Info 834

Il n'y aura qu'un seul sujet pour l'ensemble des 8h de ce TP sur MongoDB. Cela ira de l'administration de la base de données, des commandes en vrac, de la programmation en Node.js et Python, puis finalement un passage à l'échelle avec le sharding et le clustering.

1. Installation de MongoDB

Je redonne la façon d'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 s'est bien faite, essayez de lancer 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*. Bien entendu, ne pas tuer l'application en fermant le terminal ou en faisant CTRL+C. Après dans un autre terminal, vous pouvez lancer le client avec *mongo*.

3. On insère de la donnée

Allez chercher sur l'EAD, le fichier communes CSV que vous trouverez dans Info 834. La solution compliquée serait de dire que nous lisons ligne par ligne en Python puis nous faisons des insert dans la base de données.

La solution, la plus simple, reste quand même de faire un *mongoimport*. Par exemple :

```
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 précédemment, nous avons un certain nombre de commandes de base :

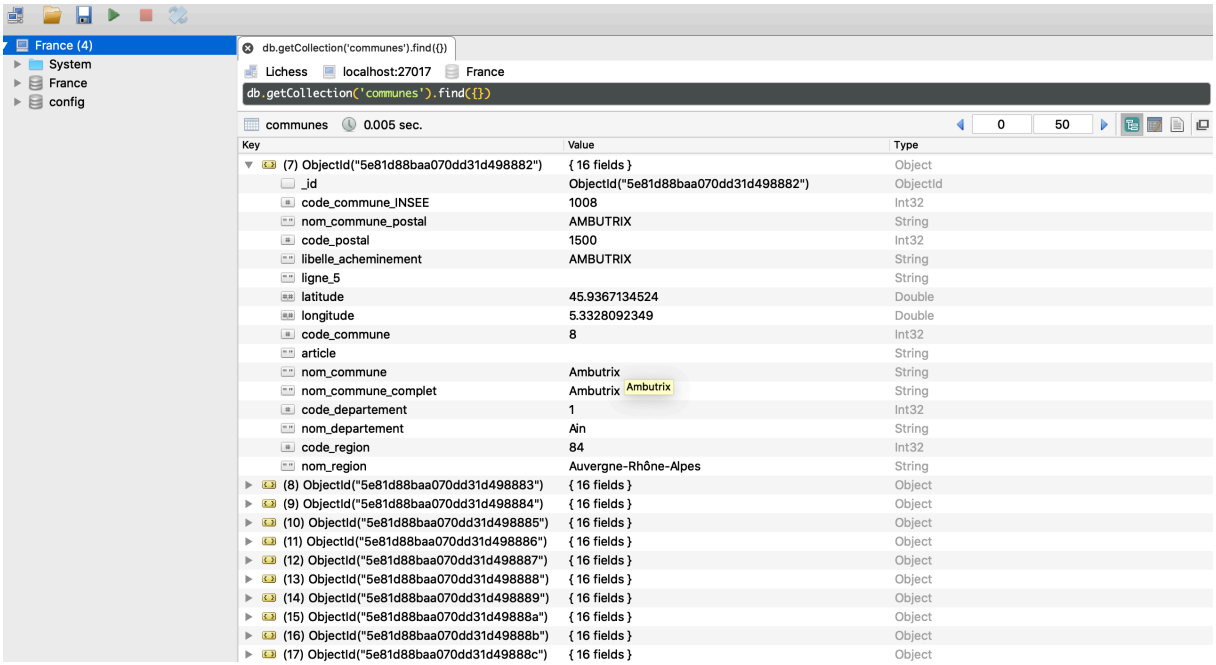
- *show databases*. Cela retourne l'ensemble des bases de données qui sont dans le répertoire data. Vous trouverez forcément les bases *admin, config* et *local*. Puis les bases de données que vous avez créées.
- *use France*. Cela vous permet de choisir la base de données XXX afin de n'avoir que les collections pour cette base de données.
- *show collections*. Cela retourne l'ensemble des collections qui est dans la base de données.
- *db.communes.count()*. Cela retourne le nombre d'enregistrements dans la collection XXX.
- *db.communes.find()*. Cela retourne les données, bien entendu s'il y a trop de données, il faut pagine et ne renvoie qu'une partie des données.
- *db.communes.find().pretty()*. Cela retourne les données en faisant en sorte que cela soit aussi bien présenté.
- *db*.drop(). Cela supprime toutes les données dans une collection.

5. L'outil de visualisation Robo 3T

Au lieu du client mongo de base, nous pouvons aussi utiliser Robo 3T qui est un GUI pour MongoDB. Il est possible de le télécharger à l'adresse suivante :
https://robomongo.org/download
(il existe pour Windows, Mac et Linux)

Installez-le. Prenez la base Communes, lancez le serveur pour MongoDB et ensuite connectez-vous par l'intermédiaire de Robo 3T.

Vérifiez que vous pouvez ajouter des documents et trouver le nombre d'éléments.



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 des données, mais cela va s'il s'agit d'une petite base de données. Si nous parlons de milliers de documents, il est préférable de penser à la pagination (ou curseur chez MongoDB). L'utilisation de cursor dans le Mongo shell (le client), c'est assez facile. Voici le code à écrire dans le client :

```
var myCursor = db.XXX.find( {} );

while (myCursor.hasNext()) {
  print(tojson(myCursor.next()));
}
```

Vous pouvez donc récupérer les pages suivantes par l'intermédiaire de .next(). Nous vous laissons essayer.

Note : si vous regardez bien, si vous faites sans le cursor, le Mongo shell vous propose les cursors puisqu'il y a *Type it for more*.

7. Un peu de benchmark

Écrivez un programme en Python ou Node.js 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 y 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 d'indexer ce champ pour que cet index soit stocké en mémoire vive afin d'accroître la vitesse.

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

Indexer un fichier est assez facile (et c'est à faire sur le Mongo shell, Python ou Node.js) :

```
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.

Prenez la base de données et ajoutez un index. Cherchez quel peut être les champs à indexer.

Finalement, rien n'interdit d'avoir plusieurs index en les séparant par des virgules :

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

Attention aussi par rapport aux index selon qu'ils doivent être uniques ou pas. Vous pouvez préciser que vous voulez un index unique en faisant de la sorte :

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

Regardez si une telle unicité est ici possible.

Maintenant, que vous avez votre index, relancez le programme de la question 6, et regardez si le calcul est plus rapide.

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 bien entendu de faire un update (ou upsert pour gagner en temps) sur le document et d'ajouter un de plus. Mais, étant donné que c'est quelque chose que nous allons devoir faire fréquemment, il serait mieux de disposer d'un moyen plus efficace. Cela s'appelle *$inc* dans MongoDB. Voyons un exemple sur une autre base, cela nous permettra de créer une base de données directement dans le Mongo shell.

Première chose : créons une nouvelle base de données. C'est (trop) simple dans Mongo shell, il suffit de donner le nom de la base de données (même si elle n'existe pas) quand vous faites *use*. Par exemple, *use Analytics.* Par contre si vous faites *show Databases*, vous ne voyez toujours votre base, parce qu'à l'heure, il n'y a pas de données dedans.

Nous allons créer un document par l'intermédiaire du Mongo shell :

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

Les trois petits points ont été ajoutés par Mongo shell car il a compris que nous n'avions pas terminé notre document. Il faut ensuite l'ajouter dans la collection :

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

Si maintenant nous faisons *show databases*, *Analytics* apparaît bien.

Le *nb* est bien de 1. Maintenant, on va modifier cette valeur très facilement :

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

db.freq.update({_id : ObjectId("5e81e0e11e95dfc6e9cd3611")}, {$inc:
{nb : 1}})
```

Et si maintenant, nous regardons le seul document que nous avons :

```
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 le document comme une mailing-list et chaque élément dans le tableau est en fait une des adresses email enregistrée pour cette mailing-list. Voyons ça sur une nouvelle base de données :

```
> use Mailing
switched to db Mailing
> list = {
... name : "Mailing-List 1",
... emails : []
... }
{ "name" : "Mailing-List 1", "emails" : [ ] }
> db.lists.insert(list)
WriteResult({ "nInserted" : 1 })
> db.lists.find().pretty()
{
    "_id" : ObjectId("5e81e5801e95dfc6e9cd3612"),
    "name" : "Mailing-List 1",
    "emails" : [ ]
}
> db.lists.update({_id : ObjectId("5e81e5801e95dfc6e9cd3612")},
... {$push : {emails : "Marc-Philippe.Huget@univ-smb.fr"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.lists.find().pretty()
{
    "_id" : ObjectId("5e81e5801e95dfc6e9cd3612"),
    "name" : "Mailing-List 1",
    "emails" : [
        "Marc-Philippe.Huget@univ-smb.fr"
    ]
}
>
```

Puis de la même façon, il est possible de retirer un élément d'une liste :

```
> db.lists.update({_id : ObjectId("5e81e5801e95dfc6e9cd3612")},
... {$push : {emails: "wrongaddress@404"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.lists.find().pretty()
{
    "_id" : ObjectId("5e81e5801e95dfc6e9cd3612"),
    "name" : "Mailing-List 1",
    "emails" : [
        "Marc-Philippe.Huget@univ-smb.fr",
        "wrongaddress@404"
    ]
}
> db.lists.update({_id : ObjectId("5e81e5801e95dfc6e9cd3612")},
{$pull : {emails : "wrongaddress@404"}})
```

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.lists.find().pretty()
{
    "_id" : ObjectId("5e81e5801e95dfc6e9cd3612"),
    "name" : "Mailing-List 1",
    "emails" : [
        "Marc-Philippe.Huget@univ-smb.fr"
    ]
}
>
```

Et il en ira de même pour Mongoose.

11. Les valeurs distinctes

Obtenir les valeurs distinctes pour une collection ne se fait pas forcément sur un index, nous pouvons le faire sur n'importe quel champ.

Par exemple, si nous voulons récupérer l'ensemble des communes pour notre région :

```
> db.communes.distinct("nom_commune", {code_region: 84})
```

Nous pouvons aussi obtenir l'ensemble des régions :

```
> db.communes.distinct("code_region")
```

12. Et si on se liait ?

La plupart des modèles que nous avons traité en Info 732 et jusqu'à présent en Info 834 était de simples modèles et il n'y avait pas de relations entre eux. Cela va être corrigé dès à présent :

Nous allons reprendre l'exemple de la mailing-liste mais cette fois au lieu de stocker l'adresse email, nous allons stocker le lien vers le document correspond à l'utilisateur.

Je crée donc ma collection Users :

```
> use Mailing
switched to db Mailing
> show collections
lists
> user1 = { name : "Doe",
... firstname: "John",
... email: "John.Doe@email.org"}
{ "name" : "Doe", "firstname" : "John", "email" :
"John.Doe@email.org" }
> db.users.insert(user1)
WriteResult({ "nInserted" : 1 })
> user2 = { name : "Claude",
... firstname : "Claude",
... email : "cloclo@email.org"}
{ "name" : "Claude", "firstname" : "Claude", "email" :
"cloclo@email.org" }
```

```
> db.users.insert(user2)
WriteResult({ "nInserted" : 1 })
> db.users.find().pretty()
{
    "_id" : ObjectId("5e81f5cd1e95dfc6e9cd3613"),
    "name" : "Doe",
    "firstname" : "John",
    "email" : "John.Doe@email.org"
}
{
    "_id" : ObjectId("5e81f6041e95dfc6e9cd3614"),
    "name" : "Claude",
    "firstname" : "Claude",
    "email" : "cloclo@email.org"
}
>
```

Maintenant, travaillons sur la mailing-liste :

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

Créez maintenant le programme en Node.js qui permet de récupérer en une seule fois la mailing liste et l'ensemble des utilisateurs. Chez mongoose et MongoDB, cela s'appelle .populate. A vous de rechercher, vous allez voir que c'est immédiat.

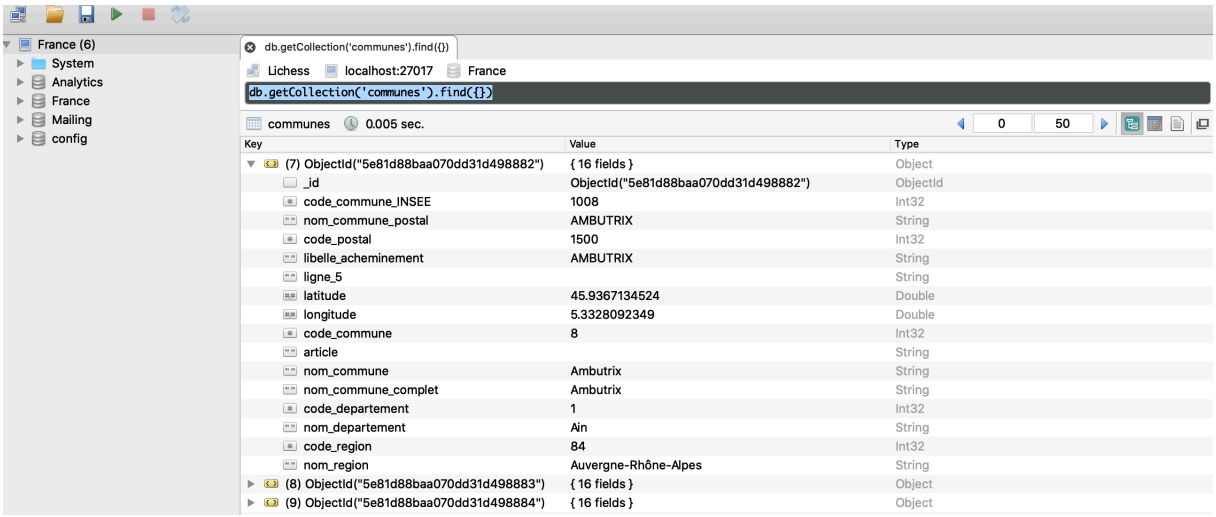
13. Montrez patte blanche

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 d'ajouter un utilisateur pour

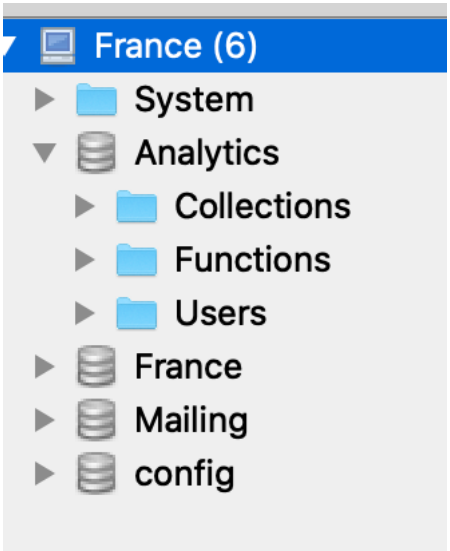
contrôler les accès et surtout éviter qu'une personne mal intentionnée vienne récupérer les données ou les effacer.

Il faut donc créer un utilisateur avec la commande *db.createUser*. C'est illustré en détails aussi à : https://docs.mongodb.com/manual/tutorial/create-users/

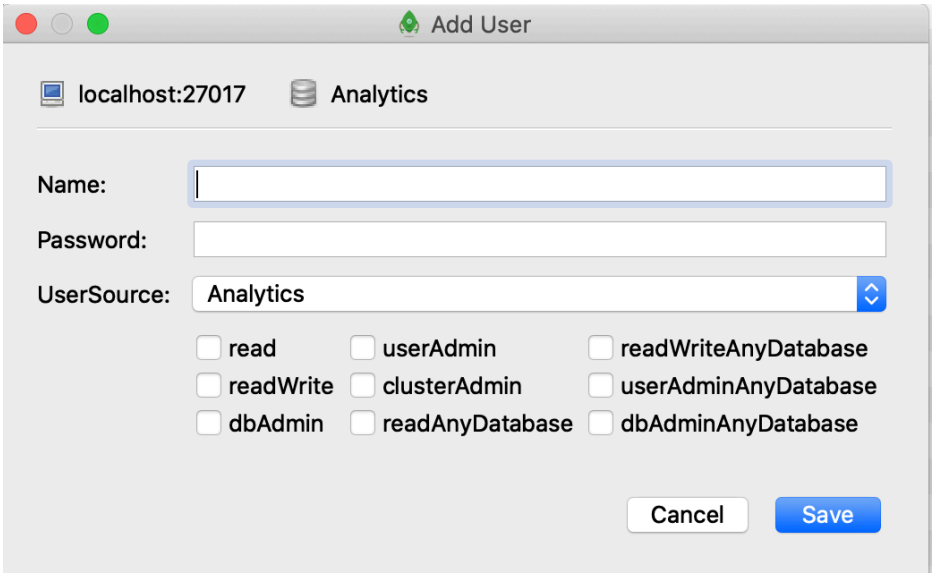
Mais pour une fois nous allons utiliser Robo 3T. Donc déjà rafraichissez la vue pour bien faire apparaître l'ensemble des bases que nous avons créées :



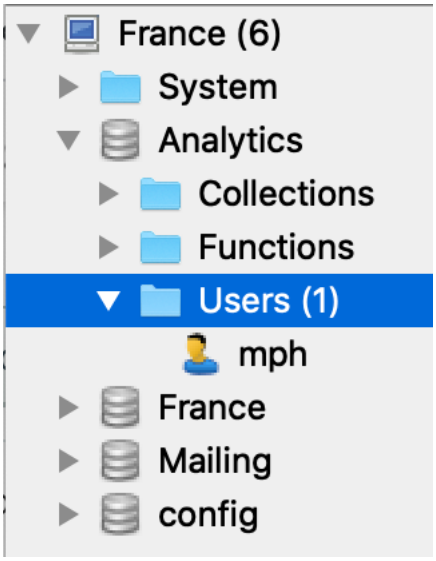
Choisissez la base de données : *Analytics.*



Vous faites un clic droit sur *Users* et ensuire *Add user*.



Créez un utilisateur avec son nom et son mot de passe, puis il faut ajouter un ou plusieurs rôles. Afin de tester, faites-en sorte que cet utilisateur ne peut que lire et pas écrire.



Il faut maintenant tester le fonctionnement de cet utilisateur.

Attention : par contre 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 :

```
> db = connect("localhost:27017/Analytics", "mph", "mph")
connecting to: mongodb://localhost:27017/Analytics
Implicit session: session { "id" : UUID("73065bce-1060-4fce-ba3a-
ddd86bd06ecc") }
MongoDB server version: 4.0.3
Analytics
> show collections
freq
```

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

Il faut maintenant se connecter avec ce nouvel utilisateur :

```
> db = connect("localhost:27017/Analytics", "mph", "mph")
connecting to: mongodb://localhost:27017/Analytics
Implicit session: session { "id" : UUID("cd77e84c-bbe3-4d78-81a5-
6996b17b0e1b") }
MongoDB server version: 4.0.3
Analytics
> db.freq.insert({url:"https://developpez.net", nb:1})
WriteCommandError({
    "ok" : 0,
    "errmsg" : "not authorized on Analytics to execute command {
insert: \"freq\", ordered: true, lsid: { id: UUID(\"cd77e84c-bbe3-
4d78-81a5-6996b17b0e1b\") }, $db: \"Analytics\" }",
    "code" : 13,
    "codeName" : "Unauthorized"
})
```

Nous obtenons bien ce que nous voulions, et nous aurions testé avec Node.js, c'est à peu près la même chose.

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

Les fonctions de map et reduce ressemblent en tout point à celles que nous avons vu pour Hadoop et elles sont écrite en JavaScript.

Commençons par voir comment écrire la fonction map :

```
> 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 génération 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
}
```

Nous obtenons le résultat après avoir appliquées les fonctions Map Reduce. Le *inputs* 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.

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

Pour commencer à voir le contenu, il suffit d'ajouter *.find()*.

Sur le même principe, tentez d'obtenir toutes les communes de la région Auvergne-Rhône Alpes qui commencent par un A.

15. 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 bien entendu 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, ce n'est pas plus compliqué, lancez un serveur MongoDB avec une base vide et faites :

```
mongorestore backup/
```

16. 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 ici :
https://media.mongodb.org/zips.json

Commencez par charger cette base dans votre MongoDB, pas la peine de vous réexpliquer comment, nous l'avons fait précédemment, si si.

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.

17. Les triggers

Les triggers sont une notion appréciée dans les bases de données en production afin de mettre en place des actions selon des situations données : faire un réassort sur un produit qui descend en dessous d'un niveau de stock par exemple. Nous allons travailler sur les triggers dans MongoDB. La mauvaise nouvelle pour vous est que les triggers n'existent pas nativement dans MongoDB. Nous allons devoir les mettre en place avec la notion de *Change Streams (*qui est dans Stitch, un add-on sur MongoDB pour ReplicatSet). Pour cela, vous disposez de change_stream dans pymongo (https://api.mongodb.com/python/current/api/pymongo/change_stream.html). Mettez en place un trigger pour être informé de la création ou de la suppression d'un user dans une collection users.
Cerise sur le gâteau : tentez d'envoyer un email à l'utilisateur (avec Mailgun) si celui-ci crée un compte ou le supprime.