Réplication et Sharding

Romain Tavenard

1 Réplication (utilisation de *Replica Set*)

Vous allez simuler l'existence d'un *cluster* de 3 machines (sauf que, dans votre cas, les trois noeuds seront situés sur la même machine : la vôtre). Pour cela, lancez un client MongoDB à l'aide de la commande :

```
mongo --nodb
```

Le paramètre `--nodb` permet d'indiquer au client MongoDB de ne pas se connecter, pour le moment, à une quelconque base. Pour créer un cluster sur lequel répliquer des données, on utilise la classe `ReplSetTest` :

```
> repSet = new ReplSetTest({"nodes" : 3})
> repSet.startSet()
> repSet.initiate()
```

Cette commande, si elle a fonctionné, a dû lancer 3 processus `mongod`. Pour le vérifier, lancez la commande suivante dans un nouveau terminal (dans le shell) :

```
ps all
```

Si la commande précédente renvoie trop de résultats, on peut limiter l'affichage des résultats aux lignes qui contiennent le terme `mongo` :

```
ps all | grep mongo
```

1. Dans la suite, tous les messages d'avertissement correspondant à ce *Replica Set* seront affichés dans la console MongoDB dans laquelle vous avez créé le cluster, rendant les sorties difficilement lisibles. Gardez donc cette fenêtre ouverte dans un coin sans y toucher et ouvrez un nouveau client MongoDB en lui indiquant de se connecter au processus tournant sur le port 31000 en utilisant la base test.

Vous devriez obtenir une invite de commande du type :

```
testReplSet:PRIMARY>
```

Cela vous indique que vous êtes connecté au noeud maître (`PRIMARY`) d'un *Replica Set*. Pour obtenir plus d'informations sur ce *Replica Set*, exécutez la commande suivante :

```
testReplSet:PRIMARY> db.isMaster()
```

Lorsque vous êtes connecté, comme c'est votre cas, au noeud maître d'un *Replica Set*, vous pouvez effectuer les opérations usuelles d'insertion / recherche de données dans les bases du *Replica Set* de manière transparente.

2. Insérez des documents dans une collection `coll` de la base courante à l'aide de la commande :

```
testReplSet:PRIMARY> for(var i=0; i < 100000; i++) {db.coll.insert({count: i})}
```

Vous pouvez également avoir une vision de l'état des noeuds esclaves du *Replica Set* à l'aide de la commande :

```
testReplSet:PRIMARY> rs.printSlaveReplicationInfo()
```

Observez notamment que lors de votre dernière insertion de données dans une collection de la base test, chaque noeud a été instantanément mis à jour. Pour finir, stoppez votre cluster en entrant la commande suivante dans la fenêtre dans laquelle vous l'aviez créé :

```
> repSet.stopSet()
```

2 Répartition (*Sharding*)

Vous allez simuler l'existence d'un cluster de 3 machines (sauf que, dans votre cas, les trois noeuds seront situés sur la même machine : la vôtre). Pour cela, lancez un client MongoDB à l'aide de la commande (ou utilisez la fenêtre précédemment ouverte avec la même commande) :

```
mongo --nodb
```

Le paramètre `--nodb` permet d'indiquer au client MongoDB de ne pas se connecter, pour le moment, à une quelconque base. Pour créer un cluster sur lequel répartir des données, on utilise la classe `ShardingTest` :

```
> cluster = new ShardingTest({"shards" : 3, "chunksize" : 1})
```

Ignorez pour le moment l'attribut `chunksize`. Cette commande, si elle a fonctionné, a dû lancer 3 processus `mongod` et un processus `mongos`. Pour le vérifier, lancez la commande suivante dans un nouveau terminal (dans le shell) :

```
ps all
```

Si la commande précédente renvoie trop de résultats, on peut limiter l'affichage des résultats aux lignes qui contiennent le terme mongo :

```
ps all | grep mongo
```

On obtient une sortie du type :

```
1797 ttys001   49:52.51 mongod
 1804 ttys002    0:01.02 mongo
54243 ttys007    0:00.13 mongo -nodb
54244 ttys007    0:02.54 mongod --port 30000 --dbpath /data/db/test0 --setParameter enableTe
54245 ttys007    0:02.05 mongod --port 30001 --dbpath /data/db/test1 --setParameter enableTe
54246 ttys007    0:02.06 mongod --port 30002 --dbpath /data/db/test2 --setParameter enableTe
54247 ttys007    0:01.20 mongos --port 30999 --configdb localhost:30000 --chunkSize 1 --setP
54274 ttys008    0:00.00 grep mongo
```

Par défaut, les processus `mongod` sont créés sur les ports 30000, 30001 et 30002 et le processus `mongos` sur le port 30999.

3. Dans la suite, tous les messages d'avertissement correspondant à ce cluster seront affichés dans la console MongoDB dans laquelle vous avez créé le cluster, rendant les sorties difficilement lisibles. Gardez donc cette fenêtre ouverte dans un coin sans y toucher et ouvrez un nouveau client MongoDB en lui indiquant de se connecter au processus mongos (dont vous devez connaître le port) en utilisant la base test.

On remarque alors que le prompt indique que l'on est connecté à un cluster :

```
mongos>
```

Dans la suite, vous allez interagir avec le processus mongos d'une manière transparente (quel que soit le nombre de *shards*). Toutefois, vous pourrez toujours obtenir de l'info sur la structure du cluster utilisé à l'aide de la commande :

```
mongos> sh.status(true)
```

4. Pour l'instant, vous n'avez pas inséré de collection dans la base `test`, elle n'apparaît donc pas dans la liste des bases présentes sur le cluster. Pour y remédier, insérez des données dans une collection `users`, vérifiez le nombre de documents insérés dans la collection puis répétez la commande `status` :

```
mongos> for(var i=0 ; i<100000 ; i++) {
  db.users.insert({"username" : "user" + i, "created_at": new Date()}) ;
}
```

Vous devez maintenant voir la base `test` apparaître, avec un *shard* primaire attribué au hasard. La propriété `partitioned` de la base étant fixée à `false`, cela signifie que toutes les données sont pour l'instant stockées sur le *shard* primaire.

Pour permettre la répartition automatique des données dans le *cluster*, il faut :

a. Autoriser le *sharding* sur la base :

```
mongos> sh.enableSharding("nomDeLaBase")
```

b. créer un index sur la collection à distribuer : cet index sera celui sur lequel la répartition sera faite (ainsi les valeurs consécutives pour cet index auront plus de chances de se retrouver sur le même *shard*)

c. Demander de répartir la collection en utilisant l'index créé (le champ correspondant sera appelé clé de répartition, ou *shard key*) :

```
mongos> sh.shardCollection("nomDeLaBase.nomDeLaCollec",
                           {"cleDeRepartition" : 1})
```

5. Mettez en oeuvre la répartition des données de la collection `users` à travers le cluster en utilisant `username` comme clé de répartition. Observez le statut du *cluster* à l'issue de cette manipulation.

Il est possible que vous n'ayez pas à faire ce qui est indiqué dans le paragraphe suivant, dépendant de la configuration de MongoDB sur votre machine. Si dans vos rapports de statut du cluster, vous voyez que la propriété `Currently enabled` de la section `Balancer` est à `false`, alors vos données ne seront pas réparties à travers les *shards* de votre *cluster*. Vous devrez donc activer le *balancer* (le processus qui se charge de l'équilibrage entre les *shards*) sur la collection visée :

```
mongos> sh.enableBalancing("test.users")
mongos> sh.setBalancerState(true)
mongos> sh.status(true)
```

Un nouvel appel à la fonction `sh.status()` doit vous montrer que les *chunks* (les morceaux de votre collection) ne se trouvent plus maintenant sur le *shard* primaire uniquement mais sont bien répartis sur tous les *shards*.

6. Affichez le contenu correspondant à l'utilisateur `"user12345"` et vérifiez que la syntaxe utilisée habituellement pour ce type de requête fonctionne ici également.
7. Affichez les détails du déroulement de la requête à l'aide de la fonction `explain()`. Combien de *shards* sont mobilisés pour cette requête ? S'agit-il du/des *shard(s)* que vous auriez pu prévoir à la lecture du statut de votre *cluster* ?
8. Qu'en est-il si vous souhaitez récupérer l'ensemble des documents de la collection ?
9. Notez (pour plus tard) la sortie de la commande suivante :

```
mongos> db.users.getShardDistribution()
```

10. Insérez de nouveau des données dans la collection `users` et observez l'évolution du nombre de *chunks* et leur répartition à travers les *shards*. Combien de *chunks* ont été impactés par cette insertion de données ?

```
mongos> for(var i=100001 ; i<110000 ; i++) {
  db.users.insert({"username" : "user" + i, "created_at": new Date()}) ;
}
```

11. Répétez la commande de la question 9 et observez que la répartition du nombre de documents par *chunk* a évolué. À votre avis, pourquoi le serveur MongoDB n'a-t-il pas effectué un équilibrage parfait des documents à travers les *chunks* ?

En début de ce sujet de TD, il vous a été indiqué, à la création du *cluster*, d'affecter à l'attribut `chunksize` la valeur 1, ce qui signifie qu'un *chunk* peut, au plus, contenir 1Mo de données.

12. Changez cette limite à l'aide de la commande suivante :

```
mongos> use config
mongos> db.settings.save({"_id": "chunksize", "value": 32})
mongos> use test
```

Le nombre de *chunks* a-t-il évolué ? Que se passe-t-il si vous répétez l'opération d'insertion de données décrite à la question 10 ?

Pour finir, stoppez votre *cluster* en entrant la commande suivante dans la fenêtre dans laquelle vous l'aviez créé :

```
> cluster.stop()
```

3 Pour aller plus loin

Si vous deviez, en pratique, mettre en place un réel cluster de réplication / *sharding*, les pages d'aide dédiée de MongoDB vous seraient d'une aide précieuse :

— https://docs.mongodb.org/manual/core/replica-set-architectures/
— https://docs.mongodb.org/manual/administration/sharded-cluster-deployment/