



# **Base De Données NoSQL**

## **Réplication**

Réalisé par : Meziane Sarah  
INFOA3-25/26

# Partie 1 :

## Introduction : Réplication et tolérance aux pannes avec MongoDB

La réplication dans MongoDB repose sur un ensemble de nœuds appelé **replica set**, conçu pour assurer la **haute disponibilité**, la **résilience** et la **continuité de service** en cas de panne. Cette architecture fonctionne en mode **maître-esclaves** :

- le nœud principal est appelé **Primary** ;
- les nœuds répliqués sont appelés **Secondary**.

Toutes les **écritures** — et par défaut les **lectures** — sont dirigées vers le Primary. Ce choix garantit que les clients accèdent toujours à la donnée la plus récente, ce qui permet à MongoDB d'offrir une **cohérence forte** (tant que l'on lit sur le Primary).

La réplication entre le Primary et les Secondary est **asynchrone** :

Les modifications sont propagées avec un léger décalage.

Si l'on interroge directement un Secondary, il existe donc un risque de lire une donnée **légèrement en retard**. C'est pourquoi la lecture sur les nœuds secondaires n'est pas recommandée dans les environnements critiques, sauf si l'on configure précisément les **read concerns** et **write concerns**, ainsi que les **read préférences** (par exemple primary, secondary, nearest, etc.).

## Détection des pannes et élection d'un nouveau Primary

Les nœuds du replica set échangent régulièrement des messages appelés **heartbeats**. Grâce à ce mécanisme :

- **Si un Secondary tombe en panne**, le Primary en est immédiatement informé. La réplication continue simplement vers les autres nœuds disponibles.
- **Si le Primary tombe en panne**, les Secondary déclenchent automatiquement un processus d'**élection** afin de désigner un **nouveau Primary**. Ce processus est entièrement automatique et ne nécessite aucune intervention humaine.

Dans le cas où le replica set est réparti sur **plusieurs sous-réseaux**, la **grappe (sous-ensemble) qui possède la majorité des nœuds** procédera à l'élection du nouveau Primary afin de garantir la cohérence globale du système. La règle de la majorité est essentielle pour éviter les situations de **split-brain** (deux Primaries en même temps).

Si **aucun sous-ensemble n'atteint la majorité**, il est possible d'ajouter un **nœud arbitre (arbiter)**. Ce nœud ne stocke pas de données mais participe au vote, permettant ainsi de rétablir une majorité dans les environnements à nombre pair de nœuds.

## 2. Mise en place du replica set : étapes détaillées

## 2.1. Préparation des terminaux et des dossiers

1. Ouvrir **3 terminaux** pour les serveurs MongoDB (un par nœud).
2. Ouvrir **1 terminal client** pour se connecter via mongosh.
3. Créer un dossier de données par serveur, par exemple :

```
mkdir disque1 #pour server1
```

```
mkdir disque2 #pour server2
```

```
mkdir disque3 #pour server3
```

## 2.2. Lancement des 3 nœuds MongoDB

Dans **chaque terminal serveur**, lancer un mongod avec le même nom de replica set mais des ports/dbpath différents.

### Serveur 1 (sera le premier Primary)

```
mongod --replSet monreplicaset --port 27018 --dbpath disque1
```

### Serveur 2

```
mongod --replSet monreplicaset --port 27019 --dbpath disque2
```

### Serveur 3

```
mongod --replSet monreplicaset --port 27020 --dbpath disque3
```

### Rôle de cette commande :

- mongod : lance le serveur MongoDB.
- --replSet monreplicaset : indique que ce nœud fait partie du replica set nommé "**monreplicaset**".
- --port 2701X : port d'écoute du serveur (on utilise un port différent par nœud).
- --dbpath disqueX : chemin vers le répertoire où les données de MongoDB seront stockées pour ce nœud.

Ces trois commandes doivent rester **en cours d'exécution** (ne pas fermer les terminaux).

## 2.3. Initialisation du replica set

Dans le **terminal client**, se connecter au premier serveur :

```
mongosh --port 27018
```

Puis lancer l'initialisation :

```
rs.initiate()
```

```
test> rs.initiate()
{
  info2: 'no configuration specified. Using a default configuration for the set',
  me: 'localhost:27018',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1764856762, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1764856762, i: 1 })
}
monremplicaset [direct: secondary] test> |
```

MongoDB va :

- créer un replica set avec l'ID monremplicaset,
- considérer localhost:27018 comme premier membre,
- élire ce nœud comme **Primary**.

Le prompt devrait ensuite ressembler à quelque chose comme :

```
monremplicaset [direct: primary] test>
```

## 2.4. Ajout des autres nœuds au replica set

Toujours dans mongosh (connecté sur le Primary), ajouter les deux autres nœuds :

```
rs.add("localhost:27019") # 2eme serveur
```

```
rs.add("localhost:27020") # 3eme serveur
```

On obtient ainsi un replica set de 3 membres.

## 2.5. Vérification de la configuration : rs.config()

Pour afficher la configuration du replica set :

```
rs.config()
```

```

monreplicaset [direct: primary] test> rs.config()
{
  _id: 'monreplicaset',
  version: 4,
  term: 1,
  members: [
    {
      _id: 0,
      host: 'localhost:27018',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 1,
      host: 'localhost:27019',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    },
    {
      _id: 2,
      host: 'localhost:27020',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    }
  ]
}

```

### Champs importants :

- `_id` : nom du replica set (monreplicaset).
- `members` : liste des nœuds du replica set.
  - `host` : adresse et port du nœud.
  - `arbiterOnly` : true si le nœud est un arbitre (sans données).
  - `priority` : influence la probabilité qu'un nœud devienne Primary (plus la priorité est haute, plus il a de chances d'être élu).
  - `votes` : nombre de votes que ce nœud possède pour les élections (souvent 1).
- `settings.heartbeatIntervalMillis` : intervalle entre deux heartbeats (en ms).
- `settings.electionTimeoutMillis` : temps après lequel un nœud considère le Primary comme mort et déclenche une élection.

On peut aussi utiliser : `rs.status()`

qui donne des **informations dynamiques** (état des nœuds, retard de réplication, etc.) :

- `stateStr` : PRIMARY, SECONDARY, ARBITER, DOWN, etc.
- `optimeDate` : dernier timestamp répliqué.
- `health` : 1 = OK, 0 = nœud injoignable.

rs.isMaster() (ou db.isMaster() selon la version) permet aussi de savoir si le nœud courant est Primary, Secondary, etc.

## 2.6. Ajout d'un arbitre

Un **arbitre** ne stocke pas de données mais garde des métadonnées pour participer au vote.

1. Lancer un 4<sup>e</sup> mongod (dans un nouveau terminal) :

```
mkdir disque4 # créer un repertoire pour l'arbitre
```

```
mongod --replSet monreplicaset --port 27021 --dbpath disque4
```

2. Ajouter l'arbitre dans le replica set (depuis le Primary, dans mongosh) :

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

Si on refais rs.config() on remarque un nouveau membre avec arbiterOnly: true.

```
{
  _id: 3,
  name: 'localhost:27021',
  health: 1,
  state: 7,
  stateStr: 'ARBITER',
  uptime: 268,
```

Remarque :

À partir des versions récentes de MongoDB, le replica set refuse certaines reconfigurations (comme l'ajout d'un arbitre) si elles modifieraient la write concern implicite.

Pour éviter cela, on définit explicitement une write concern par défaut au niveau du cluster avec la commande :

```
db.adminCommand({
  setDefaultRWConcern: 1,
  defaultWriteConcern: { w: "majority" }
})
```

## 3. Simulation de panne : scénario pas à pas

On propose de faire **2 scénarios** principaux :

- **Scénario 1 : panne d'un Secondary**
- **Scénario 2 : panne du Primary et élection automatique**

### 3.1. Scénario 1 : panne d'un Secondary

**Objectif :** montrer que si un Secondary tombe, le Primary continue à fonctionner normalement.

#### Étapes à exécuter

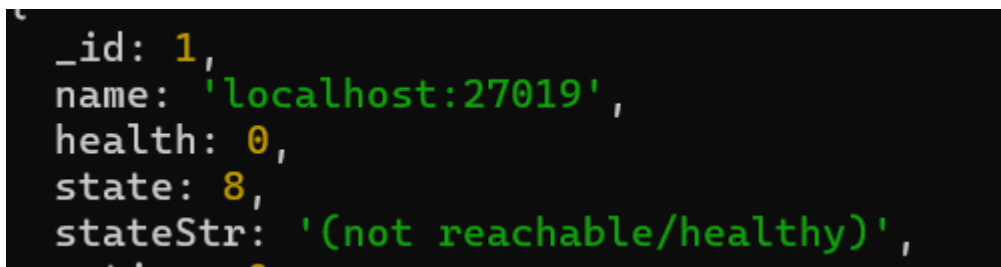
1. Vérifier l'état actuel du replica set (dans mongosh, connecté au Primary) : `rs.status()`

#### 2. Simuler la panne d'un Secondary

Par exemple, arrêter le nœud sur le port **27019**.

- Aller dans le terminal où tourne mongod --port 27019
- Faire Ctrl + C pour arrêter le processus.

Revenir dans mongosh (toujours connecté au Primary, port 27018) et relancer : `rs.status()`



```
_id: 1,
name: 'localhost:27019',
health: 0,
state: 8,
stateStr: '(not reachable/healthy)',
```

Le Primary reste **PRIMARY**.

L'autre Secondary (27020) reste **SECONDARY**.

Donc, la panne d'un Secondary n'empêche ni les écritures ni les lectures sur le Primary. Le replica set reste fonctionnel, mais la redondance est réduite (moins de copies des données).

### 3.2. Scénario 2 : panne du Primary et élection d'un nouveau Primary

#### Simuler la panne du Primary

Si localhost:27018 est le Primary :

- Aller dans le terminal où tourne mongod --port 27018
- Faire Ctrl + C pour arrêter ce processus.

Se connecter maintenant à un des autres nœuds avec mongosh, par exemple sur 27019 :

`mongosh --port 27019`

Vérifier l'état du replica set depuis ce nœud : `rs.status()`

On remarque que le nouveau primary est 27020

```
_id: 2,  
name: 'localhost:27020',  
health: 1,  
state: 1,  
stateStr: 'PRIMARY',  
uptime: 84,
```

### 3.3. Scénario 3 : panne + arbitre

1. Replica set composé de :
  - 27018 (données)
  - 27019 (données)
  - 27021 (arbitre)
2. Arrêter 27019 (un nœud de données).
3. Tant que 27018 + 27021 sont en ligne, il y a encore 2 votes sur 3 → majorité atteinte.
4. Si tu arrêtes **le Primary** mais gardes un Secondary + un arbitre, ce Secondary pourra encore devenir Primary (il a la majorité avec l'arbitre).

## 4. Création d'une collection et insertion d'éléments

Pour ajouter des documents dans la base de données, les opérations doivent être effectuées **sur le Primary** (dans notre cas : le nœud sur le port **27018**).

### Étapes sur le Primary (27018)

1. **Changer de base de données** : use demo1
2. **Créer la collection** : db.createCollection("personnes")
3. **Insérer des documents** :

```
db.personnes.insert( { "nom": "sarah" }
```

```
monreplicaset [direct: primary] test> db.personnes.insert({ "nom": "sarah" })  
{  
  acknowledged: true,  
  insertedIds: { '0': ObjectId('6936bb1689c1e1954b9dc2a0') }  
}
```

4. **Afficher le contenu de la collection** : db.personnes.find()

```
monrempliset [direct: primary] demo1> db.personnes.find()
[
  { _id: ObjectId('6936bddb38aeb954359dc29d'), name: 'sarah' },
  { _id: ObjectId('6936bde338aeb954359dc29e'), name: 'noah' },
  { _id: ObjectId('6936bde838aeb954359dc29f'), name: 'fatima' }
]
```

### Étapes sur un Secondary (27019)

1. Changer de base : use demo1
2. Afficher les données répliquées :db.personnes.find()

```
monrempliset [direct: secondary] test> use demo1
switched to db demo1
monrempliset [direct: secondary] demo1> db.personnes.find()
[
  { _id: ObjectId('6936bddb38aeb954359dc29d'), name: 'sarah' },
  { _id: ObjectId('6936bde338aeb954359dc29e'), name: 'noah' },
  { _id: ObjectId('6936bde838aeb954359dc29f'), name: 'fatima' }
]
```

On constate que les données insérées sur le Primary sont bien visibles sur le Secondary grâce à la réplication.

Important : impossibilité d'écrire sur un Secondary

Si l'on tente une insertion sur un Secondary, MongoDB renvoie une erreur :

```
monrempliset [direct: secondary] demo1> db.personnes.insert({"name":"haha"})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
Uncaught:
MongoBulkWriteError[NotWritablePrimary]: not primary
Result: BulkWriteResult {
  insertedCount: 0,
  matchedCount: 0,
  modifiedCount: 0,
  deletedCount: 0,
  upsertedCount: 0,
  upsertedIds: {},
  insertedIds: { '0': ObjectId('6936bf05016f7021889dc29d') }
}
Write Errors: []
```

C'est un comportement normal :

MongoDB impose que toutes les écritures passent par le Primary pour garantir une cohérence forte et éviter les conflits entre versions.

## Section 2 :

# Partie 1 — Compréhension de base

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

Un replica set est un ensemble de nœuds MongoDB qui contiennent la même donnée afin d'assurer la haute disponibilité et la tolérance aux pannes. Un nœud est élu Primary, les autres sont des Secondaries qui répliquent les données du Primary.

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

Le Primary reçoit **toutes les écritures** et, par défaut, la majorité des lectures. C'est le seul nœud autorisé à modifier la base.

### 3. Quel est le rôle essentiel des Secondaries ?

Ils répliquent de manière asynchrone les données du Primary. Ils servent de **sauvegarde en cas de panne** et peuvent devenir Primary lors d'une élection.

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

Pour garantir la **cohérence forte** et éviter les conflits entre versions. Les Secondaries n'ont pas forcément la dernière version de la donnée au moment d'une écriture.

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

Cela signifie que lorsqu'on lit sur le Primary, on obtient **toujours la dernière valeur confirmée**. Aucune lecture obsolète n'est possible.

### 6. Différence entre readPreference: "primary" et "secondary" ?

- primary : garantit une lecture toujours à jour.
- secondary : peut améliorer la répartition de charge, mais les données peuvent être légèrement en retard.

### 7. Dans quel cas souhaiterait-on lire sur un Secondary malgré les risques ?

Pour des **analyses**, des **rapports**, des **requêtes lourdes** ou des **lectures non critiques**, afin de décharger le Primary.

---

## Partie 2 — Commandes & configuration

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

```
rs.initiate()
```

### 9. Comment ajouter un nœud après initialisation ?

```
rs.add("host:port")
```

### 10. Quelle commande affiche l'état actuel du Replica Set ?

```
rs.status()
```

### 11. Comment identifier le rôle actuel d'un nœud ?

Avec :

- `rs.status()` → champs `stateStr`
- ou `db.isMaster()` → indique si le nœud est Primary / Secondary / Arbiter

### 12. Quelle commande force un basculement du Primary ?

```
rs.stepDown()
```

### 13. Comment désigner un nœud comme Arbitre ? Pourquoi ?

```
rs.addArb("host:port")
```

On le fait lorsqu'on veut **ajouter un vote** supplémentaire sans stocker plus de données, par exemple dans un ensemble avec nombre pair de nœuds.

### 14. Commande pour configurer un délai de réplication (`slaveDelay`)

Cela se fait en modifiant la configuration d'un membre :

```
cfg = rs.config()
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 et qu'il n'y a pas de majorité ?

Aucun nouveau Primary ne peut être élu → le replica set passe en **lecture seule**.

### 16. Comment MongoDB choisit-il un nouveau Primary ?

Sur la base de :

- priorité (priority)
- nombre de votes disponibles
- état des nœuds
- retard de réplication (le plus à jour est favorisé)

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

Processus automatique où les Secondaries votent pour élire un nouveau Primary lorsque l'ancien devient injoignable.

### 18. Que signifie auto-dégradation du Replica Set ?

Le Primary se met volontairement en mode Secondary lorsqu'il **perd la majorité** pour éviter un split-brain.

### 19. Pourquoi avoir un nombre impair de nœuds ?

Pour maximiser la probabilité d'obtenir une **majorité** et garantir une élection même en cas de panne.

### 20. Effets d'une partition réseau sur le cluster ?

Les nœuds isolés peuvent perdre la majorité, se dégrader en Secondary et devenir inaccessibles en écriture.

## Partie 4 — Scénarios pratiques

**\*\*21. 3 nœuds : 27017 (Primary), 27018 (Secondary), 27019 (Arbitre).**

Que se passe-t-il si le Primary devient injoignable ?\*\*

Le Secondary et l'Arbitre possèdent la majorité des votes → le Secondary devient **Primary**.

**22. secondary avec slaveDelay=120 sec : utilité ?**

Utile comme **protection contre les erreurs humaines** :

si une mauvaise manipulation est écrite sur le Primary, le Secondary retardé permet de récupérer une copie plus ancienne.

Exemple : restauration suite à un drop.

**23. Client exige une lecture toujours à jour même en cas de bascule :**

Je recommanderais :

- `writeConcern: "majority"`
- `readConcern: "majority"`

Cela garantit que les données lues et écrites sont confirmées par plusieurs nœuds.

**24. Pour garantir que l'écriture est confirmée par au moins deux nœuds ?**

`writeConcern: { w: 2 }`

**25. Pourquoi un étudiant a lu une donnée obsolète depuis un Secondary ?**

Parce que la réplication est **asynchrone**.

Pour éviter cela :

- lire sur le Primary
- ou utiliser `readConcern: "majority"`

**26. Commande pour vérifier quel nœud est Primary**

`rs.status()`

ou

`db.isMaster()`

## 27. Comment forcer un basculement manuel du Primary ?

Depuis le Primary :

```
rs.stepDown()
```

## 28. Ajouter un nouveau nœud secondaire dans un replica set actif

1. Lancer mongod avec le bon --replSet
2. Depuis le Primary :

```
rs.add("host:port")
```

## 29. Retirer un nœud défectueux du Replica Set

```
rs.remove("host:port")
```

## 30. Rendre un Secondary caché (hidden) et pourquoi ?

```
cfg = rs.config()
cfg.members[1].hidden = true
cfg.members[1].priority = 0
rs.reconfig(cfg)
```

On fait cela pour un **nœud dédié aux sauvegardes** ou aux analyses, invisible aux applications.

## 31. Modifier la priorité d'un nœud pour qu'il devienne Primary préféré

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

## 32. Vérifier le délai de réplication entre un Secondary et le Primary

```
rs.status()
```

Regarder optimeDate ou replicationLag.

## 33. Que fait rs.freeze() ?

Gèle un Secondary pour empêcher temporairement qu'il devienne Primary pendant X secondes.  
Utile pour des opérations de maintenance.

### 34. Redémarrer un replica set sans perdre la configuration

La configuration est stockée dans les données du Primary.

Il suffit de **redémarrer chaque mongod** normalement : la configuration est automatiquement rechargée.

### 35. Surveiller la réplication en temps réel

- Via les logs (mongod --verbose)
- Ou via rs.status()
- Ou via db.printReplicationInfo() et db.printSlaveReplicationInfo()

---

## Questions complémentaires

### 37. Qu'est-ce qu'un Arbitre ? Pourquoi ne stocke-t-il pas de données ?

Un arbitre est un nœud qui **participe au vote** mais n'a pas de données.

Il sert uniquement à maintenir la **majorité**, il n'a pas besoin de stockage.

### 38. Vérifier la latence de réplication

Via les timestamps optime dans rs.status().

### 39. Afficher le retard de réplication

db.printSlaveReplicationInfo()

### 40. Différence entre réplication asynchrone et synchrone ? Quel type utilise MongoDB ?

- **Synchrone** : tous les nœuds confirment avant validation.
- **Asynchrone** : les nœuds répliquent après coup.  
MongoDB utilise **une réplication asynchrone**.

### 41. Modifier la configuration d'un Replica Set sans redémarrer ?

Oui : via

rs.reconfig(cfg)

#### **42. Si un Secondary est en retard de plusieurs minutes ?**

Il continue à rattraper, mais :

- il ne peut pas devenir Primary
- les lectures dessus risquent d'être très obsolètes

#### **43. Comment MongoDB gère les conflits de données ?**

Le Primary est la source de vérité → les Secondaries appliquent ses journaux.  
Donc **pas de conflit possible**, contrairement à des modèles multi-master.

#### **44. Peut-on avoir plusieurs Primaries ? Pourquoi ?**

Non, à cause du mécanisme de **majorité** et de **détection de split-brain**.

#### **45. Pourquoi déconseiller l'écriture sur Secondary même en readPreference secondaire ?**

Parce que MongoDB **n'autorise pas les écritures sur les Secondaries** → risque de confusion et d'incohérence.

#### **46. Conséquences d'un réseau instable sur un Replica Set ?**

- pertes de majorité
- ré-élections fréquentes
- bascules intempestives
- risque d'indisponibilité en écriture