INFO834 – Mini-projet – BD NoSQL MongoDB, Redis

Le sujet couvre l'ensemble des 12h de TP restants. Il se fera en 2 étapes :

- **Partie I** : sur MongoDB : administration d'une base de données MongoDB, des commandes en vrac, de la programmation en Python (bibliothèque PyMongo). La mise en place d'un réseau de serveurs devra être réalisée sur MongoDB afin de faciliter la tolérance aux pannes avec des replicasets. Cette étape vous servira pour la suivante.
- **Partie II** : un mini-projet pour la réalisation d'une application de chat permettant à des utilisateurs de converser. Les messages/conversations devront être gérés dans une base de données MongoDB avec un historique des connexions stocké sur une base de données Redis.

* Organisation : Travail par équipe de 4 personnes pour le mini-projet (Partie II), seul.e ou en binôme pour la partie préparatoire (Partie I).

* Livrables : 2 rapports au format *pdf* : 1 pour chaque partie contenant les liens vers les dépôts git de vos sources (2 dépôts différents sur Moodle). Le rapport sur MongoDB devra faire figurer des copies d'écran des commandes testées avec les résultats de même que les fonctions Python avec les tests qui les accompagnent. Le rapport du mini-projet devra faire figurer en pourcentage, la contribution estimée de chacun dans l'équipe : exemple 20%, 30%, 30%, 20% (évaluation par les pairs).

* Echéances : dépôt du rapport de la partie I avant lundi 06 février, 8h ; dépôt du rapport de la partie II avant mercredi 02 mars, 8h.

PARTIE I

1. Installation de MongoDB

Rappel : pour installer MongoDB sur votre machine, rendez-vous à l'adresse suivante https://www.mongodb.com/download-center/community et récupérez le fichier nécessaire pour l'installation qui doit correspondre à votre système d'exploitation.
Pour vous assurer que l'installation se soit bien faite, lancez la commande *mongo -version*. Si vous voyez apparaître un numéro de version, c'est que tout va bien. S'il y a une erreur, c'est que vous n'avez pas mongo dans votre PATH, à corriger ou sinon placez le chemin complet avant le nombre de l'exécutable.

2. Exécution de MongoDB

L'exécutable *mongod* est le serveur alors que *mongo* est le client. Pour lancer le serveur (bien entendu avant le client...), il faut disposer d'un répertoire pour le stockage des données. Par défaut, nous pourrions prendre un répertoire data. Donc pour lancer le serveur, cela donne : *mongod –dbpath ./data*. Si tout se passe bien, vous allez voir un flot de données pour finalement voir apparaître : *waiting for connections*. Attention à ne pas tuer l'application en fermant le terminal ou en faisant CTRL+C. Dans un autre terminal, vous pouvez lancer le client avec $*mongo --port <numport>*.

3. On insère de la donnée

Téléchargez depuis l'espace du cours Moodle, le fichier *communes.csv* que vous pourrez importer comme une collection *communes* dans une BD *France* à l'aide de la commande *mongoimport* depuis une fenêtre de commande :

$mongoimport -d=France -c=communes --type csv communes-departement-region.csv --headerline

Il faut bien entendu que le serveur soit en fonctionnement. Ici, nous n'avons pas considéré qu'il y avait besoin d'une authentification. Normalement, si tout se passe bien, vous devriez voir qu'il y a eu insertion de *39201* documents.

4. Les commandes de base

Comme nous avons pu le voir en TD, nous avons un certain nombre de commandes de base :

- *show databases :* retourne l'ensemble des bases de données qui sont dans le répertoire *data*. Vous y trouverez forcément les bases *admin*, *config* et *local* de même que les bases de données que vous avez créées.
- *use XXX* : permet de choisir la base de données *XXX* (par exemple *France*) afin de n'avoir que les collections pour cette base de données.
- *show collections :* retourne l'ensemble des collections qui est dans la base de données.
- *db.XXX.count()* : retourne le nombre d'enregistrements dans la collection *XXX*.
- *db.XXX.find()* : retourne les données. S'il y en a trop, seulement une partie est affichée.
- *db.XXX.find().pretty()* : retourne les données mieux présentées.
- *db.drop()* : supprime toutes les données dans une collection.

5. L'outil de visualisation Robo 3T

Rappel : vous pouvez le télécharger à l'adresse suivante : https://robomongo.org/download. Installez-le. Lancez le serveur MongoDB et ensuite connectez-vous par l'intermédiaire de Robo 3T. Vérifiez que vous pouvez ajouter des documents et trouver le nombre d'éléments de la base *communes*.

6. Quand il y a trop de données… Utilisation de curseurs

Du moment qu'il n'y a pas trop de données, il est possible d'afficher directement l'ensemble mais s'il s'agit de milliers de documents, il est préférable de penser à la pagination (ou *curseur* dans MongoDB). L'utilisation de *cursor* est assez facile. Voici le code à écrire côté client (shell) :

```
> var myCursor = db.XXX.find( {} );
while (myCursor.hasNext()) {
  print(tojson(myCursor.next()));
}
```

Vous pouvez donc récupérer les pages suivantes par l'intermédiaire de *.next()*. Testez sur la base des communes.

Note : même si vous faites sans le curseur, MongoDB utilise par défaut les curseurs (affichage de *Type it for more*).

7. Un peu de benchmark

Écrivez un programme en Python qui calcule le temps nécessaire pour lire chaque document de la collection en recherchant par le nom de la commune. Conservez le temps nécessaire.

8. Quand ça presse et qu'il faut aller vite

Les *index* en MongoDB permettent d'améliorer la vitesse d'une base de données. Supposez que vous recherchez très fréquemment sur un champ particulier, il devient intéressant de créer un index sur ce dernier de manière à ce qu'il soit stocké en mémoire vive et raccourcir ainsi le temps de récupération des données.

N.B. L'indexation d'un champ devrait quasiment être envisagée dès le départ car indexer des Gigaoctets ou des Teraoctets peut s'avérer long.

Indexer un fichier en MongoDB se fait de la manière suivante :

```
> db.communes.createIndex( { nom_commune : 1 } )
```

Cela crée un index sur la collection communes pour le champ *nom_commune*. La valeur précisée après le « *:* » donne le tri de la sauvegarde. Pour -1, c'est un tri descendant, et pour 1, c'est un tri ascendant.

Ajoutez un index à la base de données *communes*. Cherchez quels peuvent être les champs à indexer. On peut créer plusieurs index en les séparant par des virgules :

```
> { item : 1, quantity: -1 }
```

Si des *index* doivent être uniques, vous pouvez le préciser :

```
> db.communes.createIndex( { code_commune_INSEE: 1 }, { unique: true } )
```

Dans la base des communes, cette unicité est-elle possible ?

Relancez le programme de la question 7, et vérifiez le temps de traitement est plus court.

9. Le nombre de visiteurs depuis 01/01/2001 est de…

Lorsque nous voulons faire de l'analyse de fréquentation, il est nécessaire de disposer d'un compteur qui doit être augmenté à chaque visite. La solution évidente est de faire un *update* (ou *upsert* pour gagner en temps) sur le document et d'ajouter 1. Mais, étant donné que cette opération sera faite fréquemment, nous allons utiliser un moyen plus efficace : *$inc* dans MongoDB (comme dans Redis).

Créez une nouvelle base de données *Analytics*. Avec *use Analytics*, si vous faites *show Databases*, la nouvelle base n'apparaitra pas car elle ne contient pas encore de données.

Créez un document :

```
> visit = { url : "https://www.google.com",
... nb : 1}
```

Les … ont été ajoutés par Mongo shell en attente de compléter le document. Ajoutez ce dernier dans la collection *freq* :

```
> db.freq.insert(visit)
```

Note : si maintenant vous demandez *show databases*, *Analytics* apparaît bien. Le *nb* est bien 1. Modifiez cette valeur en exécutant la séquence suivante (récupération de *_id* de l'objet puis son *update* avec *$inc*) :

```
> db.freq.find().pretty()
{
"_id" : ObjectId("5e81e0e11e95dfc6e9cd3611"),
"url" : "https://www.google.com",
"nb" : 1
}
> db.freq.update({_id : ObjectId("5e81e0e11e95dfc6e9cd3611")}, {$inc:
{nb : 1}})
```

En exécutant de nouveau la commande d'affichage, nous avons bien *nb = 2* :

```
> db.freq.find().pretty()
{
"_id" : ObjectId("5e81e0e11e95dfc6e9cd3611"),
"url" : "https://www.google.com",
"nb" : 2
}
```

10. Utilisation d'Array dans les documents

Jusqu'à présent, les données sauvegardées étaient uniquement des chaines de caractères ou des nombres mais il est possible de sauvegarder des tableaux de données.

Un exemple simple est de considérer lue mailing-list comme un document contenant un tableau où chaque élément est une des adresses mel enregistrées pour cette mailing-list. Créez une base de données *mailing* puis créez un document *list* avec les attributs *name* (nom de la mailing-list) et un attribut *emails*, tableau qui contiendra les adresses mel. Insérez ensuite de document dans une collection *lists* et affichez le résultat pour récupérer l'identifiant *_id* du document. Mettez à jour le tableau *emails* en y insérant une adresse mel, exemple :

```
> db.lists.update({_id : ObjectId("5e81e5801e95dfc6e9cd3612")},
... {$push : {emails : "Marc-Philippe.Huget@univ-smb.fr"}})
```

Puis de la même façon, il est possible de retirer un élément du tableau à l'aide de la commande *update* avec *$pull.* Ajoutez une 2ème adresse mel au tableau puis retirez-la. Vérifiez le résultat.

11. Et si liait les documents ?

Nous allons modifier la mailing-list de manière à ce qu'au lieu de stocker des adresses mel, elle stocke le lien vers les documents correspondant aux utilisateurs. Créez dans la base *mailing* 3 documents avec les attributs *name*, *firstname* et *email* correspondants à 3 utilisateurs. Ajoutez ces documents à une nouvelle collection *users*. Vérifiez le résultat.
Supprimez ensuite la collection *lists* puis recréez-la de manière à ce qu'elle contienne cette fois-ci des documents avec les attributs *name* et *users,* un tableau contenant les *_id* des 3 documents utilisateurs créés précédemment. Le résultat devrait ressembler à ceci :

```
> db.lists.find().pretty()
{
"_id" : ObjectId("5e81f8d01e95dfc6e9cd3615"),
"name" : "Mailing-List 1",
"users" : [
ObjectId("5e81f5cd1e95dfc6e9cd3613"),
ObjectId("5e81f6041e95dfc6e9cd3614")
]
}
```

En utilisant la bibliothèque *PyMongo*, créez le programme en Python qui permet de récupérer en une seule fois la mailing liste et l'ensemble des utilisateurs. En MongoDB, la commande *.populate* permet de le faire.

12. Montrez patte blanche

Jusqu'à présent, nous n'avons pas protégé nos bases de données, et il est possible d'y accéder du moment que nous savons où chercher. Il devient important de contrôler les accès et protéger les données. Pour cela, il faut donc créer un utilisateur avec la commande *db.createUser* (https://docs.mongodb.com/manual/tutorial/create-users/).Vous pouvez également le faire avec Robo 3T (base *Analytics*, clic droit sur *Users* et ensuite *Add user*).
Faites-en sorte que cet utilisateur ne peut que lire et pas écrire puis testez le fonctionnement.
Attention : il faut relancer *mongod avec l'option - -auth* sinon cela ne prend pas en compte les utilisateurs.
Vous pouvez tester directement dans le Mongo shell, avec la commande *connect,* exemple :

> db = connect("localhost:27017/Analytics", "mph", "mph")

Essayez d'insérer un document, normalement vous aurez un message « Unauthorized » puisque l'utilisateur n'a pas le droit d'écriture sur la base.

13. On se ferait bien un petit coup de Map Reduce ?

Commençons par voir comment écrire la fonction *map* sur la collection *communes* :

> use France

switched to db France

> show collections

communes

> var mapFunction = function() {
... if (this.code_region == 84) emit(this.nom_commune, 1);
... }

La fonction *mapFunction* recherche l'ensemble des communes qui est de la région Auvergne-Rhône Alpes (code 84). Si la commune correspond à ce critère, il y a émission des données à destination de la fonction *reduce* par l'intermédiaire de *emit*.

> var reduceFunction = function(nom, index) {
... return Array.sum(index);
... }

La fonction *reduceFunction* se limite à sommer le nombre de communes qui proviennent de la fonction *map*.

> db.communes.mapReduce(mapFunction, reduceFunction, {out : "map reduce example"})

{
"result" : "map reduce example",
"timeMillis" : 606,
"counts" : {
"input" : 39201,

"emit" : 4467,
"reduce" : 328,
"output" : 4020
},
"ok" : 1
}

Dans le résultat obtenu, le *input* semble logique puisqu'il correspond au nombre de documents dans la base MongoDB. Par contre le nombre d'*output* varie un peu par rapport au nombre de communes que nous sommes censés obtenir (*4095*). Cela tient vraisemblablement au rassemblement de communes.
Pour voir le contenu, il suffit d'ajouter *.find()* à la commande précédente.
Sur le même principe, tentez d'obtenir toutes les communes de la région Auvergne-Rhône Alpes qui commencent par un A.

14. Sauvegarde et restauration des données

La solution la plus simple pour une sauvegarde de la base de données est de copier le répertoire *./data* que vous avez créé. MongoDB fournit des outils pour effectuer la même chose de façon plus professionnelle : *mongodump* et *mongorestore*.

> mongodump --out=./backup/

Il faut que le serveur MongoDB soit en état de fonctionnement car *mongodump* lit toutes les données présentes dans la base. Cela peut avoir des conséquences par rapport à la base de données si elle est en production. *Mongodump* produit des fichiers binaires *bson*.
Pour la restauration, lancez un serveur MongoDB avec une base vide et faites :

> mongorestore backup/

15. Aggregation

Nous avons pu voir que Map et Reduce étaient disponibles dans MongoDB, par contre cela reste limité à une fonction *map* et une fonction *reduce*. Supposons que nous voulons faire plusieurs raffinements ou alors des regroupements. MongoDB *Aggregation* nous permet de créer un *pipeline* de traitement sur les données. Les données nécessaires à la réalisation de cette partie se trouvent dans le fichier *zips.json* (cf. Moodle).
Commencez par charger cette base dans votre MongoDB puis rendez-vous à cette adresse : *https://fr.blog.businessdecision.com/tutoriel-mongodb-agregation/* pour réaliser l'ensemble des exemples sur *l'Aggregation* afin d'en comprendre le fonctionnement.

PARTIE II

L'objectif de cette partie est de réaliser une application de chat permettant à des utilisateurs de converser. Les messages/conversations devront être gérés dans une base de données MongoDB avec un historique des connexions stocké sur une base de données Redis.

L'interface de l'application pourrait être une interface web (REST ou non) ou bien une interface Python reposant sur des sockets (avec ou sans threads). Vous pourrez dans ce second cas partir de bases existantes que vous adapterez en les modifiant, exemples :

https://python.developpez.com/cours/apprendre-python3/?page=page_20
https://codinginfinite.com/python-chat-application-tutorial-source-code/

Fonctionnalités à réaliser, celles qui permettent de :

- connaître quels sont les utilisateurs connectés et les afficher (en utilisant Redis)
- stocker l'ensemble des messages dans MongoDB
- utiliser un *ReplicaSet* pour permettre une meilleure tolérance aux pannes
- afficher une conversation précédente entre deux utilisateurs
- d'autres requêtes pertinentes : utilisateur le plus sollicité, celui qui communique le plus, etc.

Une étape de réflexion en amont entre les membres de l'équipe sur l'architecture à mettre en place est cruciale.