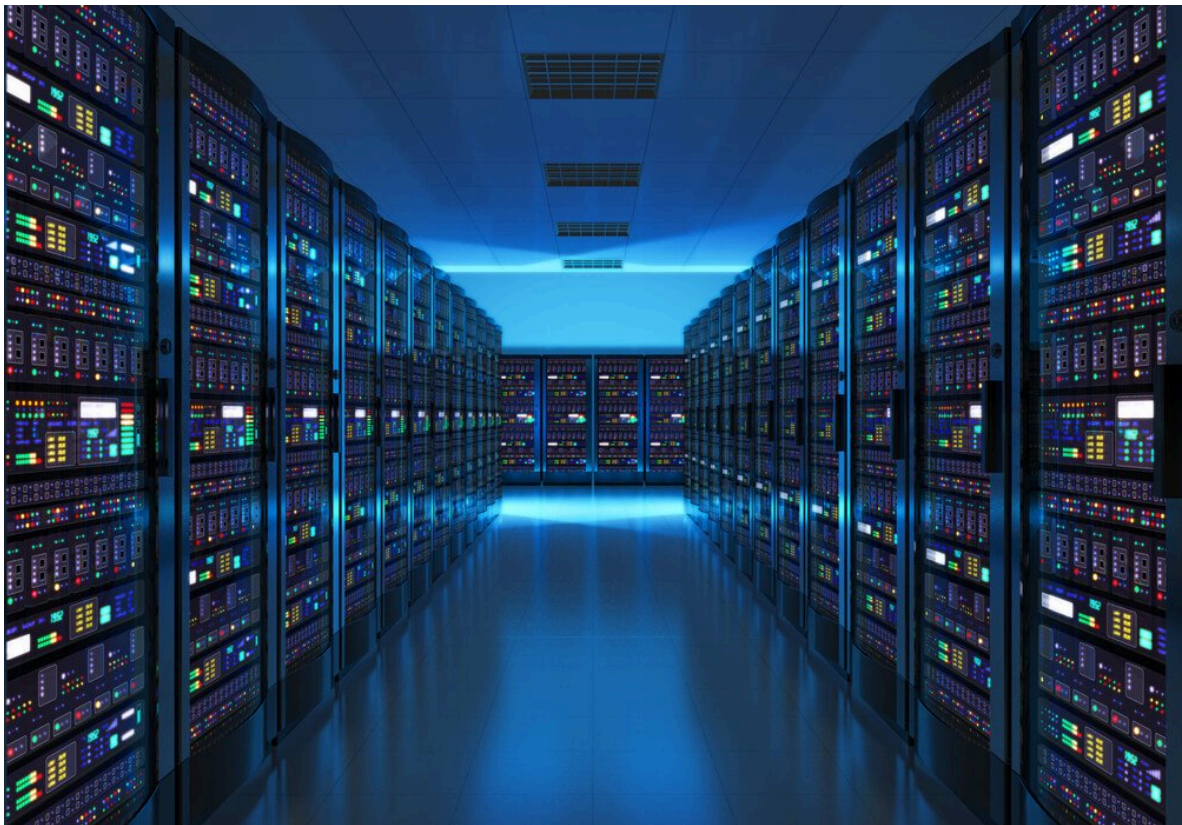


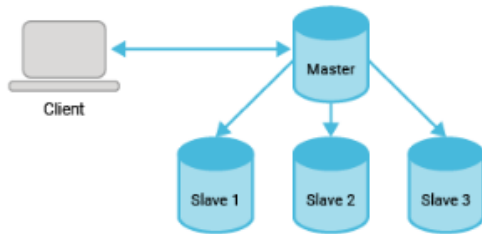
TP Réplication et Partitionnement NoSQL



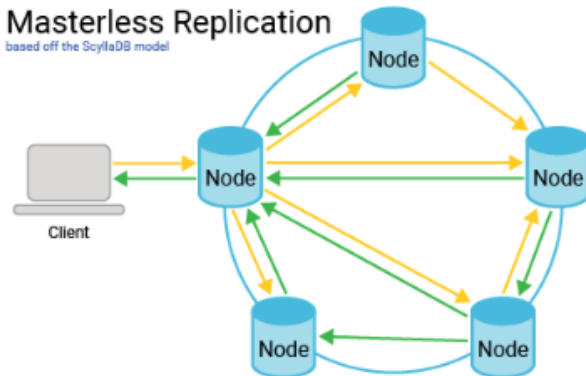
Zeineb Boujmil

La réplication en NoSQL

Master-Slave Replication based off the MongoDB model



Masterless Replication based off the ScyllaDB model



Deux grands types de réplication NoSQL : Maître-Esclave et Multi-maitres :

- **Maître Esclave :**

1. Principe général de la réplication

La réplication signifie copier automatiquement les données d'un serveur vers d'autres. Dans une base NoSQL comme MongoDB, on a un Primary et plusieurs Secondaries.

2. Rôle des nœuds

Primary : accepte toutes les écritures (writes) et peut lire.

Secondaries : ne font pas d'écriture, répliquent continuellement les données, peuvent servir en lecture.

Arbiter = vote mais ne stocke pas de données.

3. Fonctionnement de la réplication

- L'application envoie les writes → Primary
- Le Primary écrit dans son oplog (journal des opérations)
- Les Secondaries lisent cet oplog et rejouent toutes les opérations (insert, update, delete)

4. Pourquoi faire de la réplication ?

- Haute disponibilité : si le Primary tombe, un Secondary devient Primary.
- Sécurité : données dupliquées.
- Scalabilité en lecture : les Secondaries peuvent servir pour les lectures.

5. Caractéristiques importantes

- Réplication asynchrone : léger retard possible entre Primary et Secondaries.
- Un seul Primary à la fois : sinon conflits.
- Élections automatiques en cas de panne du Primary.

La réplication en NoSQL consiste à avoir un Primary qui gère les écritures et des Secondaries qui copient automatiquement les données pour assurer la haute disponibilité et la tolérance aux pannes.

- **Multi-Maître:**

Contrairement au modèle Primary/Secondary, le masterless (comme Cassandra ou ScyllaDB) ne possède aucun nœud maître. Tous les nœuds sont égaux.

1.Principe général du masterless

La réplication consiste à distribuer automatiquement les données sur plusieurs nœuds, et tous les nœuds peuvent écrire et lire.

2. Rôle des nœuds

Tous les nœuds :

Acceptent les écritures

Acceptent les lectures

Se répliquent entre eux

Aucun nœud n'est maître ou esclave

3. Fonctionnement de la réplication

- Le client peut envoyer un write à n'importe quel nœud
- Le nœud réplique l'opération vers les autres nœuds concernés
- Si un nœud tombe, le cluster continue à fonctionner normalement
- Des mécanismes internes gèrent les conflits éventuels

4. Pourquoi utiliser du masterless ?

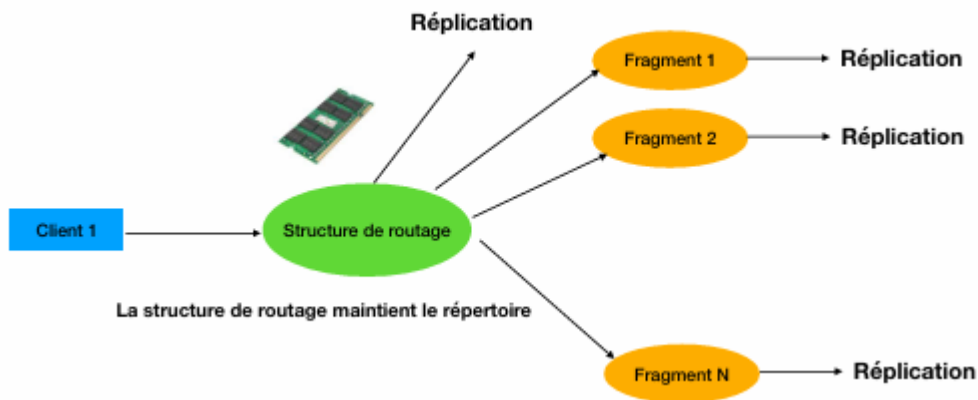
- Très haute disponibilité
- Tolérance aux pannes
- Scalabilité horizontale exceptionnelle
- Idéal pour les systèmes distribués massifs

5. Caractéristiques importantes

- Pas de Primary
- Réplication souvent asynchrone
- Gestion interne des conflits d'écriture
- Tous les nœuds ont le même rôle

Le modèle masterless permet à tous les nœuds d'être maîtres. Chaque nœud peut lire/écrire et réplique automatiquement les données aux autres nœuds pour offrir une disponibilité maximale.

Le partitionnement en NoSQL



Le partitionnement est une technique utilisée dans les systèmes distribués pour diviser une grande base de données en plusieurs fragments plus petits, appelés *partitions* ou *shards*. L'objectif est d'améliorer les performances, la scalabilité et la gestion de grandes quantités de données.

Au lieu de stocker toutes les données sur un seul serveur, elles sont séparées logiquement en plusieurs fragments, chacun pouvant être géré par un nœud différent.

Une structure de routage (router, coordinator, query planner...) se charge de déterminer où se trouvent les données, et envoie la requête du client vers le fragment approprié.

La répartition permet :

- d'augmenter la capacité de stockage,
- de répartir la charge de travail entre plusieurs serveurs,
- d'accélérer les requêtes, et d'offrir une meilleure disponibilité.

Dans un système distribué moderne, chaque fragment peut également être **répliqué** pour garantir la tolérance aux pannes.

Dans les systèmes distribués, la base de données peut être découpée selon différentes méthodes.

Les deux plus courantes sont :

Partitionnement par intervalle (Range Partitioning):

Les données sont réparties selon des plages de valeurs.

Chaque fragment contient les valeurs dans un intervalle précis.

Exemple: On stocke des utilisateurs par leur âge :

Fragment 1 → âges 0 à 20

Fragment 2 → âges 21 à 40

Fragment 3 → âges 41 à 60

Une requête sur un utilisateur de 25 ans ira directement dans le fragment 2.

Idéal pour les données ordonnées (dates, prix, IDs).

Partitionnement par hachage (Hash Partitioning):

On applique une fonction de hachage sur la clé (ex : ID, nom) pour déterminer le fragment. Cela répartit les données de manière uniforme et aléatoire.

Exemple:

On stocke des utilisateurs par leur ID.

On calcule : $\text{hash}(\text{id}) \% 3$ pour savoir dans quel fragment les mettre :

Si $\text{hash}(\text{id}) \% 3 = 0 \rightarrow$ Fragment 1

Si $\text{hash}(\text{id}) \% 3 = 1 \rightarrow$ Fragment 2

Si $\text{hash}(\text{id}) \% 3 = 2 \rightarrow$ Fragment 3

Deux utilisateurs proches (ID 101 et 102) peuvent aller dans des fragments différents.

Idéal pour équilibrer la charge et éviter les déséquilibres.

Partie 1 — Compréhension de base

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

Un **Replica Set** = un groupe de serveurs MongoDB contenant la **même base** copiée automatiquement.

Idée :

Serveur A → copie les données → Serveur B, Serveur C...

But : tolérance aux pannes + haute disponibilité.

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

Le **Primary** est le seul serveur qui accepte les écritures (insert, update, delete).

Les autres répliquent passivement.

Exemple :

On écrit un document → il va sur le *Primary* → puis il est répliqué vers les Secondaries.

3. Quel est le rôle essentiel des Secondaries ?

Les Secondaries copient les données du Primary.

Ils servent à :

- * être prêts à devenir *Primary* en cas de panne

- * répondre à certaines lectures (si on l'autorise)

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

Parce que MongoDB veut éviter les **conflits de versions**.

Si chacun écrivait dans son coin : les copies seraient différentes → perte de cohérence.

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

La cohérence forte est lire toujours la dernière version validée par le Primary.

Si on fait une lecture immédiate après un insert → je suis sûr de voir la donnée.

6. Quelle est la différence entre readPreference : "primary" et "secondary" ?

valeur	Lecture faite sur	Garanties
"Primary"	serveur principale	toujours à jour
"secondary"	l'un des secondaires	peut être en retard

7. Dans quel cas pourrait-on souhaiter lire sur un Secondary malgré les risques

Pour soulager le Primary si on a énormément de lectures.

Exemple réel :

Un site web reçoit 10 000 lectures/s → on répartit les lectures sur les Secondaries pour éviter de surcharger le Primary.

Mais : risque d'avoir un *retard de réplication*.

Partie 2 — Commandes & configuration

8. Quelle commande permet d'initialiser un Replica Set ?

```
rs.initiate()
```

9. Comment ajouter un nœud à un Replica Set après son initialisation ?

```
rs.add("adresse_du_secondaire:27017")
```

10. Quelle commande permet d'afficher l'état actuel du Replica Set ?

```
rs.status()
```

11. Comment identifier le rôle actuel (Primary / Secondary / Arbitre) d'un nœud ?

```
rs.isMaster()
```

12. Quelle commande permet de forcer le basculement du Primary ?

```
sur le primary : rs.stepDown()
```

13. Comment peut-on désigner un nœud comme Arbitre ? Pourquoi le faire ?

```
rs.addArb("adresse_arbitre:27019")
```

Un arbitre, bien qu'il ne puisse jamais devenir Primary ni stocker de données, joue un rôle dans ce processus puisqu'il participe au vote. Son intervention permet d'atteindre plus facilement une majorité, mais il n'est jamais considéré comme candidat. L'arbitre sert donc uniquement à déverrouiller les élections dans les clusters où un nœud supplémentaire est nécessaire pour obtenir la majorité, sans influencer le choix du candidat autrement que par son vote.

14. Donnez la commande pour configurer un nœud secondaire avec un délai de réplication (slaveDelay)

```
cfg = rs.conf()
```

```
cfg.members[1].slaveDelay = 120
```

```
rs.reconfig(cfg)
```

Partie 3 — Résilience et tolérance aux pannes

15. Que se passe-t-il si le Primary tombe en panne et qu'il n'y a pas de majorité ?

Dans MongoDB, pour élire un Primary, il faut une majorité des nœuds du Replica Set.

Si le Primary tombe ET qu'aucune majorité n'est disponible alors :

- Aucun nouveau Primary ne peut être élu

- Le Replica Set passe en mode lecture seule

- Toutes les écritures deviennent impossibles

- On peut encore lire, mais pas écrire

16. Comment MongoDB choisit-il un nouveau Primary ? Quels critères utilise-t-il ?

Lorsqu'une élection est déclenchée, MongoDB doit choisir le nœud le plus apte à devenir Primary.

Le premier critère utilisé est la priorité configurée dans la topologie : un nœud avec une priorité plus élevée est préféré pour devenir Primary.

Ensuite, MongoDB analyse l'état de mise à jour des nœuds, c'est-à-dire leur retard éventuel sur l'oplog. Le nœud qui possède les données les plus récentes est favorisé afin d'éviter toute perte ou incohérence.

Enfin, le système vérifie que le nœud candidat est capable de communiquer avec une majorité des autres membres du Replica Set, car un nœud isolé ne peut jamais devenir Primary.

=> L'élection combine donc priorité, fraîcheur des données et connectivité pour garantir un choix cohérent et sûr.

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

Une élection est le processus interne par lequel les nœuds d'un Replica Set se mettent d'accord pour désigner un nouveau Primary. Ce mécanisme se déclenche automatiquement lorsqu'un Primary tombe en panne, devient injoignable ou lorsqu'un nœud secondaire estime être plus apte à devenir Primary (par exemple lors d'une modification de priorité). Chaque nœud vote en analysant l'état des autres membres, et le candidat qui obtient la majorité absolue des voix devient le nouveau Primary. Cette élection garantit la continuité des écritures sans intervention humaine.

18. Que signifie auto-dégradation du Replica Set ? Dans quel cas cela survient-il ?

L'auto-dégradation est un mécanisme de sécurité par lequel un nœud se rétrograde volontairement de Primary à Secondary. Cela survient lorsqu'un Primary constate qu'il ne voit plus une majorité de nœuds, souvent à cause d'une partition réseau. Pour éviter une situation de "split-brain", où deux Primary pourraient exister simultanément dans des parties différentes du réseau, le nœud se désactive automatiquement du rôle de Primary.

=> Cette auto-dégradation permet de maintenir l'unicité du Primary et de préserver la cohérence globale des données.

19. Pourquoi est-il conseillé d'avoir un nombre impair de nœuds dans un Replica Set ?

Il est recommandé d'utiliser un nombre impair de nœuds afin de faciliter l'obtention d'une majorité lors des élections. Avec un nombre impair, il n'existe pas de situation d'égalité, ce qui simplifie le vote et améliore la disponibilité du cluster. À l'inverse, un Replica Set contenant un nombre pair de nœuds peut facilement se retrouver coupé en deux groupes ayant chacun exactement la moitié des membres, ce qui empêcherait l'élection d'un Primary dans l'une des deux parties et rendrait les écritures impossibles.

=> Un nombre impair limite donc les risques de blocage en cas de panne ou de partition réseau.

20. Quelles conséquences a une partition réseau sur le fonctionnement du cluster ?

Une partition réseau scinde le Replica Set en plusieurs groupes de nœuds incapables de communiquer entre eux. Le fonctionnement du cluster dépend alors du groupe qui détient la majorité.

Le groupe majoritaire peut continuer à fonctionner normalement, élire un Primary et accepter les écritures.

En revanche, tout groupe minoritaire est contraint de passer en mode lecture seule, car il n'a pas le droit d'élire un Primary.

Cette restriction évite que différentes parties du réseau créent chacune leur propre version des données. Une partition réseau entraîne donc un risque de retards de réplication, de basculements fréquents et de disponibilité partielle selon l'emplacement de la majorité.

Partie 4 — Scénarios pratiques

21. Vous avez 3 nœuds : 27017 (Primary) , 27018 (Secondary) , et Que se passe-t-il si le Primary devient injoignable ?

Lorsque le Primary devient injoignable, le Replica Set analyse immédiatement s'il reste une majorité disponible. Dans ce cas, le Secondary et l'Arbitre forment ensemble une majorité de deux nœuds sur trois, ce qui permet de déclencher une élection. L'arbitre, bien qu'il ne puisse pas devenir Primary, participe à l'élection en apportant un vote. Le Secondary, qui possède les données les plus à jour parmi les membres restants, est alors élu nouveau Primary. Le cluster continue donc de fonctionner normalement et continue d'accepter les écritures sans interruption. Cette situation montre l'importance d'un arbitre dans une

configuration réduite, puisqu'il permet au cluster de conserver une majorité même lors de la panne du Primary.

22. Vous avez configuré un Secondary avec un `slaveDelay` de 120 secondes. Quelle est son utilité ? Quels usages peut-on en faire dans la vraie vie ?

Un Secondary configuré avec un `slaveDelay` de 120 secondes réplique les données du Primary avec un retard volontaire de deux minutes. Ce mécanisme est très utile pour se prémunir contre les erreurs humaines ou applicatives. Par exemple, si un utilisateur supprime par erreur une collection ou modifie massivement des documents, ces opérations seront appliquées immédiatement sur le Primary mais ne le seront sur ce Secondary qu'après 120 secondes. Cela laisse un délai d'intervention pour récupérer les données non altérées avant que la modification ne se propage. Dans la pratique, ce type de nœud sert souvent pour des restaurations rapides après incident, pour des analyses ou des audits nécessitant un état légèrement antérieur des données, ou encore pour isoler des traitements lourds qui ne doivent pas interférer avec le rythme normal de la réplification.

23. Un client exige une lecture toujours à jour, même en cas de bascule. Quelles options de `readConcern` et `writeConcern` recommanderiez-vous ?

Pour assurer à un client une lecture toujours parfaitement à jour, même lors d'un basculement du Primary, il est recommandé d'utiliser un `readConcern` configuré à "majority". Ce mode de lecture garantit que la donnée retournée est visible et validée par la majorité des nœuds du Replica Set. Parallèlement, il convient d'utiliser un `writeConcern` configuré également à "majority", de manière à ce que chaque écriture ne soit considérée comme terminée que lorsqu'elle a été confirmée par la majorité des nœuds. L'association de ces deux options assure que les lectures obtiennent toujours une version cohérente des données et qu'aucune lecture obsolète n'est possible lorsqu'un basculement survient.

24. Dans une application critique, vous voulez garantir que l'écriture est confirmée par au moins deux nœuds. Quelle option de `writeConcern` devez-vous utiliser ?

Pour garantir que chaque écriture soit validée par au moins deux nœuds du Replica Set, il suffit d'utiliser un `writeConcern` égal à 2. Cette configuration impose au Primary d'attendre la confirmation d'un Secondary avant de répondre au client. Cette contrainte augmente la sécurité des données en réduisant le risque de perte d'informations en cas de panne subite du Primary, puisque l'opération aura déjà été répliquée sur un second membre du cluster.

25. Un étudiant a lu depuis un Secondary et récupéré une donnée obsolète. Expliquez pourquoi et comment éviter cela.

Un Secondary peut parfois renvoyer une donnée obsolète parce que la réplification dans MongoDB est asynchrone. Cela signifie que les modifications appliquées sur le Primary ne sont pas immédiatement visibles sur les Secondaries, qui peuvent accumuler un léger retard. Ainsi, un étudiant qui interroge un Secondary peut recevoir une version antérieure des données. Pour éviter cette situation, il suffit soit de forcer les lectures sur le Primary, soit d'utiliser un `readConcern` configuré à "majority", qui garantit que la donnée retournée a été validée par plusieurs nœuds et n'est donc pas obsolète. Ces mécanismes permettent d'assurer une cohérence forte pour les applications qui l'exigent.

26. Montrez la commande pour vérifier quel nœud est actuellement Primary dans votre Replica Set.

Pour vérifier quel membre du Replica Set joue actuellement le rôle de Primary, il suffit d'exécuter la commande `rs.isMaster()`. Cette commande retourne une structure d'information contenant notamment le champ `primary`, qui mentionne l'adresse du nœud

actuellement élu Primary. Elle permet ainsi d'identifier rapidement l'état du cluster et le nœud maître actif.

27. Expliquez comment forcer une bascule manuelle du Primary sans interruption majeure.

Il est possible de forcer une bascule du Primary en exécutant la commande `rs.stepDown()`. Cette commande ordonne au Primary actuel de renoncer à son rôle, ce qui déclenche immédiatement une élection parmi les autres membres du Replica Set. L'opération peut être réalisée sans redémarrer les nœuds et n'interrompt pas significativement le service, car l'élection est généralement très rapide. Cette méthode est souvent utilisée lors de mises à jour planifiées ou de maintenance nécessitant une prise de rôle par un autre membre.

28. Décrivez la procédure pour ajouter un nouveau nœud secondaire dans un Replica Set en fonctionnement.

Pour ajouter un nouveau nœud secondaire à un Replica Set déjà en production, il suffit d'utiliser la commande `rs.add("adresse_du_nouveau_noeud:port")`. Dès que cette commande est exécutée, MongoDB commence automatiquement la synchronisation initiale du nouveau nœud, en recopiant l'intégralité des données depuis le Primary ou un autre Secondary. Le nœud devient pleinement opérationnel lorsqu'il a rattrapé l'état actuel de la réplication. Cette opération peut être effectuée à chaud, sans interruption du cluster.

29. Quelle commande permet de retirer un nœud défectueux d'un Replica Set ?

Lorsqu'un nœud devient défectueux ou doit être retiré définitivement, il suffit d'utiliser la commande `rs.remove("adresse_du_noeud:port")`. Cette commande met à jour la configuration du Replica Set et supprime officiellement le membre concerné. Les autres nœuds ajustent alors automatiquement leur état et leurs votes en conséquence.

30. Comment configurer un nœud secondaire pour qu'il soit caché (non visible aux clients) ? Pourquoi ferait-on cela ?

Un nœud Secondary peut être configuré comme caché en modifiant sa configuration pour lui attribuer l'attribut `hidden: true` et en lui fixant une priorité égale à 0. Un membre caché ne participe pas aux lectures envoyées par les clients et ne peut pas devenir Primary. Ce type de nœud est particulièrement utile pour réaliser des opérations d'analyse, de reporting ou des sauvegardes sans risquer d'impacter les performances du cluster ou de recevoir des requêtes client.

31. Montrez comment modifier la priorité d'un nœud afin qu'il devienne le Primary préféré.

Pour modifier la priorité d'un nœud, il faut d'abord récupérer la configuration du Replica Set, puis modifier le champ `priority` du membre souhaité avant d'exécuter `rs.reconfig(cfg)`. En attribuant une priorité plus élevée à un nœud donné, on augmente ses chances d'être élu Primary lors des élections futures. Cela permet de contrôler précisément quel membre doit idéalement jouer le rôle principal dans le cluster.

32. Expliquez comment vérifier le délai de réplication d'un Secondary par rapport au Primary.

Le délai de réplication entre un Secondary et le Primary peut être contrôlé grâce à la commande `rs.printSlaveReplicationInfo()`. Cette commande affiche le retard exact en secondes pour chaque Secondary du Replica Set. Elle permet de vérifier si les nœuds répliquent correctement ou s'ils accusent un retard significatif qui pourrait affecter la cohérence des lectures.

33. Que fait la commande `rs.freeze()` et dans quel scénario est-elle utile ?

La commande `rs.freeze()` permet d'empêcher un nœud de participer aux élections pendant une durée déterminée. Cela peut être utile lorsqu'un nœud ne doit pas devenir Primary temporairement, par exemple lors d'une maintenance ou d'une synchronisation importante. En le « gelant », on garantit qu'il restera Secondary le temps nécessaire, ce qui stabilise le comportement du cluster durant les opérations sensibles.

34. Comment redémarrer un Replica Set sans perdre la configuration ?

La configuration d'un Replica Set est stockée dans la base locale de chaque nœud et n'est pas perdue lors d'un redémarrage normal. Ainsi, il suffit de relancer le service MongoDB sur chaque nœud pour que le Replica Set se rétablisse automatiquement avec sa configuration intacte. Aucun réenregistrement manuel n'est nécessaire, ce qui facilite la maintenance du cluster.

35. Expliquez comment surveiller en temps réel la réplication via les logs MongoDB ou commandes shell.

La réplication peut être surveillée en temps réel grâce aux logs MongoDB, qui affichent les opérations appliquées par les Secondaries et leurs éventuels retards.

Du côté du shell MongoDB, la commande `rs.printSlaveReplicationInfo()` fournit immédiatement le retard de chaque nœud. D'autres commandes comme `rs.status()` permettent également d'obtenir un aperçu complet de l'état du Replica Set, incluant la synchronisation, les rôles et les éventuelles anomalies.

Questions complémentaires

37. Qu'est-ce qu'un Arbitre (Arbiter) et pourquoi ne stocke-t-il pas de données ?

Un Arbitre est un membre particulier d'un Replica Set qui participe uniquement aux élections mais ne stocke aucune donnée. Son rôle est de fournir un vote supplémentaire afin de faciliter l'obtention d'une majorité lors du choix d'un Primary. Il ne possède pas de base de données ce qui permet de réduire les ressources matérielles nécessaires à son fonctionnement. Le fait qu'un Arbitre ne stocke pas de données garantit qu'il ne peut ni devenir Primary ni perturber la cohérence du cluster. Il est utilisé principalement dans des configurations où un nombre impair de nœuds est nécessaire pour maintenir la majorité, mais où l'on souhaite limiter les coûts liés à l'ajout d'un serveur complet.

38. Comment vérifier la latence de réplication entre le Primary et les Secondaries ?

La latence de réplication peut être vérifiée à l'aide des outils internes fournis par MongoDB.

La commande `rs.printSlaveReplicationInfo()` permet de connaître précisément le retard accumulé par chaque Secondary dans l'application des opérations en provenance du Primary. Cette commande affiche notamment la durée de retard en secondes, ce qui permet de savoir si les Secondaries répliquent les données à un rythme normal ou s'ils prennent du retard. Il est également possible d'observer la latence dans les logs du serveur, où MongoDB indique la progression du flux de réplication.

39. Quelle commande MongoDB permet d'afficher le retard de réplication des membres secondaires ?

La commande qui permet d'afficher clairement le retard de réplication des membres secondaires est `rs.printSlaveReplicationInfo()`. Cette commande renvoie, pour chaque Secondary, la différence temporelle entre l'opération la plus récente exécutée sur le Primary et celle appliquée sur ce Secondary. Elle constitue l'outil principal pour surveiller en temps réel l'état de la réplication.

40. Quelle est la différence entre la réplication asynchrone et synchrone ? Quel type utilise MongoDB ?

Dans une **réplication synchrone**, le Primary doit attendre que toutes les copies des données soient appliquées sur les nœuds secondaires avant de considérer une opération comme validée. Ce mécanisme assure une cohérence forte mais introduit une latence importante. En revanche, la **réplication asynchrone** permet au Primary de confirmer les écritures immédiatement, sans attendre la propagation aux Secondaries. MongoDB utilise une réplication asynchrone, car elle offre de meilleures performances et une plus grande disponibilité. Il est toutefois possible de renforcer la sécurité des écritures grâce à des `writeConcern` adaptés.

41. Peut-on modifier la configuration d'un Replica Set sans redémarrer les serveurs ?

Oui, MongoDB permet de modifier la configuration d'un Replica Set à chaud, c'est-à-dire sans redémarrer les serveurs. Pour cela, il suffit de récupérer la configuration actuelle, d'effectuer les modifications nécessaires puis d'exécuter la commande `rs.reconfig()`. Le cluster applique immédiatement les changements, ce qui rend la gestion plus souple et évite les interruptions de service.

42. Que se passe-t-il si un nœud Secondary est en retard de plusieurs minutes ?

Lorsqu'un Secondary accuse plusieurs minutes de retard, il est susceptible de renvoyer des données obsolètes si on l'utilise pour les lectures. De plus, ce retard important peut l'empêcher de participer efficacement aux élections, puisqu'un nœud qui n'est pas suffisamment à jour ne peut pas devenir Primary. Si le retard devient trop important, MongoDB peut également décider de réinitialiser la synchronisation du Secondary, ce qui implique une copie complète des données depuis le Primary ou un autre Secondary.

43. Comment MongoDB gère-t-il les conflits de données lors de la réplication ?

MongoDB utilise un modèle dans lequel l'opération la plus récente, déterminée par un horodatage provenant de l'oplog, prévaut en cas de conflit. Ce mécanisme, souvent appelé "last write wins", permet de résoudre automatiquement les divergences entre les nœuds du Replica Set. Comme toutes les écritures doivent passer par le Primary, les conflits sont rares. Lorsqu'un retard de réplication survient, l'ordre temporel impose la cohérence et garantit que tous les nœuds convergent vers le même état.

44. Est-il possible d'avoir plusieurs Primarys simultanément dans un Replica Set ? Pourquoi ?

Il n'est pas possible d'avoir plusieurs Primarys en même temps dans un Replica Set. MongoDB utilise un protocole d'élection strict basé sur la majorité, ce qui garantit qu'un seul nœud peut obtenir suffisamment de votes pour être élu. De plus, un nœud isolé perd automatiquement son statut de Primary afin d'éviter une situation de "split-brain", où deux Primarys indépendants pourraient exister et créer des versions divergentes des données. Cette conception assure une cohérence stricte dans toutes les opérations d'écriture.

45. Pourquoi est-il déconseillé d'utiliser un Secondary pour des opérations d'écriture même en lecture préférée secondaire ?

Même si un Secondary peut être utilisé pour certaines lectures, il ne doit jamais être utilisé pour les écritures. En effet, toutes les écritures doivent transiter par le Primary afin de garantir un ordre global unique et de maintenir un oplog cohérent entre les membres du Replica Set. Si l'on écrivait directement sur un Secondary, cela entraînerait des conflits irréconciliables et casserait le modèle de réplication. De plus, les Secondary sont conçus pour être en retard éventuel, ce qui rendrait toute écriture non fiable.

46. Quelles sont les conséquences d'un réseau instable sur un Replica Set ?

Un réseau instable peut provoquer des pertes de connectivité intermittentes entre les nœuds du Replica Set. Cela peut entraîner des élections répétées, dans lesquelles les nœuds perdent ou reprennent temporairement leur vision de la majorité, ce qui perturbe la stabilité du Primary. Les Secondaries peuvent accumuler du retard dans la réplication, ce qui augmente le risque de lectures obsolètes. Dans les cas extrêmes, un nœud isolé peut s'auto-dégrader pour éviter d'être un Primary fantôme. Un réseau instable réduit donc la cohérence, la disponibilité et la performance du cluster.