

# ReplicatSet

Pr. Zainab OUFQIR



# MongoDB



# Qu'est-ce que la RéPLICATION ?

## Définition:

- La réPLICATION dans MongoDB est un mécanisme qui permet d'avoir plusieurs copies de vos données sur différents serveurs.
- C'est un processus fondamental qui consiste à synchroniser les données entre plusieurs serveurs.
- Elle s'organise sous forme d'un "Replica Set", qui est un groupe de serveurs MongoDB travaillant ensemble pour assurer la redondance et la disponibilité des données.

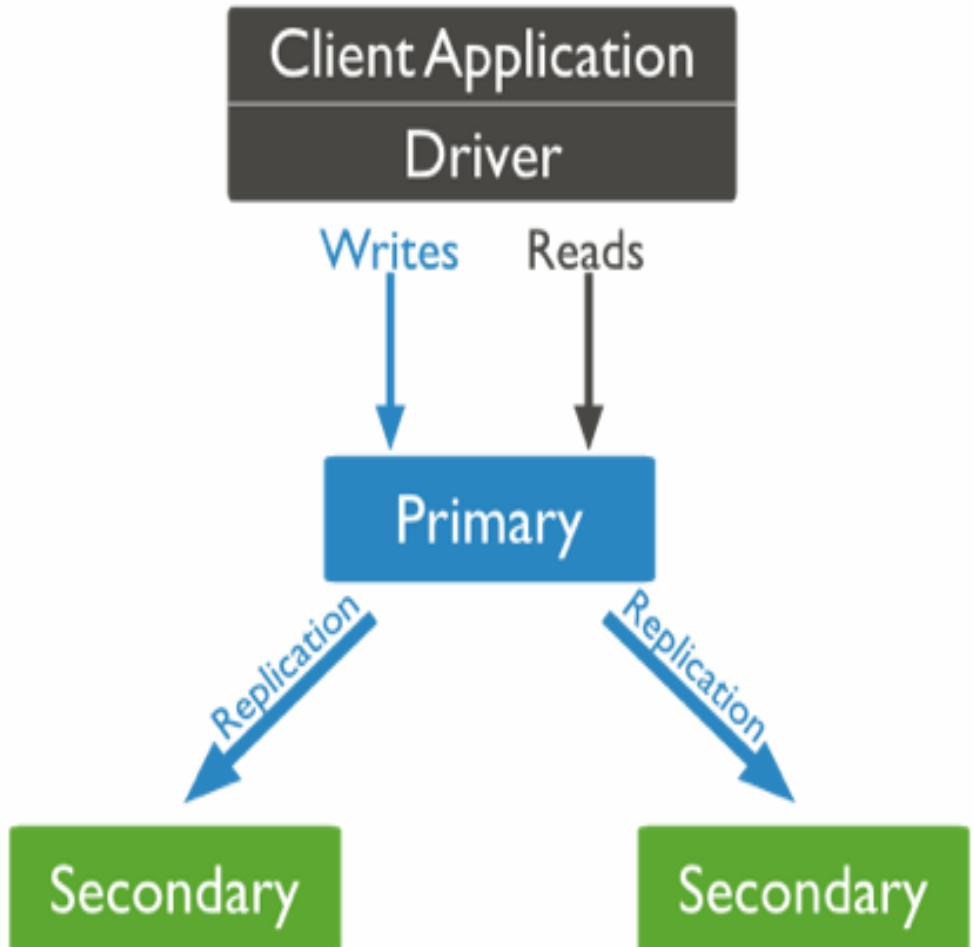
# Pourquoi Utiliser la RéPLICATION ?

Voici les principales raisons d'utiliser la réPLICATION dans MongoDB :

1. **Haute Disponibilité:** La réPLICATION assure la continuité du service en maintenant plusieurs copies des données sur différents serveurs, permettant un basculement automatique en cas de panne du serveur principal.
2. **Protection des Données:** Les données sont sécurisées grâce à la maintenance de plusieurs copies sur différents serveurs, offrant une protection contre les pannes matérielles et les corruptions tout en permettant des sauvegardes à chaud.
3. **Performance Améliorée:** La distribution des lectures sur les nœuds secondaires permet de réduire la charge sur le serveur principal et d'optimiser les performances globales du système.
4. **Distribution Géographique:** Les réPLICAS peuvent être placés dans différentes zones géographiques pour réduire la latence pour les utilisateurs locaux et assurer une protection contre les pannes de datacenter.

# ReplicaSet MongoDB

- Un ReplicaSet est une forme d'organisation où plusieurs serveurs MongoDB qui garantit que chaque serveur membre maintient une copie identique des données.
- Architecture basée sur le principe maître/esclave.
- Minimum recommandé de 3 membres pour la production.
- Les membres du ReplicaSet communiquent continuellement entre eux pour s'assurer que les données restent synchronisées.

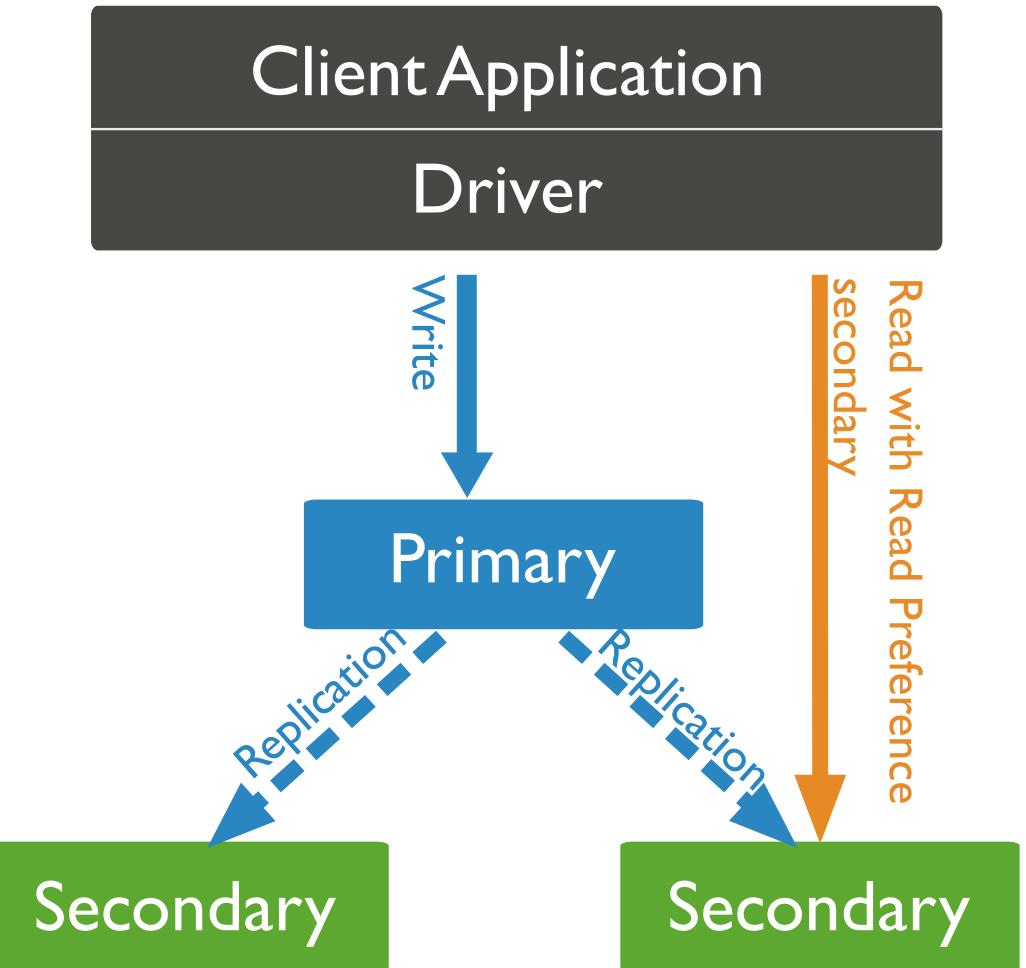


# ReplicaSet MongoDB

Par défaut, toutes les lectures se font sur le nœud primaire, mais MongoDB offre la flexibilité de configurer différentes préférences de lecture pour plusieurs raisons essentielles :

- Répartit la charge de lecture sur plusieurs serveurs
- Réduit la pression sur le nœud primaire qui gère déjà toutes les écritures
- Améliore les temps de réponse globaux

Les lectures sont toujours dirigées vers le primaire, garantissant la cohérence des données.



# Architecture Maître/Esclave

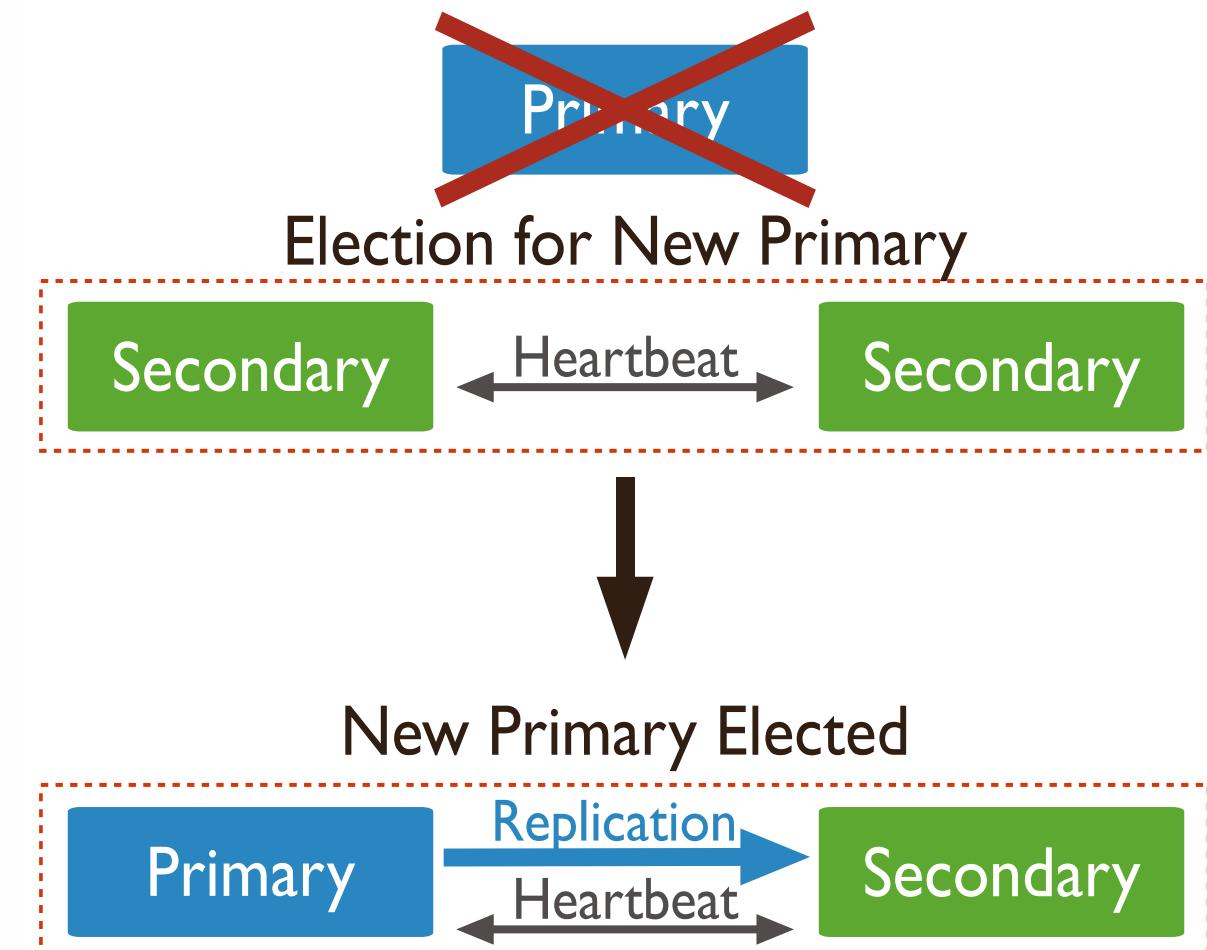
Le Maître (Primary) :

- Il est unique dans le ReplicaSet à un instant donné : Un seul nœud peut être primaire à la fois pour garantir la cohérence des données.
- C'est le récepteur exclusif de toutes les opérations d'écriture : Toutes les écritures doivent passer par le primaire pour maintenir la cohérence.
- Il enregistre chaque opération dans l'oplog : Chaque modification est enregistrée dans ce journal chronologique pour permettre la synchronisation.
- Il envoie des heartbeats aux secondaires : Envoie régulièrement des signaux pour indiquer qu'il est fonctionnel.

# Architecture Maître/Esclave

Les Esclaves (Secondaries) :

- Maintiennent une copie exacte des données : Gardent une réplique complète et à jour des données du primaire.
- Appliquent les opérations dans le même ordre : Respectent l'ordre chronologique des opérations du primaire (oplog).
- Peuvent servir les requêtes de lecture : Capables de gérer les lectures pour répartir la charge si configurées.
- Surveillent l'état du primaire : Vérifient constamment que le primaire est fonctionnel.
- Peuvent devenir primaire : Prennent le relais si le primaire actuel tombe en panne.



# Architecture Maître/Esclave

## L'Arbitre (Arbiter):

- Ne stocke aucune donnée du ReplicaSet : Contrairement aux autres membres, l'arbitre ne conserve aucune copie des données, ce qui le rend très léger en ressources.
- Participe uniquement aux votes lors des élections : Son seul rôle est de voter lors des élections d'un nouveau primaire.
- Permet d'avoir un nombre impair de votes : Il ajoute un vote supplémentaire pour éviter les égalités lors des élections, ce qui pourrait bloquer le système.
- Ne peut jamais devenir primaire : Il n'est pas éligible pour devenir primaire car il ne contient pas de données.

# Instancier un ReplicaSet

Un ReplicaSet nécessite plusieurs serveurs MongoDB (un primaire, des secondaires et éventuellement un arbitre) qui travaillent ensemble pour maintenir le même jeu de données.

Pour cela, les paramètres de chaque serveur doivent être définis :

- Un nom de ReplicaSet : **rs0**, c'est un identifiant commun qui permet à tous les membres du ReplicaSet de se reconnaître et de communiquer entre eux.
- Les ports différents (**27018, 27019, 27020**) permettent de faire fonctionner plusieurs instances MongoDB sur la même machine, ce qui est essentiel pour les tests et le développement.
- Créer un répertoire dédié : **/data/R0S1** (ReplicaSet 0/Serveur 1) - Sous Windows, utilisez le répertoire C:\data\R0S1
- Ouvrir une console et aller dans le répertoire de MongoDB (\$CheminMongoDb/bin) ou bien l'ajouter dans les variables d'environnement système
- La commande **mongod --replSet rs0 --port 27018 --dbpath /data/R0S1** démarre le premier serveur du ReplicaSet. Les paramètres définissent son identité et son emplacement de stockage.

# Instancier un ReplicaSet

Cette opération devra être répétée pour chaque serveur du ReplicaSet, les répertoires R0S1, R0S2, R0S3 stockent les données de chaque serveur du ReplicaSet. Chaque serveur doit avoir son propre espace de stockage. La structure des répertoires sous Windows est la suivante:

- C:\data\R0S1 - Pour le premier serveur
- C:\data\R0S2 - Pour le deuxième serveur
- C:\data\R0S3 - Pour le troisième serveur

# Processus d'Initialisation

La connexion avec `mongosh --port 27018` permet d'accéder à l'interface de commande pour configurer le serveur:

```
PS C:\> mongosh --port 27018
Current Mongosh Log ID: 6730ffb80680dd3b9586b01c
Connecting to:          mongodb://127.0.0.1:27018/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.3.2
Using MongoDB:          8.0.1
Using Mongosh:          2.3.2
```

La commande `rs.initiate()` transforme le serveur standalone en membre d'un ReplicaSet, c'est la première étape pour mettre en place la réPLICATION des données entre plusieurs serveurs MongoDB:

```
test> rs.initiate()
{
  info2: 'no configuration specified. Using a default configuration for the set',
  me: 'localhost:27018',
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1731268549, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAA='), 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1731268549, i: 1 })
}
```

# Ajouter des Membres Secondaires

Les serveurs secondaires sont démarrés de la même manière que le primaire mais avec des ports différents.

La commande `rs.add("hostname:port")` permet d'ajouter un nouveau serveur en tant que membre secondaire au ReplicaSet, ce serveur va automatiquement se synchroniser avec le primaire et participer aux élections.

```
rs0 [direct: primary] test> rs.add("localhost:27019")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1731271243, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAA
AAAAAAA='), 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1731271243, i: 1 })
}
```

```
rs0 [direct: primary] test> rs.add("localhost:27020")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1731271323, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAA
AAAAAAA='), 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1731271323, i: 1 })
}
```

# Configuration du ReplicaSet

La commande `rs.conf()` affiche la configuration complète du ReplicaSet, montrant comment les serveurs sont organisés et leurs rôles respectifs:

- `_id` : Nom du ReplicaSet (ici 'rs0')
- `version` : Version de la configuration
- `members` : Liste des serveurs avec leurs paramètres spécifiques
  - `host` : Adresse et port du serveur
  - `priority` : Priorité dans les élections (1 par défaut)
  - `arbiterOnly` : Indique si c'est un arbitre
- Paramètres de monitoring (`settings`) :
  - `heartbeatIntervalMillis`: Intervalle entre les vérifications (2000ms)
  - `heartbeatTimeoutSecs`: Délai avant de considérer un membre comme inaccessible (10s)

```
rs0 [direct: primary] test> rs.config()
{
  _id: 'rs0',
  version: 1,
  term: 1,
  members: [
    {
      _id: 0,
      host: 'localhost:27018',
      arbiterOnly: false,
      buildIndexes: true,
      hidden: false,
      priority: 1,
      tags: {},
      secondaryDelaySecs: Long('0'),
      votes: 1
    }
  ],
  protocolVersion: Long('1'),
  writeConcernMajorityJournalDefault: true,
  settings: {
    chainingAllowed: true,
    heartbeatIntervalMillis: 2000,
    heartbeatTimeoutSecs: 10,
    electionTimeoutMillis: 10000,
    catchUpTimeoutMillis: -1,
    catchUpTakeoverDelayMillis: 30000,
    getLastErrorModes: {},
    getLastErrorDefaults: { w: 1, wtimeout: 0 },
    replicaSetId: ObjectId('67310fc5e12a4b640a5be41b')
  }
}
```

# L'Arbitre dans le ReplicaSet

Un arbitre est un membre léger du ReplicaSet qui **ne stocke pas de données mais participe uniquement aux votes lors des élections du primaire.**

L'arbitre est essentiel pour maintenir un nombre impair de votants dans le ReplicaSet, évitant ainsi les situations de "split-brain" où deux serveurs pourraient se considérer comme primaire à cause d'une égalité de votes.

L'arbitre est un membre léger qui consomme peu de ressources système car il ne stocke et ne réplique aucune donnée, contrairement aux autres membres du ReplicaSet.

# Mise en Place de l'Arbitre

Pour configurer un arbitre, nous devons créer un répertoire dédié: `C:\data\arb`, puis démarrer une instance MongoDB avec les paramètres spécifiques : le port 30000, le chemin vers le répertoire de données et le nom du ReplicaSet: `mongod --port 30000 --dbpath /data/arb --replSet rs0`

L'ajout d'un arbitre modifie la structure du ReplicaSet et donc le nombre de votes possibles. Pour cela, nous allons exécuter la commande `db.adminCommand` établit au préalable la règle de validation des écritures (ici, confirmation par le primaire seulement avec `w: 1`), évitant ainsi des problèmes de cohérence lors de l'ajout de l'arbitre qui ne stocke pas de données mais participe aux votes.

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

# Mise en Place de l'Arbitre

La commande `rs.addArb("hostname:port")` permet d'ajouter un arbitre au ReplicaSet pour assurer une meilleure stabilité dans le processus d'élection en maintenant un nombre impair de votes.

```
rs0 [direct: primary] test> rs.addArb("localhost:30000")
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1731274401, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAA='),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1731274401, i: 1 })
}
```

# Configuration du ReplicaSet

Exécuter la commande `rs.conf()` pour afficher la nouvelle configuration de notre replicaset:

```
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
  },
  {
    _id: 3,
    host: 'localhost:30000',
    arbiterOnly: true,
    buildIndexes: true,
    hidden: false,
    priority: 0,
    tags: {},
    secondaryDelaySecs: Long('0'),
    votes: 1
  }
],
```

# Statue du ReplicaSet

Affiche un aperçu complet de l'état de santé du ReplicaSet avec la commande `rs.status()`, notamment :

- Le rôle de chaque membre (PRIMARY, SECONDARY, ARBITER)
- L'état de la réPLICATION et la synchronisation
- Les temps de réponse et la connectivité entre les membres

```
_id: 0,  
name: 'localhost:27018',  
health: 1,  
state: 1,  
stateStr: 'PRIMARY',
```

```
_id: 2,  
name: 'localhost:27020',  
health: 1,  
state: 2,  
stateStr: 'SECONDARY',
```

```
_id: 1,  
name: 'localhost:27019',  
health: 1,  
state: 2,  
stateStr: 'SECONDARY',
```

```
_id: 3,  
name: 'localhost:30000',  
health: 1,  
state: 7,  
stateStr: 'ARBITER',
```

# Tester la réplication

Pour tester la haute disponibilité du ReplicaSet, nous allons simuler une [panne en arrêtant le serveur PRIMARY](#) qui utilise le port 27018, ce qui [déclenchera automatiquement l'élection d'un nouveau PRIMARY](#) parmi les serveurs [SECONDARY](#).

```
_id: 0,  
name: 'localhost:27018',  
health: 0,  
state: 8,  
stateStr: '(not reachable/healthy)',
```

```
_id: 1,  
name: 'localhost:27019',  
health: 1,  
state: 2,  
stateStr: 'SECONDARY',
```

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

```
_id: 3,  
name: 'localhost:30000',  
health: 1,  
state: 7,  
stateStr: 'ARBITER',
```

Suite à l'arrêt du PRIMARY, c'est l'arbitre (port 30000) qui a participé au vote pour élire le serveur sur le port 27020 comme nouveau PRIMARY.