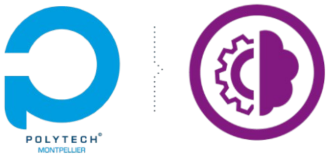

Bases de Données Avancées

MongoDB : Réplication & Sharding

Réalisé par

Axel FRACHE

Étudiant en DO4 à Polytech Montpellier

Année universitaire 2025-2026

Table des matières

Introduction

Ce rapport présente la mise en œuvre pratique de deux mécanismes essentiels des bases de données distribuées : la **réplication** et le **sharding**. Ces mécanismes répondent à deux problématiques majeures des systèmes modernes : la **haute disponibilité** et la **scalabilité**.

La réplication permet de conserver plusieurs copies identiques des données sur différents serveurs afin d'assurer la continuité du service en cas de panne. Le sharding, quant à lui, consiste à répartir les données sur plusieurs serveurs afin de pouvoir gérer des volumes de données et de requêtes plus importants qu'avec une seule machine.

Dans ce TP, ces deux mécanismes sont étudiés à travers MongoDB, une base de données NoSQL orientée documents, qui propose nativement la réplication via les *ReplicaSets* et le sharding via les *Sharded Clusters*. L'ensemble des fichiers de configuration, scripts d'automatisation et ressources associées à ce TP est disponible dans un dépôt GitHub dédié :

https://github.com/axelfrache/mongodb-ha-replication-sharding

I Réplication MongoDB

1.1 Principe de la réplication

Un **ReplicaSet** MongoDB est un groupe de processus `mongod` maintenant le même jeu de données. La réplication permet avant tout d'assurer la disponibilité du système en cas de défaillance d'un nœud. En maintenant plusieurs copies des données, elle garantit qu'un autre serveur peut prendre le relais sans interruption du service. Elle permet également une redondance des données et, dans certains cas, de répartir les lectures sur les nœuds secondaires.

Le ReplicaSet fonctionne selon un modèle **PRIMARY/SECONDARY** :

— Le **PRIMARY** reçoit toutes les écritures

— Les **SECONDARY** répliquent les données via l'*oplog*

— Un **ARBITER** participe aux votes sans stocker de données

1.2 Architecture RS1

Le ReplicaSet RS1 déployé comprend 5 nœuds :

Nœud	Port	Priorité	Rôle
mongo-primary	27017	2	PRIMARY (écriture)
mongo-secondary1	27018	1	SECONDARY
mongo-secondary2	27019	1	SECONDARY
mongo-secondary3	27020	1	SECONDARY
mongo-arbiter	30000	0	ARBITER (vote uniquement)

TABLE 1 – Architecture du ReplicaSet RS1

La priorité 2 du PRIMARY assure qu'il sera systématiquement réélu après un redémarrage.

1.3 Mise en œuvre

Le déploiement s'effectue via Docker Compose. Chaque conteneur représente un nœud MongoDB configuré avec l'option `--replSet RS1`. L'initialisation du ReplicaSet est réalisée via la commande `rs.initiate()` avec la configuration des membres.

```
mongodb-ha-replication-sharding main ? ❯ docker compose -f docker-compose.replication.yml ps
NAME                IMAGE       COMMAND                  SERVICE             CREATED          STATUS                    PORTS
mongo-arbiter       mongo:7.0   "docker-entrypoint.s…"   mongo-arbiter       49 seconds ago   Up 48 seconds (healthy)   0.0.0.0:30000->27017/tcp, [::]:30000->27017/tcp
mongo-primary       mongo:7.0   "docker-entrypoint.s…"   mongo-primary       50 seconds ago   Up 49 seconds (healthy)   0.0.0.0:27017->27017/tcp, [::]:27017->27017/tcp
mongo-secondary1    mongo:7.0   "docker-entrypoint.s…"   mongo-secondary1    49 seconds ago   Up 48 seconds (healthy)   0.0.0.0:27018->27017/tcp, [::]:27018->27017/tcp
mongo-secondary2    mongo:7.0   "docker-entrypoint.s…"   mongo-secondary2    49 seconds ago   Up 48 seconds (healthy)   0.0.0.0:27019->27017/tcp, [::]:27019->27017/tcp
mongo-secondary3    mongo:7.0   "docker-entrypoint.s…"   mongo-secondary3    49 seconds ago   Up 48 seconds (healthy)   0.0.0.0:27020->27017/tcp, [::]:27020->27017/tcp
```

FIGURE 1 – Conteneurs Docker composant le ReplicaSet RS1

Une fois initialisé, le ReplicaSet élit automatiquement un PRIMARY. Les nœuds SECONDARY synchronisent les données à partir de l'*oplog*, journal des opérations enregistré sur le PRIMARY et rejoué par les secondaires.

1.4 Tests de réplication

1.4.1 Vérification de l'état du cluster

La commande `rs.status()` permet de confirmer l'état de chaque membre du ReplicaSet :

— 1 nœud en état `PRIMARY`

— 3 nœuds en état `SECONDARY`

— 1 nœud en état `ARBITER`

```
mongodb-ha-replication-sharding main ? ❯ docker exec mongo-primary mongosh --eval "rs.status()
"
{
  set: 'RS1',
  date: ISODate('2026-02-08T21:10:19.454Z'),
  myState: 1,
  term: Long('1'),
  syncSourceHost: '',
  syncSourceId: -1,
  heartbeatIntervalMillis: Long('2000'),
  majorityVoteCount: 3,
  writeMajorityCount: 3,
  votingMembersCount: 5,
  writableVotingMembersCount: 4,
  optimes: {
    lastCommittedOpTime: { ts: Timestamp({ t: 1770585018, i: 1 }), t: Long('1') },
    lastCommittedWallTime: ISODate('2026-02-08T21:10:18.690Z'),
    readConcernMajorityOpTime: { ts: Timestamp({ t: 1770585018, i: 1 }), t: Long('1') },
    appliedOpTime: { ts: Timestamp({ t: 1770585018, i: 1 }), t: Long('1') },
    durableOpTime: { ts: Timestamp({ t: 1770585018, i: 1 }), t: Long('1') },
    lastAppliedWallTime: ISODate('2026-02-08T21:10:18.690Z'),
    lastDurableWallTime: ISODate('2026-02-08T21:10:18.690Z')
  },
  lastStableRecoveryTimestamp: Timestamp({ t: 1770584988, i: 1 }),
  electionCandidateMetrics: {
    lastElectionReason: 'electionTimeout',
    lastElectionDate: ISODate('2026-02-08T21:08:08.632Z'),
    electionTerm: Long('1'),
    lastCommittedOpTimeAtElection: { ts: Timestamp({ t: 1770584877, i: 1 }), t: Long('-1') },
    lastSeenOpTimeAtElection: { ts: Timestamp({ t: 1770584877, i: 1 }), t: Long('-1') },
    numVotesNeeded: 3,
    priorityAtElection: 2,
    electionTimeoutMillis: Long('10000'),
    numCatchUpOps: Long('0'),
    newTermStartDate: ISODate('2026-02-08T21:08:08.674Z'),
    wMajorityWriteAvailabilityDate: ISODate('2026-02-08T21:08:09.199Z')
  },
  members: [
    {
      _id: 0,
      name: 'mongo-primary:27017',
      health: 1,
      state: 1,
      stateStr: 'PRIMARY',
      uptime: 143,
      optime: { ts: Timestamp({ t: 1770585018, i: 1 }), t: Long('1') },
      optimeDate: ISODate('2026-02-08T21:10:18.000Z'),
      lastAppliedWallTime: ISODate('2026-02-08T21:10:18.690Z'),
      lastDurableWallTime: ISODate('2026-02-08T21:10:18.690Z'),
```

FIGURE 2 – État du ReplicaSet RS1 obtenu via la commande `rs.status()`

1.4.2 Test de propagation des données

Les tests réalisés permettent de vérifier le bon fonctionnement de la réplication :

1. Insertion d'un document sur le PRIMARY
2. Attente de quelques secondes (réplication asynchrone)
3. Lecture réussie du même document sur un nœud SECONDARY

```
"MovieLens").users.insertOne({name: "Test"})'
{
  acknowledged: true,
  insertedId: ObjectId('6988fc05b657d678478ce5b0')
}

mongodb-ha-replication-sharding main ? ❯ docker exec mongo-secondary1 mongosh --eval 'db.getMongo()
.setReadPref("secondary"); db.getSiblingDB("MovieLens").users.findOne({name: "Test"})'
{ _id: ObjectId('6988fc05b657d678478ce5b0'), name: 'Test' }
```

FIGURE 3 – Insertion sur le PRIMARY et lecture du document sur un SECONDARY

Le comptage des documents est identique sur l'ensemble des nœuds : **6043 utilisateurs** (6040 issus du dataset MovieLens et 3 documents de test), confirmant la cohérence des données répliquées.

1.5 Tolérance aux pannes (Failover)

Un test de failover est réalisé afin de valider la haute disponibilité du ReplicaSet :

1. Arrêt du conteneur `mongo-primary`
2. Attente de l'élection automatique (environ 10 à 15 secondes)
3. Vérification de l'élection d'un nouveau PRIMARY

```
mongodb-ha-replication-sharding main ? ❯ docker stop mongo-primary
mongo-primary

mongodb-ha-replication-sharding main ? ❯ docker exec mongo-secondary1 mongosh --eval "rs.status().members.forEach
(m => print(m.name + ' -> ' + m.stateStr))"
mongo-primary:27017 -> (not reachable/healthy)
mongo-secondary1:27017 -> PRIMARY
mongo-secondary2:27017 -> SECONDARY
mongo-secondary3:27017 -> SECONDARY
mongo-arbiter:27017 -> ARBITER
```

FIGURE 4 – Élection automatique d'un nouveau PRIMARY après arrêt du nœud principal

Nœud	**État après failover**
mongo-primary	indisponible
mongo-secondary1	PRIMARY
mongo-secondary2	SECONDARY
mongo-secondary3	SECONDARY
mongo-arbiter	ARBITER

TABLE 2 – État du cluster après failover

Ces observations montrent que le service reste opérationnel malgré la panne du nœud initial, validant ainsi le mécanisme de haute disponibilité du ReplicaSet. Lors du redémarrage de `mongo-primary`, celui-ci retrouve automatiquement son rôle de PRIMARY grâce à sa priorité supérieure.

II Sharding MongoDB

2.1 Principe du sharding

Le sharding a pour objectif de distribuer horizontalement les données sur plusieurs serveurs appelés **shards**. Chaque shard ne contient qu'une partie des données, définie par une **clé de sharding**.

L'architecture d'un cluster shardé comprend trois composants :

— **Shards** : stockent les données (chacun est un ReplicaSet)

— **Config Servers** : stockent les métadonnées du cluster

— **Mongos** : routeur qui dirige les requêtes vers les bons shards

2.2 Architecture du cluster

Le cluster déployé comprend 13 conteneurs MongoDB :

Composant	Nœuds	Ports
Config Servers (RSCFG)	cfg1, cfg2, cfg3	28017–28019
Shard 1 (RSSHARD1)	shard1-1, shard1-2, shard1-3	29017–29019
Shard 2 (RSSHARD2)	shard2-1, shard2-2, shard2-3	29020–29022
Shard 3 (RSSHARD3)	shard3-1, shard3-2, shard3-3	29023–29025
Routeur	mongos	31000

TABLE 3 – Architecture du cluster shardé MongoDB

Chaque shard est lui-même un ReplicaSet de trois nœuds, assurant la haute disponibilité de chaque partition.

```
mongodb-ha-replication-sharding main ? ❯ docker compose -f docker-compose.sharding.yml ps
NAME       IMAGE       COMMAND                  SERVICE    CREATED              STATUS                        PORTS
cfg1       mongo:7.0   "docker-entrypoint.s…"   cfg1       About a minute ago   Up About a minute (healthy)   0.0.0.0:28017->27017/tcp, [::]:28017->27017/tcp
cfg2       mongo:7.0   "docker-entrypoint.s…"   cfg2       About a minute ago   Up About a minute (healthy)   0.0.0.0:28018->27017/tcp, [::]:28018->27017/tcp
cfg3       mongo:7.0   "docker-entrypoint.s…"   cfg3       About a minute ago   Up About a minute (healthy)   0.0.0.0:28019->27017/tcp, [::]:28019->27017/tcp
mongos     mongo:7.0   "docker-entrypoint.s…"   mongos     About a minute ago   Up About a minute (healthy)   0.0.0.0:31000->27017/tcp, [::]:31000->27017/tcp
shard1-1   mongo:7.0   "docker-entrypoint.s…"   shard1-1   About a minute ago   Up About a minute (healthy)   0.0.0.0:29017->27017/tcp, [::]:29017->27017/tcp
shard1-2   mongo:7.0   "docker-entrypoint.s…"   shard1-2   About a minute ago   Up About a minute (healthy)   0.0.0.0:29018->27017/tcp, [::]:29018->27017/tcp
shard1-3   mongo:7.0   "docker-entrypoint.s…"   shard1-3   About a minute ago   Up About a minute (healthy)   0.0.0.0:29019->27017/tcp, [::]:29019->27017/tcp
shard2-1   mongo:7.0   "docker-entrypoint.s…"   shard2-1   About a minute ago   Up About a minute (healthy)   0.0.0.0:29020->27017/tcp, [::]:29020->27017/tcp
shard2-2   mongo:7.0   "docker-entrypoint.s…"   shard2-2   About a minute ago   Up About a minute (healthy)   0.0.0.0:29021->27017/tcp, [::]:29021->27017/tcp
shard2-3   mongo:7.0   "docker-entrypoint.s…"   shard2-3   About a minute ago   Up About a minute (healthy)   0.0.0.0:29022->27017/tcp, [::]:29022->27017/tcp
shard3-1   mongo:7.0   "docker-entrypoint.s…"   shard3-1   About a minute ago   Up About a minute (healthy)   0.0.0.0:29023->27017/tcp, [::]:29023->27017/tcp
shard3-2   mongo:7.0   "docker-entrypoint.s…"   shard3-2   About a minute ago   Up About a minute (healthy)   0.0.0.0:29024->27017/tcp, [::]:29024->27017/tcp
shard3-3   mongo:7.0   "docker-entrypoint.s…"   shard3-3   About a minute ago   Up About a minute (healthy)   0.0.0.0:29025->27017/tcp, [::]:29025->27017/tcp
```

FIGURE 5 – Conteneurs Docker composant le cluster shardé MongoDB

2.3 Mise en œuvre

L'initialisation du cluster shardé s'effectue en plusieurs étapes :

1. Initialisation du ReplicaSet des Config Servers (RSCFG)
2. Initialisation des ReplicaSets de chaque shard (RSSHARD1, RSSHARD2, RSSHARD3)
3. Connexion du routeur `mongos` aux Config Servers
4. Ajout des shards au cluster via la commande `sh.addShard()`
5. Activation du sharding sur les bases de données
6. Configuration des clés de sharding sur les collections

```
mongodb-ha-replication-sharding main ? ❯ docker exec mongos mongosh --eval "sh.status()"
shardingVersion
{ _id: 1, clusterId: ObjectId('6988fec68db06d1123bb317a') }
---
shards
[
  {
    _id: 'RSSHARD1',
    host: 'RSSHARD1/shard1-1:27017,shard1-2:27017,shard1-3:27017',
    state: 1,
    topologyTime: Timestamp({ t: 1770585847, i: 2 })
  },
  {
    _id: 'RSSHARD2',
    host: 'RSSHARD2/shard2-1:27017,shard2-2:27017,shard2-3:27017',
    state: 1,
    topologyTime: Timestamp({ t: 1770585847, i: 6 })
  },
  {
    _id: 'RSSHARD3',
    host: 'RSSHARD3/shard3-1:27017,shard3-2:27017,shard3-3:27017',
    state: 1,
    topologyTime: Timestamp({ t: 1770585848, i: 1 })
  }
]
---
```

FIGURE 6 – État global du cluster shardé obtenu via la commande `sh.status()`

2.4 Clés de sharding

Deux collections sont configurées pour le sharding :

Collection	Clé de sharding	Type
test.test_collection	user_id	hashed
MovieLens.movies	title	hashed

TABLE 4 – Configuration des clés de sharding

Le type **hashed** assure une distribution uniforme des données en calculant un hash de la valeur de la clé. Cela permet d'éviter les *hot spots*, où un shard recevrait une charge disproportionnée de données ou de requêtes.

2.5 Distribution des données et balancer

2.5.1 Test avec 1 million de documents

Un script génère un million de documents de test avec des valeurs aléatoires afin d'observer la répartition des données entre les shards.

```
mongodb-ha-replication-sharding main ? ❯ docker exec mongos mongosh --eval 'db.getSiblingDB("test"
).test_collection.getShardDistribution()'
Shard RSSHARD3 at RSSHARD3/shard3-1:27017,shard3-2:27017,shard3-3:27017
{
  data: '20.01MiB',
  docs: 333951,
  chunks: 2,
  'estimated data per chunk': '10MiB',
  'estimated docs per chunk': 166975
}
---
Shard RSSHARD1 at RSSHARD1/shard1-1:27017,shard1-2:27017,shard1-3:27017
{
  data: '19.9MiB',
  docs: 332230,
  chunks: 2,
  'estimated data per chunk': '9.95MiB',
  'estimated docs per chunk': 166115
}
---
Shard RSSHARD2 at RSSHARD2/shard2-1:27017,shard2-2:27017,shard2-3:27017
{
  data: '20MiB',
  docs: 333819,
  chunks: 2,
  'estimated data per chunk': '10MiB',
  'estimated docs per chunk': 166909
}
---
Totals
{
  data: '59.92MiB',
  docs: 1000000,
  chunks: 6,
  'Shard RSSHARD3': [
    '33.39 % data',
    '33.39 % docs in cluster',
    '62B avg obj size on shard'
  ],
  'Shard RSSHARD1': [
    '33.22 % data',
    '33.22 % docs in cluster',
    '62B avg obj size on shard'
  ],
  'Shard RSSHARD2': [
    '33.38 % data',
    '33.38 % docs in cluster',
    '62B avg obj size on shard'
  ]
}
```

FIGURE 7 – Distribution des documents de `test.test_collection` entre les shards

La distribution observée est la suivante :

Shard	Documents	Données	Pourcentage
RSSHARD1	332 230	19.9 MiB	33.22%
RSSHARD2	333 819	20.0 MiB	33.38%
RSSHARD3	333 951	20.0 MiB	33.39%
Total	**1 000 000**	**59.9 MiB**	**100%**

TABLE 5 – Distribution de la collection `test.test_collection`

2.5.2 Distribution MovieLens

Les 3 883 films du dataset MovieLens sont également répartis de manière homogène entre les shards :

Shard	Documents	Données	Pourcentage
RSSHARD1	1 249	90 KiB	32.16%
RSSHARD2	1 288	93 KiB	33.17%
RSSHARD3	1 346	97 KiB	34.66%
Total	**3 883**	**282 KiB**	**100%**

TABLE 6 – Distribution de la collection `MovieLens.movies`

2.5.3 Le balancer

Le **balancer** est un processus automatique chargé d'équilibrer la distribution des *chunks* entre les shards. Il s'active lorsqu'un déséquilibre est détecté et migre les chunks vers les shards les moins chargés.

Les résultats obtenus montrent une distribution quasi uniforme (environ 33 % par shard), validant à la fois le bon fonctionnement du balancer et la pertinence du choix des clés de sharding.

III Analyse

3.1 Réplication vs Sharding

Aspect	Réplication	Sharding
Objectif principal	Haute disponibilité	Scalabilité horizontale
Données	Copies complètes	Partitions distinctes
Écriture	PRIMARY uniquement	Tous les shards
Lecture	PRIMARY ou SECONDARY	Routée par mongos
Tolérance pannes	Failover automatique	Par shard (ReplicaSet)

TABLE 7 – Comparaison réplication vs sharding

3.2 Choix des clés de sharding

Le choix d'une clé de sharding est critique pour les performances. Une bonne clé distribue uniformément les données et les requêtes, tandis qu'une mauvaise clé crée des hot spots et déséquilibre le cluster.

Le type `hashed` utilisé dans ce TP garantit une distribution uniforme des données entre les shards. En contrepartie, il limite les requêtes par plage sur la clé de sharding, ce qui illustre le compromis classique entre performance et flexibilité des requêtes.

3.3 Limites observées

Quelques points d'attention lors de la mise en œuvre :
— Le temps d'initialisation du cluster est significatif (~45 secondes)
— Le mongos nécessite que les Config Servers soient opérationnels avant de répondre
— La commande `use` ne fonctionne pas avec `-eval` ; `getSiblingDB()` est préférable

IV Conclusion

Ce TP a permis de mettre en pratique les concepts fondamentaux de distribution des données avec MongoDB :

1. La **réplication** via un ReplicaSet de 5 nœuds, avec validation du failover automatique
2. Le **sharding** via un cluster de 3 shards, avec validation de la distribution uniforme (~33% par shard)

L'utilisation de Docker et Docker Compose a permis de simuler une infrastructure distribuée complexe sur une seule machine. Les scripts d'automatisation facilitent le déploiement reproductible de l'ensemble.

Les résultats obtenus montrent que MongoDB met en œuvre efficacement ces mécanismes, permettant de combiner haute disponibilité et scalabilité horizontale.