

Claire LANDREAT

Big Data

31/12/2023

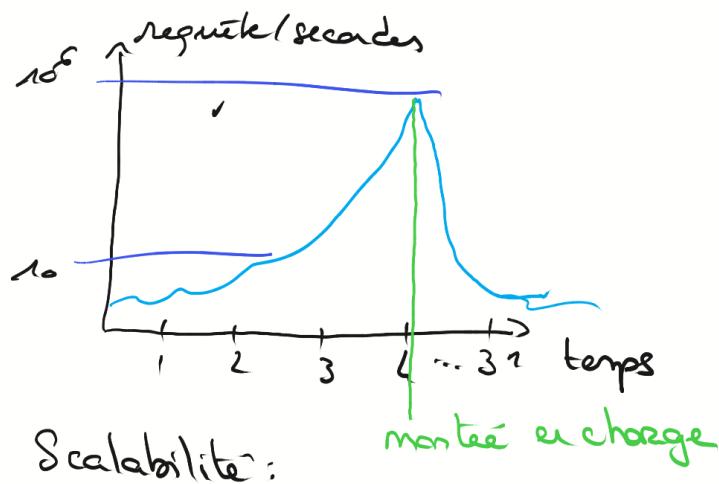
Rappels :Mongo DB :

Création d'une séquence d'opération:

- \$ match {}
- \$ Sort {}
- \$ Project {}
- \$ Set {}
- \$ Unset {}

Cours :

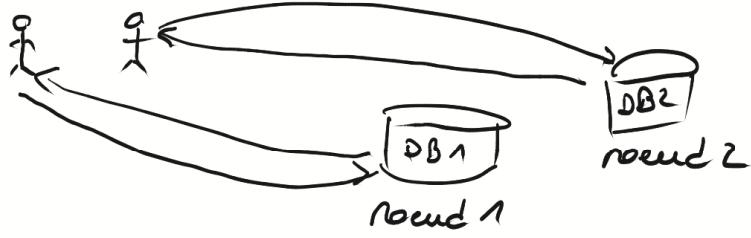
La RéPLICATION et
la reprise sur panne



Scalabilité : montée en charge

lecture : distribution des requêtes à lecture sur plusieurs nœuds

écriture : distribution des requêtes d'écriture sur plusieurs noeuds (problème de concurrence et de réconciliation)



La réPLICATION et la reprise sur panne

Architecture tolérante aux pannes

Solution intuitive : sauvegardes régulières



Encore plus loin pour assurer

1. Disponibilité : la probabilité pour qu'une donnée peut être accessible par un utilisateur.

Comment ? En cas de panne (serveur, noeud, d'un disque), la (les) tâche(s) peut (peuvent) être prise(s) en compte par un autre composant.

2. Scalabilité :

2.1. lecture : distribution des requêtes de lecture sur plusieurs noeuds

2.2. écriture : distribution des requêtes d'écriture sur plusieurs noeuds (problème de concurrence et de réconciliation)

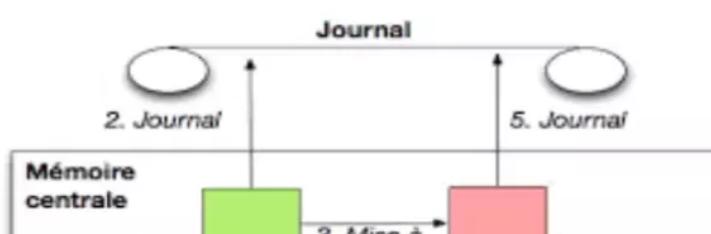
Scalabilité : la capacité d'un produit à s'adapter à un changement d'ordre de grandeur de la demande, en particulier sa capacité à maintenir ses fonctionnalités et ses performances en cas de forte demande (wikipedia)

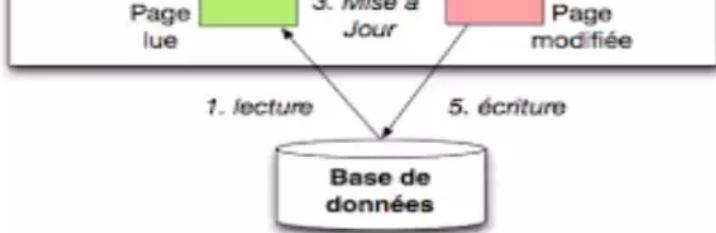
La réPLICATION et la reprise sur panne

Une "bonne réPLICATION" ?

3 copies très bon niveau de sécurité (2 copies dans un même réseau et une troisième dans un autre, en cas de coupure)

Fonctionnement : si une perte d'une copie est constatée, alors une réPLICATION est mise en place





id	Nom	Péson	Age
1			
:			
10 ⁹			

$\lceil id = 101 \rceil \rightarrow$ utilisateur que l'on cherche dans la table
 → en parcourant la table mais on ne peut le faire qu'au niveau de la mémoire puisqu'il faut écrire

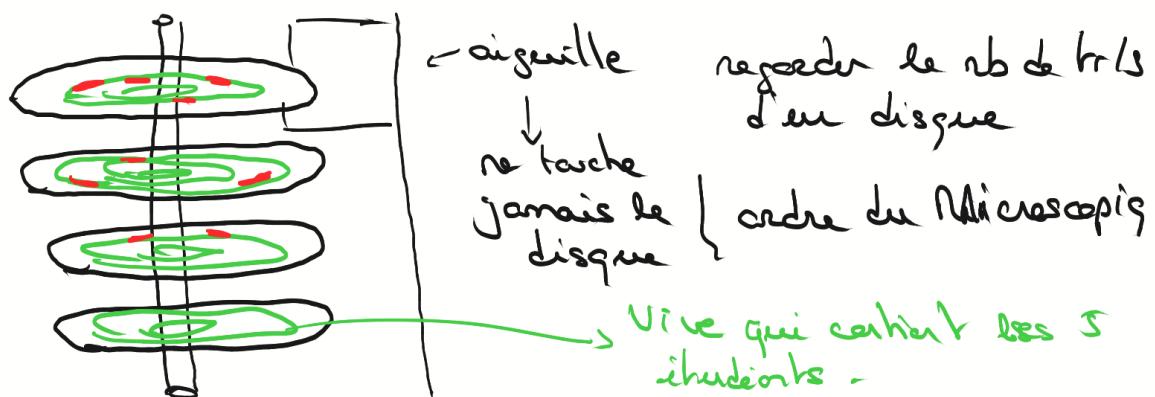
dans, donc le parcours se fera au niveau de la mémoire vive
 Mais 1, on peut avoir des données qui dépassent la mémoire vive (\rightarrow au Go RAM) → RangeDB va utiliser la notion d'index. On ne charge jamais toute la base dans la mémoire. Cela dépend de ce que l'on souhaite et faire ensuite

Quels sont les 4/5 types de mémoire ?

1. Mémoire cache (qq Ko)
2. Mémoire vive (qq Go)
3. Mémoire flash (SSD) \rightarrow Disque
4. CD, USB \rightarrow disque magnétique

RangeDB fait de la réplication
 2 notions : accès direct et accès séquentiel

Disque :



→ plusieurs platters

- blocs

↳ zone mémoire de taille fixe stockée sur disque
 bloc de X Ko

Sur MySQL : taille d'un bloc par défaut

10^{19} utilisateurs



address

au niveau du disque

Nouvel utilitaire: $104 \leftarrow id$

On cherche par dichotomie : je dois sauvegarder son address par ordre croissant dans le 2nd tableau

Dans l'index, on trouve des mots clefs ; exemple : recherche de recette de galette au citron : check index : mot clef : citron

→ nous donne la page (p76)

On a donc chercher par dichotomie

Si on a un bloc rempli, il faut renvoyer tout le bloc au niveau de la mémoire vive

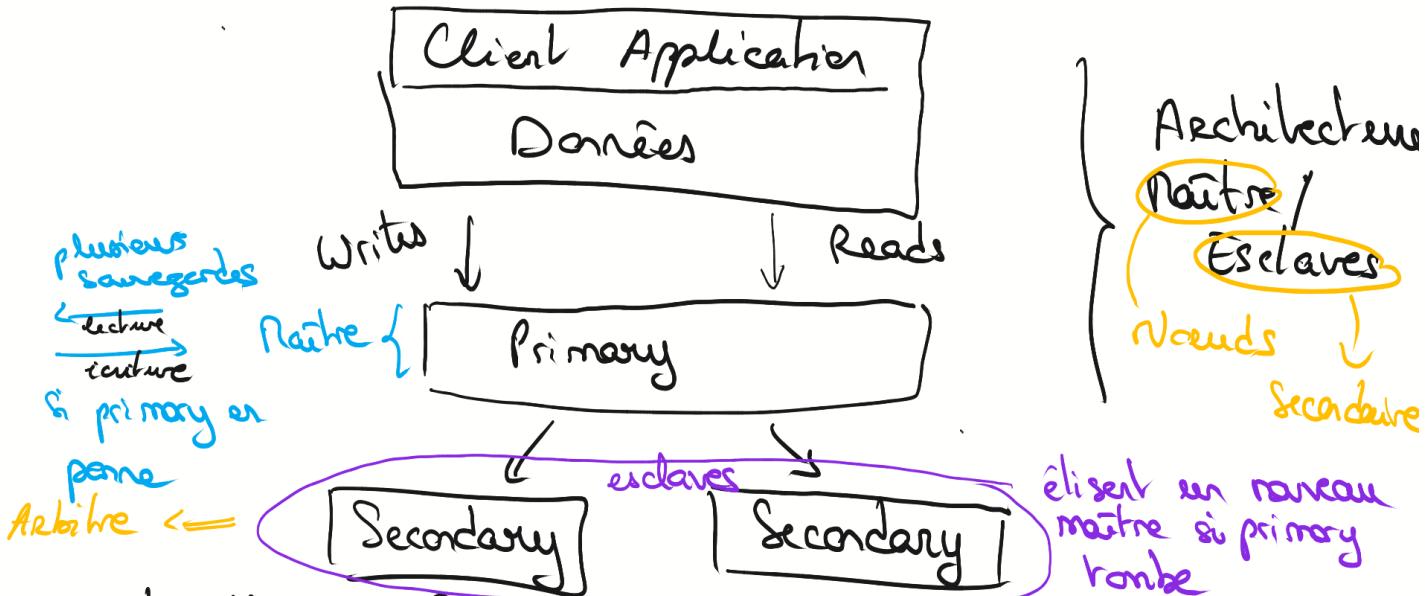
Si par ordre alphabétique, rechercher avec ordre alphabétique

↳ Structure index toute simple mais qui fait gagner du temps

Architecture tolérante aux pannes

On fait en sorte de répliquer les données sur plusieurs sites ≠, ceci pas sur le même support.

NongoDB utilise Replicaset: répond en un temps équivalent au primary si celui-ci tombe



Aller sur site officiel NongoDB

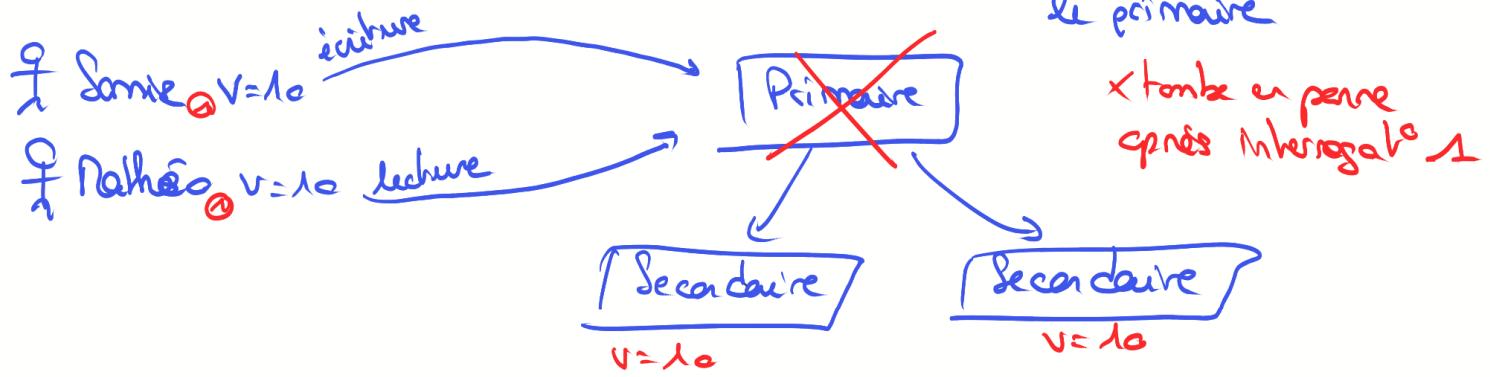
Lecture de doc sur les sauvegardes régulières Replicaset

1) Disponibilité : constante du système, si cas de panne d'un serveur (nœuds), la tâche effectuée par ce serveur peut être effectuée par un autre serveur (nœud)

2) Scalabilité : pouvoir distribuer

↳ en lecture { écrire sur le primaire & sur le secondaire
en écriture { sauf sur MongoDB → juste sur le primaire
mais il accepte lecture sur les secondaires. écriture

Configuration par défaut : écriture et lecture uniquement sur le primaire



↳ Après panne:

lecture sur les secondaires

dernière valeur
- Fortement cohérent
pas à jour
- Faiblement "
- Cohérent à terme

à terme, ce sera à jour

Si on autorise Sonnie & Nathéo

à lire sur les secondaires :

Sac Nathéo n'a pas accès à Primary, il demande au secondaire
réPLICATION pas encore terminée, Nathéo va recevoir $V=10$ au
lieu de $V=11$ (valeur qui a changé entre temps)

→ MongoDB accepte de lire sur secondaire mais exposition à
un risque de réPLICATION non terminée donc pas à jour -
→ faiblement cohérent.

CouchDB et MongoDB sont orientées documents.

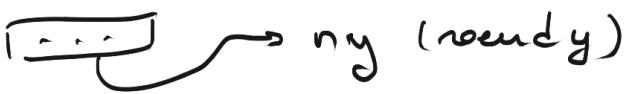
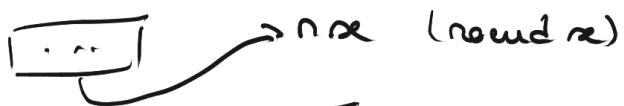
Théorème : CAP

C : Cohérence

A : Disponibilité (availability)

? : tolérance au partitionnement

↳ est-ce que le système continue de fonctionner en cas de partitionnement ? Oui même en cas de partitionnement risque

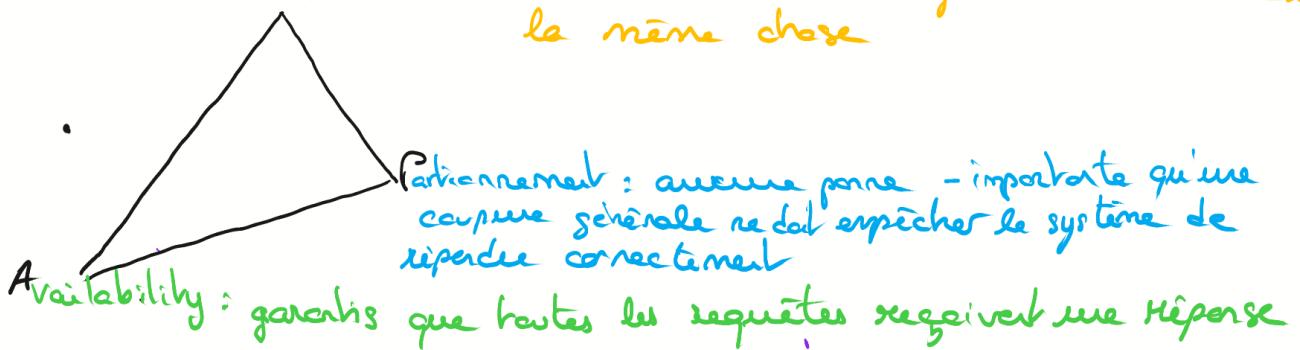


16h00

Ce système peut-il fonctionner avec les 3 ?

Schéma CAP :

Cohérence : tous les noeuds du système voient exactement la même chose



Partitionnement : aucune panne - importante qu'une coupure générale ne doit empêcher le système de répondre correctement

Availability : garantit que toutes les requêtes reçoivent une réponse

Les étapes pour instancier une réplication :

- ① Un nom d'un Replicaset rs018;
- ② Utiliser un port d'écoute pour chaque serveur
 $\begin{matrix} 27018, 27019, 27020 \\ \text{maître} \qquad \qquad \qquad \text{wclave} \end{matrix}$

Lancer les serveurs :

mongod -- replicaset rs018 -- port 27018

-- dbpath data/rs1

↓
3 répertoires

Il y a place d'une architecture tolérante aux pannes avec MongoDB
on procède de la façon suivante :

- ① Définir un répertoire de sauvegarde pour chacun des serveurs.
Ici, nous parlons sur 3 serveurs dont les répertoires sont nommés

rs1, rs2, rs3, avec différents ports d'écoute

② Nous faisons les répertoires définis, et lance 3 serveurs comme suit:

On utilise la commande <mkdir> pour créer un répertoire.

Avec les commandes suivantes, nous associons les répertoires de stockages à chacun de nos replicaset que l'on met en écoute sur les ports:

```
C:\Program Files\MongoDB\Server\4.0\bin>mongod --replSet rs0 --port 27018 --dbpath rs1
```

```
C:\Program Files\MongoDB\Server\4.0\bin>mongod --replSet rs0 --port 27019 --dbpath rs2
```

```
C:\Program Files\MongoDB\Server\4.0\bin>mongod --replSet rs0 --port 27020 --dbpath rs3
```

Nous n'avons pas encore mis en place le replicaset, nous utilisons donc la commande suivante afin de le mettre en place sur le port [27018](#) correspondant au serveur principal:

```
C:\Program Files\MongoDB\Server\4.0\bin>mongo --port 27018
```

Pour initialiser notre replicaSet on se connecte sur le serveur principal et on applique la commande suivante :

Cela nous permet de voir la machine présente dans le replicaset. Ici on s'aperçoit que c'est bien la machine rs1 sur le port [27018](#).

```
> rs.initiate()
{
    "info2" : "no configuration specified. Using a default configuration for the set",
    "me" : "localhost:27018",
    "ok" : 1,
    "operationTime" : Timestamp(1607528625, 1),
    "$clusterTime" : {
        "clusterTime" : Timestamp(1607528625, 1),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }
}
```

La configuration de notre réplicaSet se vérifie en utilisant la commande « rs.conf() » cela nous permet de renvoyer les membres qui appartiennent au réplicaSet.

```
rs0:SECONDARY> rs.conf()
```

On peut maintenant ajouter nos différents réplicaSet en les nommant par le numéro de port précédemment établi en les associant à notre machine

```
rs0:PRIMARY> rs.add("localhost:27019")
{
    "ok" : 1,
    "operationTime" : Timestamp(1607529018, 1),
    "$clusterTime" : {
```

```
        "clusterTime" : Timestamp(1607529018, 1),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }

rs0:PRIMARY> rs.add("localhost:27020")
{
    "ok" : 1,
    "operationTime" : Timestamp(1607529023, 1),
    "$clusterTime" : {
        "clusterTime" : Timestamp(1607529023, 1),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }
}
```

Nous utilisons la commande rs.conf(); de nouveau afin d'observer la bonne mise en place de notre replicaSet qui a bien les 3 membres que nous avons ajouté avant:

```
rs0:PRIMARY> rs.conf()
{
    "_id" : "rs0",
    "version" : 3,
    "protocolVersion" : NumberLong(1),
    "writeConcernMajorityJournalDefault" : true,
    "members" : [
        {
            "_id" : 0,
            "host" : "localhost:27018",
            "arbiterOnly" : false,
            "buildIndexes" : true,
            "hidden" : false,
            "priority" : 1,
            "tags" : {

            },
            "slaveDelay" : NumberLong(0),
            "votes" : 1
        },
        {
            "id" : 1,
            "host" : "localhost:27019",
            "arbiterOnly" : false,
            "buildIndexes" : true,
            "hidden" : false,
            "priority" : 1,
            "tags" : {

            },
            "slaveDelay" : NumberLong(0),
            "votes" : 1
        }
    ]
}
```

```
        },
        {
            "id" : 2,
            "host" : "localhost:27020",
            "arbiterOnly" : false,
            "buildIndexes" : true,
            "hidden" : false,
            "priority" : 1,
            "tags" : {

            },
            "slaveDelay" : NumberLong(0),
            "votes" : 1
        }
    ],
    "settings" : {
        "chainingAllowed" : true,
        "heartbeatIntervalMillis" : 2000,
        "heartbeatTimeoutSecs" : 10,
        "electionTimeoutMillis" : 10000,
        "catchUpTimeoutMillis" : -1,
        "catchUpTakeoverDelayMillis" : 30000,
        "getLastErrorMessage" : {

        },
        "getLastErrorDefaults" : {
            "w" : 1,
            "wtimeout" : 0
        },
        "replicaSetId" : ObjectId("5fd0f9b1eb5f1bdcb321b14e")
    }
}
```

La commande <rs.status()> permet de voir les membres de notre replicaSet qui sont définis comme serveur primaire et secondaire :

```
"members" : [
    {
        "id" : 0,
        "name" : "localhost:27018",
        "health" : 1,
        "state" : 1,
        "stateStr" : "PRIMARY",
        "uptime" : 1072,
        "optime" : {
            "ts" : Timestamp(1607529495, 1),
            "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2020-12-09T15:58:15Z"),
        "syncingTo" : "",
        "syncSourceHost" : "",
        "syncSourceId" : -1,
        "infoMessage" : "",
        "electionTime" : Timestamp(1607528625, 2),
        "electionDate" : ISODate("2020-12-09T15:43:45Z"),
        "configVersion" : 3,
        "configTerm" : 2
    }
]
```

```

        "self" : true,
        "lastHeartbeatMessage" : ""
    },
    {
        "_id" : 1,
        "name" : "localhost:27019",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 486,
        "optime" : {
            "ts" : Timestamp(1607529495, 1),
            "t" : NumberLong(1)
        },
        "optimeDurable" : {
            "ts" : Timestamp(1607529495, 1),
            "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2020-12-09T15:58:15Z"),
        "optimeDurableDate" : ISODate("2020-12-09T15:58:15Z"),
        "lastHeartbeat" : ISODate("2020-12-09T15:58:23.880Z"),
        "lastHeartbeatRecv" : ISODate("2020-12-09T15:58:24.910Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncingTo" : "localhost:27020",
        "syncSourceHost" : "localhost:27020",
        "syncSourceId" : 2,
        "infoMessage" : "",
        "configVersion" : 3
    }
},

```

(De même pour le id:2) ↗ SECONDARY

On remarque que le serveur rs1 est le serveur primaire dit maître et rs2 et rs3 sont des serveurs secondaires dits esclaves

Pour confirmer que le replicaSet est bien configuré on peut quitter le serveur rs1 en enlevant l'écoute sur le port [27018](#). On se connecte sur le port [27019](#) (rs1) et on applique la commande « rs.status() » pour savoir quel serveur est passé en primaire.

```

{
    "_id" : 1,
    "name" : "localhost:27019",
    "health" : 1,
    "state" : 1,
    "stateStr" : "PRIMARY",
    "uptime" : 1656,
    "optime" : {
        "ts" : Timestamp(1607530117, 1),
        "t" : NumberLong(2)
    },
    "optimeDate" : ISODate("2020-12-09T16:08:37Z"),
    "syncingTo" : "",
    "syncSourceHost" : "",
    "syncSourceId" : -1,
    "infoMessage" : "",
    "electionTime" : Timestamp(1607530107, 1),
    "electionDate" : ISODate("2020-12-09T16:08:27Z"),
    "configVersion" : 3,
    "self" : true,
    "lastHeartbeatMessage" : ""

    "_id" : 2,
    "name" : "localhost:27020",
    "health" : 1,
    "state" : 2,
    "stateStr" : "SECONDARY",
    "uptime" : 1000
}

```

```
        "uptime" : 1098,
        "optime" : {
            "ts" : Timestamp(1607530117, 1),
            "t" : NumberLong(2)
        },
        "optimeDurable" : {
            "ts" : Timestamp(1607530117, 1),
            "t" : NumberLong(2)
        },
        "optimeDate" : ISODate("2020-12-09T16:08:37Z"),
        "optimeDurableDate" : ISODate("2020-12-09T16:08:37Z"),
        "lastHeartbeat" : ISODate("2020-12-09T16:08:41.810Z"),
        "lastHeartbeatRecv" : ISODate("2020-12-09T16:08:40.633Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncingTo" : "localhost:27019",
        "syncSourceHost" : "localhost:27019",
        "syncSourceId" : 1,
        "infoMessage" : "",
        "configVersion" : 3
```

On remarque que le serveur rs2 est passé en serveur primaire et rs3 en secondaire.

Maintenant, si on relance notre serveur rs1 et que l'on affiche le status sur le serveur rs2 on s'aperçoit que le rs2 est resté en primaire et rs1 et rs3 en secondaire. En effet une fois qu'un serveur a été élu primaire il reste primaire tant qu'il n'est pas déconnecté.

Afin de réduire les temps de latence lors de l'élection d'un nouveau serveur primaire, nous allons donc créer un arbitre pour que les serveurs arrivent à se mettre d'accord.

Pour cela on crée un répertoire « arb » qui va jouer le rôle de l'arbitre:

```
C:\Program Files\MongoDB\Server\4.0\bin>mkdir arb
```

On peut maintenant lancer notre serveur et notre replicaSet à ce niveau sur le port [30000](#)

```
C:\Program Files\MongoDB\Server\4.0\bin>mongod --port 30000 --dbpath arb --replSet rs0
```

On se connecte à présent sur notre serveur primaire (rs2) sur le port [27019](#) pour y ajouter notre arbitre:

```
rs0:PRIMARY> rs.addArb("localhost:30000")
{
    "ok" : 1,
    "operationTime" : Timestamp(1607531379, 1),
    "$clusterTime" : {
        "clusterTime" : Timestamp(1607531379, 1),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }
}
```

On vérifie la présence de l'arbitre sur notre serveur :

```
        "_id" : 3,
        "name" : "localhost:30000",
        "health" : 1,
        "state" : 7,
        "stateStr" : "ARBITER",
        "uptime" : 157,
        "lastHeartbeat" : ISODate("2020-12-09T16:32:15.149Z"),
        "lastHeartbeatRecv" : ISODate("2020-12-09T16:32:15.281Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncingTo" : "".
```

```
"syncSourceHost" : "",  
"syncSourceId" : -1,  
"infoMessage" : "",  
"configVersion" : 4
```

On s'aperçoit que l'arbitre a bien été configuré.

Lorsque qu'un serveur tombe, nous sommes désormais capables de déterminer rapidement quel serveur va reprendre la main et passer en primaire.

Cette configuration va nous permettre d'assurer une la continuité de service et de garantir que nos données soient disponibles en permanence.

Comment accepter de lire des informations sur un esclave ?

→ cette technique se situe au niveau des esclaves et non au niveau des serveurs, c'est <read preference> en configurant les drivers. Cela permet de définir vers quels types de serveurs, primaires, secondaire, ou les deux, le client ira se connecter. Selon la préférence, la disponibilité ou l'intégrité des données sera garantie

Que signifie le C du théorème CAP ?

→ Le client reçoit systématiquement la dernière version d'un document

Décrire les étapes à suivre pour le sharding (partitionnement)

→ le sharding est une surcouche qui est basée sur du Master / Slave ou du ReplicaSet

Pour instancier le cluster, il faut :

- lancer les ConfigServer en ReplicaSet
- lancer chaque Shard (serveur de données) en ReplicaSet
- lancer un mongo (routeur):

1. Connecter les ConfigServer
2. Connecter les shard
3. lancer le sharding sur une collection