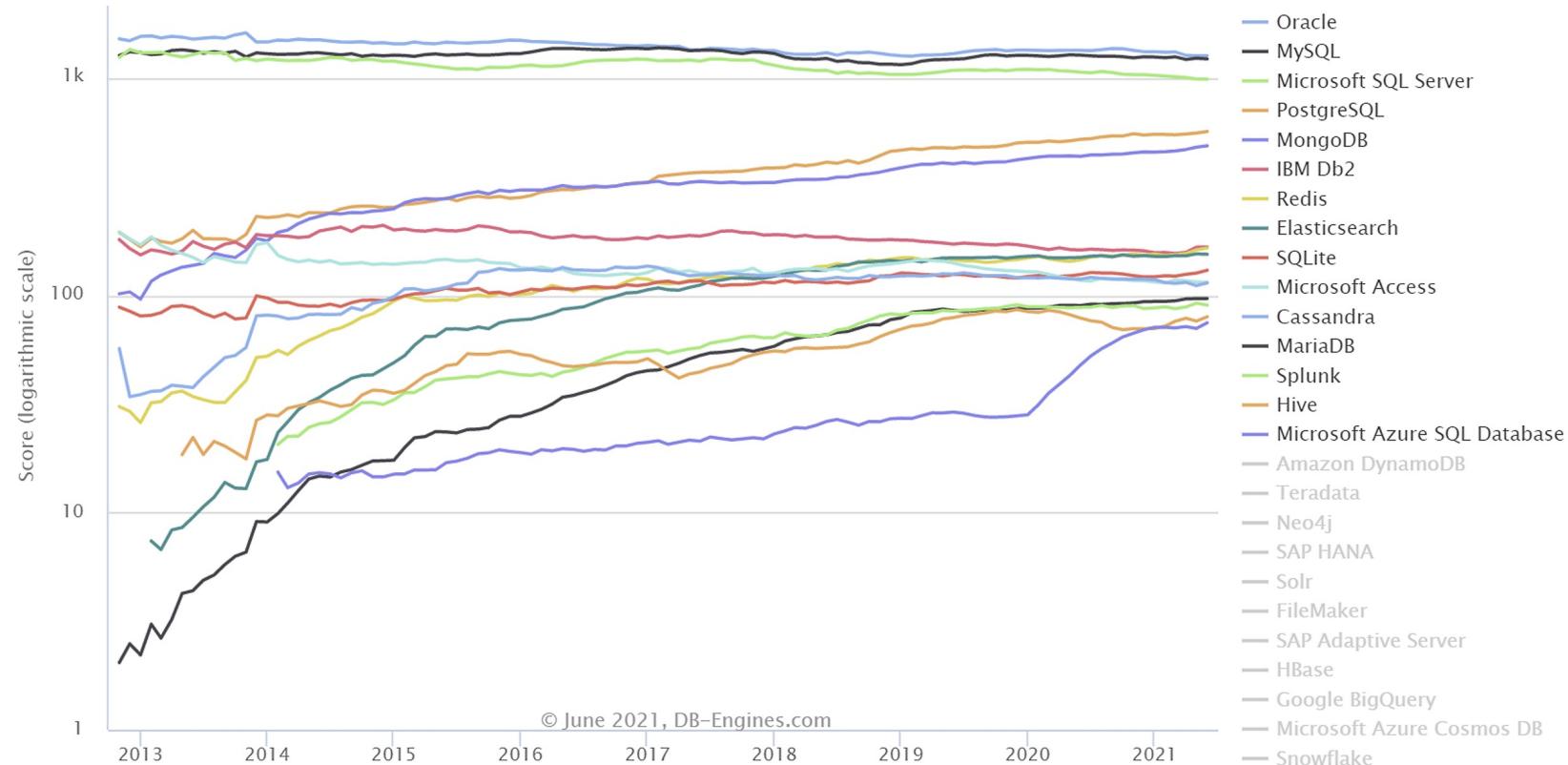
Comme toutes les sociétés basant leurs modèles sur « l'Opensource », MongoDB Inc. complète son offre avec des outils spécifiques (interface d'administration Compass par exemple) ou du support ... Payants

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



Osman AIDEL / MongoDB

Structure d'un document Json

Je suis un
document
JSON

```
{  
    first_name: 'Paul',  
    surname: 'Miller',  
    cell: 447557505611,  
    city: 'London',  
    location: [45.123,47.232],  
    Profession: ['banking', 'finance', 'trader'],  
    cars: [  
        { model: 'Bentley',  
         year: 1973,  
         value: 100000, ... },  
        { model: 'Rolls Royce',  
         year: 1965,  
         value: 330000, ... }  
    ]  
}
```

Fields

String

Number

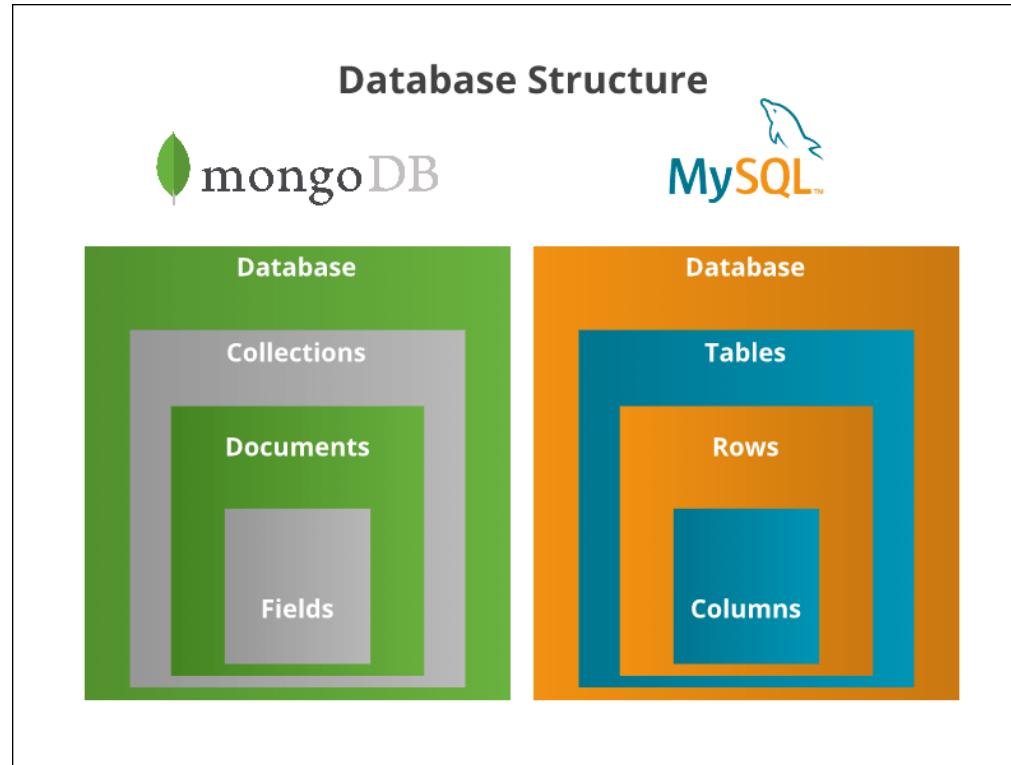
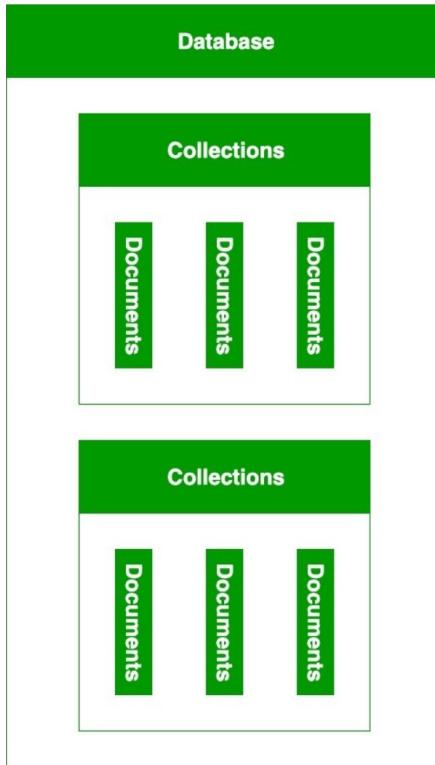
Geo-Coordinates

Typed field values

Fields can contain arrays

Fields can contain an array of sub-documents

MongoDB



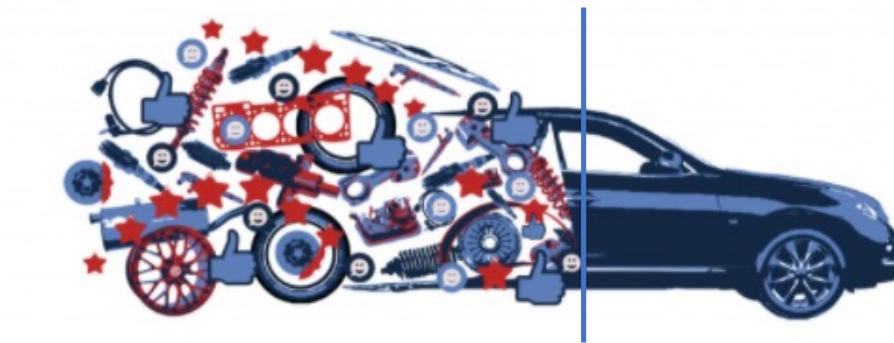
Osman AIDEL / MongoDB



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

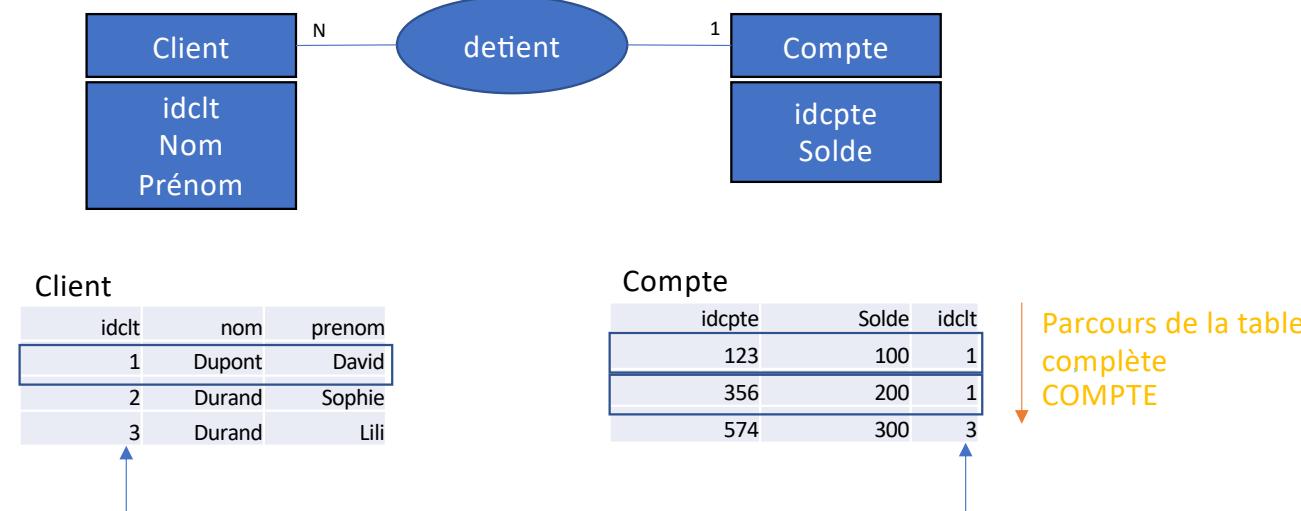


Représentation logique sous forme de documents



Introduction au modèle orienté document

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



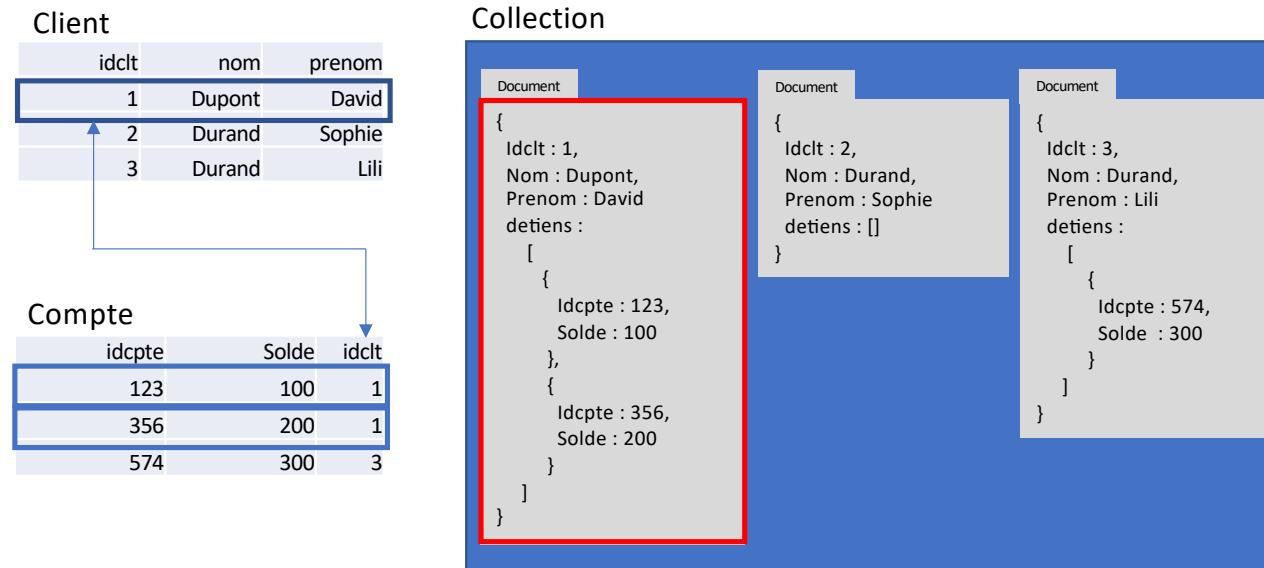
Comment obtenir le solde des comptes du Client David Dupont ?

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

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

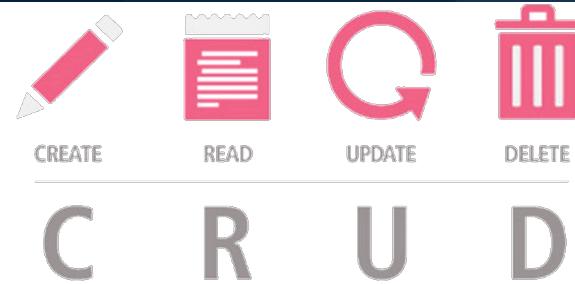
Introduction au modèle orienté document

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**

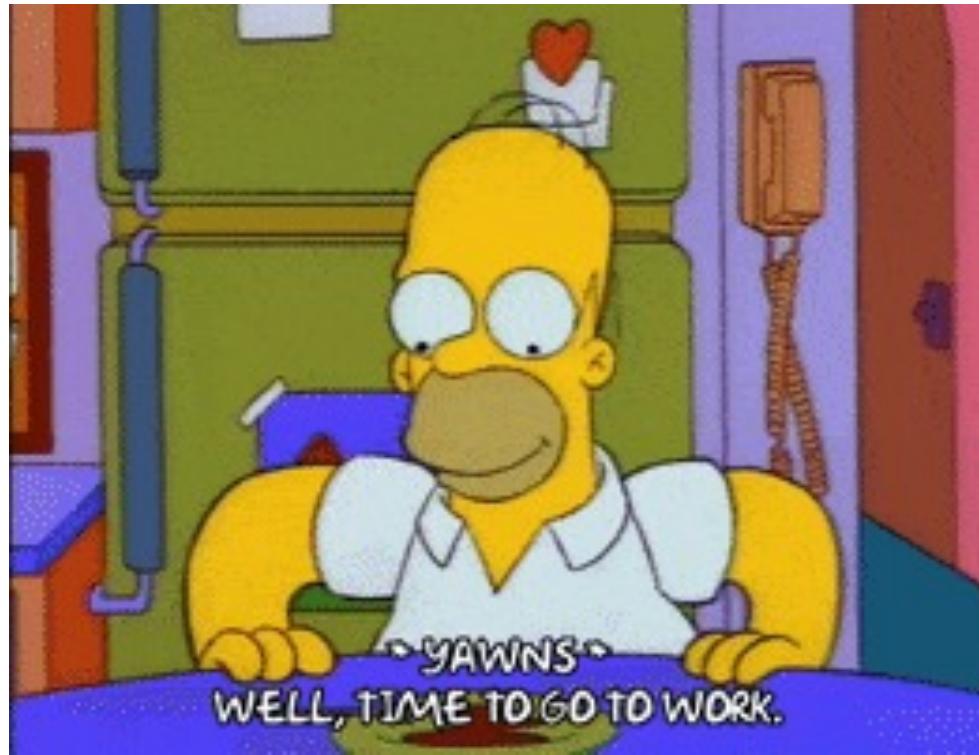


TP



Mongo_TP01_introduction.ipynb

Mongo_TP02 CRUD.ipynb



Aggregation framework



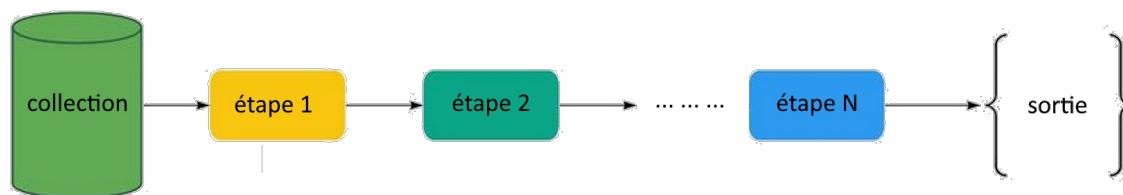
Aggregation framework



MongoDB intègre une plateforme qui permet d'enchâiner 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

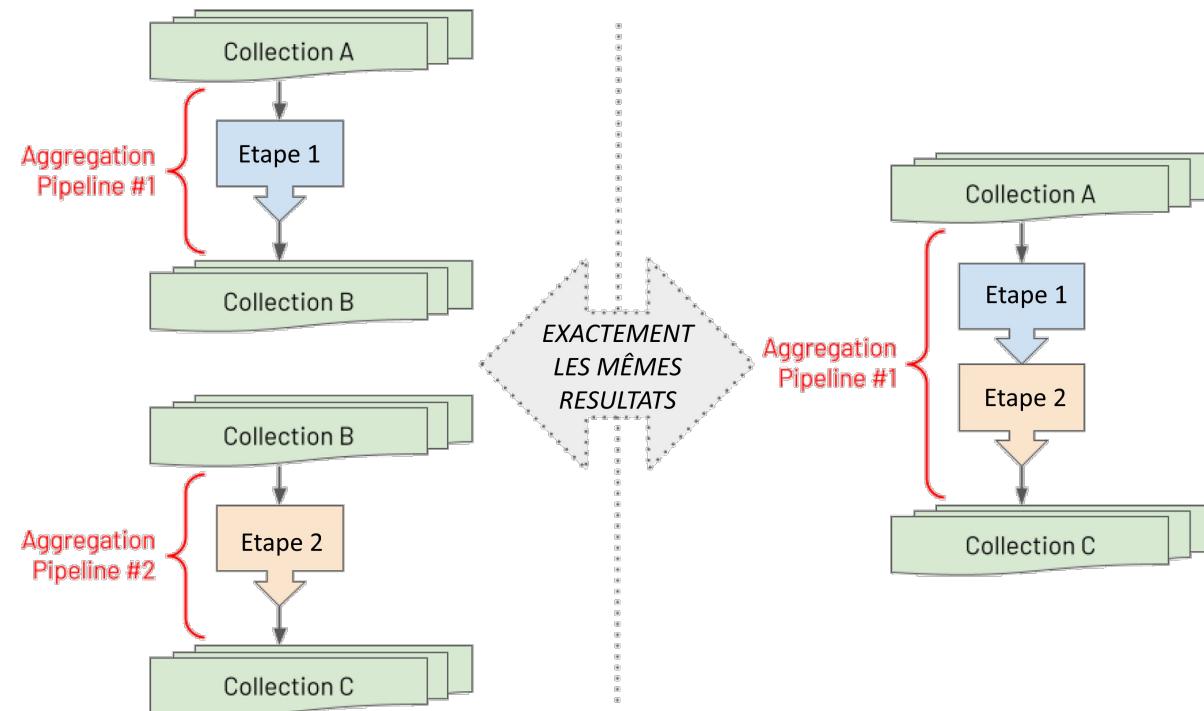


Aggregation framework

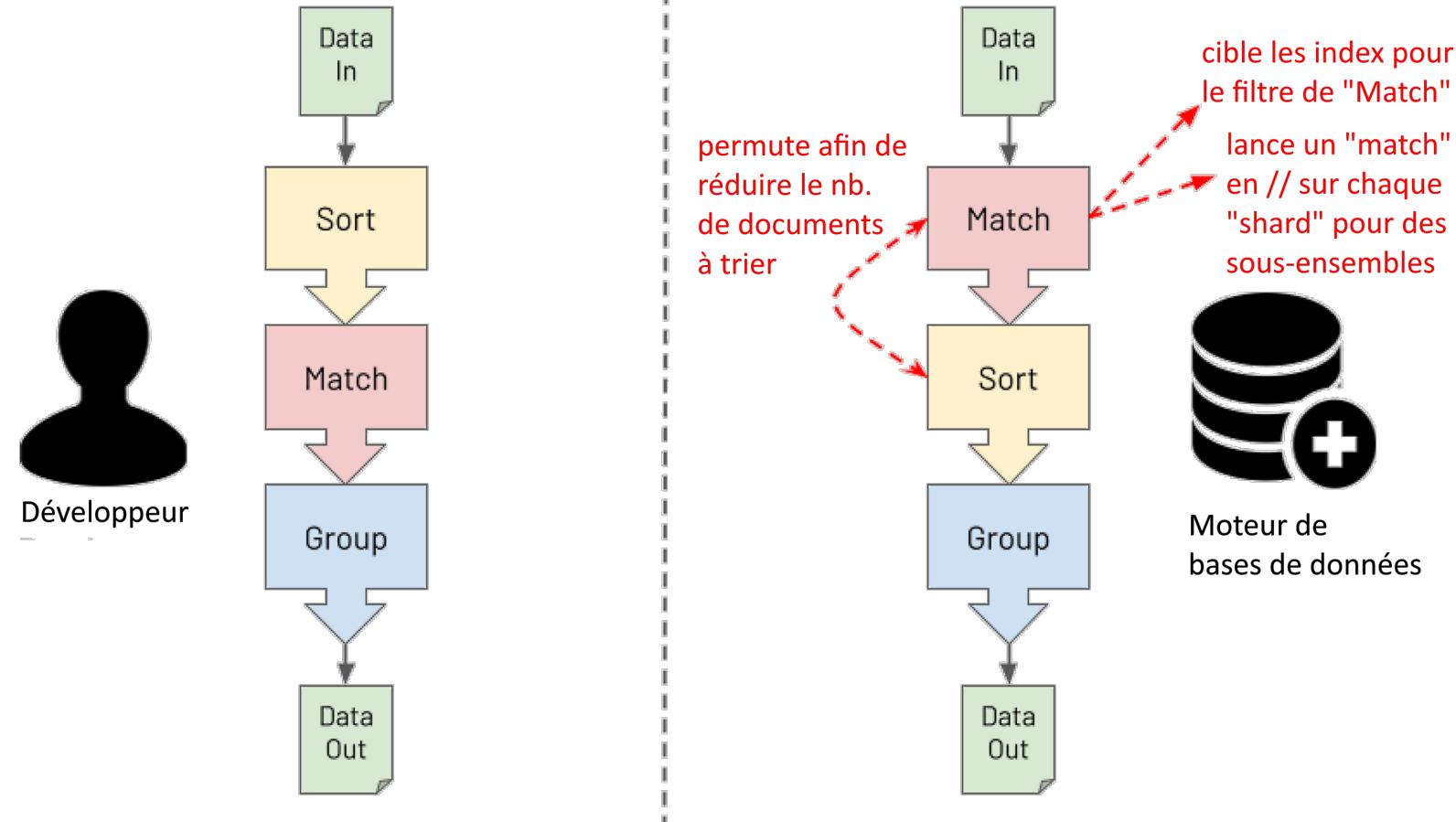


Les étages sont autonomes et sans état

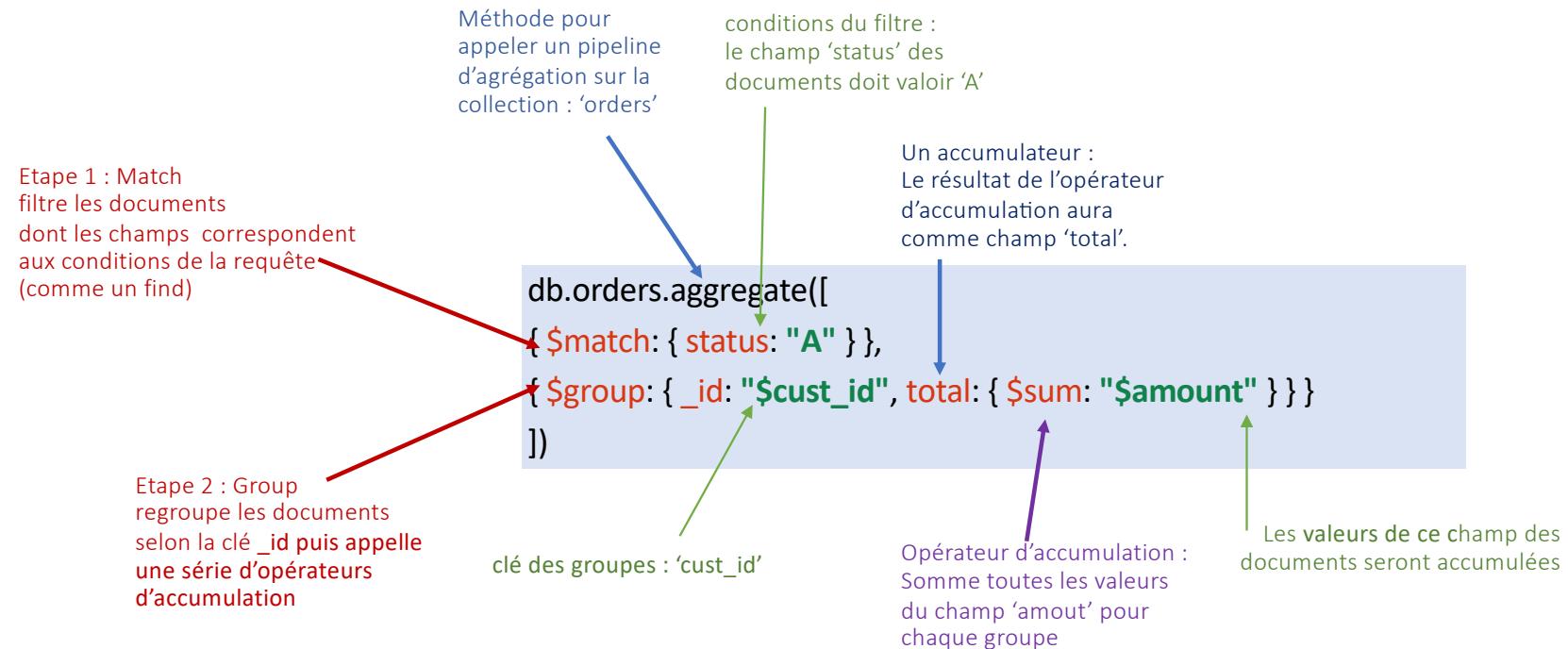
Avec l'agrégation, vous pouvez prendre un problème complexe demandant une agrégation complexe et le décomposer en série d'étapes plus simples qui peuvent être testées et validées indépendamment.



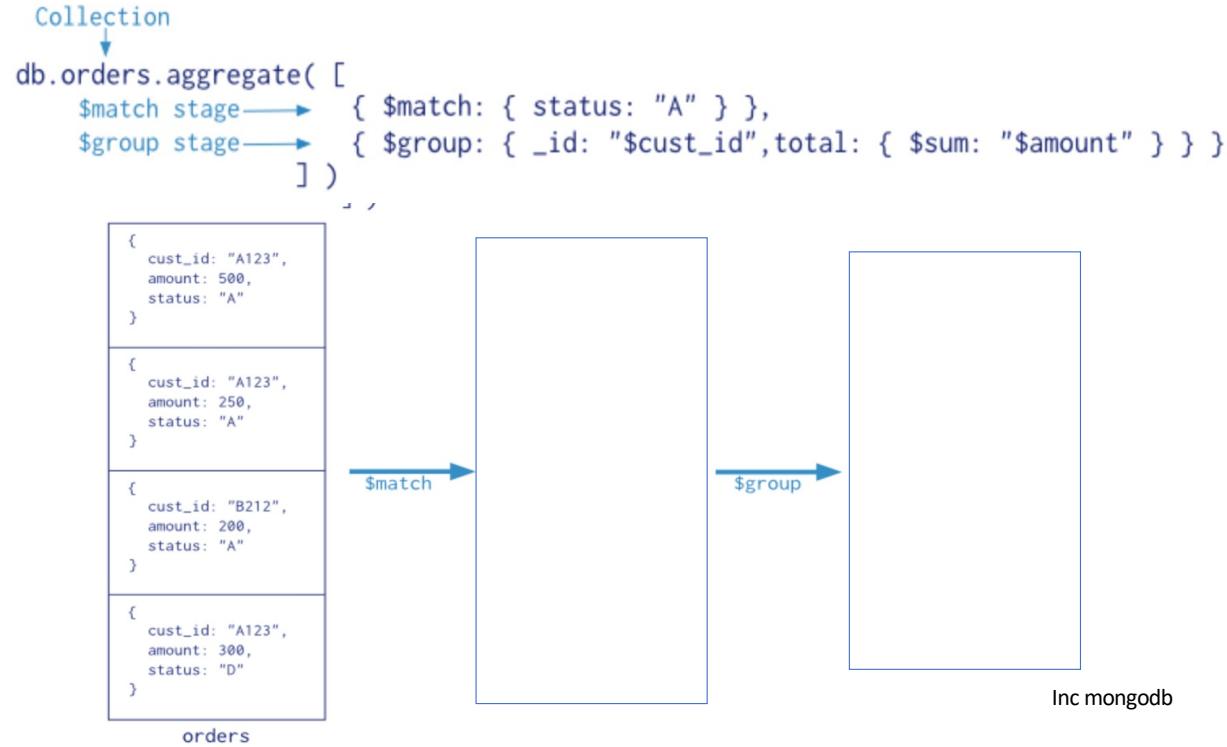
Aggregation framework



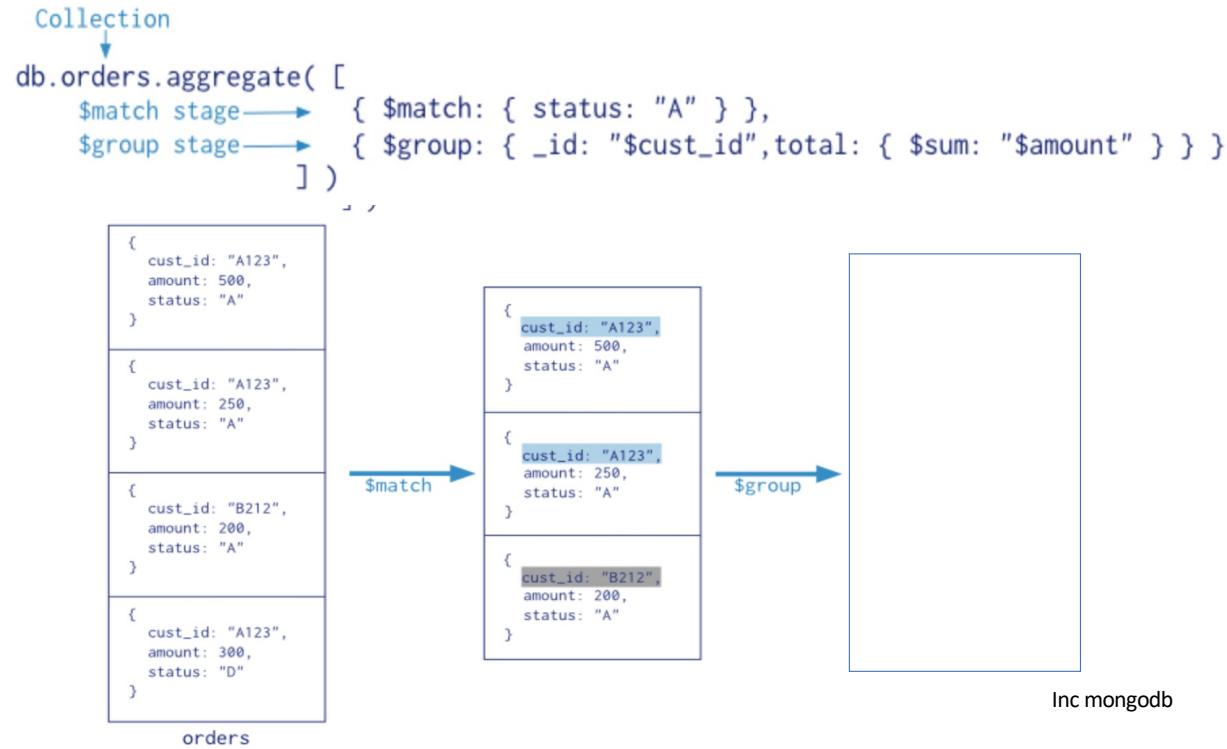
Aggregation framework



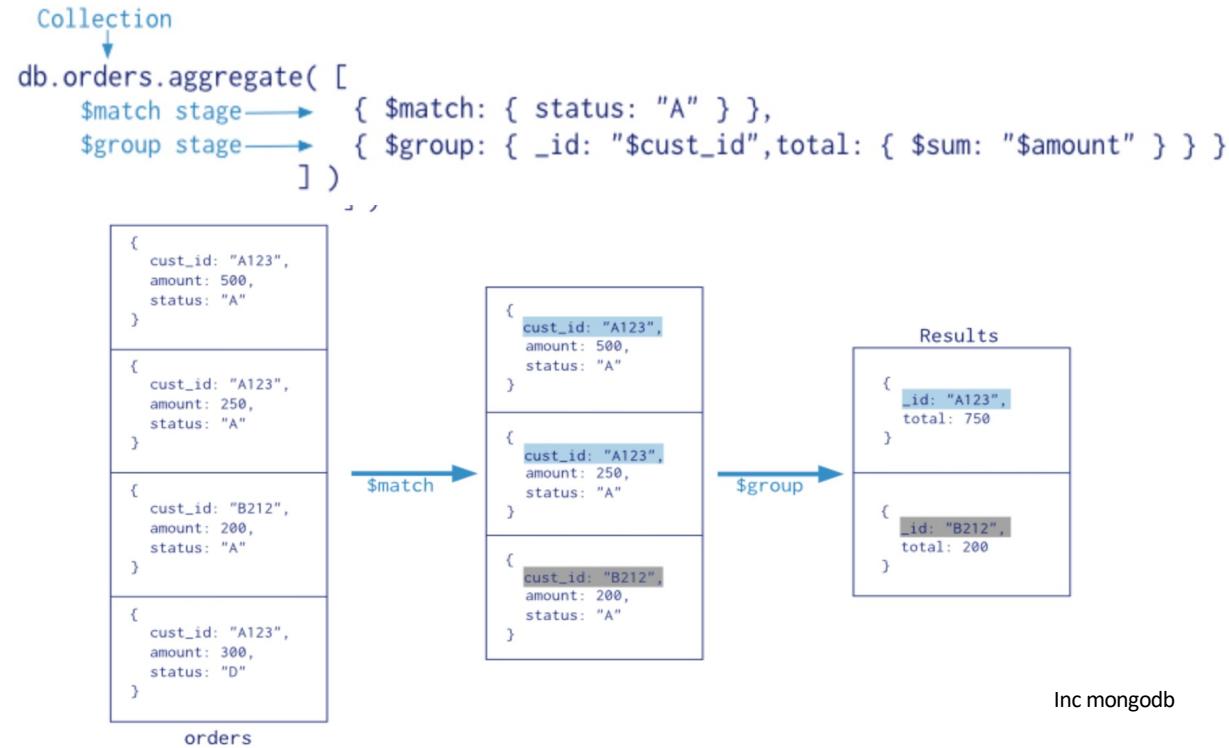
Aggregation framework



Aggregation framework



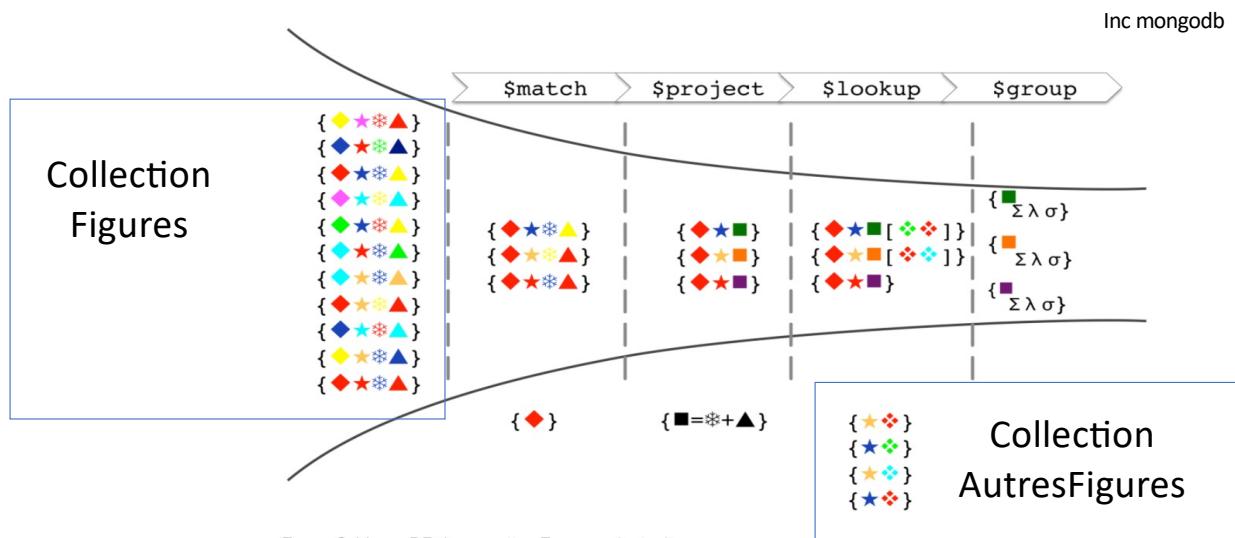
Aggregation framework



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 :

```
magasins:{  
    bsonType: ["array"],  
    description : "magasins est un tableau non obligatoire et si il existe doit contenir au moins 2 valeurs quelconque 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.



```

{
  required : [ "nom", "products"],
  properties: {
    nom: {
      bsonType: "string",
      description : "nom est obligatoire et doit être une string"
    },
    counter: {
      bsonType:"int",
      description : "counter est un type entier dont la valeur doit être comprise en tres 0 et 100 000 et est obligatoire ",
      minimum: 0,
      maximum: 100000
    },
    productid: {
      oneOf:
      [
        {
          bsonType: "string",
          description : "le champs products est soit un string ou un entier dans la valeur est obligatoire "
        },
        {
          bsonType: "int"
        }
      ]
    },
    state: {
      enum: ["AL","NY"],
      description : "state est un champs non obligatoire de type enum avec pour valeur uniquement AL ou NY"
    },
    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}[- /.]([0-9]{1}[0-9])[- /.]([0-9]{1}[0-9])|[12][0-9]{3}[01])"
    },
    loc:{ 
      bsonType: "object",
      description : "loc est un objet (un document) non obligatoire et si il existe doit comporter un champs x et y de type double",
      required: ["x","y"],
      properties:
      {
        x : {bsonType: "double"},
        y : {bsonType: "double"}
      }
    },
    magasins:{
      bsonType: ["array"],
      description : "magasins est un tableau non obligatoire et si il existe doit contenir au moins 2 valeurs quelconque mais non-redondantes",
      minItems: 2,
      uniqueItems: true
    },
    couleurs:{
      bsonType: ["array"],
      description : "magasins est un tableau non obligatoire et si il existe 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 : exemple

```

{
  required : [ "nom", "products" ],
  properties : {
    nom : {
      bsonType: "string",
      description : "nom est obligatoire et doit etre une string"
    },
    counter : {
      bsonType:"int",
      description : "counter est un type entier dont la valeur doit etre comprise entre 0 et 100 000 et est obligatoire",
      minimum: 0,
      maximum: 100000
    },
    productid: {
      oneOf:
      [
        {
          bsonType: "string",
          description : "le champs products est soit un string ou un entier dans la valeur est obligatoire"
        },
        {
          bsonType: "int"
        }
      ]
    },
    state: {
      enum: ["AL","NY"],
      description : "state est un champs non obligatoire de type enum avec pour valeur uniquement AL ou NY"
    },
    lastmodification : {
      bsonType : "string",
      description : "lastmodification est un champs non obligatoire de type string avec un format de date respectant le format indique dans pattern (ex: 2009-03-20 / 2010.12.20 ... )",
      pattern : "(19|20)[0-9]{2}[- .][0-9]{1}[1|0|12][- .][0-9]{1}[1|2|0-9|3|012]"
    },
    loc:{ 
      bsonType: "object",
      description : "loc est un objet (un document) non obligatoire et si il existe doit comporter un champs x et y de type double",
      required: ["x","y"],
      properties:{ 
        x : {bsonType: "double"}, 
        y : {bsonType: "double"} 
      }
    },
    magasins:{ 
      bsonType: "[array]",
      description : "magasins est un tableau non obligatoire et si il existe doit contenir au moins 2 valeurs qui sont des documents non-redondantes",
      minItems: 2,
      uniqueItems: true
    },
    couleurs:{ 
      bsonType: "[array]",
      description : "magasins est un tableau non obligatoire et si il existe doit contenir au moins 1 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"
          }
        ]
      }
    }
  }
}

```

Définition des clés obligatoires et non-obligatoires

Définition de la clé nom

Garfield



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 !!!

Osman AIDEL / MongoDB

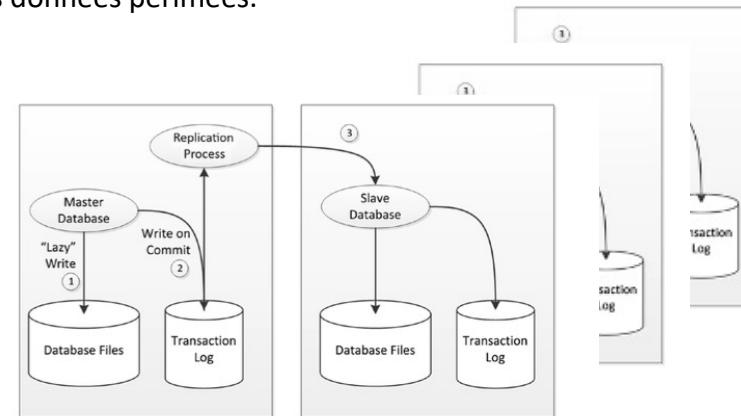


La réPLICATION avec MongoDB

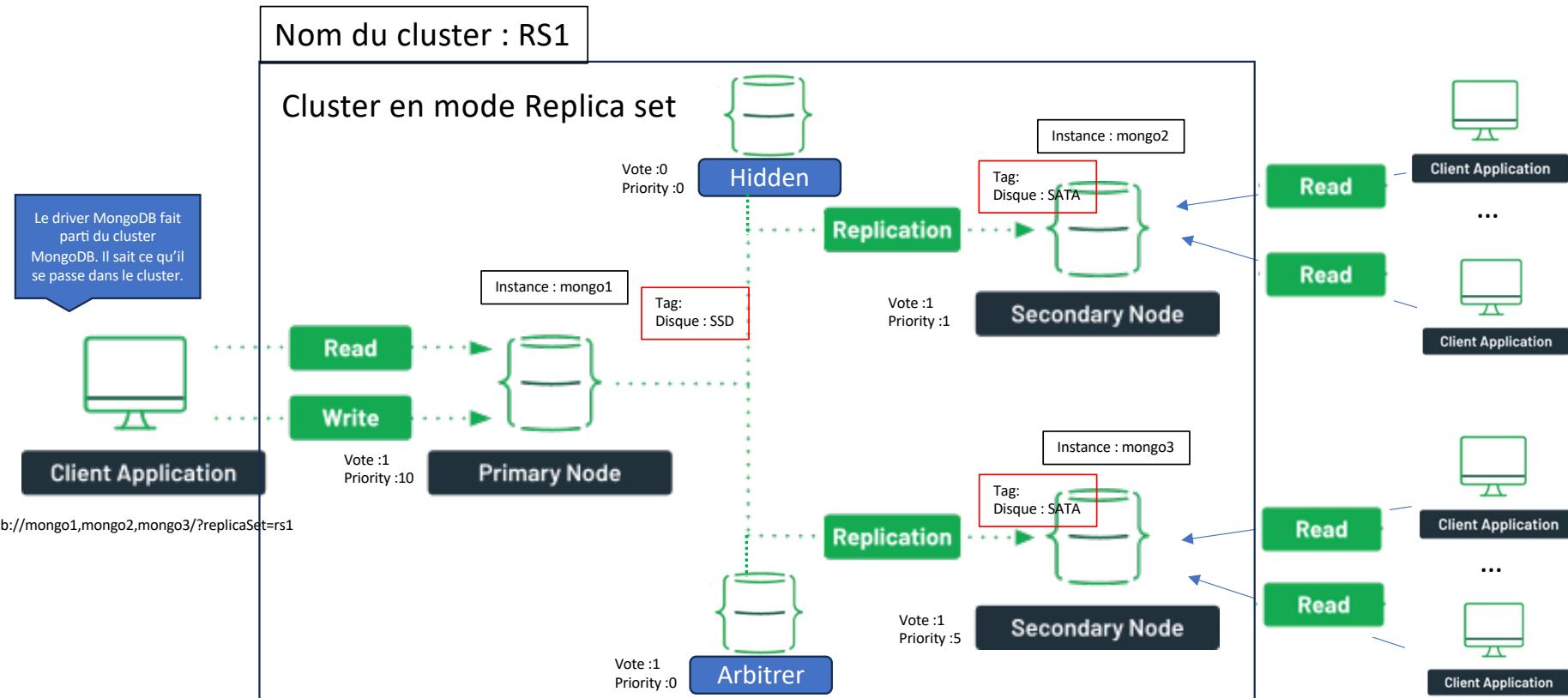


La réPLICATION avec MongoDB

- 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.

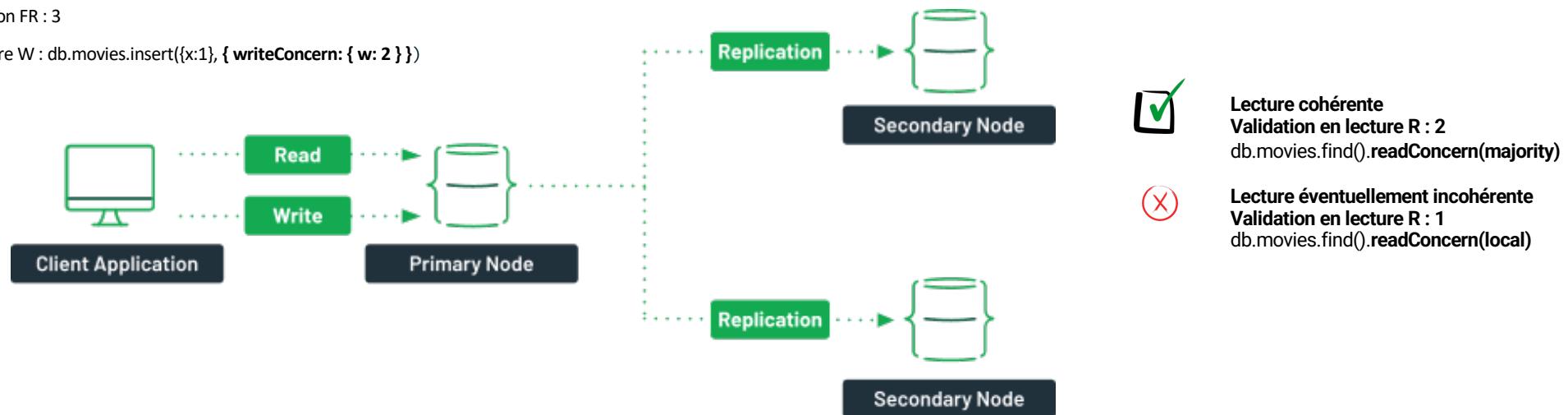
La réPLICATION avec MongoDB

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

