

NoSQL, systèmes distribués et passage en production de projets Data

Thierry GAMEIRO MARTINS

Séances

1. Introduction et prise en main d'Onyxia

2. Le stockage des données en NoSQL

3. Les systèmes de traitement distribués

4. Le passage en production

5. Orchestration et pratique DevOps

6. Déploiement conteneurisé sous Kubernetes

MongoDB

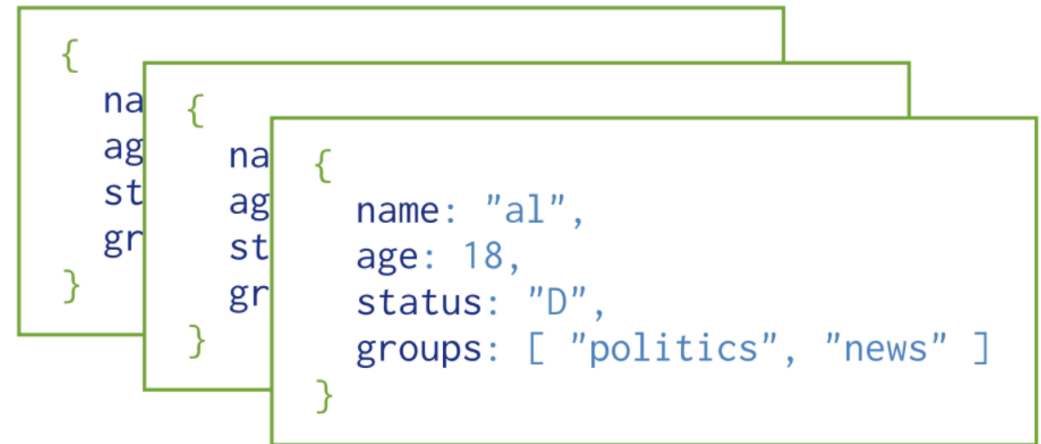
- Stocke les données sous forme de documents **JSON** (semi-structuré) dans des collections
- Modèle flexible sans schéma prédéfini
- Scalabilité horizontale
- Stockage compact **Binary JSON** (**BSON**)

Quand l'utiliser ?

- Applications nécessitant une flexibilité des schémas
- Applications en temps réel nécessitant des opérations rapides

Connexion avec le client Python

```
from pymongo import MongoClient  
  
client = MongoClient('mongodb', 27017)  
  
db = client['mydatabase']
```



Collection

La dénormalisation des données

Les données métiers sont souvent stockées dans un format relationnel ou en `csv`

Principe de la normalisation

- Évite la duplication de données
- Assure une meilleure fiabilité de la donnée

⚠ À éviter dans une base document

orders

order_id	name	store_id
0	alex	0
1	alice	1
2	tom	0

store

store_id	store_city	store_size
0	Paris	Small
1	Marseille	Large

Dénormalisation

Combiner l'ensemble des données au sein d'un même document plutôt que d'utiliser des jointures entre documents

Pourquoi ?

- Stockage à faible coût aujourd'hui
- Référence à plusieurs documents augmente le temps de réponse
- Simplification des requêtes

orders

```
{
  "_id": 1,
  "name": "alex"
  "store": {
    {"store_id": 0, "store_city": "Paris", "store_size": "Small"}
  }
}
```

- **Imbrication de documents** : idéal pour les relations 1-1
- **Référence partielle** : répliquer une partie des informations couramment utilisée et référencer les détails via un id car :
 - **risque des modifications** : coûteux si une modification implique de modifier tous les documents
 - **vérifier la taille des document** : dans le cas de relation 1-n, peut alourdir la taille du document (max 16Mo)

```
{
  "_id": ObjectId("6149eae5a1c2f1001e5e8a23"),
  "title": "How to scale MongoDB",
  "author": {
    "name": "John Doe",
    "job": "Data Scientist",
    "user_id": ObjectId("6149eae5a1c2f1001e5e8b24")
  },
  "address": {
    "city": "Paris",
    "state": "France",
    "zip": "75014"
  },
  "comments": [
    {
      "comment_id": ObjectId("6149eae5a1c2f1001e5e8a25"),
      "text": "Great article!",
      "commenter": {
        "name": "Jane Smith",
        "user_id": ObjectId("6149eae5a1c2f1001e5e8b26")
      },
      "created_at": "2023-09-21T14:15:22Z"
    },
    {
      "comment_id": ObjectId("6149eae5a1c2f1001e5e8a26"),
      "text": "I found it very helpful!",
      "commenter": {
        "name": "Alice Brown",
        "user_id": ObjectId("6149eae5a1c2f1001e5e8b27")
      },
      "created_at": "2023-09-21T14:20:15Z"
    }
  ]
}
```

Les types de données MongoDB

MongoDB stocke les données sous le format **BS0N** qui permet de stocker davantage d'information que le format **JS0N**

Le type ObjectID

ID généré par MongoDB est généré à partir de plusieurs facteurs :

- La date d'insertion au sens Unix
- Un nombre aléatoire
- Un compteur par rapport à la valeur insérée

D'autres types :

- **Date** afin de manipuler des dates (Timestamp)
- **Entiers** afin de gérer l'espace de stockage (Long, Int32, Decimal128, etc.)

Insertion de données

Ajouter un document

```
db.collection.insertOne({"name": "Alice", "age": 25})
```

Insérer plusieurs documents

```
db.collection.insertMany([{"name": "Bob", "age": 30},  
{"name": "Charlie", "age": 35}], { ordered: false })
```

`ordered` : n'impose pas d'ordre à l'insertion et tente l'ensemble

`writeConcern` : gestion de la vérification de l'écriture (`majority` , `0` , `1`)

Utilisation de `mongoimport` pour de l'import en masse (fichiers CSV, JSON, etc.).

```
mongoimport --db mydatabase  
--collection mycollection  
--file data.json --jsonArray
```

- Très efficace pour de l'import massif
- Supporte différents formats (JSON, CSV)

MongoDB crée automatiquement les collections lors de la première insertion

Recherche de données

Trouver un document

```
db.collection.findOne({ "name": "Alice" })
```

Trouver plusieurs documents

```
db.collection.find({ "age": { "gt": 30 } })
```

Filtrage

Valeur exacte : { "name": "Alice" }

Sous-clé : { "address.city": "Metz" }

Regex : { "name": { \$regex: "^A" } }

Paramètres

- **Premier paramètre** : critère de recherche
- **Deuxième paramètre** : projection (champ à inclure/exclure) dans la sortie

```
db.collection.find(  
  { "age": { $gt: 30 } }, // critère  
  { "name": 1, "age": 1 } // projection  
)
```

{ "_id": 0 } pour exclure l'id

Comparaisons

- `$gt` et `$lt` : supérieur ou inférieur
`{ "age": { $gt: 30 } }`
- `$gte` et `$lte` : supérieur/inférieur ou égal à
`{ "age": { $gte: 30 } }`
- `$and` ou `$or` : combinaison
`{"age": {"and": [{"$lte": 30}, {"$gte": 40}] }}`
- `$ne` : non égal à
`{ "age": { $ne: 25 } }`

Inclusions/exclusions

- `$in` : inclus dans une liste
`{ "status": { $in: ["Active", "Pending"] } }`
- `$nin` : non inclus dans une liste
`{ "status": { $nin: ["Banned"] } }`
- `$exists` : si la clé existe
`{ "age": { $exists: 1 } }`

Suppression de données

- `deleteOne` : supprime un seul document correspondant au critère
- `deleteMany` : supprime tous les documents correspondant au critère
- Pour supprimer tous les documents d'une collection

```
db.collection.deleteOne({ name: "Alice" })
```

```
db.collection.deleteMany({ age: { $lt: 30 } })
```

```
db.collection.deleteMany({})
```

 Vérifier le critère avant toute suppression

Mise à jour des données

- `updateOne` : met à jour un document correspondant au critère
- `updateMany` : met à jour tous les documents correspondants au critère (ou tous)
- `replaceOne` : remplace entièrement un document correspondant au critère
- `replaceMany` : remplace entièrement tous les documents correspondants au critère

- **Premier paramètre** : critère de recherche (quel document modifier)
- **Second paramètre** : transformation à appliquer (quelles modifications apporter)

```
db.customers.updateOne(  
  { "name": "Alex" },  
  { $set: { "email": "newemail@example.com" } }  
)
```

`$set` : définit une nouvelle valeur pour un champ spécifique

```
{ $set: { "email": "updated_email@example.com" } }
```

`$inc` : incrémente ou décrémente une valeur numérique

```
{ $inc: { "age": 1 } } // Ajoute 1 à l'âge
```

`$rename` : renomme une clé existante

```
{ $rename: { "oldFieldName": "newFieldName" } }
```

`$unset` : supprime un champ

```
{ $unset: { "address": "" } } // Supprime le champ
```

`$mul` : multiplie la valeur par un nombre spécifié

```
{ $mul: { "salary": 1.1 } } // Augmente de 10%
```

le paramètre optionnel `{ upsert: true }` permet de créer un nouveau document si aucun n'est trouvé avec le critère de recherche

Pipeline de données

- Ensemble d'étapes qui traitent les documents
- Chaque étape passe les résultats à la suivante

```
db.orders.aggregate([  
  { $match: { status: "completed" } },  
  { $group: { _id: "$customerId", totalAmount: { $sum: "$amount" } } }  
]);
```

- `merge` : permet d'écrire le résultat dans un document (toujours en dernière étape)

```
{ $merge: { into: "myOutput", on: "_id",  
  whenMatched: "replace",  
  whenNotMatched: "insert" } }
```

- `$match` : permet de filtrer les documents afin de les passer à l'étape suivante
- `$project` : permet de sélectionner les champs à passer dans la prochaine étape d'une pipeline
- `$group` : permet de regrouper des données et d'y appliquer une expression à chaque groupe

D'autres étapes sont disponibles [ici](#)

- `unwind` : applatit une liste à afin de produire une clé pour chaque élément de la liste
- `$lookup` : permet de faire correspondre des colonnes d'un document avec la colonne d'un autre document

inventory

```
{ prodId: 100, price: 20, quantity: 125 },
{ prodId: 101, price: 10, quantity: 234 },
{ prodId: 101, price: 10, quantity: 20 },
{ prodId: 102, price: 15, quantity: 432 }
```

orders

```
{ orderId: 201, custid: 301, prodId: 100, numPurchased: 20 },
{ orderId: 202, custid: 302, prodId: 101, numPurchased: 10 },
{ orderId: 203, custid: 303, prodId: 102, numPurchased: 5 },
```

Sur la table `orders` la pipeline suivante :

```
{ $lookup:
  {
    from: "inventory",
    localField: "prodId",
    foreignField: "prodId",
    as: "inventoryDocs"
  }
}
```

```
{ '_id': ObjectId('6714d5fc313563a5c86e36cd'),
  'custid': 301,
  'inventoryDocs': [{ '_id': ObjectId('6714d5fa313563a5c86e36c9'),
    'price': 20,
    'prodId': 100,
    'quantity': 125}],
  'numPurchased': 20,
  'orderId': 201,
  'prodId': 100 }
{ '_id': ObjectId('6714d5fc313563a5c86e36ce'),
  'custid': 302,
  'inventoryDocs': [{ '_id': ObjectId('6714d5fa313563a5c86e36ca'),
    'price': 10,
    'prodId': 101,
    'quantity': 234 },
    { '_id': ObjectId('6714d5fa313563a5c86e36cb'),
    'price': 10,
    'prodId': 101,
    'quantity': 20 } ],
  'numPurchased': 10,
  'orderId': 202,
  'prodId': 101 }
```


Comparaison SQL et MongoDB



- Une page de la documentation est dédiée à la comparaison aux équivalences entre MongoDB et le SQL :

<https://www.mongodb.com/docs/manual/reference/sql-comparison/>

<pre>SELECT * FROM people</pre>	<pre>db.people.find()</pre>
<pre>SELECT id, user_id, status FROM people</pre>	<pre>db.people.find({ }, { user_id: 1, status: 1 })</pre>
<pre>SELECT user_id, status FROM people</pre>	<pre>db.people.find({ }, { user_id: 1, status: 1, _id: 0 })</pre>
<pre>SELECT * FROM people WHERE status = "A"</pre>	<pre>db.people.find({ status: "A" })</pre>
<pre>SELECT user_id, status FROM people WHERE status = "A"</pre>	<pre>db.people.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })</pre>
<pre>SELECT * FROM people WHERE status != "A"</pre>	<pre>db.people.find({ status: { \$ne: "A" } })</pre>
<pre>SELECT * FROM people WHERE status = "A" AND age = 50</pre>	<pre>db.people.find({ status: "A", age: 50 })</pre>

Optimisation des performances : indexation

Un index permet d'éviter de scanner l'ensemble des documents lors d'une recherche

-  Améliore le temps de lecture
-  Diminue le temps d'écriture

Lister les index

```
db.collection.getIndexes()
```

Supprimer un index

```
db.collection.dropIndex("indexName")
```

Plusieurs types d'index :

- Index par défaut (`_id`)
- Index simple (sur un seul champ)

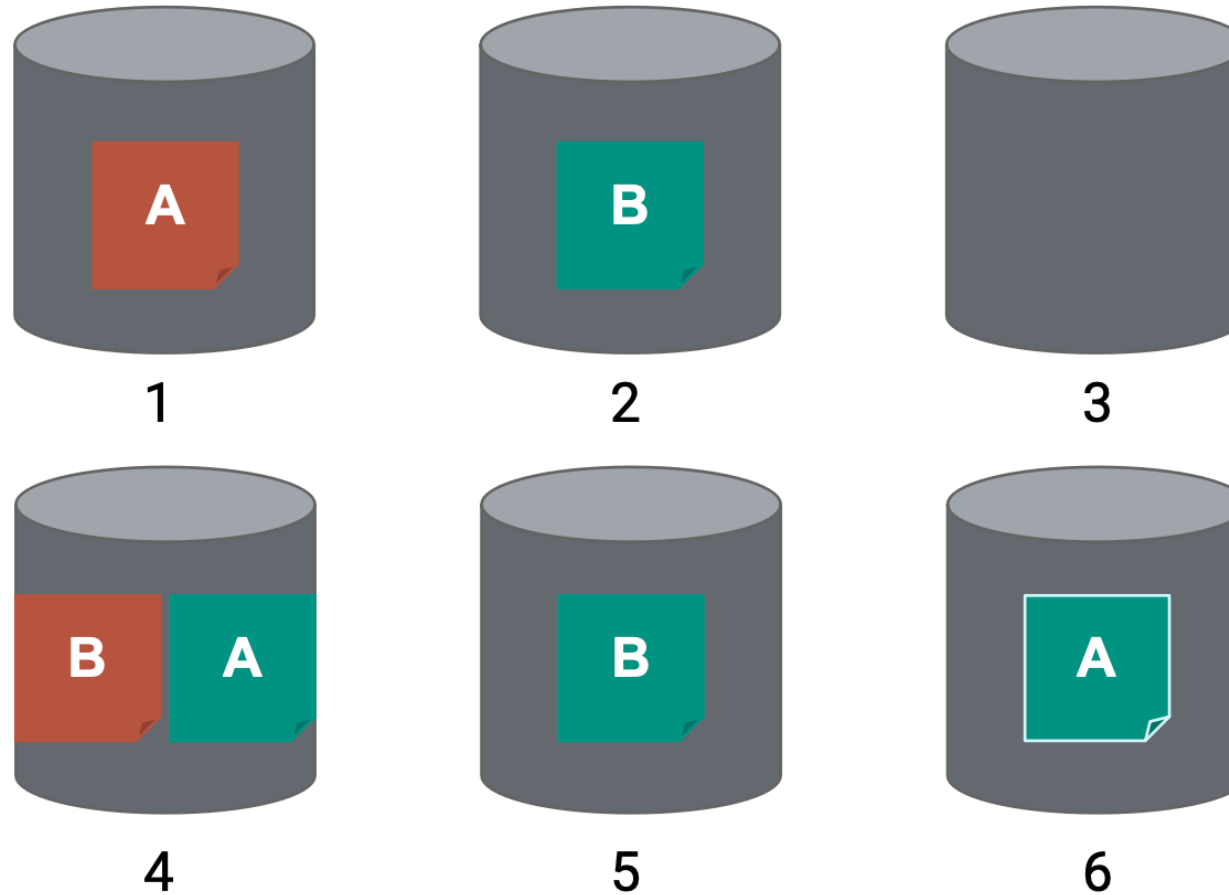
```
db.collection.createIndex({  
  field: 1  
})
```
- Index composés (plusieurs champs)

```
db.collection.createIndex({  
  field1: 1, field2: -1  
})
```

 Vérifier que votre index tient en RAM

```
db.collection.totalIndexSize()
```

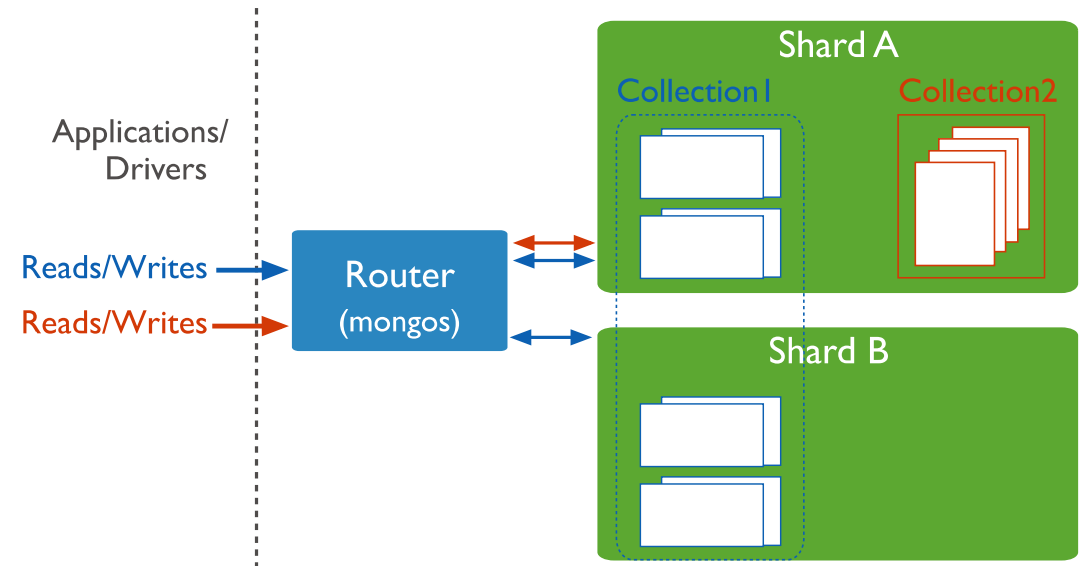
Concept : Réplication vs Partitionnement



Optimisation des performances : sharding

Principe de partitionnement horizontal
pour distribuer les données à travers
plusieurs serveurs

- `shards` : chaque serveur `shard` contient une partie des données (`chunk`).
- `mongos` : agit comme un routeur de requêtes selon les données et les configurations de préférences



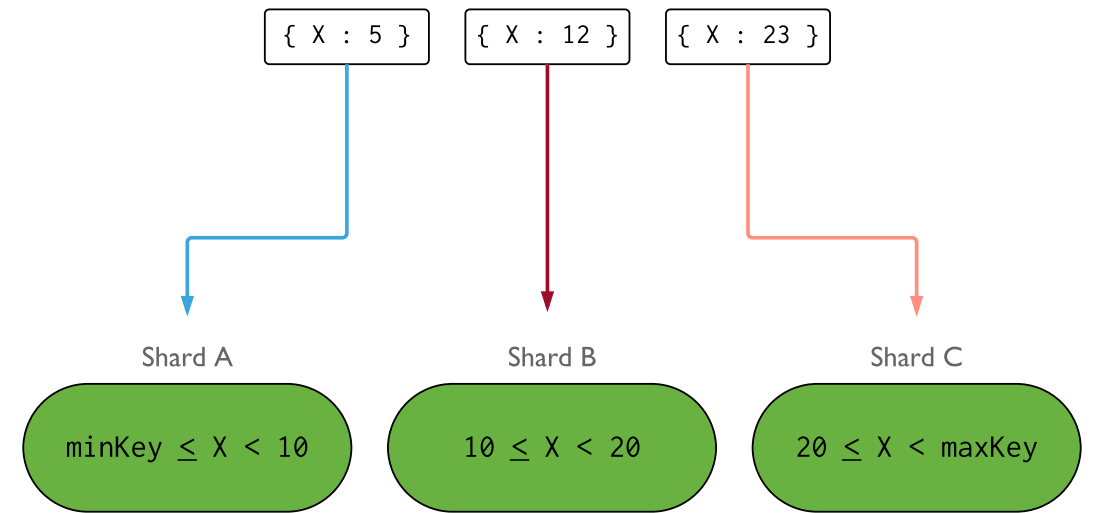
```
sh.enableSharding("mydb")
```

Shard Key : clé de répartition de la donnée

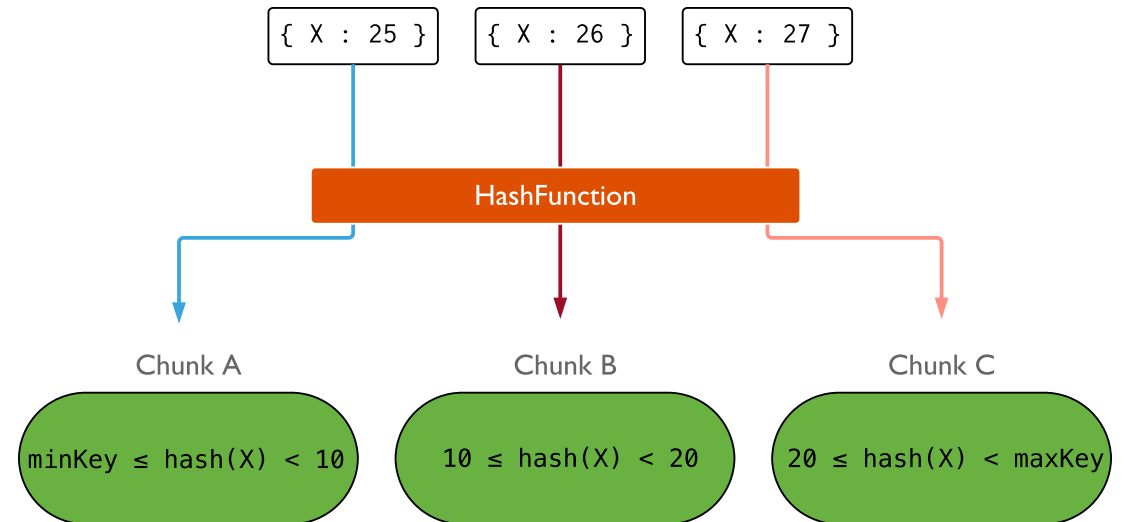
- Cardinalité large (nombre de **chunks**)
- Fréquence faible (uniformément répartie)

Deux types de *sharding* :

- **Hashed sharding** : fonction appliquée à la clé
 - Accès aléatoires
 - Scalabilité plus simple
- **Ranged sharding** : organisation par plage de valeurs
 - ⚠ déséquilibre possible
 - Idéal pour les accès par plage de valeurs (*logs, transactions*)



```
sh.shardCollection("mydb.collection2", { shardKey: "hashed" })
```



```
sh.shardCollection("mydb.collection", { shardKey: 1 })
```

- **Replica Sets** : ensemble de shards qui maintiennent la même données (pour la redondance et disponibilité)
 - Serveur primaire (écriture) et secondaire (lecture)
 - Réplication asynchrone
 - Principe d'élection avec *l'arbiter* (externe) et *Read Preference*
- **Balancer** : processus qui gère la répartition des chunks entre les shards

