INFO833

TP3 – PubSub avec Redis

Tanguy MARITON – Loïck COMBRIE

Polytech Annecy-Chambéry
IDU4

Partie 1 – Découverte

Le code est disponible sur le git suivant :

https://github.com/TanguyMrtn/Redis-with-geospatial-data

Question 1 : Insérer les différentes données dans des clés appropriées.

```
# Connexion à redis

import redis

r = redis.Redis(port=6379)

# Question 1
r.geoadd("Ville",49.8986514,2.2145979,"Amiens")
r.geoadd("Ville",50.1101392,1.7962242,"Abbeville")
r.geoadd("Ville",50.0033988,2.6340819,"Albert")
r.geoadd("Ville",50.1514695,2.2819405,"Doullens")
r.geoadd("Ville",49.9216618,2.4800254,"Corbie")
r.geoadd("Ville",49.647487,2.5486319,"Montdidier")
r.geoadd("Ville",49.6907257,2.7521182,"Roye")
r.geoadd("Ville",49.8726485,2.3500182,"Longueau")
r.geoadd("Ville",49.7529828,3.0523779,"Ham")

a="Technicien de maintenance"
b="Assistant chef de projet"
c="Conseiller en clientèle"
d="Consultant informatique"

r.lpush("Amiens",a,b,d,d)
r.lpush("Péronne",a)
r.lpush("Corbie",a,d)
r.lpush("Longueau",a)
r.lpush("Abbeville",b,d)
r.lpush("Roye",c)
r.lpush("Ham",c)
```

Tout d'abord on effectue la connexion à Redis. Ensuite, on ajoute les villes, avec leurs coordonnées, dans une clé « Ville ». Pour gérer l'aspect géographique, on utilise la fonction GEOADD de redis, dont voici le début de la documentation : « Adds the specified geospatial items (latitude, longitude, name) to the specified key. Data is stored into the key as a sorted set, in a way that makes it possible to later retrieve items using a query by radius with the GEORADIUS or GEORADIUSBYMEMBER commands. » Dans notre cas, la key est « Ville », et une ville que l'on ajoute est un name (membre) avec sa latitude et longitude. Comme l'indique la documentation, on peut faire des requêtes en précisant un rayon autour d'une ville (GEORADIUS), ce qui nous sera utile pour la suite. Ensuite, on créait des listes avec pour clé le nom d'une ville et en valeur les offres d'emploi (donc une liste d'offre d'emploi).

Question 2 : Récupérez les différentes données des différentes clés.

```
# Question 2
print(r.geopos("Ville","Amiens"))
print(r.geopos("Ville","Abbeville"))
print(r.geopos("Ville","Albert"))
print(r.geopos("Ville","Doullens"))
print(r.geopos("Ville","Corbie"))
print(r.geopos("Ville","Montdidier"))
print(r.geopos("Ville","Roye"))
print(r.geopos("Ville","Longueau"))
print(r.geopos("Ville","Ham"))

print(r.lrange("Amiens",0,-1))
print(r.lrange("Péronne",0,-1))
print(r.lrange("Corbie",0,-1))
print(r.lrange("Longueau",0,-1))
print(r.lrange("Abbeville",0,-1))
print(r.lrange("Roye",0,-1))
print(r.lrange("Ham",0,-1))
```

Pour récupérer les données géographiques, on utilise GEOPOS, en précisant la clé (« Ville ») et le membre dont on souhaite récupérer les coordonnées (« Amiens » par exemple).

Pour récupérer les données des listes d'offre d'emploi, on utilise LRANGE, en précisant la clé (« Amiens » par exemple pour les offres d'emploi à Amiens). On va de 0 à -1 pour la range, afin de récupérer l'entièreté de la liste. Voici le résultat :

Question 3 : Récupérez les villes à 35km ou moins d'Amiens.

```
# Question 3
print(r.georadiusbymember("Ville", "Amiens", 35,unit="km"))
```

On utilise la fonction GEORADIUSBYMEMBER, qui nous permet de récupérer les membres situés dans un rayon autour d'un membre spécifique. Ici, on travaille sur la clé « Ville », on précise que le membre central de notre requête est Amiens, et que l'on veut les autres membres situés dans un rayon de 35 km autour de Amiens (membre central de la requête). Le résultat est « [b'Amiens', b'Longueau', b'Corbie', b'Doullens'] ».

Question 4 : Imaginez une méthode (sans la mettre en place) qui permettrait de récupérer des offres dans un rayon donné autour d'une ville.

On récupère les villes à x km autour d'une certaine ville (commande question 3 par exemple), et pour chaque résultat, donc chaque ville qui est dans le rayon de x km, on récupère les offres (avec r.lrange(Ville,0,-1)). Si on stocke les données comme ce que l'on a fait précédemment, il est possible d'appliquer cette méthode.

Partie 2 – Recherches additionnelles

Question 1 : Les principaux SGBD relationnels disposent-ils de fonctionnalités pour du requêtage Géospatial ? Si oui, dans quel cadre l'utilisation d'un SGBD relationnel peut-être intéressante ?

Oui, les SGBD relationnels disposent de fonctionnalités pour du requêtage géospatial, par exemple : sur MySQL --> Type GEOMETRY, sur PostGreSQL --> PostGIS... A partir des SGBD relationnels on peut faire des jointures facilement, des grosses requêtes facilement, tout en utilisant un langage qui est beaucoup plus expressif que celui de redis par exemple, est donc beaucoup plus compréhensible.

Question 2 : Un de vos collègues vous dit qu'Elasticsearch, accompagné de Kibana, pourrait apporter une grande plus-value. Qu'en pensez-vous ?

C'est vrai car Elasticsearch peut gérer les données géospatiales, et on a l'outil de datavisualisation Kibana, directement branché sur la base ES, qui nous permet de facilement avoir des visualisations, dont des visualisations géographiques. De plus, Elasticsearch supporte les requêtes SQL (https://www.elastic.co/fr/what-is/elasticsearch-sql) donc on a le même avantage qui si on utilisait un SGBD relationnel, avec l'outil de datavisualisation Kibana en plus.

Partie 3 – PUB/SUB avec Redis

Question 1 : Supposez que nous voulons développer une application permettant à un demandeur d'emploi de s'abonner à un flux lui permettant de donner le nom d'une ville et de recevoir toutes les annonces à 30 km de cette ville. Comment exploiteriez-vous le PUBSUB redis pour le faire?

Premièrement, voici une explication rapide de ce qu'est PUB/SUB : « In software architecture, publish–subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers' express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are. »

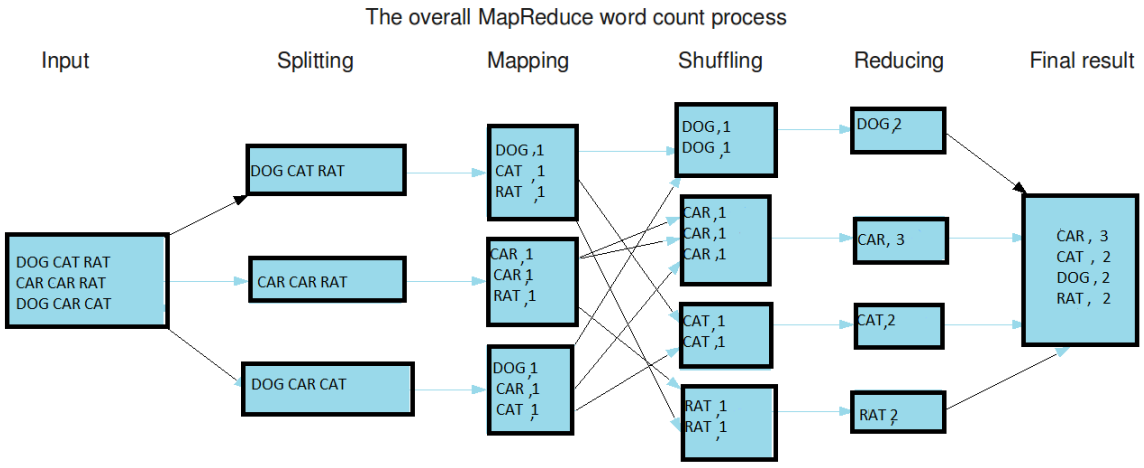
Redis permet d'utiliser PUB/SUB, dont voici la documentation: « SUBSCRIBE, UNSUBSCRIBE and PUBLISH implement the Publish/Subscribe messaging paradigm where (citing Wikipedia) senders (publishers) are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterized into channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology. »

Dans notre cas, le demandeur d'emploi est le subscriber, le flux un chanel, un message une offre d'emploi et les personnes / entités postant des offres d'emploi les publishers. Voici comment il est possible de répondre au problème en utilisant le PUB/SUB de Redis : si on garde la structure précédente (toutes les offres d'emploi sont associées à des villes), alors on aura un chanel par ville. Lorsque le demandeur d'emploi va préciser la ville et le rayon pour récupérer les offres d'emploi, l'application va inscrire le demandeur d'emploi sur les chanels des villes se situant à 30km de la ville précisée par le demandeur (on peut récupérer les villes comme ce que l'on a fait précédemment, partie 1 question 3). Le demandeur sera donc inscrit aux chanels des villes (un chanel par ville) se situant à 30km de la ville qu'il souhaite. Il recevra donc que les offres d'emploi de cette zone.

Question 2 : Redis est fréquemment utilisé comme cache de mémoire pour les applications distribuées. Supposez le scénario suivant. Une application de traitement de la donnée à grande échelle génère des données qu'elle doit partager avec un certain nombre d'autres composants. Comment utiliseriez-vous Redis et pubsub pour le faire. Comment implémenteriez-vous le projet map-reduce que vous avez fait le semestre dernier avec redis et le PubSub (indice il est possible de faire la totalité du projet en moins de 50 lignes de code et en une heure de travail).

Pour l'application de traitement, c'est très simple. L'application publie ses données sur un ou plusieurs chanels (en fonction de ce que l'on souhaite), et les composants s'inscrivent sur les chanels adéquats afin qu'ils récupèrent les données qui les intéressent.

Pour Map-Reduce, on rappelle que ça consiste à répartir le travail sur les différents nœuds du cluster (map), puis à organiser et réduire les résultats fournis par chaque nœud en une seule réponse cohérente à une requête. Une approche par pub/sub peut alors prendre tout son sens, elle va nous permettre de transmettre facilement nos messages entre les différentes entités. Les chanels PUB/SUB de redis sont notre « flow » de notre projet MapReduce, ils permettent de faire circuler les différents messages. Notre projet suivait l'architecture MapReduce suivante :



On envoie un premier message dans le chanel « initial-input ». Une première entité chargée du splitting (on peut par exemple utiliser un Thread), inscrite sur ce chanel, va récupérer les données en entrée (un message par nouvelles données en entrée, par exemple le contenu d'un fichier texte), séparer chaque contenu d'un message en n partie (n étant le nombre de mapper souhaitée) et ensuite publier n message à un chanel « mapper-input ». Ensuite, une deuxième entité chargée du mapping, inscrite sur ce chanel, va récupérer les messages et lancer n mapper, avec chacun pour données à traiter le contenu d'un des messages (chaque mapper s'occupe d'un message différent des autres). Chaque mapper retourne une valeur (ensemble clé valeur par exemple), et l'entité va créer un message pour chaque résultat des mappers, qu'elle va publier dans un chanel « mapper-output ». Ensuite, une troisième entité chargée du shuffling, inscrite sur ce chanel, va récupérer les messages et regrouper toutes les apparitions de chaque mots (comme montré sur le schéma). Elle publiera un message par regroupement sur un chanel « shuffler-output ». Ensuite, une entité chargée du reducing, inscrite sur ce chanel, va lancer un reducer par message, avec chacun pour données à traiter le contenu d'un des messages (chaque reducer s'occupe d'un message différent des autres). Elle publiera un message par résultat sur un chanel « reducer-output ». Pour finir, une entité chargée du résultat finale, inscrite sur ce chanel, va regrouper tous ces messages et donner le résultat.