TP Réplication MongoDB

Réponses aux Questions

Rayane Sendjakedine

1 Partie 1 - Compréhension de base

Question 1 : Qu'est-ce qu'un Replica Set dans MongoDB ?

Un Replica Set est un groupe de processus `mongod` qui maintiennent le même ensemble de données. Il est composé d'un nœud Primary (qui gère les écritures) et de plusieurs nœuds Secondaries (qui répliquent les données). C'est le mécanisme de base pour assurer la haute disponibilité et la tolérance aux pannes dans MongoDB.

Question 2 : Quel est le rôle du Primary dans un Replica Set ?

Le Primary est le seul nœud qui accepte les écritures. Il :

— Reçoit toutes les opérations d'insertion, mise à jour et suppression
— Enregistre les opérations dans l'oplog (journal d'opérations)
— Envoie un accusé de réception au client
— Initie la réplication vers les Secondaries

Question 3 : Quel est le rôle essentiel des Secondaries ?

Les Secondaries maintiennent une copie à jour des données du Primary en :

— Lisant continuellement l'oplog du Primary
— Appliquant les mêmes opérations sur leurs propres données
— Pouvant servir les requêtes de lecture (si configuré)
— Participant aux élections en cas de panne du Primary

Question 4 : Pourquoi MongoDB n'autorise-t-il pas les écritures sur un Secondary ?

Pour éviter les conflits de mise à jour et garantir la cohérence des données. Si plusieurs nœuds pouvaient accepter des écritures simultanément, on aurait des versions divergentes des données et la réconciliation serait très complexe. La centralisation des écritures sur le Primary assure un ordre unique et non ambigu des opérations.

Question 5 : Qu'est-ce que la cohérence forte dans le contexte MongoDB ?

La cohérence forte garantit qu'une fois qu'une écriture est validée, toute lecture ultérieure retourne la dernière valeur écrite. MongoDB l'assure en forçant par défaut les lectures et écritures sur le Primary. Ainsi, on est toujours sûr d'avoir accès à la version la plus récente des données.

Question 6 : Différence entre readPreference : "primary" et "secondary" ?

— `primary` : Toutes les lectures se font sur le Primary (cohérence forte garantie, valeur par défaut)
— `secondary` : Les lectures se font sur les Secondaries (meilleure répartition de charge mais risque de données légèrement obsolètes)

Question 7 : Dans quel cas lire sur un Secondary malgré les risques ?

On peut lire sur un Secondary pour :

— Réduire la charge sur le Primary dans une application à forte lecture
— Servir des requêtes analytiques ou de reporting qui tolèrent des données légèrement obsolètes
— Améliorer la latence en lisant depuis le nœud géographiquement le plus proche
— Effectuer des sauvegardes sans impacter le Primary

2 Partie 2 - Commandes & configuration

Question 8 : Commande pour initialiser un Replica Set ?

```
rs.initiate()
```

Cette commande initialise le Replica Set sur le nœud connecté qui devient le Primary initial.

Question 9 : Comment ajouter un nœud après initialisation ?

```
rs.add("localhost:27019")
```

On utilise `rs.add()` en spécifiant l'adresse et le port du nouveau membre.

Question 10 : Commande pour afficher l'état actuel du Replica Set ?

```
rs.status()
```

Cette commande affiche l'état en temps réel de chaque nœud, le Primary actuel, le replication lag, et la santé de chaque membre.

Question 11 : Comment identifier le rôle actuel d'un nœud ?

Plusieurs méthodes :

```
rs.isMaster()          # Indique si le noeud connecte est Primary
rs.status()            # Affiche le role de chaque membre (PRIMARY/SECONDARY/
    ARBITER)
db.hello()             # Alternative a rs.isMaster()
```

Question 12 : Commande pour forcer le basculement du Primary ?

```
rs.stepDown(120)       # Force le Primary a devenir Secondary pendant 120
    secondes
```

Cela déclenche une élection et un Secondary prend le relais.

3 Partie 3 - Résilience et tolérance aux pannes

Question 13 : Désigner un nœud comme Arbitre ? Pourquoi ?

Commande :

```
rs.addArb("localhost:27021")
```

Pourquoi ? L'Arbitre ne stocke pas de données mais participe aux votes d'élection. Il permet d'avoir un nombre impair de votants (éviter les situations sans majorité) sans le coût de stockage d'un nœud complet. Utile pour éviter le split-brain dans un cluster avec nombre pair de nœuds de données.

Question 14 : Configurer un nœud secondaire avec slaveDelay ?

```
cfg = rs.conf()
cfg.members[2].slaveDelay = 3600  # 1 heure de retard
cfg.members[2].priority = 0       # Ne peut pas devenir Primary
cfg.members[2].hidden = true      # Cache aux clients
rs.reconfig(cfg)
```

Question 15 : Primary tombe en panne sans majorité ?

Si le Primary tombe et qu'il n'y a pas de majorité de nœuds disponibles, aucune élection ne peut avoir lieu. Le cluster entre en mode dégradé :

— Aucun nouveau Primary n'est élu
— Aucune écriture n'est acceptée
— Les lectures peuvent continuer sur les Secondaries (si configuré)

C'est pour éviter le split-brain.

Question 16 : Comment MongoDB choisit-il un nouveau Primary ?

Critères d'élection (par ordre de priorité) :

1. Le nœud doit avoir `priority > 0`
2. Le nœud doit posséder les données les plus récentes (optime le plus élevé)
3. Le nœud avec la priorité la plus élevée est préféré
4. En cas d'égalité, le nœud avec l'ID le plus bas est choisi
5. Le nœud doit recevoir la majorité des votes

Question 17 : Qu'est-ce qu'une élection dans MongoDB ?

Une élection est le processus par lequel les membres d'un Replica Set votent pour désigner un nouveau Primary. Elle est déclenchée quand :

— Le Primary actuel tombe en panne ou devient injoignable
— Un nouveau Replica Set est initialisé
— Un membre fait `rs.stepDown()`
— La configuration du Replica Set change

Elle utilise un algorithme de consensus distribué (basé sur Raft).

Question 18 : Auto-dégradation du Replica Set ?

L'auto-dégradation survient quand le Primary détecte qu'il ne peut plus communiquer avec la majorité des membres. Il se dégrade automatiquement en Secondary pour éviter le split-brain. Le cluster reste en mode lecture seule jusqu'à ce qu'un nouveau Primary soit élu par un groupe majoritaire.

Question 19 : Pourquoi un nombre impair de nœuds ?

Pour faciliter l'obtention d'une majorité. Avec un nombre impair :

— 3 nœuds : majorité = 2 (tolère 1 panne)
— 5 nœuds : majorité = 3 (tolère 2 pannes)

Avec un nombre pair (4 nœuds), la majorité est 3, même tolérance qu'avec 3 nœuds mais coût supérieur. On ajoute plutôt un Arbitre.

Question 20 : Conséquences d'une partition réseau ?

Une partition réseau divise le cluster en groupes isolés. Seul le groupe possédant la majorité peut continuer à fonctionner et élire un Primary. Les autres groupes :

— Ne peuvent pas accepter d'écritures
— Restent en lecture seule (Secondaries)
— Se resynchronisent automatiquement quand le réseau est rétabli

4 Partie 4 - Scénarios pratiques

Question 21 : 3 nœuds (Primary 27017, Secondary 27018, Arbitre 27019). Primary injoignable ?

Si le Primary (27017) devient injoignable :

1. Le Secondary (27018) et l'Arbitre (27019) détectent la panne
2. Ils lancent une élection
3. Le Secondary vote pour lui-même, l'Arbitre vote aussi
4. Le Secondary obtient 2 votes sur 3 (majorité atteinte)
5. Le Secondary (27018) devient le nouveau Primary
6. Les écritures peuvent reprendre

Question 22 : Secondary avec slaveDelay de 120 secondes - utilité ?

Un Secondary avec délai permet de se protéger contre les erreurs humaines. Usages :

— **Protection contre suppression accidentelle** : Si on supprime des données par erreur, on a 120 secondes pour les récupérer depuis ce nœud avant qu'il ne réplique la suppression
— **Sauvegarde "vivante"** : Point de restauration récent sans backup complet
— **Rollback manuel** : Possibilité de revenir à un état antérieur

Question 23 : Client exige lecture toujours à jour, même en bascule. Options ?

```
// Configuration recommandee
readConcern: "linearizable"   // Garantit lecture de la derniere ecriture
writeConcern: { w: "majority", j: true }  // Ecriture confirmee par majorite
readPreference: "primary"     // Toujours lire sur le Primary
```

Cela garantit la cohérence la plus forte possible, au prix de la latence.

Question 24 : Garantir écriture confirmée par au moins 2 nœuds ?

```
db.collection.insertOne(
    { data: "value" },
    { writeConcern: { w: 2, wtimeout: 5000 } }
)
```

`w: 2` force la confirmation par le Primary + au moins 1 Secondary. `wtimeout` évite l'attente infinie.

Question 25 : Lecture obsolète depuis Secondary - Pourquoi et solution ?

Pourquoi ? MongoDB utilise une réplication asynchrone. Quand le Primary écrit, il confirme immédiatement au client et réplique en arrière-plan. Il y a donc un délai (lag) pendant lequel le Secondary n'a pas encore reçu les dernières modifications.

Solutions :

— Lire depuis le Primary (`readPreference: "primary"`)
— Utiliser `readConcern: "majority"` qui garantit des données répliquées sur la majorité
— Accepter le délai si les données temps réel ne sont pas critiques

Question 26 : Vérifier quel nœud est actuellement Primary ?

```
rs.status().members.forEach(function(member) {
    if (member.stateStr == "PRIMARY") {
        print("Primary: " + member.name);
    }
});

# Ou plus simplement
rs.status()  # Chercher "stateStr": "PRIMARY"
```

Question 27 : Forcer bascule manuelle du Primary sans interruption ?

```
# Methode 1 : Step down temporaire
rs.stepDown(60)  # Primary devient Secondary pendant 60s

# Methode 2 : Step down avec election immediate
rs.stepDown(10, 5)  # Step down si Secondary a <5s de retard
```

Les clients avec retry automatique se reconnectent au nouveau Primary sans perte de données.

Question 28 : Ajouter un nouveau Secondary en fonctionnement ?

Procédure :

1. Démarrer le nouveau serveur mongod avec le même `--replSet`
2. Se connecter au Primary actuel
3. Ajouter le nouveau membre : `rs.add("localhost:27021")`
4. Le nouveau nœud se synchronise automatiquement (initial sync)
5. Vérifier avec `rs.status()` que l'état passe à SECONDARY

Question 29 : Retirer un nœud défectueux ?

```
rs.remove("localhost:27020")
```

Le nœud est retiré de la configuration du Replica Set. Il faut ensuite l'arrêter manuellement.

Question 30 : Configurer un nœud caché ? Pourquoi ?

```
cfg = rs.conf()
cfg.members[2].priority = 0
cfg.members[2].hidden = true
rs.reconfig(cfg)
```

Pourquoi ? Un nœud caché :

— N'apparaît pas aux clients (pas de lectures)
— Ne peut pas devenir Primary
— Utile pour backups, analytics, ou reporting sans impacter la prod

Question 31 : Modifier priorité pour Primary préféré ?

```
cfg = rs.conf()
cfg.members[0].priority = 5  # Priorite elevee pour ce noeud
cfg.members[1].priority = 1  # Priorite par defaut
cfg.members[2].priority = 1
rs.reconfig(cfg)
```

```
# Forcer election immediate
rs.stepDown()
```

Le nœud avec priorité 5 sera préféré lors des élections.

Question 32 : Vérifier délai de réplication d'un Secondary ?

```
rs.printSlaveReplicationInfo()

# Ou via rs.status()
rs.status().members.forEach(function(m) {
    if (m.stateStr == "SECONDARY") {
        print(m.name + " - Lag: " +
              (m.optime.ts.getTime() - Date.now()/1000) + "s");
    }
});
```

5 Questions complémentaires

Question 33 : Que fait rs.freeze() ?

```
rs.freeze(120)  # Empeche le noeud de devenir Primary pendant 120s
```

Utile pour :

- Maintenance d'un nœud sans risque qu'il devienne Primary
- Tests de failover
- Empêcher temporairement un nœud de participer aux élections

Question 34 : Redémarrer un Replica Set sans perdre la configuration ?

Méthode sûre :

1. Redémarrer les Secondaries un par un (attendre qu'ils soient SECONDARY avant le suivant)
2. Redémarrer le Primary en dernier (ou faire `rs.stepDown()` avant)
3. La configuration est persistée sur disque, elle sera rechargée automatiquement

Pas besoin de réinitialiser le Replica Set.

Question 35 : Surveiller la réplication en temps réel ?

Via commandes shell :

```
# Boucle de surveillance
while true; do
    mongo --eval "rs.printSlaveReplicationInfo()"
    sleep 5
done
```

Via logs :

```
# Logs de replication
tail -f /var/log/mongodb/mongod.log | grep -i repl
```

Via MongoDB Compass ou outils de monitoring : Atlas, Ops Manager, ou Prometheus avec mongodb_exporter.

Question 37 : Qu'est-ce qu'un Arbitre et pourquoi ne stocke-t-il pas de données ?

Un Arbitre est un membre léger du Replica Set qui :

— Participe uniquement aux votes d'élection
— Ne réplique pas les données (pas de stockage)
— Ne peut jamais devenir Primary
— Consomme très peu de ressources (CPU/RAM/disque)

Pourquoi pas de données ? Pour réduire les coûts. Il sert juste à atteindre une majorité de votes, pas à assurer la redondance des données.

Question 38 : Vérifier latence de réplication Primary-Secondaries ?

```
rs.printSlaveReplicationInfo()
```

Affiche pour chaque Secondary :

— Timestamp de la dernière opération répliquée
— Secondes de retard par rapport au Primary
— Dernière synchronisation

Question 39 : Commande pour afficher retard de réplication ?

```
rs.printSlaveReplicationInfo()
# ou
rs.printSecondaryReplicationInfo()  # Meme chose, nom mis a jour
```

Question 40 : Différence réplication asynchrone/synchrone ? Type MongoDB ?

Asynchrone :

— Primary confirme l'écriture immédiatement
— Réplication vers Secondaries en arrière-plan
— Faible latence mais risque de perte si Primary crash avant réplication

Synchrone :

— Primary attend confirmation des Secondaries avant d'acquitter
— Cohérence forte garantie
— Latence plus élevée

MongoDB utilise : Réplication asynchrone par défaut, mais peut simuler le comportement synchrone avec `writeConcern: "majority"`.

Question 41 : Modifier configuration sans redémarrer les serveurs ?

Oui, avec `rs.reconfig()` :

```
cfg = rs.conf()
cfg.members[1].priority = 2
rs.reconfig(cfg)
```

Les changements prennent effet immédiatement sans redémarrage. Attention : certains changements peuvent déclencher une élection.

Question 42 : Secondary en retard de plusieurs minutes ?

Conséquences :

— Les lectures sur ce Secondary retournent des données obsolètes
— Il peut être exclu des élections s'il est trop en retard
— Si le retard dépasse la taille de l'oplog, il faudra une resynchronisation complète (initial sync)

Causes possibles : charge réseau, I/O disque lent, charge CPU élevée.

Question 43 : Comment MongoDB gère les conflits lors de la réplication ?

MongoDB évite les conflits par conception :

— Toutes les écritures passent par le Primary (source unique)
— Les Secondaries appliquent les opérations dans le même ordre (oplog)
— Pas d'écritures concurrentes possibles
— En cas de rollback (Primary isolé qui revient), les opérations non répliquées sont annulées et sauvegardées dans des fichiers de rollback

Question 44 : Plusieurs Primarys simultanément possible ? Pourquoi ?

Non, c'est impossible par conception. MongoDB utilise :

— Un algorithme de consensus (basé sur Raft) qui garantit un seul Primary
— La règle de majorité : seul un groupe majoritaire peut élire un Primary
— L'auto-dégradation : un Primary qui perd la majorité redevient Secondary

Cela évite le split-brain et garantit la cohérence des données.

Question 45 : Pourquoi déconseillé d'écrire sur Secondary même en lecture préférée ?

Il est techniquement impossible d'écrire sur un Secondary, pas juste déconseillé :

— MongoDB rejette toute tentative d'écriture sur un Secondary (erreur "not master")
— Cela garantit l'intégrité des données
— Même avec `readPreference: "secondary"`, seules les lectures sont autorisées

Pour l'écriture distribuée, il faudrait utiliser le sharding.

Question 46 : Conséquences d'un réseau instable sur un Replica Set ?

— **Élections fréquentes** : Le Primary peut être élu/destitué à répétition
— **Indisponibilité temporaire** : Pendant les élections, aucune écriture n'est acceptée
— **Replication lag** : Retard accru des Secondaries
— **Split-brain potentiel** : Partitionnement du cluster
— **Rollbacks** : Si un Primary isolé accepte des écritures avant d'être destitué

Solutions :

— Augmenter les timeouts d'élection
— Utiliser un réseau stable et rapide entre les nœuds
— Placer un Arbitre dans un datacenter neutre
— Configurer des priorités pour éviter les élections inutiles