Big Data : Informatique pour les données et calculs massifs

2A – campus de Metz – 2018-2019

TD-3-4 : exploitation d'une BdD NoSQL en MongoDB

Exercice 1 : démarrage de MongoDB et importation d'une BdD

L'utilisation d'une BdD MongoDB nécessite le lancement d'un serveur puis d'un (ou plusieurs) clients, ce qui nécessitera d'ouvrir plusieurs fenêtre DOS pour ce TD.

Question 1 : lancement du serveur

- En préambule il est nécessaire de créer un répertoire où seront stockées les données de la base MongoDB. Mais un répertoire **C:\data** a déjà été créé sur votre poste de TP.
- Ouvrez une fenêtre DOS (une fenêtre de « commande »), et postionnez vous sur le disque ''**C''** **en entrant** `C:`
- Lancez le serveur **en entrant :** `mongod --dbpath \data`

Question 2 : importation de données JSON

- Ouvrez une autre fenêtre DOS, et allez sur C (entrez `C:`)
- Importez un ensemble de données depuis un fichier JSON à l'aide de la commande **mongoimport** : voir les exemples dans les **slides du chapitre 9**.
  - La base de données s'appellera `td3`,
  - La collection s'appellera `publistat`,
  - Le fichier sera : `\json\fr-esr-publications-statistiques.json`
    Attention : ce fichier est en fait un ''array'' JSON (vous pouvez l'ouvrir dans un notepad pour visualiser son allure).
  - Choisissez le mode d'insertion qui vous semble convenir : `insert` ? `upsert` ?

N'hésitez pas à consulter aussi la **documentation de mongoimport sur Internet !!**

Question 3 : lancement du client

- Réutilisez la fenêtre DOS de la question 2
- Lancez le client (il s'agit du « mongo shell ») en entrant :

  mongo

  Il se connecte alors au serveur **mongod**, et est prêt à interroger la base NoSQL, le prompt du mongo shell apparaît : `>`

- Interrogez le serveur pour connaître les BdD stockées dans le répertoire géré par le serveur :

```
> show dbs
```

- Positionnez le client pour une utilisation de la base `td3` dans laquelle vous avez importé des données :

```
> use td3
```

Exercice 2 : exploitation de la BdD par *queries*

La commande permettant de faire des requêtes à MongoDB en langage natif est : **`find(…)`**. Voir les exemples dans les **slides du chapitre 9**, et dans la **documentation MongoDB sur Internet**.

Principe de fonctionnement :

- On choisit une BdD qui devient la BdD courante : fait avec "**`use td3`**" à la question précédente)

- On interroge cette BdD par :

  `db.`*`laCollection`*`.find(…)`

  où ***`laCollection`*** est le nom de la collection interrogée.

Question 1 : découvrir les collections et ce qu'elles contiennent

- Listez les collections de votre BdD à l'aide de : **`show collections`**

- Visualisez ce que contient UN enregistrement (un document JSON) de la collection que vous avez importée grâce à **`findOne()`**, en remplacement de **`find(…)`**.

- Demandez le contenu de la collection avec :

  `db.`*`laCollection`*`.find().pretty()`

  Sans arguments, la requête find ne fait aucune sélection, et projette tous les champs de chaque document. Donc … elle retourne toute la collection !
  Enchaîner la commande **`pretty()`** permet un affichage agréable comparable à celui de **`findOne()`**.

Question 2 : requêtes simples sur la collection *publistat*

En vous aidant des slides du chapitre 9 et de la documentation MongoDB sur Internet, réalisez les opérations suivantes :

- L'un des champs de chaque enregistrement est en fait une description d'une image : à l'aide de **`find(…)`** projetez le champ **`fields.publication_image_fichier`** de tous les enregistrements de la collection **`publistat`**.

- N'affichez maintenant que les hauteurs et largeurs des images au format PNG.

- Ordonnez les enregistrements sélectionnés selon leur hauteur d'image croissante.

- Affinez la requête précédente pour ne retenir que les images au format PNG dont la hauteur se trouve dans l'intervalle]500 ;700[

Attention : tester (si hauteur > 500) et (si hauteur < 700) **ne fonctionnera pas** avec deux tests séparés et associés par une « , » dans le premier argument du **`find({…},{…})`** !!

- Solution 1 : faire « un test double » :
  `{…height : {$gt : 500, $lt : 700}}`
- Solution 2 : ne faire que des tests reliés par un opérateur **$and** explicite :
  {$and : [{"…format" : "PNG"}, {"...height" : {$gt : 500}}, {"…height" : {$lt : 700}}] }

- Comptez le nombre d'enregistrements satisfaisant la requête précédente.

Question 3 : requêtes et enrichissement des enregistrements

La méthode **`forEach(…)`** peut être appliquée au document résultant d'un **`find(…)`** :

```
db.laCollection.find().forEach(<function>)
```

La fonction dans le forEach peut être complètement définie dans le forEach, ou bien n'être qu'un appel à une fonction déjà définie dans le Mongo Shell.

La requête **`update`**, associée à l'opérateur **`$set`** permet de changer le contenu d'un champ d'un enregistrement sans modifier les autres, ou d'ajouter un nouveau champ pour enrichir les données.

```
db.laCollection.update({_id : xxxx},{$set : {leChamp : laValeur}})
```

Voir les **slides du chapitre 9**, et la documentation MongoDB sur Internet.

- A partir d'une requête **`find(…)`** et d'une méthode **`forEach(…)`** : **calculez et affichez le nombre de pixels de chaque enregistrement décrivant une image au format PNG.**

  La fonction définie dans le **`forEach(…)`** est écrite en JavaScript, et supporte la définition de variables locales, l'utilisation d'opérateurs arithmétiques, et le print (…)

  Ex : **`function(doc) { var x = doc.a + doc.b; print("a.b = " + x);}`**

- Modifiez la fonction précédente du **`forEach(…)`** pour qu'elle affiche aussi le champ **`_id`** de l'objet avant d'afficher son nombre de pixels.

- Faites une mise à jour du dernier enregistrement (copiez-collez son **`_id`** depuis la fenêtre DOS) pour lui ajouter un champ dans la description de son image PNG, à l'aide de la requête **`update`** associée à l'opérateur **`$set`**.

- **Modifiez votre `forEach` pour qu'en plus d'afficher le nombre de pixels de l'image PNG, la fonction définie dans le `forEach` insère un champ `nbpix` à chaque description d'image PNG.**

- A l'aide d'une requête **`find(…)`** et d'une méthode **`sort(…)`** ordonner tous les enregistrements par taille d'image PNG croissante.

Exercice 3 : traitements *mapReduce* en MongoDB

Rappel : le Map-Reduce de MongoDB est différent de celui d'Hadoop, est est nommé « mapReduce ». Globalement il est moins générique et supportent moins d'optimisations, mais la principale différence réside dans le fonctionnement de l'étape ***reduce****. Elle évite de provoquer des débordements mémoire, mais est plus contraignante (voir* ***slides des chapitres 8 et 9****).*

Question 1 : importer une collection de plus grande taille

Nous allons maintenant travailler avec une collection de données stockées dans un fichier JSON de **7.5Go**, et qui prendra environ **1.5Go** une fois chargé dans MongoDB et stocké en BSON (Binary JSON).

Dans la même base que précédemment (base ''td3'') importez dans la collection `inscription` le fichier (sur C:) :

```
\json\fr-esr-sise-effectifs-d-etudiants-inscrits-esr-public.json
```

Rmq1 : c'est encore une fois un ''jsonArray''.

Rmq2 : l'importation peut prendre un peu de temps.

La collection `inscription` contient maintenant des données sur des inscriptions d'étudiants dans l'enseignement supérieur avec de nombreuses informations sur le bac qu'ils ont obtenu.

Question 2 : Compter le nombre de bacs de chaque type

Visualiser rapidement un des enregistrements de la collection `inscription`. Un des champs intitulé `fields.bac` (vers la fin de chaque enregistrement) code chaque type de bac par une chaine de caractère contenant un simple entier.

Ecrivez un programme `mapReduce` qui compte le nombre d'occurrences de chaque type de bac :

- **Ecrivez une fonction `map` et une fonction `reduce`**

  Rappel de la syntaxe d'une fonction `map` de *mapReduce* en MongoDB :

```
map2 = function() {
    var k = this.…;
    var v = …;
    emit(k,v);
}
```

  Rappel de la syntaxe d'une fonction `reduce` de *mapReduce* en MongoDB :

```
reduce2 = function(k,v) {
    …;
    return(…);
}
```

- **Exécutez un mapReduce selon les 2 syntaxes possibles**

```
db.runCommand( {"mapreduce" : "inscription", "map" : map2,
                "reduce" : reduce2, "out" : "res2a"} )
```

Ou bien

```
db.inscription.mapReduce(map2, reduce2, {"out" : "res2b"})
```

- Les résultats (les collections `res2a` et `res2b`) doivent être identiques. Vous pouvez les visualiser avec la commande `find()`.

- Dans les deux cas, l'opération **mapReduce** retourne un rapide bilan de son exécutions, et indique notamment son temps d'exécution.

  Ce temps est normalement plus long lors du premier **mapReduce** qui provoque un chargement de nombreuses données en RAM. **Quel premier et second temps d'exécution avez-vous observé ?**

  En lançant le gestionnaire de tâches on peut visualiser qu'un seul cœur est utilisé à la fois**. Le traitement de ce mapReduce sur une collection non-distribuée reste séquentiel.**

Question 3 : tri des données d'entrée et impact des index

Le **mapReduce** de MongoDB permet de trier/ordonner les données d'entrée (la collection `inscription` dans notre cas) avec l'option `sort` (en précisant l'attribut et le sens du tri)**:**

```
Ex: db.lacollection.mapReduce(
        map, reduce,
        {"out" : "results", "sort" : {"a.b.c.d" : -1}, …}
    )
```

Voir la documentation du **mapReduce** de MongoDB sur Internet pour plus d'information sur l'option `sort`.

- Ré-exécutez votre mapReduce **en triant les données d'entrées selon le champ `fields.bac` en ordre croissant**, puis en ordre **décroissant**.

  Quel « problème » apparait ?

On peut faciliter le travail du `sort` en **créant un index sur le champ trié** (`fields.bac`). On accélère alors les traitements et on évite le problème précédent.

- Créez un index sur cette collection et sur le champ `fields.bac` , dans le sens croissant.

  Utilisez : `db.laCollection.createIndex(…)`. Voir les slides du chapitre 9 et la documentation de `createIndex` sur Internet.

- Ré-exécutez votre mapReduce en triant les données d'entrées selon le champ `fields.bac` en ordre croissant, puis en ordre décroissant.

- Le problème apparu précédemment a-t-il disparu ? dans l'ordre croissant et décroissant ?

Question 4 : limitation des données d'entrée

En plus de trier les données d'entrées, on peut limiter leur nombre avec l'option `limit` du mapReduce. Cela permet de tester un algorithme, une commande sur un sous-ensemble des données pour ne pas attendre longtemps.

- Utilisez conjointement les options `sort` et `limit`, et limitez le nombre d'entrées à 1000.
- Les résultats sont-ils identiques avec un tri en ordre croissant et en ordre décroissant ? Pourquoi ?

Question 5 : reformatage des résultats

Le **mapReduce** de MongoDB impose que les paires clé-valeur de sortie du **reduce** soient au même format que celles de sortie du **map**. Mais une fonction `finalize(k,v)` permet de reformater les sorties du **reduce**.

- Implantez une fonction `finalize(k,v){newVal = …; return(newVal);}` pour générer des sorties au format :

  `{"bac" : code du bac, "nb" : nombre}`

- Exécutez votre mapReduce en précisant d'exécuter votre fonction `finalize`
- Visualisez le résultats avec la commande `find()`

Question 6 : ajout d'une *query* sur les données d'entrée

Il est possible de préciser une *query* sur les données d'entrée afin (par exemple) de les filtrer très simplement avant d'entrer dans le **map**.

- Enrichissez l'exécution de votre mapReduce avec une *query* qui ne retient que les enregistrements dont le champ ''`fields.dep_etab_lib`'' vaudra ''`Moselle`''
- Réexécutez votre mapReduce et visualiser le résultat

Exercice 4 : identification de toutes les valeurs possibles d'un champ

Le champ ''`fields.dep_etab_lib`'' peut prendre des valeurs variées, difficile à prévoir à cause de l'accentuation de certains caractères, mais en nombre fini (d'après la nature des valeurs). On souhaite obtenir la liste de toutes les valeurs possibles de ce champ, en ne présentant au final chaque valeur qu'une fois et une seule.

Question 1 : code mapReduce

- Ecrivez et exécutez un programme mapReduce pour recenser une fois et une seule chaque valeur du champ ''`fields.dep_etab_lib`''. De quel patron allez-vous vous inspirer ?

Question 2 : comptage des solutions

- Utilisez la méthode `count()` pour compter le nombre de réponses du résultat (voir les slides du chapitre 9 ou la documentation sur Internet).

Exercice 5 : réalisation d'un « JOIN » entre deux collections

Les bases NoSQL restent faibles pour réaliser des « join ». On part toujours du principe que le « join » doit être fait à l'écriture, c'est-à-dire que l'on doit mettre dans une collection « tout ce qu'il faut » pour pouvoir l'interroger ensuite sans avoir besoin d'autres données.

Il existe quand même des solutions pour réaliser les « join » qui s'avèrent nécessaires, mais elles sont couteuses en temps de traitement. Hadoop propose ainsi un Map-Reduce sur deux flux d'entrées (deux fichiers distribués). Mais MongoDB limite son mapReduce à une seule collection, ce qui l'empêche de réaliser une jointure sur deux collections.

Un « join » sur deux collections se réalise en MongoDB à partir de l'opérateur $lookup dans une agrégation. Voir slides du chapitre 9.

Question 1 : génération de deux tables spécialisées

à partir de la (grosse) collection `inscription` on commence par générer deux collections simples, que l'on « joindra » ensuite.

- On souhaite générer une collection ''`res51`'' de documents contenant les champs :
  - `etablissement_lib` le nom de l'établissement
  - `cursus_lmd` « L », « M » ou « D »

  En fait, on générera des documents/enregistrements contenant uniquement des clés, composées de ces deux champs :

  - _id : {''`etablissement_lib`'' : *leEtablissementLib*, ''`cursus_lmd`'' : *leCursusLMD*}

  On bénéficiera ainsi de la capacité de déduplication du mécanisme de gestion des clés, et on obtiendra autant d'enregistrements qu'il y a de couples différents de ces deux champs.

  Ecrivez et exécutez le petit programme mapReduce qui produira cette collection ''`res51`'' d'enregistrements de clés à deux champs.

- **On pratiquera de même pour obtenir la collection ''`res52`'' réunissant des enregistrements de clés avec les champs :**
  - `etablissement_lib` le nom de l'établissement
  - `aca_etab_lib` le nom de l'académie de rattachement

  On cherche dons à obtenir des enregistrements contenant :

  - _id : {''`etablissement_lib`'' : *leEtablissementLib*, ''`aca_etab_lib`'' : *laAcademie*}

Question 2 : réalisation d'un JOIN par l'opérateur $lookup dans une agrégation

On va maintenant réaliser une jointure des collections "**res51**" et "**res52**" en prenant le champ commun "**etablissement_lib**" comme clé de jointure, et en utilisant l'opérateur **$lookup** dans un **aggregate** fait spécialement pour lui.

- Implantez une agrégation réalisant cette jointure et stockant le résultat dans la collection "**res5j**".

```
db.res51.aggregate([
    {"$lookup" : {
                    ..........
                  }
    },
    {"$out" : "res5j"}
]);
```

Voir les slides du chapitre 9, et la documentation de MongoDB sur Internet.

- Visualiser le résultat dans la collection **res5j**. Le format de sortie devrait contenir toutes les informations pertinentes, mais parfois de manière redondante et pas très simple à exploiter.

Question 3 : transformation de la sortie du JOIN

- Compléter l'agrégation avec les opérateurs **$project** et **$unwind** pour obtenir une collection de sortie dont les enregistrements soient proches de :

```
db.res5j.findOne()
{
  "_id" : ObjectId("592d91222bb10509205a4f77"),
  "etablissement_lib" : "Aix-Marseille université",
  "cursus_lmd" : "D",
  "aca_etab_lib" : "Aix-Marseille"
}
```

- Visualiser le résultat dans la nouvelle collection **res5j**.