MongoDB Révision

1. Introduction à MongoDB :

MongoDB est une base de données NoSQL orientée document, conçue pour stocker et gérer des données semi-structurées et non structurées. Elle est largement utilisée pour le développement d'applications web, mobiles et IoT.

2. Les caractéristiques de MongoDB

- **Schéma flexible** : MongoDB ne nécessite pas de schéma rigide pour les données, ce qui signifie que les développeurs peuvent stocker des données semi-structurées ou non structurées dans la base de données.
- **Langage de requête** : MongoDB utilise une syntaxe de requête similaire à celle du langage de programmation JSON pour faciliter les interactions avec la base de données.
- **Évolutivité** : MongoDB est conçu pour être évolutif, ce qui signifie qu'il peut être utilisé pour stocker et gérer de grandes quantités de données. Il est également conçu pour être réparti sur plusieurs serveurs.
- **Haute disponibilité** : MongoDB est conçu pour être résilient aux pannes grâce à sa capacité à répliquer des données sur plusieurs serveurs.
- **Performance** : MongoDB est rapide et efficace grâce à son architecture interne et à sa capacité à gérer de grandes quantités de données.
- **Indexation** : MongoDB offre des options d'indexation pour accélérer la recherche et l'accès aux données.
- **Sécurité** : MongoDB prend en charge les fonctionnalités de sécurité, telles que l'authentification et le chiffrement des données.

3. Commandes de gestion des collections :

Voici quelques exemples de commandes d'exécution de MongoDB :

- Démarrage de MongoDB :

Pour démarrer MongoDB, vous devez exécuter la commande **mongod** en ligne de commande. Voici un exemple de commande pour démarrer un serveur MongoDB sur le port 27017 :

```
mongod --port 27017 --dbpath /data/db
```

- Connexion à une base de données MongoDB :

Pour vous connecter à une base de données MongoDB, vous pouvez utiliser la commande **mongo** en ligne de commande. Voici un exemple de commande pour vous connecter à une base de données locale :

```
mongo
```

- Sélection d'une base de données :

```
use myDatabase
```

Voici quelques exemples de gestion et de création de collections dans MongoDB :

- Création d'une collection : **db.createCollection()**

```
db.createCollection("maCollection")
```

- Suppression d'une collection : **db.collection.drop()**

```
db.maCollection.drop()
```

- Renommage d'une collection : **db.collection.renameCollection()**

```
db.maCollection.renameCollection("nouveauNomCollection")
```

- Obtention de la liste des collections dans une base de données : **db.getCollectionNames()**

```
db.getCollectionNames()
```

- Obtention d'une référence à une collection : **db.collection**

  Cet exemple obtient une référence à la collection **maCollection**. La variable **maCollection** peut ensuite être utilisée pour effectuer des opérations sur la collection, comme l'insertion, la recherche, la mise à jour et la suppression de documents.

```
var maCollection = db.getCollection("maCollection")
```

4. Opérations de manipulation de données :

- **Insertion** : L'opération d'insertion permet d'ajouter des données à une collection MongoDB. Elle peut être effectuée avec la méthode insertOne() pour insérer un document unique, ou insertMany() pour insérer plusieurs documents.
- **Recherche** : L'opération de recherche permet de récupérer des données à partir d'une collection MongoDB. Elle peut être effectuée avec la méthode find() et peut être filtrée à l'aide de critères de recherche.
- **Mise à jour** : L'opération de mise à jour permet de modifier des données existantes dans une collection MongoDB. Elle peut être effectuée avec la méthode updateOne() pour mettre à jour un seul document, ou updateMany() pour mettre à jour plusieurs documents.
- **Suppression** : L'opération de suppression permet de supprimer des données d'une collection MongoDB. Elle peut être effectuée avec la méthode deleteOne() pour supprimer un seul document, ou deleteMany() pour supprimer plusieurs documents.
- **Agrégation** : L'opération d'agrégation permet de traiter et de regrouper des données de plusieurs documents dans une collection MongoDB. Elle peut être effectuée avec la méthode aggregate() et peut être utilisée pour effectuer des calculs complexes sur les données.
- **Indexation** : L'opération d'indexation permet de créer des index sur les données d'une collection MongoDB. Cela permet d'accélérer les opérations de recherche et de tri sur les données.
- **Transactions** : Les transactions permettent de gérer plusieurs opérations de manière atomique dans MongoDB. Cela garantit que toutes les opérations sont exécutées avec succès ou qu'elles sont toutes annulées si une opération échoue.

Voici quelques exemples d'opérations d'insertion de données dans MongoDB à l'aide de la méthode **insertOne()** :

- Exemple 1 : Insertion d'un document unique dans une collection :

```
db.maCollection.insertOne({
  nom: "Dupont",
  prenom: "Jean",
  age: 30,
  ville: "Paris"
})
```

- Exemple 2 : Insertion d'un document avec un champ tableau dans une collection :

```
db.maCollection.insertOne({
  nom: "Durand",
  prenom: "Marie",
  age: 25,
  hobbies: ["lecture", "cinéma", "voyage"]
})
```

- Exemple 3 : Insertion d'un document avec des champs imbriqués dans une collection :

```
db.maCollection.insertOne({
  nom: "Martin",
  prenom: "Pierre",
  age: 35,
  adresse: {
    rue: "Rue de la Paix",
    ville: "Lyon",
    pays: "France"
  }
})
```

- Exemple 4 : Insertion de plusieurs documents dans une collection :

```
db.maCollection.insertMany([
  { nom: "Dubois", prenom: "Lucie", age: 28, ville: "Toulouse" },
  { nom: "Moreau", prenom: "Paul", age: 40, ville: "Marseille" },
  { nom: "Petit", prenom: "Sophie", age: 22, ville: "Bordeaux" }
])
```

Voici quelques exemples d'opérations de recherche de données dans MongoDB à l'aide de la méthode find() :

- Exemple 1 : Recherche de tous les documents dans une collection :

```
db.maCollection.find()
```

- Exemple 2 : Recherche de documents avec un critère de recherche :

```
db.maCollection.find({ ville: "Paris" })
```

- Exemple 3 : Recherche de documents avec une opération de comparaison :

```
db.maCollection.find({ age: { $gt: 30 } })
```

- Exemple 4 : Recherche de documents avec une opération logique :

```
db.maCollection.find({ $or: [{ ville: "Paris" }, { ville: "Marseille" }] })
```

- Exemple 5 : Recherche de documents avec un champ imbriqué :

```
db.maCollection.find({ "adresse.ville": "Lyon" })
```

- Exemple 6 : Recherche de documents avec une projection de champs :

```
db.maCollection.find({ ville: "Paris" }, { nom: 1, prenom: 1 })
```

Voici une liste de types d'opérateurs couramment utilisés dans MongoDB :

1. Opérateurs de comparaison : **$eq**, **$ne**, **$gt**, **$gte**, **$lt**, **$lte**, **$in**, **$nin**.
2. Opérateurs logiques : **$and**, **$or**, **$not**, **$nor**.
3. Opérateurs de projection : **$project**, **$slice**, **$elemMatch**.
4. Opérateurs de regroupement : **$group**, **$sum**, **$avg**, **$min**, **$max**.
5. Opérateurs de mise à jour : **$set**, **$unset**, **$inc**, **$push**, **$pull**, **$addToSet**.
6. Opérateurs de présence : **$exists**.
7. Opérateurs de type de données : **$type**.
8. Opérateurs d'expression régulière : **$regex**.
9. Opérateurs de tous les éléments : **$all**.
10. Opérateurs de correspondance d'élément : **$elemMatch**.
11. Opérateurs de taille de tableau : **$size**.
12. Opérateurs de tri : **$sort**.

- Opérateur d'égalité : **$eq**

```
db.maCollection.find({ ville: { $eq: "Paris" } })
```

- Opérateur de comparaison : **$gt**, **$gte**, **$lt**, **$lte**

```
db.maCollection.find({ age: { $gt: 30 } })
```

- Opérateur d'appartenance : **$in**

```
db.maCollection.find({ ville: { $in: ["Paris", "Marseille"] } })
```

- L'opérateur **$nin** (not in) permet de sélectionner les documents où la valeur d'un champ n'est pas dans une liste de valeurs données

```
db.users.find({ age: { $nin: [25, 30, 35] } })
```

- Opérateur de négation : **$ne**

```
db.maCollection.find({ ville: { $ne: "Paris" } })
```

- Opérateur logique : **$and**, **$or**

```
db.maCollection.find({ $and: [{ ville: "Paris" }, { age: { $gt: 30 } }] })
db.maCollection.find({ $or: [{ ville: "Paris" }, { ville: "Marseille" }] })
```

- L'opérateur **$not** est utilisé pour sélectionner les documents où une condition n'est pas vraie.

```
db.products.find({ price: { $not: { $gt: 100 } } })
```

- L'opérateur **$nor** (not or) est utilisé pour sélectionner les documents où aucune des conditions données n'est vraie.

```
db.users.find({ $nor: [ { age: 25 }, { age: 30 }, { age: 35 } ] })
```

- Opérateur de texte : **$text**

```
db.maCollection.find({ $text: { $search: "foo" } })
```

- Opérateur de regex : **$regex**

```
db.maCollection.find({ nom: { $regex: /^D/ } })
```

- **$exists** : opérateur de présence

```
db.maCollection.find({ adresse: { $exists: true } })
```

- **$type** : opérateur de type de données

```
db.maCollection.find({ age: { $type: "number" } })
```

- **$all** : opérateur de tous les éléments

```
db.maCollection.find({ loisirs: { $all: ["tennis", "natation"] } })
```

- **$elemMatch** : opérateur de correspondance d'élément

```
db.maCollection.find({ amis: { $elemMatch: { nom: "Paul", age: { $gt: 30 } } } })
```

- **$size** : opérateur de taille de tableau

```
db.maCollection.find({ loisirs: { $size: 3 } })
```

- **$sort** : opérateur de tri

```
db.maCollection.find().sort({ age: 1 })
```

- **$group** : permet de regrouper des documents en fonction d'une clé et d'effectuer des opérations sur les documents regroupés.

```
db.sales.aggregate([
  { $group: { _id: "$item", total: { $sum: "$price" } } }
])
```

- **$sum** : permet de calculer la somme d'un champ numérique pour les documents regroupés

```
db.sales.aggregate([
  { $group: { _id: "$item", totalSales: { $sum: "$qty" } } }
])
```

- **$avg** : permet de calculer la moyenne d'un champ numérique pour les documents regroupés.

```
db.sales.aggregate([
  { $group: { _id: "$item", averageSales: { $avg: "$qty" } } }
])
```

- **$min** : permet de trouver la valeur minimale d'un champ pour les documents regroupés.

```
db.sales.aggregate([
  { $group: { _id: "$item", minPrice: { $min: "$price" } } }
])
```

- **$max** : permet de trouver la valeur maximale d'un champ pour les documents regroupés.

```
db.sales.aggregate([
  { $group: { _id: "$item", maxPrice: { $max: "$price" } } }
])
```

- Opération de tri : **sort()**

```
db.maCollection.find().sort({ age: -1 })
```

- Opération de limitation : **limit()**

```
db.maCollection.find().limit(10)
```

- Opération de saut : **skip()**

```
db.maCollection.find().skip(5)
```

- Opération de projection : **projection()**

```
db.maCollection.find({ ville: "Paris" }).projection({ nom: 1, prenom: 1 })
```

- Opération de comptage : **count()**

```
db.maCollection.find({ ville: "Paris" }).count()
```

- Opération d'agrégation : **aggregate()**

Cet exemple retourne la somme de l'âge de chaque personne dans la collection **maCollection** où la valeur du champ **ville** est égale à "Paris", regroupée par nom.

```
db.maCollection.aggregate([
   { $match: { ville: "Paris" } },
   { $group: { _id: "$nom", total: { $sum: "$age" } } }
])
```

Voici quelques exemples de mises à jour couramment utilisées dans MongoDB :

- Mise à jour d'un champ : **updateOne()**

```
db.maCollection.updateOne({ _id: ObjectId("60e02063f79bae39d784f2c2") }, { $set: { age: 35 } })
```

- Mise à jour multiple : **updateMany()**

```
db.maCollection.updateMany({ ville: "Paris" }, { $set: { pays: "France" } })
```

- Opération d'incrémentation : **increment()**

```
db.maCollection.updateOne({ _id: ObjectId("60e02063f79bae39d784f2c2") }, { $inc: { age: 1 } })
```

- Mise à jour conditionnelle : **updateOne()** avec **filter**

Cet exemple met à jour le champ **ville** du premier document de la collection **maCollection** où le champ **nom** est égal à "Dupont" et le champ **age** est supérieur à 30.

```
db.maCollection.updateOne({ nom: "Dupont", age: { $gt: 30 } }, { $set: { ville: "Marseille" } })
```

- Mise à jour avec un tableau : **updateOne()** avec **$push**

Cet exemple ajoute "Martin" à la liste des amis du document de la collection **maCollection** où le champ **nom** est égal à "Dupont". Si le champ **amis** n'existe pas encore, il est créé.

```
db.maCollection.updateOne({ nom: "Dupont" }, { $push: { amis: "Martin" } })
```

Voici quelques exemples de suppressions couramment utilisées dans MongoDB :

- Suppression d'un document : **deleteOne()**

```
db.maCollection.deleteOne({ _id: ObjectId("60e02063f79bae39d784f2c2") })
```

- Suppression de plusieurs documents : **deleteMany()**

```
db.maCollection.deleteMany({ ville: "Paris" })
```

- Suppression de tous les documents : **deleteMany()** sans filtre

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

- Suppression d'un champ dans un document : **updateOne()** avec **$unset**

```
db.maCollection.updateOne({ _id: ObjectId("60e02063f79bae39d784f2c2") }, { $unset: { age: "" } })
```

- Suppression d'un élément d'un tableau dans un document : **updateOne()** avec **$pull**

```
db.maCollection.updateOne({ nom: "Dupont" }, { $pull: { amis: "Martin" } })
```

Voici quelques exemples d'opérations d'agrégation dans MongoDB :

- Calculer la somme d'un champ :

```
db.orders.aggregate([
  { $group: { _id: null, total: { $sum: "$amount" } } }
])
```

- Calculer la moyenne d'un champ :

```
db.orders.aggregate([
  { $group: { _id: null, average: { $avg: "$amount" } } }
])
```

- Regrouper les résultats par champ :

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

- Rechercher des documents correspondant à un critère, puis les regrouper :

```
db.orders.aggregate([
  { $match: { status: "completed" } },
  { $group: { _id: "$product", count: { $sum: 1 } } }
])
```

- Utiliser la pipeline d'agrégation pour effectuer plusieurs opérations :

```
db.orders.aggregate([
  { $match: { status: "completed" } },
  { $group: { _id: "$product", total: { $sum: "$amount" } } },
  { $sort: { total: -1 } },
  { $limit: 10 }
])
```

Voici quelques exemples d'indexation dans MongoDB :

- Index simple sur un champ :

```
db.orders.createIndex({ customer_id: 1 })
```

- Index composé sur plusieurs champs :

```
db.orders.createIndex({ customer_id: 1, date: -1 })
```

- Index sur un champ de type tableau :

```
db.orders.createIndex({ "items.product_id": 1 })
```

- Index textuel pour effectuer des recherches de texte intégral :

```
db.articles.createIndex({ content: "text" })
```

- Index géospatial pour effectuer des requêtes géospatiales :

```
db.stores.createIndex({ location: "2dsphere" })
```

- Index TTL (Time To Live) pour supprimer les documents après un certain temps :

```
db.sessions.createIndex({ created_at: 1 }, { expireAfterSeconds: 3600 })
```

- Index de hachage :

```
db.users.createIndex({ password: "hashed" })
```

- Index de texte intégral avancé :

```
db.articles.createIndex({ content: "text" }, { default_language: "french", weights: { title: 10, content: 5 } })
```

- Index de texte intégral avec poids :

```
db.articles.createIndex({ content: "text" }, { weights: { title: 10, content: 5 } })
```

- Index avec option unique :

```
db.users.createIndex({ email: 1 }, { unique: true })
```

- Index avec option sparse :

```
db.users.createIndex({ address: 1 }, { sparse: true })
```

Les transactions dans MongoDB permettent d'exécuter un ensemble d'opérations de manière cohérente, en garantissant que toutes les opérations réussissent ou échouent ensemble. Voici quelques exemples de transactions :

- Exécution d'une transaction simple :

```
session = db.getMongo().startSession();
session.startTransaction();
try {
  db.customers.updateOne(
    { _id: 1 },
    { $set: { status: "unpaid" }, $inc: { balance: -10 } },
    { session }
  );
  db.orders.insertOne(
    { customerId: 1, items: [{ name: "item1", quantity: 1, price: 10 }] },
    { session }
  );
  session.commitTransaction();
} catch (error) {
  session.abortTransaction();
  throw error;
}
```

Dans cet exemple, nous créons une session MongoDB, puis nous commençons une transaction avec **session.startTransaction()**. Nous effectuons ensuite deux opérations : nous mettons à jour un document dans la collection **customers**, puis nous insérons un document dans la collection **orders**. Nous utilisons l'option **session** pour garantir que les opérations sont effectuées dans la même session. Enfin, nous validons la transaction avec **session.commitTransaction()** si toutes les opérations ont réussi, sinon nous l'annulons avec **session.abortTransaction()**.

- Exécution d'une transaction avec plusieurs collections :

```
session = db.getMongo().startSession();
session.startTransaction();
try {
  db.users.updateOne(
    { _id: 1 },
    { $set: { name: "Alice" } },
    { session }
  );
  db.accounts.updateOne(
    { userId: 1 },
    { $inc: { balance: -50 } },
    { session }
  );
  session.commitTransaction();
} catch (error) {
  session.abortTransaction();
  throw error;
}
```

Dans cet exemple, nous mettons à jour deux collections différentes (**users** et **accounts**) dans la même transaction. Nous utilisons les options **session** pour garantir que les opérations sont effectuées dans la même session et qu'elles sont validées ou annulées ensemble.

- Exécution d'une transaction avec des documents imbriqués :

```
session = db.getMongo().startSession();
session.startTransaction();
try {
  db.orders.insertOne(
    {
      customerId: 1,
      items: [
        { name: "item1", quantity: 1, price: 10 },
        { name: "item2", quantity: 2, price: 20 }
      ],
      total: 50
    },
    { session }
  );
  db.customers.updateOne(
    { _id: 1 },
    { $inc: { balance: -50 } },
    { session }
  );
  session.commitTransaction();
} catch (error) {
  session.abortTransaction();
  throw error;
}
```

Dans cet exemple, nous insérons un document dans la collection **orders** qui contient un tableau de documents imbriqués **items**. Nous mettons également à jour un document dans la collection **customers**. Encore une fois, nous utilisons les options **session** pour garantir que les opérations sont effectuées dans la même session et qu'elles sont validées ou annulées ensemble.

Voici quelques exemples de mesures de sécurité que vous pouvez mettre en place dans MongoDB :

- Authentification des utilisateurs :

MongoDB offre plusieurs méthodes d'authentification pour protéger votre base de données contre les accès non autorisés. Vous pouvez créer des utilisateurs avec des noms d'utilisateur et des mots de passe, et leur accorder des rôles et des privilèges spécifiques. Voici un exemple de création d'un utilisateur :

```
use admin
db.createUser(
  {
    user: "myUser",
    pwd: "myPassword",
    roles: [ { role: "readWrite", db: "myDatabase" } ]
  }
)
```

- Autorisations de base de données :

Vous pouvez accorder des autorisations spécifiques pour chaque base de données de votre cluster. Par exemple, vous pouvez définir des rôles qui permettent de lire et d'écrire des données, de créer des index ou d'exécuter des commandes spécifiques. Voici un exemple de définition d'un rôle personnalisé :

```
use myDatabase
db.createRole(
  {
    role: "myCustomRole",
    privileges: [
      { resource: { db: "myDatabase", collection: "" }, actions: [ "find", "insert", "update", "remove" ] }
    ],
    roles: []
  }
)
```

- Chiffrement des données :

MongoDB prend en charge le chiffrement des données pour protéger les informations sensibles stockées dans votre base de données. Vous pouvez chiffrer les données en transit en utilisant TLS/SSL, et chiffrer les données au repos en utilisant des options telles que le chiffrement du stockage avec LUKS.

- Pare-feu :

Pour protéger votre cluster MongoDB contre les accès non autorisés, vous pouvez utiliser un pare-feu pour restreindre l'accès à votre cluster à partir de certaines adresses IP. Vous pouvez également configurer les options de sécurité de votre cluster pour désactiver l'accès à distance ou limiter l'accès à des adresses IP spécifiques.

- Audit de sécurité :

MongoDB offre des fonctionnalités d'audit qui vous permettent de surveiller les accès à votre base de données et les actions effectuées par les utilisateurs. Vous pouvez configurer des règles d'audit pour surveiller les connexions réussies et échouées, les opérations de lecture et d'écriture, et les commandes système. Voici un exemple de configuration des règles d'audit :

```
use admin
db.createCollection("system.auditLog")
db.runCommand({ auditLog: "enable", destination: "file", format: "JSON", path:
"/var/log/mongodb/auditLog.json" })
```

L'importation et l'exportation de données sont des opérations courantes dans MongoDB pour déplacer des données entre différentes bases de données ou systèmes. Voici quelques exemples d'importation et d'exportation avec des explications détaillées :

- Exporter une collection dans un fichier JSON :

Pour exporter une collection dans un fichier JSON, vous pouvez utiliser la commande **mongoexport** en ligne de commande. Voici un exemple de commande pour exporter une collection **users** dans un fichier **users.json** :

```
mongoexport --db myDatabase --collection users --out users.json
```

- Importer des données à partir d'un fichier JSON :

Pour importer des données à partir d'un fichier JSON dans une collection, vous pouvez utiliser la commande **mongoimport** en ligne de commande. Voici un exemple de commande pour importer des données à partir d'un fichier **users.json** dans une collection **users** :

```
mongoimport --db myDatabase --collection users --file users.json
```

- Exporter une base de données entière :

Pour exporter une base de données entière dans un fichier JSON, vous pouvez utiliser la commande **mongodump** en ligne de commande. Voici un exemple de commande pour exporter la base de données **myDatabase** dans un dossier **myDatabaseDump** :

```
mongodump --db myDatabase --out myDatabaseDump
```

- Importer une base de données entière :

Pour importer une base de données entière à partir d'un fichier BSON, vous pouvez utiliser la commande **mongorestore** en ligne de commande. Voici un exemple de commande pour importer la base de données **myDatabase** à partir du dossier **myDatabaseDump** :

```
mongorestore --db myDatabase myDatabaseDump/myDatabase
```

MongoDB est un système de gestion de bases de données NoSQL très populaire qui offre une grande flexibilité et évolutivité pour les applications modernes. Dans cet exemple, nous allons voir comment utiliser MongoDB avec Python en utilisant le package **pymongo**. Nous allons couvrir les opérations CRUD de base ainsi que les opérations d'agrégation.

- Installation de **pymongo**

Avant de pouvoir utiliser **pymongo**, vous devez l'installer en utilisant pip. Ouvrez une fenêtre de terminal et exécutez la commande suivante :

```
pip install pymongo
```

- Connexion à la base de données

Avant de pouvoir effectuer des opérations sur une base de données MongoDB, vous devez vous connecter à celle-ci en utilisant **pymongo**. Pour ce faire, vous pouvez utiliser la méthode **MongoClient()** en spécifiant l'adresse IP et le port du serveur MongoDB :

```
from pymongo import MongoClient

client = MongoClient('localhost', 27017)
```

➢ Création d'une collection

Une fois connecté, vous pouvez créer une collection en utilisant la méthode **insert_one()** de **pymongo**. Voici un exemple de code qui crée une collection **users** et ajoute un document à celle-ci :

```
db = client['myDatabase']
users = db['users']

user = {"name": "John", "age": 30}
result = users.insert_one(user)
print(result.inserted_id)
```

Dans cet exemple, nous avons créé une collection **users** dans la base de données **myDatabase** en utilisant la méthode **insert_one()**. Nous avons également ajouté un document dans la collection **users** en spécifiant un dictionnaire de clés-valeurs pour les champs **name** et **age**. La méthode **insert_one()** renvoie un objet **InsertOneResult** qui contient l'ID du document inséré.

➢ Lecture de données

Pour lire des données à partir d'une collection, vous pouvez utiliser la méthode **find()** de **pymongo**. Voici un exemple de code qui lit tous les documents de la collection **users** :

```
for user in users.find():
  print(user)
```

Dans cet exemple, nous avons utilisé la méthode **find()** pour récupérer tous les documents de la collection **users**. La méthode **find()** renvoie un objet **Cursor** qui permet d'itérer sur tous les documents de la collection.

Vous pouvez également spécifier des filtres pour limiter les documents retournés en utilisant des expressions de requête MongoDB. Voici un exemple qui lit les documents de la collection **users** où **age** est supérieur à 25 :

```
for user in users.find({"age": {"$gt": 25}}):
  print(user)
```

Dans cet exemple, nous avons utilisé l'opérateur **$gt** pour spécifier que l'âge doit être supérieur à 25.

➢ Mise à jour de données

Pour mettre à jour des données dans MongoDB avec Python, on peut utiliser la méthode **update_one()** ou **update_many()** de la classe **Collection**.

➢ update_one()

La méthode **update_one()** permet de mettre à jour un seul document dans la collection.

```
query = {"name": "John"}
new_values = {"$set": {"age": 30}}
col.update_one(query, new_values)

for x in col.find():
  print(x)
```

Dans cet exemple, la méthode **update_one()** met à jour le premier document de la collection qui a pour champ "name" la valeur "John". Le document est mis à jour avec la nouvelle valeur de l'âge.

➢ update_many()

La méthode **update_many()** permet de mettre à jour plusieurs documents dans la collection.

```
query = {"address": {"$regex": "^S"}}
new_values = {"$set": {"name": "Minnie"}}
col.update_many(query, new_values)

for x in col.find():
  print(x)
```

Dans cet exemple, la méthode **update_many()** met à jour tous les documents de la collection dont l'adresse commence par la lettre "S". Les documents sont mis à jour avec la nouvelle valeur du champ "name".

- Suppression de données

Pour supprimer des données dans MongoDB avec Python, on peut utiliser la méthode **delete_one()** ou **delete_many()** de la classe **Collection**.

- **delete_one()**

La méthode **delete_one()** permet de supprimer un seul document dans la collection.

```
query = {"name": "John"}
col.delete_one(query)

for x in col.find():
  print(x)
```

Dans cet exemple, la méthode **delete_one()** supprime le premier document de la collection qui a pour champ "name" la valeur "John".

- **delete_many()**

La méthode **delete_many()** permet de supprimer plusieurs documents dans la collection.

```
query = {"address": {"$regex": "^S"}}
col.delete_many(query)

for x in col.find():
  print(x)
```

- Indexation :

Les index dans MongoDB sont similaires aux index dans les bases de données relationnelles. Ils accélèrent les requêtes de recherche en permettant à la base de données de localiser les documents plus rapidement. Les index peuvent être créés sur une ou plusieurs clés de champs dans une collection.
Voici un exemple de création d'index :

```
mycollection.create_index("nom")
```

- Agrégation :

L'agrégation est le processus de traitement des données pour retourner un résultat calculé. Les opérations d'agrégation dans MongoDB sont effectuées en utilisant la méthode **aggregate()** sur un objet de collection. Les opérations d'agrégation peuvent être combinées pour effectuer des calculs complexes.
Voici un exemple d'opération d'agrégation :

```
pipeline = [
  {"$group": {"_id": "$ville", "count": {"$sum": 1}}}
]

result = mycollection.aggregate(pipeline)
for r in result:
  print(r)
```

- Transactions :

Les transactions permettent de garantir l'intégrité des données en assurant que toutes les opérations dans une transaction sont effectuées avec succès ou qu'aucune opération n'est effectuée du tout. Les transactions dans MongoDB sont effectuées en utilisant la méthode **with_transaction()** sur un objet de base de données.

Voici un exemple de transaction :

```
from pymongo.client_session import ClientSession

with client.start_session() as session:
    with session.start_transaction():
        # Effectuer les opérations à l'intérieur de la transaction
        mycollection.insert_one({"nom": "Durand", "ville": "Paris"})
        mycollection.update_one({"nom": "Dupont"}, {"$set": {"ville": "Marseille"}})
```

- Sécurité :

MongoDB fournit plusieurs fonctionnalités de sécurité pour protéger les données stockées dans la base de données. Cela inclut l'authentification, l'autorisation, le chiffrement et les audits. L'authentification est le processus de vérification de l'identité de l'utilisateur, tandis que l'autorisation est le processus de contrôle des actions qu'un utilisateur peut effectuer.

Voici un exemple d'authentification et d'autorisation :

```
from pymongo import MongoClient
from pymongo.auth import authenticate

client = MongoClient("mongodb://localhost:27017/")

# Authentification
authenticate(client.admin, "user", "password")

# Autorisation
mycollection = client.mydatabase.mycollection
mycollection.insert_one({"nom": "Dupont", "age": 30})
```