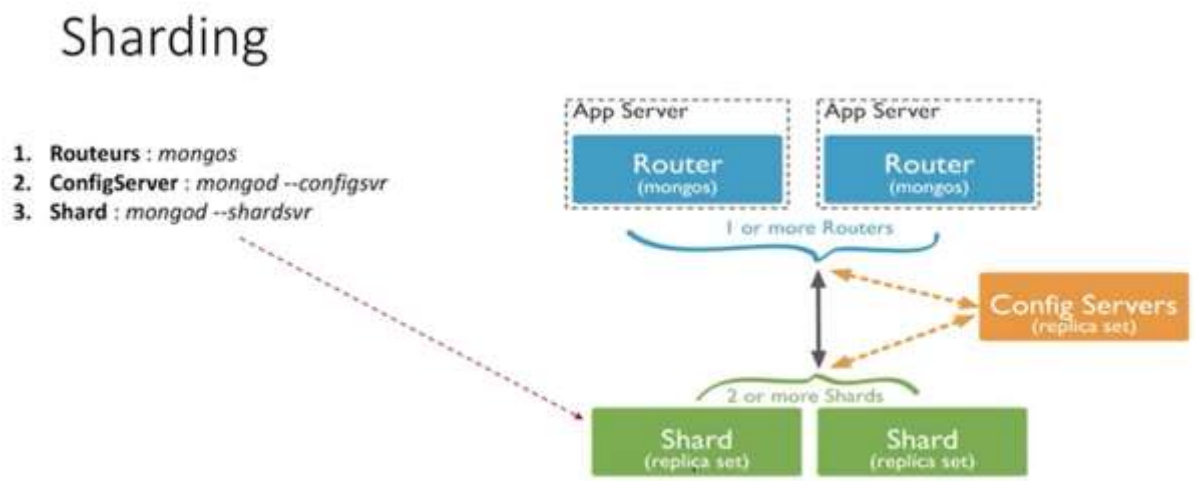
TP : NoSQL

Sharding



1- Création des ConfigServer (ici 2 ConfigServer)

Pour cela on doit :

1-1- créer 2 répertoires /data/config1 et /data/config1

1-2- Lancer deux serveurs de configuration (2 config Serveur) (**--configsvr**) qui vont fonctionner ensemble en un seul ReplicaSet (de nom **configReplSet**) et qui seront liés aux 2 répertoires créés.

On utilise mongod avec paramètre configsvr et chacun a son port, pour permettre d'interroger les shards.
Les config server doivent avoir le même replicaset.

```
mongod --configsvr --replSet configReplSet --port 27019 --dbpath /data/config1
mongod --configsvr --replSet configReplSet --port 27020 --dbpath /data/config2
```

2- Création des shards

2-1-Créer 2 repertoire /data/sh1 et /data/sh2

2-2-Créer deux shard pour tester la distribution.

Pour chaque shard, le paramètre --shardsvr est nécessaire pour permettre son intégration. Les shards ont chacun son **replicaset (sh1 et sh2, pour faire replication sur chacun), et son port (27031 et 27032).**

on peut avoir plusieurs shards

```
mongod --shardsvr --replSet sh1 --port 27031 --dbpath /data/sh1
mongod --shardsvr --replSet sh2 --port 27032 --dbpath /data/sh2
```

3-Relier les serveurs

On doit relier tous ces serveurs les un avec les autres :
On exécute mongo.exe pour lancer une console d'administration et pour aussi évaluer (--eval) une commande. Cette commande est l'initialisation d'un ReplicaSet sur le port 27019 dans notre cas qui est le Config Server, pour lequel je vais directement aller chercher mon deuxième serveur qui est le Config Server 27020.
L'option **--eval** permet de faire passer la commande **rs.initiate()** directement au serveur puis de récupérer la main sur la console.

```
mongo --port 27019 --eval "rs.initiate()";rs.add(localhost :27020 ")" ;
```

puis j initialise pour le serveur 27031 et 27032

```
mongo --port 27031 --eval "rs.initiate()"
mongo --port 27032 --eval "rs.initiate()"
```

Remarque : 27020 et 27019 ont même replicaset

3-Lancer le routeur :

3-1-Lancer le routeur mongos pour lequel je vais aller chercher le *ReplicaSet* du *ConfigServer* qui est là, avec le port de 27019. Le port du routeur est 27017.

Avec « ; » on peut ajouter la liste de tous les Config Servers qui correspondent à notre replica set.

```
mongos --configdb configReplSet/localhost:27019 --port 27017
```

3-2-on peut rajouter chacun des serveurs du ReplicaSet, en les séparant par des virgules :

```
configReplSet/localhost:27019,localhost:27020,localhost:27021
```

3-3-Les *shards* peuvent alors être ajoutés les uns après les autres au niveau du mongos en mode console :

On lance une nouvelle fenêtre avec mongo

```
mongo --port 27017
```

```
sh.addShard( "sh1/localhost:27031");
```

```
sh.addShard( "sh2/localhost:27032");
```

4-Distribution de la base de données

L'architecture de distribution est mise en place. Il suffit maintenant de définir la collection que l'on veut distribuer. Pour cela, nous allons créer une base « testDB » et une collection « test ».

```
use testDB;

--on va autoriser la sharding sur cette base de donnée

sh.enableSharding("testDB");

--on crée une collection de nom test sur cette base

db.createCollection("test");

-- Créer un petit index qui va nous permettre au niveau de MongoDB de connaître le placement des données, et de trier directement les données, puis de rajouter un Sharding sur cette collection,  sur l'identifiant de notre collection qui ici est  _id,

db.test.createIndex({"_id":1});

sh.shardCollection("testDB.test",{"_id":1});

exit
```

La collection **test** est maintenant **distribuée automatiquement sur nos deux shards,** les documents sont placés dans les *chunks* en fonction de leur valeur d'identifiant "_id", un tri de ces valeurs est effectué pour déterminer le placement

Remarque : On ne se connecte jamais au shard, directement, sinon vous aurez des problèmes de chunks, vous n'aurez pas de Sharding.

Nous pouvons maintenant importer les données des restaurants dans cette base de **test**, le port du routeur est 27017

```
mongoimport --db testDB --collection test --port 27017   restaurants.jsonf
mongo --port 27017 --eval "sh.status()"
```

https://s3-eu-west-1.amazonaws.com/course.oc-static.com/courses/4462426/restaurants.json_.zip

Résultat sh.status()" :

```
--- Sharding Status ---
```

```
sharding version: {
"_id" : 1,
"minCompatibleVersion" : 5,
"currentVersion" : 6,
"clusterId" : ObjectId("5967a3f7695a5111518224c6")
}
shards:
--ici on a 2 shard
{  "_id" : "sh1",  "host" : "rs0/local:27031",  "state" : 1 }
{  "_id" : "sh2",  "host" : "rs1/local:27032",  "state" : 1 }
active mongoses:
"3.4.3" : 1
autosplit:
Currently enabled: yes
balancer:
Currently enabled:  yes
Currently running:  no
      Balancer lock taken at Thu Jul 13 2017 18:46:47 GMT+0200 (CEST) by
ConfigServer:Balancer
Failed balancer rounds in last 5 attempts:  0
Migration Results for the last 24 hours:
      2 : Success
databases:
{  "_id" : "testDB",  "primary" : "rs0",  "partitioned" : true }
      testDB.test
            shard key: { "_id" : 1 }
            unique: false
            balancing: true
```

DB partionnée

On utilise ID comme clée de prtitionnement

```
        chunks:
                sh1     2
                sh2     2

        { "_id" : { "$minKey" : 1 } } -->> { "_id" :
ObjectId("5967a455b6c0223b91eefbd1") } on : rs1 Timestamp(2, 0)

        { "_id" : ObjectId("5967a455b6c0223b91eefbd1") } -->> { "_id" :
ObjectId("5967a455b6c0223b91eeffba") } on : rs1 Timestamp(3, 0)

        { "_id" : ObjectId("5967a455b6c0223b91eeffba") } -->> { "_id" :
ObjectId("5967a455b6c0223b91ef01af") } on : rs0 Timestamp(3, 1)



        { "_id" : ObjectId("5967a455b6c0223b91ef01af") } -->> { "_id" : { "$maxKey" : 1
} } on : rs0 Timestamp(1, 4)
```

On a 2 chunks sur sh1 et 2 sur sh2

Nous pouvons constater dans la partie « **databases** » que la collection *testDB.test* est composée de 4 *chunks*, répartis uniformément sur les deux *shards*.

enableSharding ne marche **que** sur une collection préalablement **vide**. Toute collection déjà existante ne peut être distribuée. Il faudra ainsi importer les données dans la collection « *shardée* ».

Stratégie de distribution (partie facultative)

Nous avons vu comment créer une architecture de distribution et comment distribuer une collection. Maintenant, il serait intéressant de savoir comment choisir une stratégie de distribution appropriée à nos données, afin de fournir les meilleures performances pour nos requêtes.

Requêtes par intervalles

Jusqu'ici, nous nous sommes contentés de choisir l'identifiant « **_id** » pour distribuer les données. Par défaut, l'indexation est basée sur un index non-dense triant les données. De fait, les requêtes par intervalles de valeurs sont optimisées. Il est donc possible de choisir une autre clé pour les documents de la collection comme stratégie de distribution.

Pour profiter des requêtes par intervalles, nous pourrions créer une « clé de sharding » sur le code postal, la clé : *zipcode*. Ainsi, toute requête cherchant à retrouver les restaurants associés à ce code postal ou une plage de codes postaux seront efficaces.

Tout changement de stratégie de sharding impose la création d'une nouvelle collection. On ne peut changer la distribution d'une collection déjà « shardée ».

-Connecter sur
Mongo - -port 2707
- Utiliser la base testdb
Use testDB

- créer une nouvelle collection de nom test2

```
db.createCollection("test2");
```

- créer un index, cette fois sur address.zipcode

```
db.test.createIndex({"address.zipcode" : 1});
```

-on va créer un shardin sur test et test2

```
sh.shardCollection("testDB.test", {"address.zipcode" : 1});
```

```
sh.shardCollection("testDB.test2", {"address.zipcode" : 1});
```

Requêtes par zones

Le *zipcode*, c'est bien, mais comment mieux maîtriser la répartition des restaurants sur les différents *chunks* ? En effet, la technique précédente ne permet pas de contrôler la répartition physique des *chunks* et les intervalles de valeurs de ceux-ci. Le *sharding* par zone permet d'associer à chaque *shard* une zone. Tous les restaurants de cette zone seront alors alloués à un serveur dédié. L'avantage est double :

1. **Optimiser** toutes **requêtes** liées à une **zone** (filtrage ou agrégation par quartier).
2. Définir un **serveurphysique** dans un *datacenter***proche** du client (zone géographique), correspondant à des requêtes de proximité.

Pour cela, il faut tout d'abord définir ces zones. Une zone est composée d'une plage de valeur et d'un « tag ». Nous pouvons choisir sur un intervalle de codes postaux correspondant aux villes :

```
sh.addTagRange("testDB.test2",
   {"address.zipcode": "10001"}, {"address.zipcode":  "11240"},
   "NYC")
sh.addTagRange("testDB.test2",
   {"address.zipcode": "94102"}, {"address.zipcode":  "94135"},
   "SFO")
```

L'avantage est de pouvoir associer des *shards* physiquement proches des utilisateurs dans chaque ville pour réduire la latence du réseau. Cela peut avoir également un intérêt si l'on souhaite stocker des données dans un cluster sécurisé.

Attention toutefois à la surcharge d'un *shard* suite à une zone mal dimensionnée. Nous pouvons découper une zone sur deux tags.

```
sh.addTagRange("testDB.test2",
   {"address.zipcode": "10001"}, {"address.zipcode": "10281"},
```

```
    "NYC-Manhattan")


sh.addTagRange("testDB.test2",
    {"address.zipcode": "11201"}, {"address.zipcode": "11240"},
    "NYC2-Brooklyn")
```

Pour information, la plage de valeurs inclut le min (10001), mais pas le max (10281). Maintenant que les tags sont associés à des plages de valeurs, il suffit d'associer à chaque tag un shard. Un tag peut être associé à plusieurs shards.

```
sh.addShardTag("sh1", "NYC")
sh.addShardTag("sh2", "NYC")
sh.addShardTag("sh2", "SFO")
```

Automatiquement, les restaurants seront placés dans les *chunks* des *shards* désignés en fonction de leurs codes postaux. Il faudra toutefois rester vigilant à la répartition de charge et le dimensionnement des *shards*. Pour cela, vous pouvez consulter la répartition des *chunks* d'un tag dans le réseau :

```
use config;
db.shards.find({ "tags" : "NYC" });
```

Requêtes par catégories

Et si nous faisions des requêtes ciblées sur un certain type de valeurs comme le quartier (*borough*) ? Il n'y a dans ce cas pas de plages de valeurs possibles, uniquement des restaurants correspondant à un quartier donné. Un simple partitionnement par hachage suffirait amplement à la segmentation des données.

MongoDB offre la possibilité de faire un hachage (via la fonction *md5*) sur les valeurs de la clé de sharding. L'avantage est soit de pouvoir **contrôler le placement** de tous les documents partageant la même valeur, soit de faire une **répartition plus uniforme** des documents sur l'ensemble des shards.

```
sh.shardCollection( "testDB.test3", { "borough" : "hashed" } )
```

Attention, le routeur n'est alors efficace que si l'on utilise des requêtes d'égalité sur le quartier. Les requêtes par intervalles ne permettent pas d'effectuer un routage sur les *shards*. De fait, la requête est transmise à tous les *shards* par *broadcast*.

Index et performances

Bien entendu, nous ne pouvons choisir qu'une seule stratégie de distribution pour optimiser nos requêtes. Mais MongoDB offre la possibilité d'interroger n'importe quelle clé. Comment éviter alors le *broadcast* lorsque la clé de *sharding* n'est pas utilisée ?

La collection « *testBD.test* » est distribuée sur l'identifiant des restaurants « *_id* ». Je souhaite maintenant exécuter la requête suivante :

```
db.test.find({"address.street" : "3 Avenue"}).explain()
{
  "queryPlanner" : {
   "mongosPlannerVersion" : 1,
   "winningPlan" : {
    "stage" : "SHARD_MERGE",
    "shards" : [
     {
       "shardName" : "sh1",
       "connectionString" : "sh1/local:27030",
       "serverInfo" : {
         "host" : "local",
         "port" : 27031,
         "version" : "3.4.3",
         "gitVersion" : "f07437fb5a6cca07c10bafa78365456eb1d6d5e1"
       },
       "plannerVersion" : 1,
       "namespace" : "testDB.test",
       "indexFilterSet" : false,
       "parsedQuery" : {
         "address.street" : {
```

```
        "$eq" : "3 Avenue"
      }
    },
    "winningPlan" : {
      "stage" : "SHARDING_FILTER",
      "inputStage" : {
        "stage" : "COLLSCAN",
        "filter" : {
          "address.street" : {"$eq" : "3 Avenue"}
        },
        "direction" : "forward"
      }
    },
    "rejectedPlans" : [ ]
  },
  {
    "shardName" : "sh2",
    "connectionString" : "sh2/local:27031",
    "serverInfo" : {
      "host" : "local",
      "port" : 27032,
      "version" : "3.4.3",
      "gitVersion" : "f07437fb5a6cca07c10bafa78365456eb1d6d5e1"
    },
    "plannerVersion" : 1,
    "namespace" : "testDB.test",
    "indexFilterSet" : false,
```

```
        "parsedQuery" : {
          "address.street" : {
            "$eq" : "3 Avenue"
          }
        },
        "winningPlan" : {
          "stage" : "SHARDING_FILTER",
          "inputStage" : {
            "stage" : "COLLSCAN",
            "filter" : {
              "address.street" : {
                "$eq" : "3 Avenue"
              }
            },
            "direction" : "forward"
          }
        },
        "rejectedPlans" : [ ]
      }
    ]
   }
  },
  "ok" : 1
}
```


La fonction "**.explain()**" appliquée à une requête permet de consulter le plan d'exécution généré. Ici, le plan « *WinningPlan* » possède une étape (stage) « *SHARD_MERGE* » qui effectue la fusion des résultats provenant de deux « *COLLSCAN* » (« *Collection Scan* »), en

l'occurrence un scan complet de la collection pour chaque *shard*. Cela correspond à une lecture complète du contenu des chunks, ce qui n'est pas très efficace.

Nous allons créer un index sur la rue :

```
db.test.createIndex({"address.street" : 1})
```

Après avoir testé à nouveau la requête, l'index est utilisé car l'étage passe en « *IXSCAN* ». Ce qui va permettre d'accéder directement aux restaurants de la 3° avenue sans avoir à parcourir toute la base de données. Bien sûr, comme ce n'est pas la clé de *sharding*, tous les *shards* effectuent cette recherche. Même si un seul restaurant correspond à un critère de recherche, tous les *shards* parcourent l'index. Il n'y a pas d'index global, uniquement des indexes locaux à chaque *shard*. Toutefois, les performances seront améliorées.

Si l'on souhaite profiter des avantages d'un index dans le cadre d'une requête « *aggregate* », il faut que le premier critère de recherche soit un filtre sur la clé utilisée dans l'index. Dans notre cas, seul un opérateur de type **$match** contenant « *address.street* » permettrait l'utilisation de l'index (idem pour la clé de *sharding*).

Pour consulter le plan d'exécution d'un « *aggregate* », le paramètre *{explain:true}* doit être ajouté après la séquence d'opérations :

```
db.test.aggregate([
   {$match:{"address.street" : "3 avenue"}},
   {$group:{_id:"$borough", total : { $sum : 1}}}
 ],
 {explain:true}
);
```