Rakib SHEIKH
Asano - ESGI 3 IABD - Machine Learning for Information Access
rsheikh1@myges.fr | noobzik@pm.me

TD/TP 3 : Amenez moi ce python !

Pour rappel, le Github est disponible à cette adresse : https://github.com/Noobzik/TP_NoSQL

Comment ça mon reuf ? Nous allons allier la puissance de MongoDB avec Python tout simplement ! Accrochez-vous bien, car cela va envoyer du lourd.

> **Caution**
>
> Cette version n'est pas la version finalisé du TP.

1. Introduction

Il existe deux drivers (modules) mongodb pour python à savoir:
- PyMongo : Utilisé dans les applications synchrones.
- Motor : Utilisé dans les applications non synchrones.

Nous utiliserons PyMongo pour la suite de ce TP. Pour ces deux drivers, il faudra voir cela comme un ORM, qui permet de connecter une base de données mongo à votre application.

1. Quelle est la principale responsabilité d'un pilote MongoDB ? (Sélectionnez une réponse.)

- A. Exécuter les pipelines d'agrégation.
- B. Établir des connexions sécurisées avec un cluster MongoDB et exécuter des opérations de base de données pour le compte d'applications clientes.
- C. Contrôler la réplication et le partage entre les serveurs.
- D. Créer différents types de graphiques de nos données.

2. Quel est le driver que nous utiliserons pour les applications synchrone de python ?
- A. Motor
- B. PyMongo
- C. PyMBD
- D. MangoPy

2. La connexion

Nous allons maintenant écrire nos premiers lignes de code. Pour cela, un fichier `requirement.txt` vous est fournit.
1. Créez un fichier `connection.py` et importez MongoClient depuis pymongo
2. Affectez à la variable MONGO_URI la valeur suivante

```
MONGODB_URI = "mongodb://user1:<password>@localhost"
```

> **Note**
>
> Ce n'est pas une bonne pratique de définir une URI dans du code. En effet, si nous oublions de l'effacer au moment du push dans un git, elle risque d'être exposé à la vue de tous. Dans le pire des cas, nous pouvons être victime d'un accès dérobé aux git, et qu'à partir de là, l'attaquant peut accéder à l'URI du fichier en

> question pour accéder à MongoDB. Exemple très récent avec New York Times : https://www.securityweek.com/new-york-times-responds-to-source-code-leak/

3. Instanciez un clien MongClient avec en paramètre le MONGO_URI.
4. Vérifiez la connexion en affichant la liste des bases de données qui est contenu dans client, à l'aide de la méthode `list_database_names()`
5. Vous vous rappelez que vous avez crée un cluster sur Atlas ? Nous allons le faire de même mais en ligne.

```
MONGOD_URI = "mongodb+srv://user1: <password>@cluster0.qtxwxpp-mongodb.net/?
retryWrites=truesw=majority"
```

3. La database

3. 1. Banques

Avant de continuer, nous allons utiliser deux collections issue de bank qui est :

- accounts
- transfers

Elles sont disponible en tant que fichier bson, et vous devez lancer le script python insert_bank.py pour insérer les données dans votre instance MongoDB.

3. 2. Sample Database

Nous allons utiliser les samples database que nous avons "vue" dans le premier TP. Mais pour cela il faudra télécharger les sample data officiel de MongoDB.

```
curl https://atlas-education.s3.amazonaws.com/sampledata.archive -o sampledata.archive
```

Et ensuite depuis notre docker, lancer une procédure de restauration de base de donnée

```
mongorestore --archive=/sample_data/sampledata.archive --username root --password example
```

4. CRUD

4. 1. Insert

Nous allons tout d'abord voir la version terminal et ensuite la version python. Il existe deux version d'insertion, insertOne et insertMany.

Nous utilisons la database suivante : `sample_analytics/accounts`

1. Inserez dans accounts la valeur suivante :

```
{
  "account_id": 111333,
  "limit": 12000,
  "products": [
    "Commodity",
    "Brokerage"
    ],
  "last_updated": new Date()
}
```

2. Maintenant nous allons inserer plusieurs valeurs avec insertMany

```
[{  "account_id":  111333,  "limit":  12000,  "products":  [  "Commodity",  "Brokerage"],
"last_updated": new Date() },
 {  "account_id": 678943, "limit": 8000, "products": [ "CurrencyService", "Brokerage",
```

```
"InvestmentStock"], "last_updated": new Date() },
  { "account_id": 321654, "limit": 10000, "products": [ "Commodity", "CurrencyService"],
"last_updated": new Date() }]
```

4. 1. 1. Maintenant en python !

Nous allons utiliser un nouveau format de donnée étendu du JSON, qui est BSON. C'est un format utilisé par mongodb pour stocker et organiser les données, sous la forme binarisé. Elle est optimisé pour le stockage, la récupération, et la transmission sur les différents composants.

La particularité est qu'elle est plus sécurisé par rapport à un JSON car cela nous permet de contrer les attaques par injections de données JSON. De plus, elle est compatible avec plus de Datatype qu'un JSON conventionnel.

PyMongo permet de traiter les documents BSON en tant que dictionnaire python, voir même sous la forme de tableau ! La conversion entre les types de données Python et BSON se fait de manière totalement automatique.

Pour cette section, nous utiliserons la collection de donnée : `bank/accounts`

4. 1. 1. 1. Insertion

Nous utiliserons le jeu de donnée suivant : `bank/accounts`.

Nous allons utiliser `insert_one()` et `insert_many()`.

- Nous devons donc ajouter une référence vers notre base de données dans un premier temps, et vers notre collection, dans un deuxème temps.

```
db = client.bank
account_collection = db.accounts
```

4. 1. 1. 1. 1. insert_one

- Il nous reste plus qu'à créer un dictionnaire

```
new_account = {
  "account_holder": "Linus Torvalds",
  "account_id": "MDB829001337",
  "account_type": "checking",
  "balance": 50352434,
  "last_updated": datetime.datetime.utcnow(),
}
```

- Pour ensuite l'insérer

```
result = account_collection.insert_one(new_account)
document_id = result.document_id
print("_id of inserted document: {document_id}")
```

4. 1. 1. 1. 2. insert_many

```
new_accounts = {
  "account_id": "MDB011235813",
  "account_holder": "Ada Lovelace",
  "account_type": "checking",
  "balance": 60218,
},
{
  "account_id": "MDB829000001",
  "account_holder": "Muhammad ibn Musa al-Khwarizmi",
  "account_type": "savings",
  "balance": 267914296,
}
```

- Pour ensuite l'insérer

```
result = account_collection.insert_many(new_accounts)
document_id = result.document_ids
print("# of  documents inserted: " + str(len(document_id))
print("_id of inserted document: {document_ids}")
```

4. 2. Find

Il existe deux manière de pouvoir récupérer un document, soit en utilisant l'opérateur `field: {$eq: <value>}` soit `{field: <value>}`

Pour cette section, la collection à utiliser sera : `sample_training/zips`

1. Récupérez l'ensemble des états avec la valeur `AZ`.
2. L'opérateur `$in` nous permet de selectionner tout les documents qui ont un champ égale une des valeurs spécifié dans le tableau. Renvoyez le nombre de zips dont la ville est soit dans `PHOENIX` soit dans `CHICAGO`.
3. Changeons de Dataset, maintenant nous souhaitons récupérer cette information depuis la base de donnée `sample_supplies/sales` `{ _id: ObjectId("5bd761dcae323e45a93ccff4")}`
4. Et enfin, nous souhaitons récupérer l'ensemble de la base de donnée `storeLocation` qui est à la fois à `New York` mais aussi à `London` `{ storeLocation: { $in: ["London", "New York"] } }`

4. 2. 1. Opérateurs

Nous utilisons `sample_supplies/sales`

Les quatres opérateurs sont les suivants :

- `$gt` pour greater than $>$: Retourne les documents dont le champ contient une valeur supérieure à la valeur spécifiée.

Exemple : `db.sales.find({ "items.price": { $gt: 50 } })`

- `$lt` pour less than $<$: Retourne les documents dont le champ contient une valeur inférieur à la valeur spécifiée.

Exemple : `db.sales.find({ "items.price": { $lt: 50 } })`

- `$lte` pour less than or equal to $\leqslant$: Retourne tous les documents qui contient un nombre inférieur ou égale d'un nombre donnée.

Exemple : `db.sales.find({ "customer.age": { $lte: 65 } })`

- `$gte` pour greater than or equal to $\geqslant$: Retourne tous les documents qui contient un nombre supérieur ou égale d'un nombre donnée.

Exemple : `db.sales.find({ "customer.age": { $gte: 65 } })`

Pour cette section, nous utiliserons la collection suivante : sample_supplies sales

1. Trouver les ventes comprenant un article dont le prix est supérieur à 200$
2. Trouvez tous les documents qui contiennent un article dont le prix est inférieur à 25$.
3. Recherchez tous les documents contenant un article dont la quantité est supérieure ou égale à 10.
4. Recherchez tous les documents concernant une vente à un client âgé de 45 ans ou moins.

4. 2. 2. Maintenant en python !

Nous utiliserons : bank account

Nous pouvons chercher un document à partir de son ObjectId, mais en général, il est assez difficile de le connaitre. Mais puisqu'on est certain de retourner une valeur au plus, on peut utiliser find_one.

```
document_to_find = {"_id": ObjectId ("62d6e04ecab6d8e1304974ae") }
cursor = accounts_collection.find_one(documents_to_find)
```

Nous pouvons également utiliser un opérateur pour chercher des documents, mais puisque nous allons potentiellement retourner plusieurs documents, on aura donc find.

```
documents_to_find = {"balance": {"$gt": 4700}}
cursor = accounts_collection.find(documents_to_find)
```

Dans les deux cas, si nous ne trouvons rien, nous avons None qui est retourné.

1. Ecrire un programme en python qui va chercher l'object id de `62d6e04ecab6d8e1304974ae` dans bank.accounts
2. Ecrire un programme en python qui va chercher tout les comptes en banque dont la somme est supérieur à 4700$

4. 3. Update

Ici la donnée utilisée est `bank/account`

Cette fois-ci, nous utiliserons directement le code python, puisque vous commencez à comprendre le principe.

- update_one() : Elle va mettre à jour un document qui répond à un critère spécifique. Sa syntaxe peut être décrite ainsi : `db.collection.update_one(<filter>, <update>)`

  Si nous supposons que nous souhaitons augmenter de 100$ un compte en banque. La première étape est de créer un filtre :

  ```
  document_to_update = { "_id": ObjectId("62d6e04ecab6d8e130497482") }
  ```

  Ensuite nous créeons un update :

  ```
  add_to_balance = {"$inc": {"balance": 100}}
  ```

  Et enfin nous appliquons la mise à jour: (ici le inc correspond à l'incrémentation)

  ```
  result = accounts_collection.update_one(document_to_update, add_to_balance)
  ```
- update_many() : Elle va mettre à jour plusieurs documents en 1 opération.

  Par exemple :

  ```
   # Filter
  select_accounts = {"account_type": "savings"}

  # Update
  set_field = {"$set": {"minimum_balance": 100}}

  # Write an expression that adds a 'minimum_balance' field to each savings acccount and sets
  its value to 100.
  result = accounts_collection.update_many(select_accounts, set_field)
  ```

1. Ecrire un programme en python qui va ajouter 200$ à tout les comptes en banques.
2. Une banque locale offre à tous ses clients titulaires d'un compte chèque une somme de 50 dollars en l'honneur de son 50e anniversaire. La banque a déposé l'argent sur les comptes en utilisant un seul transfert, dont le numéro d'identification est `TR413308000`.

   Votre tâche consiste à mettre à jour tous les comptes chèques en ajoutant le numéro d'identification du virement à la liste transfers_complete.

   Le filtre permettant de sélectionner les comptes chèques est affecté à une variable nommée select_accounts. La modification de la liste transfers_complete est affectée à une variable nommée add_transfer.

Les données relatives aux comptes se trouvent dans la collection de comptes de la base de données bancaire (`bank/account`). La collection de comptes est affectée à la variable accounts_collection.

Ecrire un code répondant permettant d'ajouter 50$ à tout le monde avec le numéro de transaction donnée dans la consigne.

4. 4. Delete

Nous allons maintenant voir pour la suppression des documents sous python.

- delete_one() : Pour supprimer un document, nous utiliserons cette commande, elle a la syntaxe suivante :

  ```
  db.collection.delete_one(<filter>)
  ```

  Nous allons donc lui donner en paramètre un filtre qui permet de savoir quel document faudra supprimer.

  Exemple :

  ```
  # Supposons que nous souhaitons supprimer le document suivant :
  documents_to_delete = {"_id": ObjectId("62d6e04ecab6d8e130497485" )}

  # Pour supprimer le document, nous allons faire appel à la fonction delete_one depuis la
  collection bank account
  db = client.bank
  accounts_collection = db.account

  # C'est ici que l'on réalise la suppression de document.
  result = account_collection.delete_one(documents_to_delete)
  ```

- delete_many() : Pour la partie Many, comme vous l'avez deviné, elle consiste à supprimer un ensemble de documents à partir d'un filtre effectué sur un document.

  Exemple :

  ```
  # Affectation de la collection account depuis la variable db qui contenait client.bank
  accounts_collection = db.accounts

  # Filtrer les comptes en banque qui ont moins de $2000
  documents_to_delete = {"balance": {"$lt": 2000}}

  # C'est ici que l'on va supprimer les documents affectés
  result = accounts_collection.delete_many(documents_to_delete)
  ```

Nous utilisons toujours le `bank/accounts`. *Pensez à recharger le fichier* `bson` *dans la base de donnée.*

1. Ecrivez un code permettant de supprimer un document ayant pour identifiant unique `ObjectId("62d6e04ecab6d8e130497485")`.
2. Ecrivez un code permettant de supprimer un ensemble de document en fonction d'un filtre suivant : Supprimer tout les comptes en banque dont le balance est inférieur à 500.

4. 5. Transaction sous python.

Comme à l'image du SGBD PostegresSQL ou bien de SQL Server chez Microsoft, nous avons également ici une notion de transaction. Une transaction est tout simplement une éxecution d'un ensemble de requête qui est temporaire, et devient définitif lorsque la commande `commit` est appelé.

C'est une sécurité qui permet de s'assurer que les requêtes ont le comportement attendu sur les données que l'on traite. En cas de problème on peut tout simplement faire l'équivalent d'un `CTRL+Z`, soit un `ROLLBACK` qui nous permet de revenir en arrière.

Pour être plus formelle, c'est le principe de l'atomicité. Il faut que l'intégralité de la requête soit en succès, ou bien en échec s'il y a un problème survenu lors de l'éxecution d'une requête.

Des exemples d'applications de transactions :
- Transfert d'argent avec une application mobile.
- Prendre un article de l'inventaire et l'ajouter dans un panier d'achat à travers une application mobile d'achat.

Sous MongoDB il est possible d'avoir le même comportement, en suivant le plan suivant :

Supposons qu'une connexion à MongoDB est active, nous devons :
- définir une fonction **callback** qui va spécifier les séquences d'opérations à réaliser lors d'une transaction.
- Démarrer une session client.
- Commencer la transaction en appelant la méthode `with_transaction()` sur l'objet de la session client. *With transaction va démarrer une transaction et à la fin, appel la fonction callback et va soit commit, soit rollback en cas d'erreur lors de l'execution.*

> **Caution**
>
> MongoDB va annuler toute transaction dont la durée totale excède 60 secondes.

Exemple :

Supposons que nous souhaitons appliquer une transaction dans un contexte bancaire, nous disposons donc de deux collection issue de cette banque qui est :
- Accounts contenant les informations client.
- transfers contenant les informations de transferts entre client.

```
{
  "account_id": "MDB574189300",
  "account_holder": "Coskun Demirbas",
  "account_type": "checking",
  "balance": 4690.87,
  "transfers_complete": [
    "TR488315128",
    "TR401663822",
  ]
}
```

```
{
  "account_id": "MDB343652528",
  "account_holder": "Marcus Jorgensen",
  "account_type": "checking",
  "balance": 2522.14,
  "transfers_complete":[
    "TR488315128",
    "TR655897500",
  ]
}
```

Ici nous avons deux exemple de compte en banque avant d'appliquer la transaction. Elles incluent des zones contenant le nom du titulaire du compte, le solde, l'ID de compte et les ID des transferts terminés.

Les opérations qui interviennent sont les suivantes :
- Dans la collection account, nous allons mettre à jour (soit Update) les docuents avec une `balance` modifié avec un identifiant de transfert.
- Dans la collection transfers, nous allons ajouter un document relative au information du transfert.

Maintenant, nous avons toutes les informations nécessaire pour réaliser une transaction.

Etape 1 : Définissons une callback

Nous allons définir une callback qui va spécifier la séquence des opérations pour appliquer une transaction.

```
def callback(
  session, transfer_id=None,
  account_id_receiver=None,
```

```
  account_id_sender=None,
  transfer_amount=None,
):
  # Nous fixons les deux collections qui vont intervenir
  accounts_collection = session.client.bank.accounts
  transfers_collection = session.client.bank.transfers

  # Nous construisons un transfert qui va être ajouté dans la collection transfers plus tard.
  transfer = {
    "transfer_id": transfer_id,
    "to_account": account_id_receiver,
    "from_account": account_id_sender,
    "amount": {"$numberDecimal": transfer_amount},
  }

  # C'est ici qu'on va débuter la transaction
  # Note : Il faut passer la session à chaque opération !

   # Opération 1 : Mettre à jour l'account sender : Soustraire le balance au montant de la
transfer du sender

  accounts_collection.update_one(
    {"account_id": account_id_sender},
    {
      "$inc": {"balance": -transfer_amount},
      "$push": {"transfers_complete": transfer_id},
    },
    session=session,
  )

  # Opération 2 : Mettre à jour l'account receiver : Ajouter le balance au montant de la transfer
du receiver

  accounts_collection.update_one(
    {"account_id": account_id_receiver},
    {
      "$inc": {"balance": transfer_amount},
      "$push": {"transfers_complete": transfer_id},
    },
    session=session,
  )

  # Opération 3 : Ajouter un nouveau transfer vers la collection transfers.

  transfers_collection.insert_one(transfer, session=session)

  # Un petit print pour dire que la transaction est réalisé
  print("Transaction sucessfull")
  return
```

Etape 2 : Définir une fonction d'encapsulation `callback_wrapper` dans le but de faire passer des paramètre à notre callback que nous venons de définir, afin de permettre un transfer entre Coskun et Marcus.

```
def callback_wrapper(s):
  callback(
    S,
    transfer_id="TR218721873",
```

```
    account_id_receiver="MDB343652528",
    account_id_sender="MDB574189300",
    transfer_amount=100,
  )
```

Etape 3 Démarrer une session client et lancer la transaction.

```
with client.start_session() as session:
  session.with_transaction(callback_wrapper)
client.close()
```

1. Rédigez deux transaction qui permet de transferer un montant de votre choix. Ces deux transaction auront une contrainte de ne pas utiliser les comptes en banque cité en exemple.

5. Conclusion

Vous avez vu dans ce TP :

- la manipulation des fichiers bson.
- Les opérations CRUD.
- Une transaction sur plusieurs documents.