

Bases de données NoSQL

- Étude de cas avec MongoDB, Cassandra et HBase -

Khaled Jouini

j.khaled@gmail.com

Institut Supérieur d'Informatique et des Technologies de Communication

La reproduction de ce document par tout moyen que ce soit, sans l'avis de l'auteur, est interdite conformément aux lois protégeant la propriété intellectuelle

Ce document peut comporter des erreurs, à utiliser donc en connaissance de cause!

Plan du cours (1/2)

1 Modèle de données : BDs Relationnelles et Big Data, c'est quoi le problème?

Transactions et distribution

Jointure et distribution

Aggregate Data Model

Discussion

Quiz

2 Bases NoSQL documentaires

NoSQL Documentaire, quésaco ?

Expression des requêtes

MongoDB : un système ACID-Compliant

Quiz

Exercices

3 RéPLICATION

RéPLICATION : pourquoi et comment?

Cohérence forte vs. Eventual Consistency

Théorème CAP

Heartbeats, Failover et élections

La réPLICATION dans la pratique - Cas de MongoDB

Quiz

Plan du cours (2/2)

4 Sharding

Partitionner : pourquoi et comment?

Élasticité et load balancing

Techniques de partitionnement

Le sharding dans la pratique - Cas de MongoDB

Quiz

5 Au-delà des document-stores

Apache Cassandra, une base peer-to-peer avec hachage cohérent

Apache HBase

Quiz

Lectures intéressantes (1/2)

- *Bigtable: A Distributed Storage System for Structured Data.*
F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber
Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2006.
- *A Comparative Study of NoSQL Databases: MongoDB, Cassandra, and HBase.*
K. Grolinger, W. A. Higashino, A. Tiwari, M. A. Capretz
Proceedings of the IEEE World Congress on Services. 2013.
- *Dynamo: Amazon's Highly Available Key-value Store.*
Zaharia, M., Chowdhury, M., G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels
Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP). 2007.
- *CAP Twelve Years Later: How the "Rules" Have Changed.*
Eric Brewer
IEEE Computer Magazine. 2012.

- *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.*

David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, Rina Panigrahy

Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC). 1997.

- *Documentation officielle de MongoDB, Apache Cassandra et Apache HBase*

Big Data, quésaco?

Thématiques générales du cours : **gestion et analyse des données massives** (données en masse ou **Big Data**)



(*Source Statista 2020)

Big Data : besoins en stockage et/ou en traitement **dépassant de très loin les capacités d'une seule machine**

⇒ **Plate-formes** de stockage et de traitement **distribués** (e.g. Hadoop, Spark, Tensorflow, MongoDB, KAFKA, etc.).

Systèmes NoSQL et Big Data

- **NoSQL (Not Only SQL)**

- Famille de BDs **non relationnelles** permettant de gérer de grandes quantités de données non structurées ou semi-structurées.
- Contrairement aux bases relationnelles classiques, les bases NoSQL sont hautement scalables (**scalabilité horizontale**).
- Elles permettent la gestion efficace de données massives distribuées sur plusieurs serveurs.
- 3 éléments clés : **Modèle de Données, RéPLICATION et Sharding**

- **Exemples de systèmes NoSQL :**

- MongoDB : Base de données orientée documents (JSON/BSON).
- Cassandra : Base de données orientée colonnes, idéale pour le traitement distribué.
- Redis : Base de données clé-valeur en mémoire, très rapide.

Types de Bases de Données NoSQL

- **Bases orientées documents :**

- Utilisent des documents (souvent JSON ou BSON) pour stocker les données.
- Exemples : MongoDB, CouchDB.

- **Bases clé-valeur :**

- Stockent les données sous forme de paires clé-valeur.
- Exemples : Redis, Amazon DynamoDB.

- **Bases orientées colonnes :**

- Stockent les données par colonnes plutôt que par lignes, optimisant les requêtes en lecture.
- Exemples : Apache Cassandra, HBase.

- **Bases orientées graphes :**

- Conçues pour stocker des relations complexes entre les données, représentées sous forme de graphes.
- Exemples : Neo4j, ArangoDB.

Chapitre 2 - Modèle de données : BDs Relationnelles et Big Data, c'est quoi le problème?

Transactions et distribution

Transactions ACID : Rappel

Le protocole 2 Phases Commit

Jointure et distribution

Aggregate Data Model

Discussion

Quiz

Transactions ACID - Rappel

Definition (Transaction)

Séquence d'opérations de lectures et d'écritures qui doivent être exécutées, soit toutes soit aucune (**Atomicité**)

- Exemples : virement d'un compte A à un compte B, validation d'un panier sur un site de vente en ligne, etc.
- Commence par un **BEGIN TRANSACTION** (ou **startTransaction()**)
- En cas de succès (validation) se termine par un **COMMIT** (ou **commitTransaction()**)
- En cas d'échec (anulation) se termine par un **ROLLBACK** (ou **abortTransaction()**)
- BDs Relationnelles : transactions **obligatoires**, i.e. toute opération doit obligatoirement appartenir à une transaction.

Transactions ACID - Rappel

Atomicité : Tout ou rien

- Une transaction effectue toutes ses actions ou aucune. En cas d'annulation, les modifications engagées doivent être défaillantes.

Cohérence : Intégrité des données

Isolation : Pas d'interférence entre transactions

- Les résultats des modifications des transactions ne sont pas visibles par les autres transactions qu'après sa validation

Durabilité : L'enregistrement des données modifiées doit être garanti même en cas de pannes

- Journalisation des mises-à-jour (Fichiers Log)
- Les modifications validées sont rejouées à partir des logs en cas de perte.

2 PC

Transaction distribuée : les opérations de la transaction ont lieu sur des **sites distants**

Exemple : virement d'une banque A à une banque B

Problème : **comment garantir l'atomicité** (si l'une des 2 opérations échoue, l'autre doit être annulée)?

Protocoles les plus utilisés : validation à 2 phases (atomicité) et MVCC (isolation).

Hypothèses du protocole 2 PC

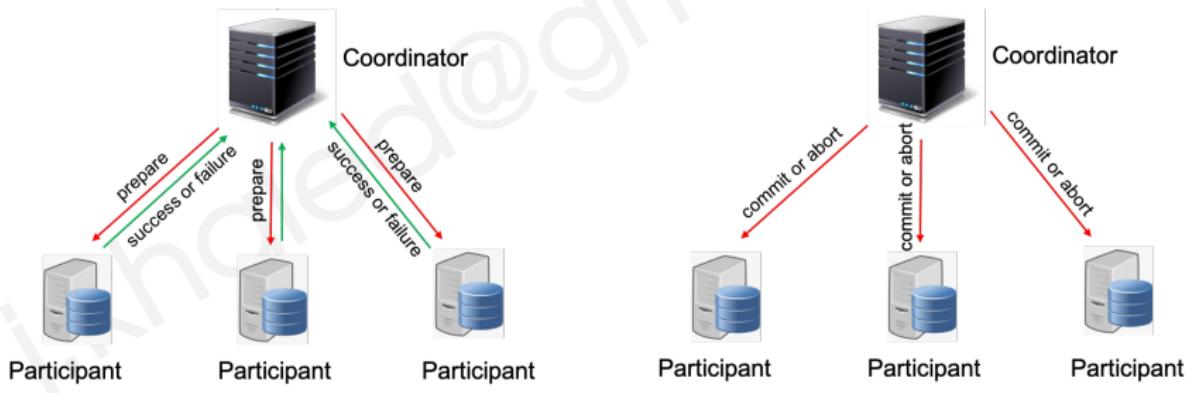
- Il existe un site coordinateur (*Coordinator*), les autres sites sont des participants (*Cohorts*)
- Chaque participant journalise ses opérations et génère ses propres données d'annulation.
- La panne sur un site est temporaire

2 PC

Principe 2 PC : 2 phases, une de vote et une de validation ou d'annulation

Phase de vote

- ① Le coordinateur envoie les opérations à chacun des participants et attend sa réponse.
- ② Lorsqu'un participant termine ses opérations avec succès, il envoie un acquittement au coordinateur.



Phase 1

Phase 2

2 PC

Phase de validation en cas de succès (tous les participants ont voté Oui)

- ① Le coordinateur envoie un message commit à tous les participants
- ② Chaque participant valide ses opérations, libère ses verrous et envoie un acquittement au coordinateur
- ③ Le coordinateur termine la transaction lorsqu'il reçoit les acquittements de tous les participants

Phase de validation en cas d'échec (au moins un des participants a voté Non)

- ① Le coordinateur envoie un message Rollback à tous les participants
- ② Chaque participant annule ses opérations et envoie un acquittement au coordinateur
- ③ Le coordinateur termine la transaction lorsqu'il reçoit les acquittements de tous les participants

Transactions ACID et Distribution à Grande Échelle : Les Problèmes

Complexité du 2PC

- Communication et coordination importantes.
- Point de défaillance unique (le coordinateur).
- Blocage possible en cas de panne du coordinateur.

Impact sur la performance

- Latence accrue due aux échanges entre les nœuds.
- Diminution du débit global du système.

Scalabilité limitée

- Difficile à mettre en œuvre sur des clusters très importants.
- Augmentation de la probabilité de pannes.

Conséquences pour le Big Data

- Les exigences ACID strictes sont souvent assouplies (BASE).
- Compromis entre cohérence et disponibilité.

Le Compromis BASE : Une Alternative à ACID pour le Big Data

BASE : Basically Available, Soft state, Eventually consistent.

Basically Available (Fondamentalement Disponible)

- Le système garantit la disponibilité des données, même en cas de pannes partielles.
- Répond aux requêtes même si certaines données ne sont pas à jour.

Soft state (État Mou)

- L'état du système peut évoluer au fil du temps, même sans intervention explicite.
- Pas de cohérence forte garantie à chaque instant.

Eventually consistent (Cohérence Éventuelle) (Nous y reviendrons!)

- Si aucune nouvelle mise à jour n'est effectuée, le système finira par atteindre un état cohérent.
- Délai de propagation des mises à jour possible.

ACID vs. BASE : Un Choix de Compromis

ACID

- Cohérence forte, transactions fiables.
- Complexité et limitations en matière de scalabilité et de disponibilité.

BASE

- Haute disponibilité, Scalabilité horizontale facilitée.
- Cohérence éventuelle, compromis sur la cohérence forte.

Le choix dépend des besoins de l'application

- Applications transactionnelles critiques (finance, banque) : ACID.
- Applications Big Data avec des exigences de scalabilité et de disponibilité élevées : BASE.

Chapitre 2 - Modèle de données : BDs Relationnelles et Big Data, c'est quoi le problème?

Transactions et distribution

Jointure et distribution

Aggregate Data Model

Discussion

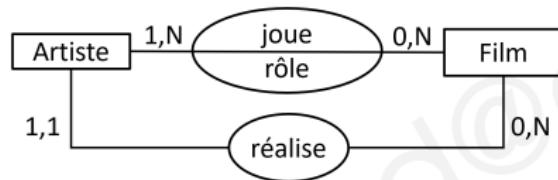
Quiz

Qu'en est-il de la jointure relationnelle?



```
{ "customer_id": 1,  
  "first_name": "Mark",  
  "last_name": "Smith",  
  "city": "San Francisco",  
  "phones": [ {  
      "type": "work",  
      "number": "1-800-555-1212"  
    },  
    { "type": "home",  
      "number": "1-800-555-1313",  
      "DNC": true  
    },  
    { "type": "home",  
      "number": "1-800-555-1414",  
      "DNC": true  
    }  
  ]  
}
```

Modèle Relationnel



id	Title	Year	Genre	Director_ID
1	Dachra	2018	Horror	1
2	Pulp Fiction	1994	Crime	37
3	The Departed	1990	Thriller	46

movie_id	artist_id	role
1	8	Walid
3	5	Billy Costigan
2	6	Mia Wallace

id	Last Name	First Name	Birth Year
1	Bouchnak	Abdelhamid	1984
37	Tarantino	Quentin	1963
46	Scorsese	Martin	1942
8	Jebali	Aziz	1986

Scalabilité du Modèle Relationnel : Les Défis des Jointures

Normalisation et Jointures : Le modèle relationnel normalise les données pour éviter la redondance, ce qui implique de nombreuses **jointures** pour reconstituer les informations.

Coût des Jointures : Les jointures sont des opérations coûteuses, en particulier sur de grands volumes de données.

Distribution et Communication : Dans un environnement distribué, les données sont réparties sur plusieurs machines.

- Les jointures nécessitent des transferts de données entre les nœuds.
- Augmentation de la latence et de la complexité de la coordination.

Écritures et Transactions ACID : il faut effectuer plusieurs écritures pour une même entité ⇒ transactions et Verrouillage distribué complexe et coûteux.

Conséquence : Ces limitations rendent le modèle relationnel et les jointures peu adaptés aux applications.

Chapitre 2 - Modèle de données : BDs Relationnelles et Big Data, c'est quoi le problème?

Transactions et distribution

Jointure et distribution

Aggregate Data Model

Discussion

Quiz

Modèle agrégé de données (Aggregate Data Model)

Documents structurées (XML, JSON)

- Permettent la représentation des données semi-structurées (*i.e.* structure et typage souples)
- Facilitent l'échange de données entre applications hétérogènes
- Fournissent des règles de représentation de l'information "neutres" (*i.e.* non propriétaires) par rapport aux applications susceptibles de traiter l'information

Notions essentielles

Les données sont auto-décrivantes : Le contenu vient avec sa propre description.

Structures riches : Le contenu se décrit avec des listes, des enregistrements imbriqués, des ensembles.

Typage flexible : Les données peuvent être typées ("c'est un entier"), et/ou structurées ("cette partie du graphe doit avoir telle forme"), et/ou ni l'un ni l'autre. Dans ce dernier cas c'est à l'application d'effectuer le contrôle.

Sérialisation : Structure et contenu doivent pouvoir être transformés ensemble en une chaîne de caractères autonome.

Représentation Relationnelle et NoSQL : Un Problème de Scalabilité

Les systèmes NoSQL visent à **minimiser l'usage des jointures** en regroupant les données pertinentes dans des entités autonomes, simplifiant ainsi la gestion des données distribuées.

NoSQL documentaire : exploite la flexibilité des documents structurés pour remplacer les jointures par l'**imbrication de structures**

Cette approche favorise une **scalabilité horizontale** en distribuant les données sur plusieurs serveurs, mais introduit une certaine **redondance** et une hiérarchisation des accès.

Modèle agrégé de données (Aggregate Data Model)

```
1  {
2      "id": 1,
3      "title": "Dachra",
4      "year": 2018,
5      "genre": "Horror",
6      "summary": "A group of students investigates ...",
7      "director": {
8          "id": 1,
9          "last_name": "Bouchnak",
10         "first_name": "Abdelhamid"
11         "birth_year": 1984
12     },
13     "actors": [
14         {
15             "id": 8,
16             "last_name": "Jebali",
17             "first_name": "Aziz"
18             "birth_year": 1986,
19             "role": "Walid"
20         }
21     ]
22 }
```

Chapitre 2 - Modèle de données : BDs Relationnelles et Big Data, c'est quoi le problème?

Transactions et distribution

Jointure et distribution

Aggregate Data Model

Discussion

Quiz

Inconvénients de l'imbrication

Redondance des données

- Les informations peuvent être dupliquées dans plusieurs documents, augmentant le risque d'**incohérences**.

Hiérarchisation des accès

- Les modèles agrégés privilient des chemins d'accès spécifiques, ce qui peut rendre certains types de requêtes plus difficiles ou plus lentes.

Limitation dans la gestion d'entités indépendantes

- Il est impossible d'ajouter une entité, telle qu'un acteur, sans l'encapsuler dans un document de film.

⇒ **Seul avantage : éviter les jointures et les écritures distribuées**

Pour aller plus loin...

Lecture intéressante : Patterns de modélisation pour les bases documentaires
<https://docs.mongodb.org/manual/data-modeling/>

Chapitre 2 - Modèle de données : BDs Relationnelles et Big Data, c'est quoi le problème?

Transactions et distribution

Jointure et distribution

Aggregate Data Model

Discussion

Quiz

Quiz

- ① Comparez et contrastez les modèles de données relationnels et NoSQL en termes de flexibilité, de performance et de scalabilité. Donnez des exemples d'applications qui conviennent mieux à l'un ou à l'autre modèle.
- ② Quelle est la différence entre la scalabilité verticale et horizontale?
- ③ Pourquoi les bases de données relationnelles sont inadaptées à la distribution à grande échelle?
- ④ Préciser la notion de "hiérarchisation des accès".
- ⑤ Considérez le modèle conceptuel ci-après. Donnez les différentes possibilités de traduction de ce modèle en un schéma de base documentaire.



- ⑥ Supposez que l'on voudrait enregistrer dans la base des produits qui n'ont pas encore été facturés. Quels schémas sont appropriés? Que pouvez vous conclure sur le modèle relationnel?

Chapitre 3 - Bases NoSQL documentaires

NoSQL Documentaire, quésaco ?

Expression des requêtes

MongoDB : un système ACID-Compliant

Quiz

Exercices

NoSQL Documentaire, quésaco ?

MongoDB

- fait partie des bases de données NoSQL **orienté document** (comme CouchDB) ;
- utilise un modèle de données semi-structuré, encodé en JSON ;
- ne nécessite pas de schéma rigide, offrant une flexibilité totale ;
- dispose d'un langage de requête original et spécifique ;
- ne supportait pas les transactions à ses débuts, mais introduit un support transactionnel depuis la version 4.0.

Dès sa conception, MongoDB a été pensé pour être un système **évolutif** et **distribué** :

- les données sont réparties par un mécanisme de **sharding** (partitionnement) ;
- la résilience aux pannes est assurée par la **réPLICATION**.

MongoDB, quésaco?

SQL Terms/Concepts	NoSQL Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	embedded documents and linking
primary key Specify any unique column or column combination as primary key.	primary key In MongoDB, the primary key is automatically set to the <code>_id</code> field.

<http://docs.mongodb.org/manual/reference/sql-comparison/>

Opérations CRUD

```

db.users.insertOne( ← collection
{
    name: "sue", ← field: value
    age: 26, ← field: value
    status: "pending" ← field: value } ) document
}

db.users.find( ← collection
    { age: { $gt: 18 } }, ← query criteria
    { name: 1, address: 1 } ← projection
).limit(5) ← cursor modifier

db.users.updateMany( ← collection
    { age: { $lt: 18 } }, ← update filter
    { $set: { status: "reject" } } ← update action
)

db.users.deleteMany( ← collection
    { status: "reject" } ← delete filter
)

```

Chapitre 3 - Bases NoSQL documentaires

NoSQL Documentaire, quésaco ?

Expression des requêtes

Aggregation Pipeline

Map/Reduce

MongoDB : un système ACID-Compliant

Quiz

Exercices

Exemples de Requêtes MongoDB

Trouver les films sortis après 1990

```
1 db.movies.find({ "year": { $gt: 1990 } })
```

Trouver les films réalisés par A. Bouchnek

```
1 db.movies.find({ "director.last_name": "Bouchnek" })
```

Trouver un film par titre avec findOne

```
1 db.movies.findOne({ "title": "Dachra" })
```

Trouver des films où "Jebali" a joué

```
1 db.movies.find({ "actors.last_name": "Jebali" })
```

Trouver les titres de films, mais afficher uniquement le titre

```
1 db.movies.find({ "year": 1994 }, { "title": 1, "_id": 0 })
```

Trouver des films appartenant aux genres Drama ou Action

```
1 db.movies.find({ "genre": { $in: [ "Drama", "Action" ] } })
```

Aggregation Pipeline

- MongoDB dispose d'un langage de requêtes dit "Aggregation Pipeline"
- **Pipeline** : **séquence** d'opérateurs inspirés des opérateurs SQL, permettant de réaliser des requêtes plus complexes que les requêtes `find()`.
- Depuis la version 3.2, MongoDB dispose d'un opérateur de jointure `$lookup`.

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	<code>\$match</code>
GROUP BY	<code>\$group</code>
HAVING	<code>\$match</code>
SELECT	<code>\$project</code>
ORDER BY	<code>\$sort</code>
LIMIT	<code>\$limit</code>
SUM()	<code>\$sum</code>

Aggregation Pipeline

Nombre de films du genre "drama"

```
1 db.movies.aggregate([
2     {$match: {genre: "drama"}},
3     {$count: "nombreFilms"},,
4 ])
```

Les 3 réalisateurs qui ont dirigé le plus de films

```
1 db.movies.aggregate([
2     {$group:{_id: "$director._id", total: {$sum: 1}}},
3     {$sort: { total: -1 }},
4     {$limit: 3}
5 ])
```

Les 3 acteurs qui ont joué dans le plus de films

```
1 db.movies.aggregate([
2     {$unwind: "$actors"},
3     {$group: {_id: "$actors._id",total: {$sum: 1}}},
4     {$sort: { total: -1 }},
5     {$limit: 3}
6 ])
```

Aggregation Pipeline

Exemple \$lookup : titre et nom du réalisateur des films.

```
1 db.moviesAvecRef.aggregate([
2     {
3         $lookup: {
4             from: "artists",
5             localField: "director._id",
6             foreignField: "_id",
7             as: "realisateur"
8         }
9     },
10    {
11        $project: {
12            "title": 1,
13            "realisateur.last_name": 1
14        }
15    }
16]);
```

Map/Reduce, quésaco?

- **Map/Reduce** : modèle de calcul **distribué** destiné à traiter efficacement de grands volumes de données sur plusieurs machines.
- Il repose sur deux opérations fondamentales :
 - **Map** : applique une transformation sur chaque donnée pour générer des paires clé-valeur.
 - **Reduce** : les paires clé-valeur intermédiaires sont regroupées par clé, puis agrégées pour produire le résultat final.
- **Exemple intuitif** : Pour compter les occurrences de mots dans un ensemble de documents, chaque mot est transformé en une clé lors de la phase *Map*. Ensuite, la phase *Reduce* additionne les occurrences des mots identiques pour fournir un décompte global.
- Modèle particulièrement adapté aux systèmes massivement parallèles, offrant scalabilité, tolérance aux pannes et une exécution efficace des tâches sur des clusters distribués.

Principe de Fonctionnement

Phase Map

- Appliquée à chaque **élément** : prend un document en entrée dans notre cas.
- Chaque document peut produire **0, 1 ou plusieurs paires (clé, valeur)** en sortie.

Phase de Shuffle et Tri

- Les **valeurs** générées par les mappers sont **regroupées par clé** ⇒ **un groupe est créé pour chaque clé**.
- Chaque groupe contient une clé et une liste de valeurs : **(k, list(v))**.
- Ces groupes **(k, liste(v))** sont envoyés aux reducers.

Phase Reduce

- Appliquée à chaque groupe **(k, liste(v))**, la fonction retourne une nouvelle paire **(k, v)**.
- En théorie, cette phase ne commence qu'une fois que tous les mappers ont terminé.
- En pratique, les reducers peuvent démarrer pendant que les mappers finissent leur travail pour accélérer le traitement.

Principe de Fonctionnement

La collection **La fonction de map** **Les groupes** **La fonction de reduce** **Le résultat**

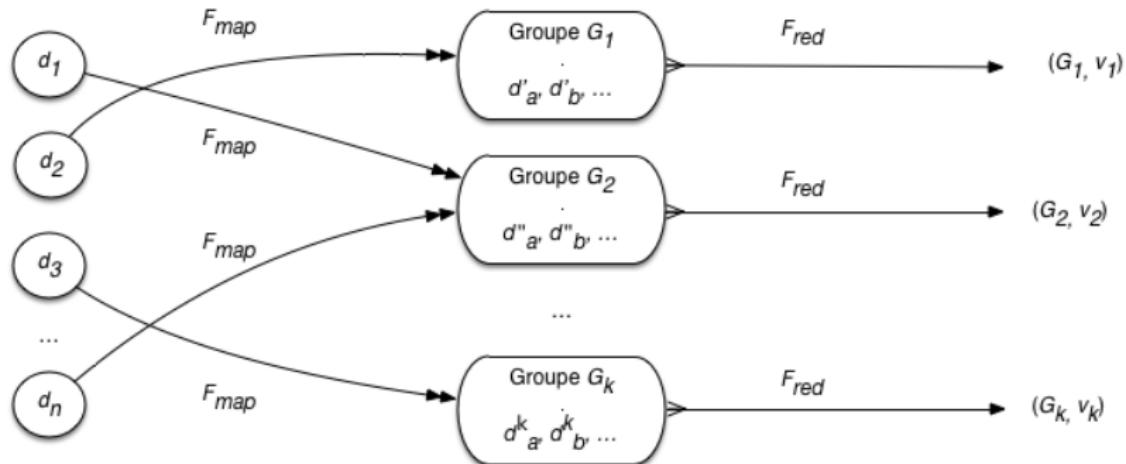


Figure reprise de <http://b3d.bdpedia.fr/>

Exemple : nombre de films par réalisateur

Fonction Map

```
1 var mapFunction = function() {  
2     emit(this.director.last_name, 1);  
3 };
```

Fonction Reduce

```
1 var reduceFunction = function(key, values) {  
2     return Array.sum(values);  
3 };
```

Exécution du MapReduce

```
1 db.movies.mapReduce(  
2     mapFunction,  
3     reduceFunction,  
4     { out: "movies_per_director" }  
5 )
```

Chapitre 3 - Bases NoSQL documentaires

NoSQL Documentaire, quésaco ?

Expression des requêtes

MongoDB : un système ACID-Compliant

Quiz

Exercices

MongoDB : un système ACID-Compliant

MongoDB : introduit le support des transactions ACID à partir de la version 4.0.

```
1 // Demarrez une session MongoDB
2 session = db.getMongo().startSession();
3 // Selectionnez la BD
4 db = session.getDatabase("nom_de_votre_base_de_donnees");
5 // Selectionnez la collection
6 collection = db.getCollection("nom_de_votre_collection");
7 // Demarrez une transaction dans la session
8 session.startTransaction();
9 try {
10     // Effectuez des operations dans la transaction
11     collection.insertOne({ name: 'Document 1' });
12     collection.insertOne({ name: 'Document 2' });
13     // Validez la transaction
14     session.commitTransaction();
15 }
16 catch (error) {
17     // Annulez la transaction en cas d'erreur
18     session.abortTransaction();
19 }
20 finally {
21     // Terminez la session
22     session.endSession();
23 }
```

Contrairement aux BDs relationnelles, les transactions ne sont pas obligatoires.

Attention au cas distribué!

Chapitre 3 - Bases NoSQL documentaires

NoSQL Documentaire, quésaco ?

Expression des requêtes

MongoDB : un système ACID-Compliant

Quiz

Exercices

Quiz

- ① Expliquez le principe de localité des données.
- ② Quel est le rôle de la clé dans le modèle Map/Reduce ?
- ③ En quoi consiste la phase Shuffle & Sort?
- ④ Quelle la différence entre la gestion des transactions dans les RDBMS et dans un système NoSQL comme MongoDB?

Chapitre 3 - Bases NoSQL documentaires

NoSQL Documentaire, quésaco ?

Expression des requêtes

MongoDB : un système ACID-Compliant

Quiz

Exercices

Exercices

Écrire en pseudo-code les fonctions Map/Reduce permettant de trouver :

① Nombre de films par genre

```
var mGenre = function() { emit(this.genre, 1); };
var rGenre = function(genre, values) { return values.length; }
```

② Nombre de films par acteur

```
var mFA = function() {
    for (var i = 0; i < this.actors.length; i++) {
        emit(this.actors[i]._id, 1);
    }
}
var rFA = function(acteur_id, values) {return values.length; }
```

③ Nombre d'acteurs dirigés par un réalisateur

```
var mAR = function(){
    emit(this.director._id, this.actors.length);
}
var rAR = function(realisateur, acteurs) {
    var res=0;
    for (var i=0; i<acteurs.length; i++) {res = res + acteurs[i];}
    return res;
}
```

Exercices

Écrire en pseudo-code les fonctions Map/Reduce permettant de :

- ① Afficher les titres des films du genre "drama" par année
- ② Trouver pour chaque acteur la liste des acteurs avec lesquels il a tourné des films
- ③ Afficher les films par genre et par année
- ④ Trouver le graphe des co-artistes

Chapitre 4 - RéPLICATION

RéPLICATION : pourquoi et comment?

Cohérence forte vs. Eventual Consistency

Théorème CAP

Heartbeats, Failover et élections

La réPLICATION dans la pratique - Cas de MongoDB

Quiz

RéPLICATION : pourquoi et comment?

RéPLICATION : dupliquer les données sur plusieurs serveurs pour assurer

- **Haute Disponibilité** : Capacité d'un système à rester fonctionnel et accessible même en cas de panne de certains serveurs.
- **Tolérance aux pannes** : Capacité de récupérer les données perdues.
- **Répartition de la charge** : Améliorer les performances en répartissant les accès aux données entre plusieurs serveurs.
- Essentielle dans les clusters distribués à grande échelle où les pannes sont fréquentes.

Dans la plupart des systèmes : **écritures centralisées et lectures distribuées**

- Les écritures sont généralement centralisées sur un seul serveur pour éviter la gestion complexe des accès concurrents distribués (e.g. résolution des conflits d'écriture, verrouillage, etc.).

RéPLICATION : pourquoi et comment?

RéPLICATION Master/Slaves (*Leader-Based Replication*)

- Lorsque les **écritures** se font sur un seul serveur, ce serveur est appelé **Master**.
- Les autres serveurs, appelés **Slaves**, contiennent des copies et répondent éventuellement aux requêtes de **lecture**.
- Panne sur un Slave : le système continue à fonctionner et à assurer les lectures et les écritures
- Panne du Master : un Slave peut être promu pour le remplacer (**élection**).

Mécanisme de Synchronisation : Fichier Journal (Log)

Les systèmes de réPLICATION utilisent des **fichiers journaux (log)** pour consigner les écritures.

Chaque modification est enregistrée dans un fichier journal et les réPLICES rejouent les modifications à partir de ce log.

La journalisation permet de se ramener à des écritures séquentielles plus rapides que les écritures aléatoires.

Idéalement un slave récupère le log du master, sinon d'un slave

Chapitre 4 - RéPLICATION

RéPLICATION : pourquoi et comment?

Cohérence forte vs. Eventual Consistency

Théorème CAP

Heartbeats, Failover et élections

La réPLICATION dans la pratique - Cas de MongoDB

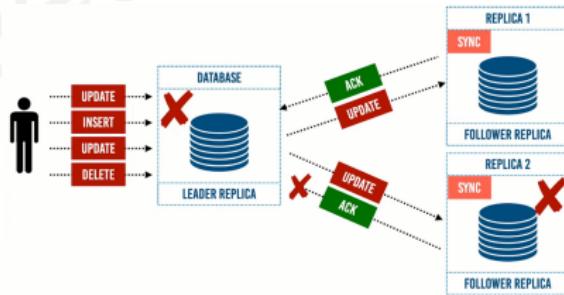
Quiz

RéPLICATION synchrone et cohérence forte

RéPLICATION Synchrone

- Une écriture n'est validée que si elle réussit sur **toutes les répliques**, garantissant ainsi que toutes les copies des données sont **toujours identiques**.
- **Cohérence forte**
 - Toute lecture renvoie la valeur de la dernière écriture validée.
 - Mode adopté par les bases relationnelles (E.g. Oracle Data Guard)
 - Entraîne un **temps de latence élevé**.

Difficile à assurer dans les environnements distribués à grande échelle, où les **pannes sont fréquentes** et les répliques ne sont pas toujours atteignables.

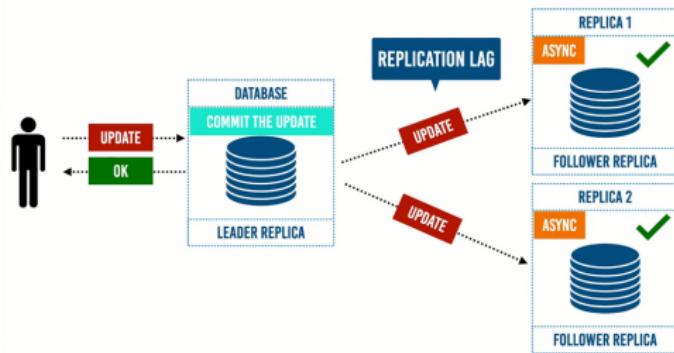


RéPLICATION asynchrone et cohérence à terme (*Eventual Consistency*)

RéPLICATION ASYNCHRONE : Les écritures sont confirmées sur le maître **sans attendre** (tous) les slaves.

Cohérence à terme (*Eventual Consistency*)

- Les données seront éventuellement cohérentes sur toutes les répliques, mais sans garantie de délai.
- ⇒ **Latence réduite, performances accrues, tolérance aux pannes;** mais **Incohérence temporaire** possible.



RéPLICATION SYNCHRONE VS. RÉPLICATION ASYNCHRONE

Critère	Synchrone	Asynchrone
Cohérence	Forte	À terme
Latence	Élevée	Faible
Tolérance aux pannes	Faible	Élevée
Mécanisme de réPLICATION	Blocage jusqu'à confirmation	RéPLICATION différée
Complexité de mise en œuvre	Élevée	Plus simple
Scalabilité	Plus difficile	Plus facile

RéPLICATION SYNCHRONE VS. RÉPLICATION ASYNCHRONE

Chapitre 4 - RéPLICATION

RéPLICATION : pourquoi et comment?

Cohérence forte vs. Eventual Consistency

Théorème CAP

Heartbeats, Failover et élections

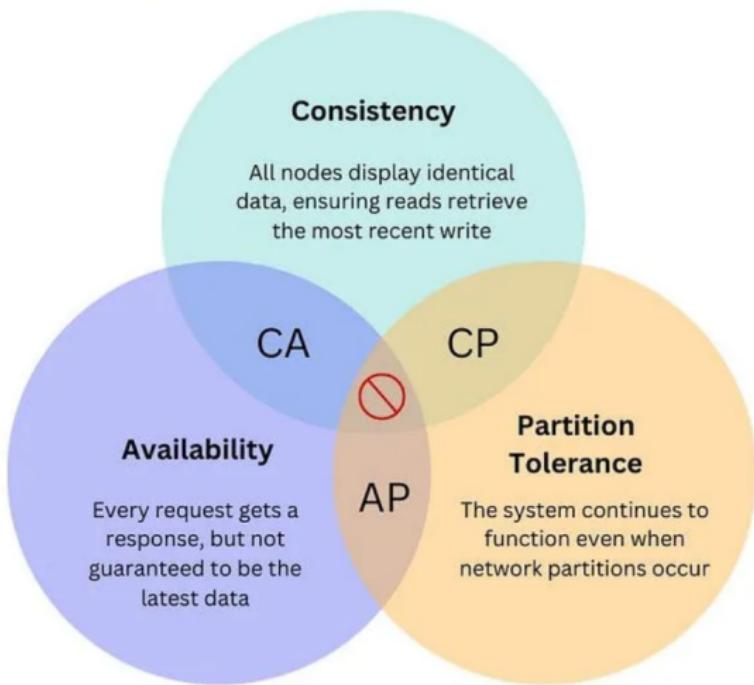
La réPLICATION dans la pratique - Cas de MongoDB

Quiz

Le Théorème CAP

- Le théorème CAP indique qu'un système distribué ne peut pas garantir simultanément :
 - **Cohérence (Consistency)** : Toutes les répliques voient les mêmes données.
 - **Disponibilité (Availability)** : Chaque requête reçoit une réponse même en cas de panne.
 - **Tolérance au Partitionnement (Partition Tolerance)** : Le système continue de fonctionner même si la communication entre les serveurs est coupée.
- Les systèmes NoSQL doivent choisir deux propriétés sur trois, souvent **disponibilité et tolérance au partitionnement**, au détriment de la cohérence forte.
- Les RDBMS privilégient la cohérence et la disponibilité

Théorème CAP



Chapitre 4 - RéPLICATION

RéPLICATION : pourquoi et comment?

Cohérence forte vs. Eventual Consistency

Théorème CAP

Heartbeats, Failover et élections

La réPLICATION dans la pratique - Cas de MongoDB

Quiz

Tolérance aux Pannes, Heartbeats et Failover

Les serveurs échangent des messages de **heartbeat** pour vérifier leur état.

Si un slave tombe en panne, rien ne se passe

Si un serveur maître ne répond plus aux heartbeats, un mécanisme de **failover** est déclenché pour promouvoir un slave comme nouveau maître. Différents algorithmes de consensus, dont notamment **PAXOS** et plus récemment **RAFT**.

De préférence nombre de votants **impair** (pourquoi?)

Critères de sélection

- L'état de synchronisation des répliques (la réplique la plus à jour est favorisée).
- La priorité des serveurs (certains serveurs peuvent être favorisés en fonction de leur rôle ou capacité).

Algorithme de Consensus RAFT - Élection du leader

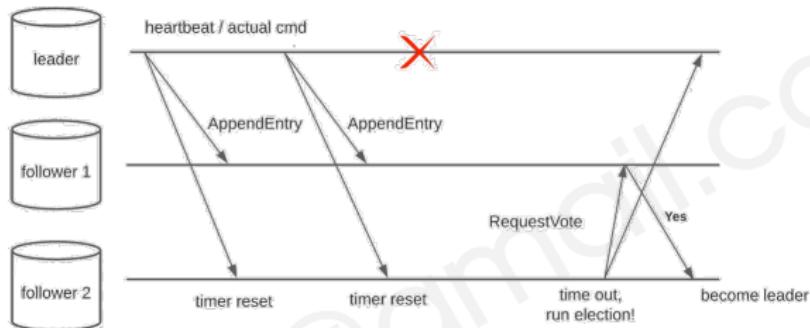


Figure reprise de (OO14)

Détection des pannes

- Chaque follower dispose d'un **timeout** (e.g. entre 150 et 300 ms). Si le follower ne reçoit pas de heartbeat pendant ce délai, il considère que le leader est défaillant et doit être remplacé.
- Le follower se transforme alors en **candidat** et lance une **élection** pour tenter de devenir le nouveau leader.
- Un candidat lance une élection en augmentant son **terme** et en envoyant des **requêtes de vote (RequestVote)** à tous les nœuds.

Algorithme de Consensus RAFT - Élection du leader

Vote d'un nœud : Un nœud recevant une **requête de vote**

- Vérifie si le **terme du candidat est supérieur** au sien. Si c'est le cas, il vote pour le candidat.
- Vérifie si son **log est plus à jour** que celui du candidat. Si c'est le cas, il refuse de voter.
- **Ne vote qu'une seule fois par terme.**

Atteinte du quorum : Un candidat devient leader lorsqu'il reçoit des votes de la **majorité des nœuds** (quorum).

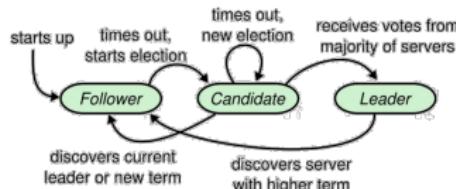


Figure reprise de (OO14)

Algorithme de Consensus RAFT - Gestion des Termes

- **Définition d'un terme :** Un terme dans Raft représente une unité de temps logique pendant laquelle un leader est élu. Chaque terme commence soit par une élection, soit par la continuité du leader précédent.
- **Mise à jour du terme :** Chaque nœud conserve un numéro de terme qui est incrémenté lors d'une élection ou lorsqu'il reçoit un message d'un nœud avec un terme supérieur. Le leader diffuse son terme lors de la réPLICATION du log, permettant aux autres nœuds de se synchroniser.
- **Vote unique par terme :** Pour prévenir les conflits de leadership, chaque nœud ne peut voter qu'une seule fois par terme. Un nœud vote pour le candidat ayant le terme le plus élevé et un log au moins aussi à jour.
- **Cohérence des journaux :** Un leader élu dans un terme plus élevé dispose d'un journal plus récent et plus complet, assurant ainsi la cohérence des données dans tout le cluster. Avant d'être élu, un candidat doit démontrer qu'il a appliqué toutes les entrées du log des autres nœuds jusqu'à un certain index.

Exemple d'Élection RAFT - Configuration Initiale et Détection de la Panne

Configuration initiale :

- **Nœud A** : Leader, terme 1
- **Nœuds B, C, D, E** : Followers, terme 1

Défaillance du leader : Le nœud A tombe en panne.

Détection de la panne :

- **Nœud B** détecte en premier l'absence de heartbeat de A.
- **Nœud C** détecte ensuite la défaillance de A.

Important : les **timeouts sont différents** d'un nœud à un autre dans RAFT (randomized timeouts)

Exemple d'Élection RAFT - Scénario 1 : un candidat obtient le quorum

Étape	Noeud A	Noeud B	Noeud C	Noeud D	Noeud E
Initial	Leader (terme 1)	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)
Détection de la panne par B	-	Terme 2, candidat	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)
Détection de la panne par C	-	Terme 2, can- didat	Terme 2, can- didat	Follower (terme 1)	Follower (terme 1)
Votes de D et E pour B	-	Terme 2, leader	Terme 2, fol- lower	Terme 2, a voté pour B	Terme 2, a voté pour B
Requête de vote de C	-	Leader	Terme 2, fol- lower	Terme 2, a voté pour B	Terme 2, a voté pour B

Exemple d'Élection RAFT - Scénario 2 : Vote divisé

Étape	Noeud A	Noeud B	Noeud C	Noeud D	Noeud E
Initial	Leader (terme 1)	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)
Détection de la panne par B	-	Terme 2, candidat	Follower (terme 1)	Follower (terme 1)	Follower (terme 1)
Détection de la panne par C	-	Terme 2, candidat	Terme 2, can- didat	Follower (terme 1)	Follower (terme 1)
Vote de D pour B	-	Terme 2, candidat	Terme 2, can- didat	Terme 2, a voté pour B	Follower (terme 1)
Vote de E pour C	-	Terme 2, candidat	Terme 2, can- didat	Terme 2, a voté pour B	Terme 2, a voté pour C

Résolution du vote divisé : Dans ce cas, aucun candidat n'a obtenu le quorum (3 votes). RAFT résout cette situation en utilisant des **timeouts d'élection aléatoires**, qui permettent à un candidat de déclencher la prochaine élection légèrement avant l'autre.

Chapitre 4 - RéPLICATION

RéPLICATION : pourquoi et comment?

Cohérence forte vs. Eventual Consistency

Théorème CAP

Heartbeats, Failover et élections

La réPLICATION dans la pratique - Cas de MongoDB

MongoDB, un système master-slaves

Niveaux de cohérence

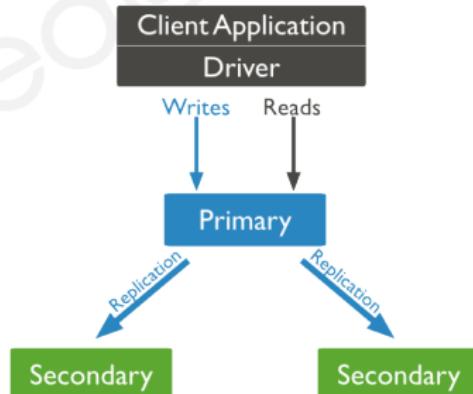
Élection du maître

Session illustrative

Quiz

MongoDB, système distribué maître-esclave

- MongoDB : Système distribué maître-esclaves, utilisant une **réPLICATION ASYNCHRONIE**
- L'ensemble de serveurs mongoDB partageant des replicas d'un même ensemble de documents, est appelé **Replica Set**
- Typiquement un RS contient un maître, *i.e.* **Primary** et deux esclaves, *i.e.* **Secondaries** et éventuellement un serveur arbitre.
- Dans une grappe MongoDB avec des documents répartis, on peut trouver plusieurs replica sets, chacun contenant un sous-ensemble d'une très grande collection (nous y reviendrons)



Replica Set, Primary, Secondaries

Primary (maître)

- Les écritures ont toujours lieu sur le **Primary** (il en existe un seul dans un replica set)
- Consigne toutes les opérations de modification des données **dans un fichier journal** (log), appelé **oplog** (*operations log*)

Secondary (esclave)

- Récupère l'opLog (du primary ou de n'importe quel autre secondary) et le stocke localement dans la collection `local.oplog.rs`
- Met à jour la copie locale à partir de l'opLog récupéré
- La mise-à-jour des copies est **asynchrone**, i.e. on n'attend pas la confirmation de l'écriture des copies.

Cohérence forte vs. cohérence à terme

2 niveaux de cohérence dans MongoDB : **Cohérence forte** et **cohérence à terme**.

Cohérence forte

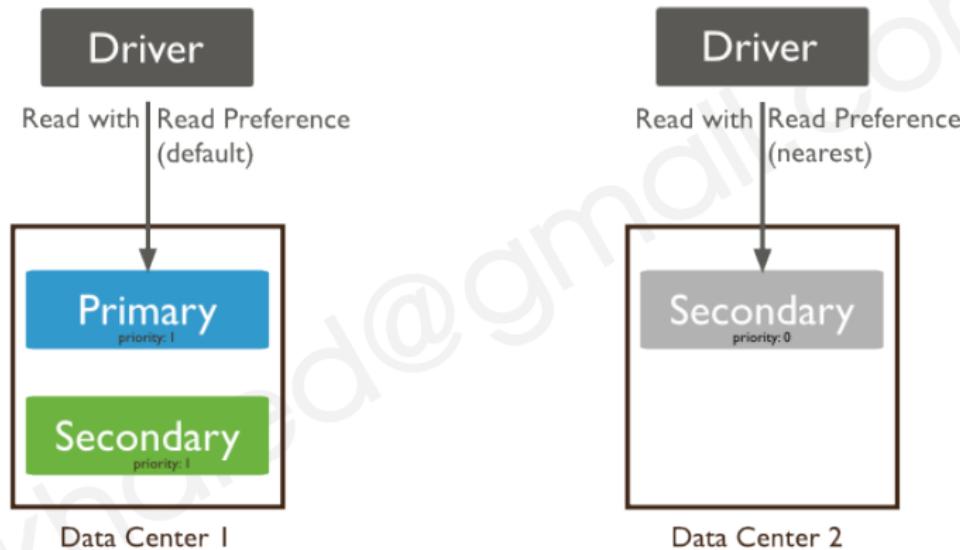
- Elle est garantie en forçant le client à **effectuer toutes les lectures directement sur le nœud maître**.
- Les nœuds esclaves ne sont pas utilisés pour répartir la charge de lecture.
- Les nœuds esclaves ont pour fonction principale de maintenir une copie en temps réel des données, et de prendre le relais du maître en cas de défaillance.

Cohérence à terme (*Eventual Consistency*)

- **Le client peut choisir de lire sur un esclave**
- Plusieurs options : lire à partir du noeud ayant la plus faible latence réseau (*nearest*), sur le noeud secondaire préféré (*SecondaryPreferred*), etc.

Read Preference et Write Concern

Read Preference

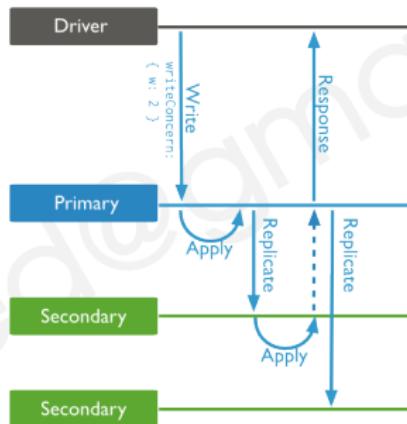


```
db.coln.find().readPref('nearest')
```

Read Preference et Write Concern

Write Concern

Option supplémentaire : on attend que **n serveurs** ou bien une **majorité** (*majority*) de serveurs aient effectué l'écriture avant de reprendre la main



```
1 mongosh --host=localhost --port=27017 --writeConcern="{w: 'majority'}" --
connectTimeoutMS=5000
ou
1 db.myCollection.insertOne({nom: "Jeanne Durand"}, {writeConcern:{w:"majority", j: true
}});
```



Élection d'un maître dans MongoDB

Reprise sur panne : identique à celle décrite précédemment

- Les serveurs se surveillent par *heartbeat* (ping)
- Perte d'un esclave : redirection des éventuelles lectures et initiation d'une nouvelle réPLICATION
- Perte du maître : éLECTION d'un nouveau maître.

ÉLECTION d'un nouveau maître : déclenchée

- Lors de l'initialisation d'un nouveau RS
- Si une majorité d'esclaves ne voient plus le maître.

Choix du nouveau maître

- Un Secondary ne peut devenir Primary que s'il est le plus "à jour" parmi tous les Secondaries connectés
- Un Secondary ne peut devenir Primary que s'il "voit" la majorité des secondaries connectés
- etc.

Serveur Arbitre

Serveur arbitre (*arbiter*)

- Pour qu'une élection aboutisse, le nombre de participants doit être impair
- On peut ajouter un serveur arbitre pour rendre impair le nombre de participants
- Arbitre : serveur ne contenant pas de replica, mais participant au vote.

# Membres	Majorité	Tolérance aux pannes
3	2	1
4	3	1
5	3	2
6	4	2

Session illustrative

Démarrez une instance MongoDB pour chaque membre en spécifiant le répertoire de données et le port, comme montré ci-après:

```
1 mongod --replSet rs0 --dbpath "C:\data\db1" --port 27017 --
  bind_ip localhost
```

Pour le deuxième membre :

```
1 mongod --replSet rs0 --dbpath "C:\data\db2" --port 27018 --
  bind_ip localhost
```

Pour le troisième membre :

```
1 mongod --replSet rs0 --dbpath "C:\data\db3" --port 27019 --
  bind_ip localhost
```

Connectez-vous à l'une des instances MongoDB via le shell mongo

```
1 mongosh --port 27017
```

Session illustrative

Initialisez le réplica set

```
1 rs.initiate()  
2 rs.add("localhost:27019")  
3 rs.add("localhost:27020")
```

Vérifiez l'état du Replica Set :

```
1 rs.status()
```

Assurez-vous que tous les membres sont en ligne et fonctionnent correctement.

Session illustrative

Connectez-vous au master et insérez un nouvel enregistrement.

```
1 db.myCollection.insert({_id:1, nom:"Foulen"})
```

Connectez-vous à un des slaves et dites ce qui se passe lors de l'exécution de ces deux commandes

```
1 db.myCollection.find()
```

```
1 db.myCollection.find().readPref("primaryPreferred")
```

Session illustrative

Exercice Utilisez la commande ci-après pour simuler des pannes sur le master et/ou ses slaves.

```
1 use admin  
2 db.shutdownServer({ force: true })
```

❶ Dites ce qui se passe lorsque (traiter chaque cas de manière indépendante):

- ❶ Le master s'arrête
- ❷ Le master est un slave s'arrêtent simultanément;
- ❸ Tous les slaves s'arrêtent simultanément;

❷ Pour chacun des cas susmentionnés, testez la commande ci-après et dites ce qui se passe

```
1 db.myCollection.insertOne({nom: "sarah"}, {writeConcern: { w: "majority", j: 1 }})
```

Chapitre 4 - RéPLICATION

RéPLICATION : pourquoi et comment?

Cohérence forte vs. Eventual Consistency

Théorème CAP

Heartbeats, Failover et élections

La réPLICATION dans la pratique - Cas de MongoDB

QUIZ

QUIZ

- ① Comment pourrait-on obtenir une cohérence forte dans un système utilisant la réPLICATION ASYNCHRONE?

- ② Expliquez le principe de l'Eventual Consistency (cohérence à terme)

- ③ Expliquez la raison pour laquelle augmenter le nombre de serveurs peut dans certains cas ne pas améliorer la tolérance aux pannes

- ④ Expliquez le théorème CAP.

- ⑤ Expliquez comment se fait le failover dans un Replica Set MongoDB

- ⑥ Supposez que vous disposez d'un Replica Set contenant 3 membres. Dites ce qui se passe lorsque
 - ① Tous les slaves tombent simultanément en panne
 - ② Le master tombe en panne
 - ③ Le master et un slave tombent simultanément en panne.

Chapitre 5 - Sharding

Partitionner : pourquoi et comment?

Partitionner, pourquoi?

Partitionner, comment?

Élasticité et load balancing

Techniques de partitionnement

Le sharding dans la pratique - Cas de MongoDB

Quiz

Partitionner, pourquoi?

Big Data : nécessite de **très hautes capacités de stockage** (disques) **et de traitement** (CPU et RAM)

Classiquement on a 2 approches pour l'augmentation des capacités : la scalabilité verticale et la scalabilité horizontale.

Scalabilité verticale : augmentation de la puissance de calcul sur une seule machine.

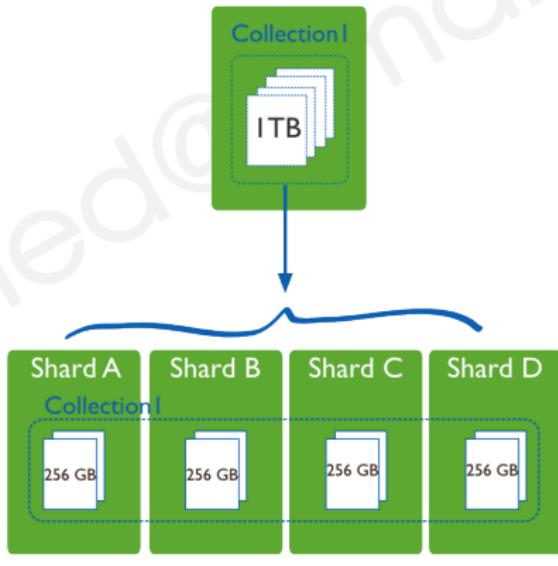
Limites de la **scalabilité verticale**

- **Scalabilité limitée** : les ressources d'une seule machine ne peuvent pas croître indéfiniment
- A performances égales, une machine à très hautes performances est **disproportionnellement** plus **chères** qu'un cluster de machines à bas coût (*commodity servers*).
- Les fournisseurs d'infrastructures cloud ne permettent que l'approvisionnement de "petites" instances (machines virtuelles).

Partitionner, pourquoi?

Scalabilité horizontale : augmentation de la puissance de calcul par l'allocation de nouvelles machines

- Les données et les calculs sont répartis sur une grappe de serveurs
- **Augmentation du débit et réduction de la charge** (nombre d'opérations) de chaque serveur



Partitionner, comment?

Déroulement du **sharding** (i.e. partitionnement)

1. Partitionner horizontalement une collection en **fragments (chunks)** en fonction d'un champ.

le champ selon lequel les données sont partitionnées est appelé **clé de partitionnement** (i.e. **shard key**).

Chunk : a généralement une **taille maximale fixe prédéfinie** (de 64 MO à qq GO). Appelé également *tablet*, *region*, *bucket*, etc.

2. Distribuer les chunks sur les serveurs (i.e. **shards**) ;
3. Maintenir un **répertoire** indiquant que tel chunk se trouve dans tel shard
4. Un serveur (le **Routeur**), ou ensemble de serveurs, consulte le répertoire et oriente les recherches vers le bon shard.

Chapitre 5 - Sharding

Partitionner : pourquoi et comment?

Élasticité et load balancing

Techniques de partitionnement

Le sharding dans la pratique - Cas de MongoDB

Quiz

Équilibrage de la charge et élasticité

Le sharding doit assurer l'**élasticité** et l'**équilibrage de charge**

Load balancing

- La charge doit être équitablement répartie sur les serveurs de la grappe, sinon le serveur le plus lent ralentit tout le traitement
- e.g. Rappelez-vous, les reducers ne peuvent démarrer que si TOUS les mappers ont terminé leur travail!

Répartition dynamique (élasticité) : adaptation automatisée

- à l'ajout/suppression de données
- à l'ajout/disparition de serveurs

Équilibrage de la charge et élasticité

L'équilibrage de charge et l'élasticité se basent sur 2 opérations : le **split** et la **migration** de chunks

- **Split** : Diviser un chunk qui a dépassé la taille maximale prédéfinie

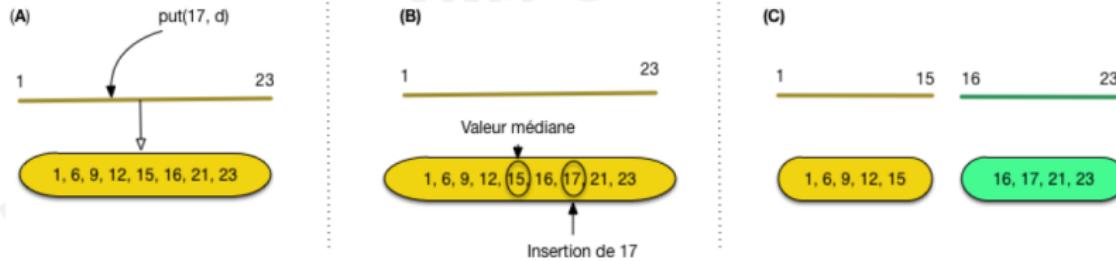
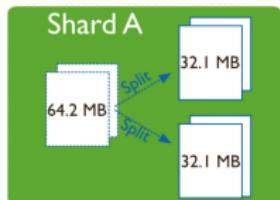


Figure reprise de <http://b3d.bdpedia.fr/>

Équilibrage de la charge et élasticité

- **Migration et équilibrage** : déplacement d'un chunk d'un shard à un autre pour une répartition équitable

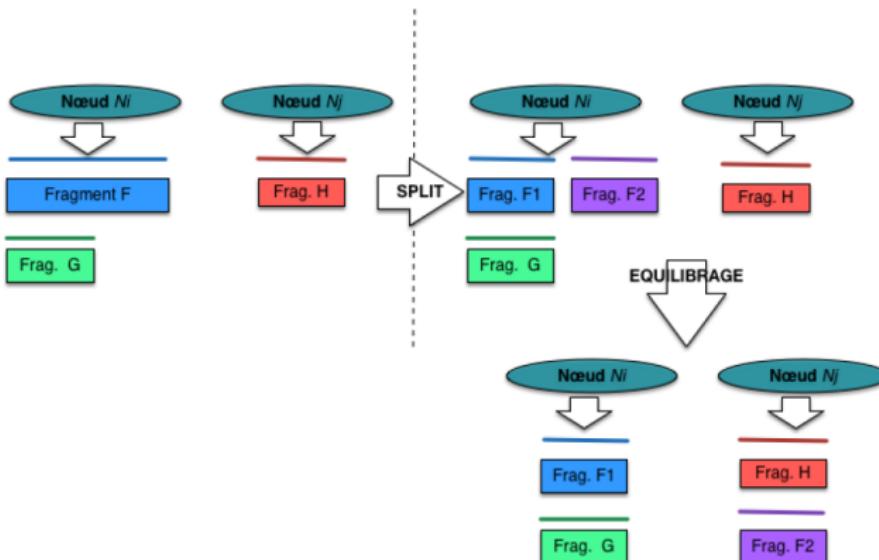


Figure reprise de <http://b3d.bdpedia.fr/>

- Le "split" n'est pas coûteux, la migration l'est. Pourquoi?

⇒ généralement le système effectue une seule migration à la fois (à un instant donné)

Tolérance aux pannes

Un système NoSQL doit savoir gérer la reprise sur pannes **failover**

- Le serveur stockant le répertoire est un SPOF \Rightarrow **Le répertoire est répliquée** : 3 copies au minimum dans un environnement de production.
- Les **fragments sont répliqués**

Chapitre 5 - Sharding

Partitionner : pourquoi et comment?

Élasticité et load balancing

Techniques de partitionnement

Le sharding dans la pratique - Cas de MongoDB

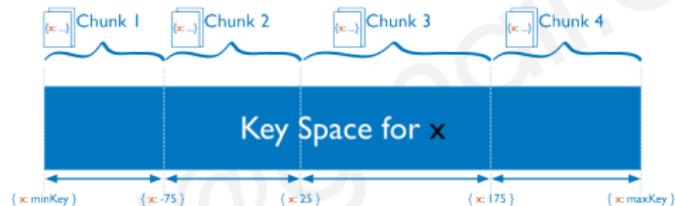
Quiz

Techniques de partitionnement

Deux techniques de partitionnement

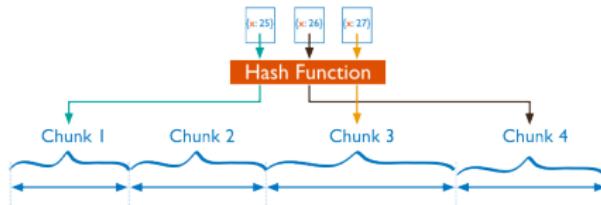
- **par intervalle** : données triées sur la clé, puis groupées par intervalles (**Arbre B+**).

Exemples représentatifs : MongoDB, HDFS (Hadoop), GFS (Google).



- **par hachage** : par hachage sur la clé, avec répertoire de hachage.

Exemples représentatifs : Chord, Dynamo, Cassandra, beaucoup d'autres.

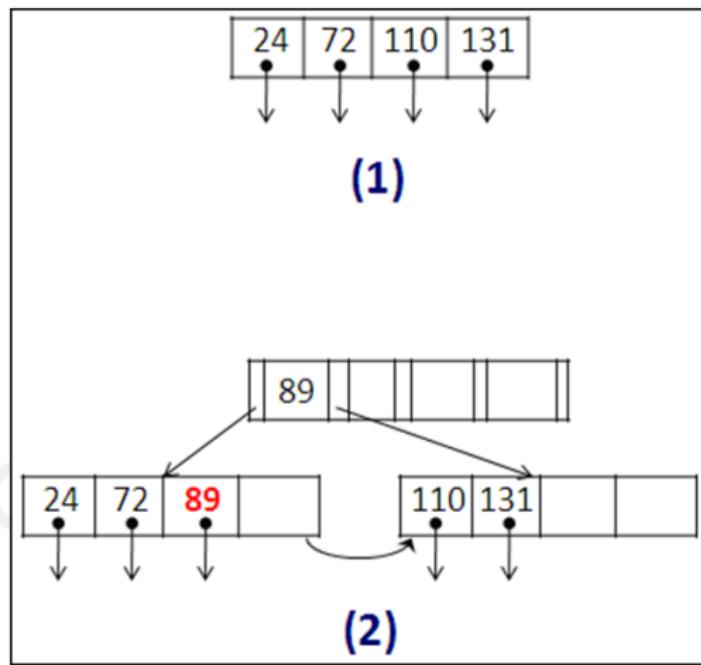


Arbre B+ - Vue d'ensemble

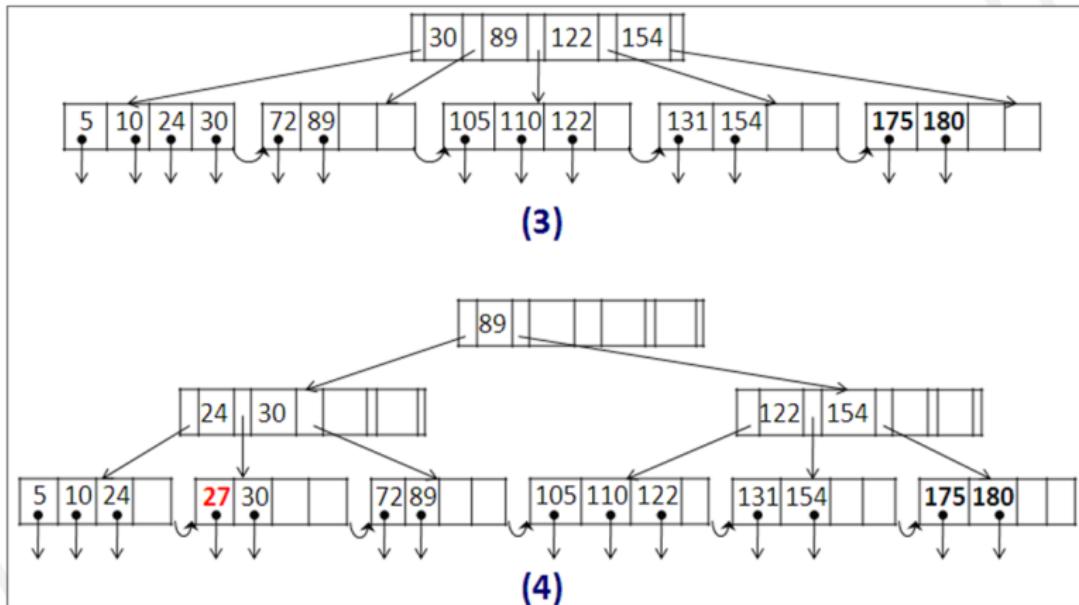
Arbre B+

- Structure d'indexation arborescente, équilibrée (*Balanced*), inspirée de l'indexation des fichiers séquentiels
- **Objectif** : amortissement des opérations de mises-à-jour
- ⇒ Création et maintenance de manière incrémentale et dynamique

Arbre B+ - Exemple



Arbre B+ - Exemple



Chapitre 5 - Sharding

Partitionner : pourquoi et comment?

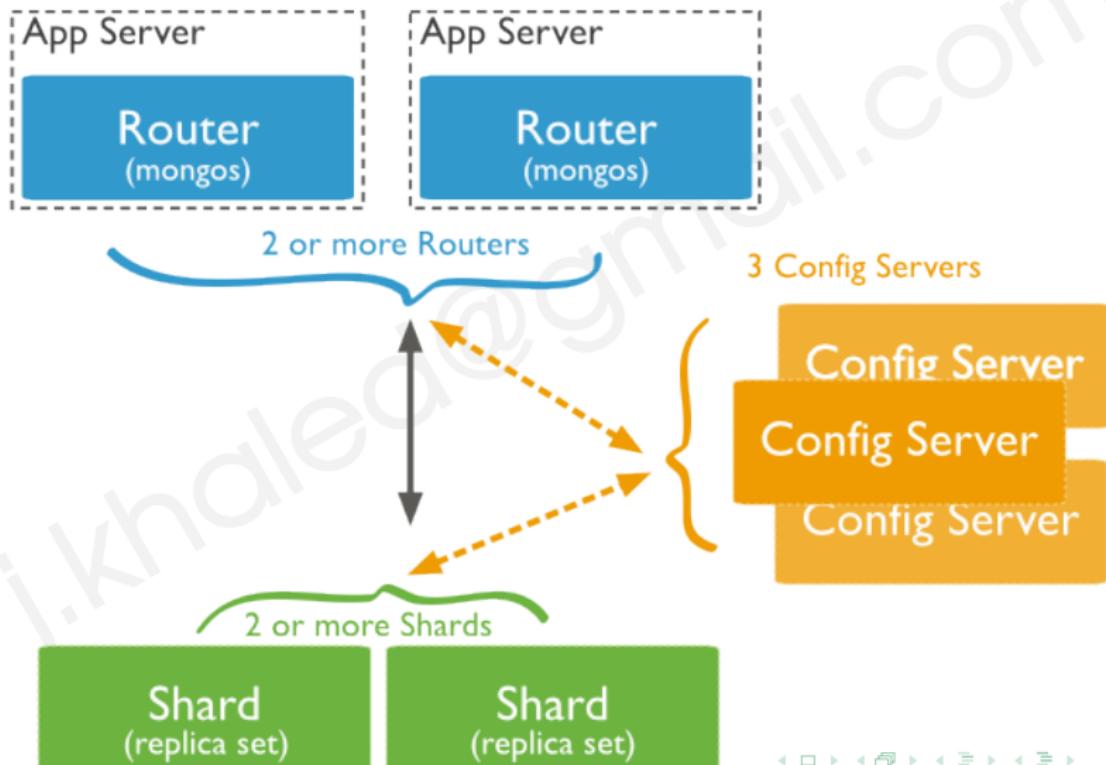
Élasticité et load balancing

Techniques de partitionnement

Le sharding dans la pratique - Cas de MongoDB

Quiz

Le sharding de MongoDB



Le sharding de MongoDB

Config Server

- Stocke le répertoire (obligatoirement répliqué) : "**Serveurs de métadonnées**"
- Processus mongod

Instances mongos (Query Routers)

- prend en charge les requêtes des clients.
- consulte le répertoire (Config Server) et oriente les requêtes vers le shard approprié
- agrège les résultats et envoie la réponse au client.
- **stateless** (ne stocke pas les données)
- il peut y avoir plusieurs routeurs (mais un client envoie une requête à un seul)

Chaque shard est un replica set complet

- Shard : "**serveur de données**"
- Il s'agit d'un processus mongod maître avec un ou plusieurs autres processus mongod secondaires et éventuellement un mongod arbitre
- On peut **ajouter ou supprimer un shard dynamiquement** (le processus d'arrière plan Balancer se charge de répartition de la charge).

Session illustrative

- Créer les répertoires pour les serveurs de configuration :

```
1 mkdir "c:\..\config1" "c:\..\config2" "c:\..\config3"
```

- Lancer les serveurs de configuration :

```
1 mongod --configsvr --dbpath "c:\..\config1" --port 27020  
    --bind_ip localhost --replSet rsConfig  
2 mongod --configsvr --dbpath "c:\..\config2" --port 27021  
    --bind_ip localhost --replSet rsConfig
```

- Initialisation du replica set :

```
1 mongosh --port 27020  
2 rs.initiate()  
2 rs.add("localhost:27021")
```

Session illustrative

- Créer des répertoires et lancer les Shard Servers :

```
1     mkdir "c:..\n11" "c:..\n12"
2     mongod --replSet rs1 --dbpath "c:..\n11" --port 27030
3         --bind_ip localhost --shardsvr
4     mongod --replSet rs1 --dbpath "c:..\n12" --port 27031
5         --bind_ip localhost --shardsvr
```

- Initialiser le replica set pour le shard :

```
1     mongosh --port 27030
2
3         rs.initiate()
4         rs.add("localhost:27031")
```

- Créer et lancer un autre replica set pour un deuxième Shard Server :

```
1     mkdir "c:\data\n21" "c:..\n22"
2     mongod --replSet rs2 --dbpath "c:..\n21" --port 27040
3         --bind_ip localhost --shardsvr
4     mongod --replSet rs2 --dbpath "c:..\n22" --port 27041
5         --bind_ip localhost --shardsvr
```

Session illustrative

- Initialiser le replica set du deuxième shard :

```
1 mongosh --port 27040
1 rs.initiate()
2 rs.add("localhost:27041")
```

- Déployer un ou plusieurs mongos

```
1 mongos --configdb rsConfig/localhost:27020,localhost
:27021 --bind_ip localhost --port 27018
```

- Ajouter des Shard Servers au mongos

```
1 mongosh --port 27018
2
1 sh.addShard("rs1/localhost:27030,localhost:27031")
2 sh.addShard("rs2/localhost:27040,localhost:27041")
3 sh.status()
```

Session illustrative

- Configurer la taille des chunks

```
1 mongosh --port 27018
1 use config
2 db.settings.save({_id: "chunksize", value: 64})
```

- Spécifier les données à partitionner

```
1 sh.enableSharding("dbTP")
2 sh.shardCollection("dbTP.myCollection", { _id: "hashed"
})
```

Chapitre 5 - Sharding

Partitionner : pourquoi et comment?

Élasticité et load balancing

Techniques de partitionnement

Le sharding dans la pratique - Cas de MongoDB

Quiz

Quiz

- ① Pourquoi est-il nécessaire de partitionner les données dans un système distribué?
- ② Dites quelle facteurs il faut considérer lors du choix de la taille des chunks
- ③ Que signifie l'élasticité et comment peut-on l'obtenir
- ④ Quelles sont les étapes nécessaires pour mettre en place un sharding dans MongoDB?
- ⑤ Comment le système de répartition des chunks est-il équilibré dans MongoDB?
- ⑥ Quels critères sont importants lors du choix d'une clé de partitionnement (shard key)?

Chapitre 6 - Au-delà des document-stores

Apache Cassandra, une base peer-to-peer avec hachage cohérent

Rappel du hachage

Hachage cohérent (*Consistent Hashing*)

Le sharding dans Cassandra

Gossip et Hinted Handoff

Lectures et écritures

Apache HBase

Quiz

Hachage statique

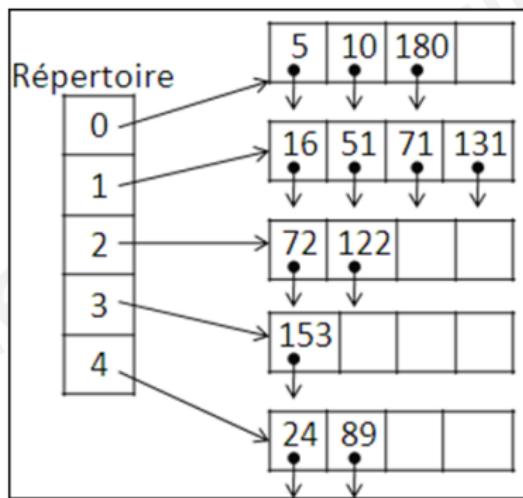
Principe

- Nous disposons de s shards ($s \ll n$, avec n le nombre total d'entrées (k, s))
- Une **fonction de hachage h** associe chaque valeur de clé à un des s shards.
Exemple : $h(k) = k \bmod s$
- Recherche d'une clé $k \Rightarrow$ calcul de $h(k)$ pour trouver l'adresse du serveur contenant k
- h est choisie de manière à répartir uniformément les clés dans les shards
- ⇒ Simple et efficace !

Hachage statique

Exemple : $n = 12, s = 5$

- $h(k) = k \bmod 5$
- Un **répertoire** de 5 entrées de 0 à 4 pointe vers les shards



Hachage statique

Limites

- Pas de recherche par intervalle
- En cas d'ajout d'un nouveau serveur, nous devons modifier la fonction de hachage (*i.e.* $h(k) = k \bmod 6$), ce qui implique de réaffecter tous les documents, régénérer le répertoire, etc.

Pas du tout efficace, ni envisageable!

⇒ Nous devons trouver un moyen pour permettre l'élasticité sans affecter la répartition déjà existante

Pas du tout simple!

Pourquoi le Hachage Cohérent ?

- Le **hachage cohérent** minimise les redistributions en répartissant les clés de manière équilibrée sans nécessiter de grands remaniements.
- Attribue des clés et des nœuds sur un "anneau virtuel" circulaire.
- Chaque nœud et chaque clé est assigné à une position sur cet anneau en utilisant une fonction de hachage.
- Une clé est stockée sur le premier nœud dont la position dépasse celle de la clé sur l'anneau.
- Lorsque des nœuds sont ajoutés ou supprimés, seules les clés proches du nœud modifié doivent être réaffectées.

Fonctionnement du Hachage Cohérent

- **Étape 1** : Chaque nœud est mappé sur une position de l'anneau en utilisant une fonction de hachage.
- **Étape 2** : Les clés sont également mappées sur l'anneau en utilisant la même fonction de hachage.
- **Étape 3** : Une clé est attribuée au premier nœud "dans le sens des aiguilles d'une montre" à partir de sa position sur l'anneau.
- **Étape 4** : Lorsqu'un nœud est ajouté ou supprimé, seules les clés affectées à ce nœud doivent être redistribuées.

Apache Cassandra

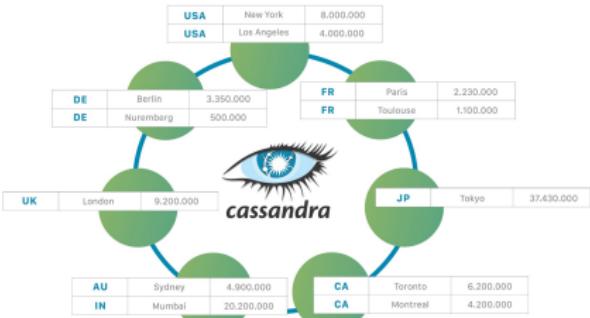
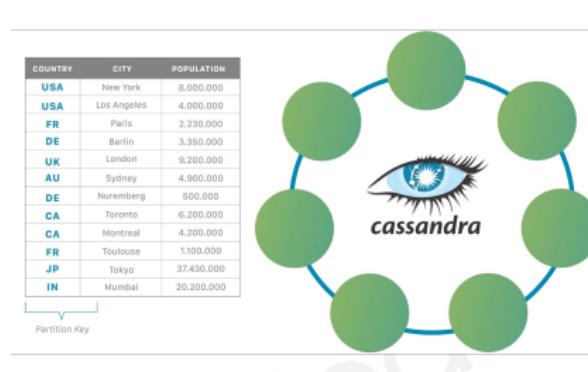
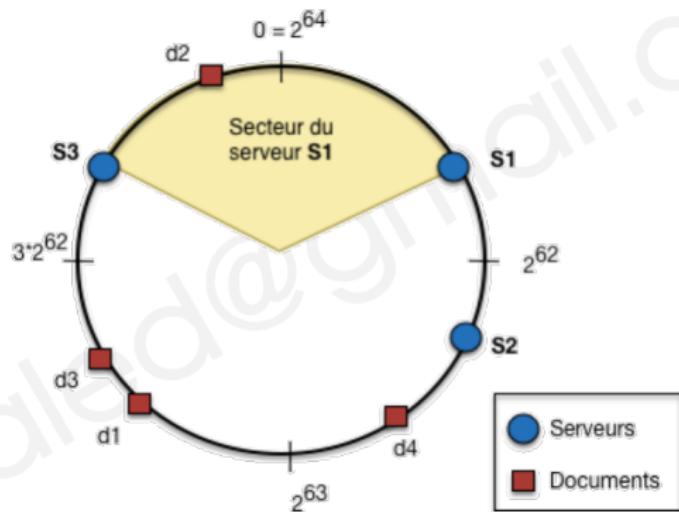


Figure reprise de <https://cassandra.apache.org>

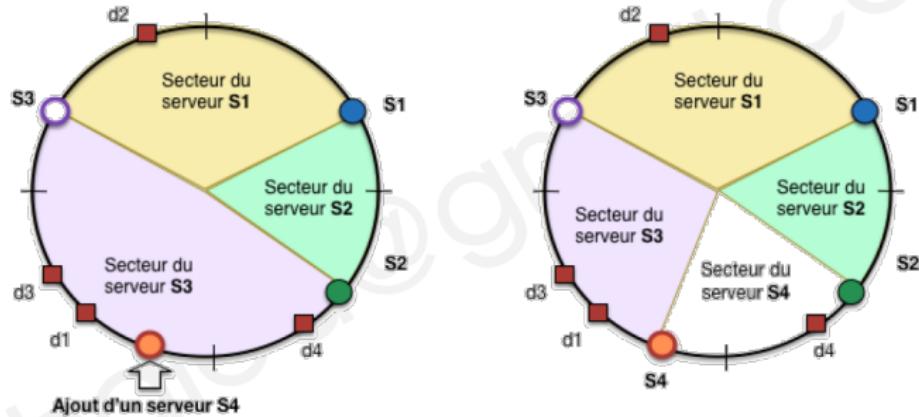
Considérations pour Cassandra

- **Cassandra** utilise le hachage cohérent pour répartir les données sur plusieurs nœuds dans un cluster.
- **Scalabilité Linéaire** : Grâce au hachage cohérent, Cassandra peut évoluer de manière linéaire en ajoutant de nouveaux nœuds sans perturber le système.
- **Cohérence** : Le modèle Eventual Consistency permet une forte disponibilité au détriment d'une cohérence immédiate.
- Lorsque de nouveaux nœuds sont ajoutés au cluster, ils s'intègrent à l'anneau sans redistribuer la majorité des données existantes.
- **Replication Factor** : Cassandra utilise la réPLICATION pour assurer la tolérance aux pannes. Chaque clé est répliquée sur plusieurs nœuds.

HRing de Cassandra



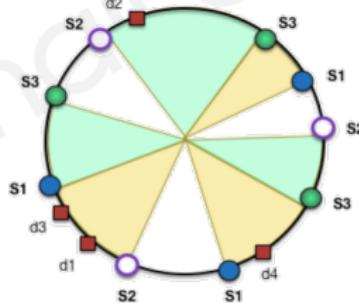
HRing de Cassandra



HRing de Cassandra

Panne ⇒ géré par la réPLICATION (surprise !) ; par exemple on copie sur la machine suivante dans l'anneau, etc.

Equilibrage ⇒ un même serveur (physique) est distribué en plusieurs points (virtuels) sur l'anneau.



Gossip Protocol et Hinted Handoff

• Gossip Protocol

- **But** : Utilisé pour la communication pair-à-pair entre les nœuds du cluster Cassandra.
- **Fonctionnement** : Chaque nœud envoie périodiquement des messages (gossips) à un sous-ensemble aléatoire de nœuds, partageant des informations sur l'état du cluster (état des nœuds, charge).
- **Avantages** : Adaptation dynamique du cluster en cas de changement (ajout/suppression de nœuds), robuste et efficace pour les grands clusters.

• Hinted Handoff

- **But** : Permet à Cassandra de gérer les écritures lorsque certains nœuds sont temporairement hors ligne.
- **Fonctionnement** : L'écriture est temporairement stockée sur un autre nœud, qui transfère ensuite les données au nœud initial une fois disponible.
- **Avantages** : Garantit que les données ne sont pas perdues lors des pannes temporaires, maintenant ainsi une haute disponibilité.

Lecture et écriture dans Apache Cassandra

Écriture :

- Les écritures sont d'abord enregistrées dans un **Commit Log** pour garantir la durabilité.
- Les données sont ensuite écrites dans un **Memtable**, une structure en mémoire.
- Lorsque le **Memtable** est plein, les données sont vidées dans un fichier **SSTable** sur le disque.

Lecture :

- Lors d'une lecture, Cassandra recherche d'abord les données dans le **Memtable**.
- Si les données ne sont pas trouvées, elles sont recherchées dans les **SSTables** stockés sur le disque.
- Les lectures sont distribuées entre les noeuds pour améliorer les performances.

Chapitre 6 - Au-delà des document-stores

Apache Cassandra, une base peer-to-peer avec hachage cohérent

Apache HBase

Hbase, quésaco?

Modèle de données

Lectures et écritures

Quiz

HBase, quésaco ?

- **HBase** : système NoSQL, distribué, **orienté colonnes**, construit sur HDFS.
- Utilise le **HDFS** comme système de fichiers distribué sous-jacent pour le stockage des données.
- **Principe important** : Les colonnes qui sont souvent utilisées ensemble sont placées dans la même famille de colonnes et stockées ensemble sur le disque, optimisant ainsi la localité spatiale.
- Modèle de données
 - **Table** : Un ensemble de lignes.
 - **Row Key** : Clé unique pour identifier une ligne.
 - **Famille de colonnes** : Un groupe logique de colonnes, regroupant celles souvent utilisées ensemble pour optimiser les lectures.
 - **Cellule** : Intersection d'une ligne, d'une famille de colonnes et d'un timestamp.

Modèle de données

HBase Data Model – LOGICAL VIEW

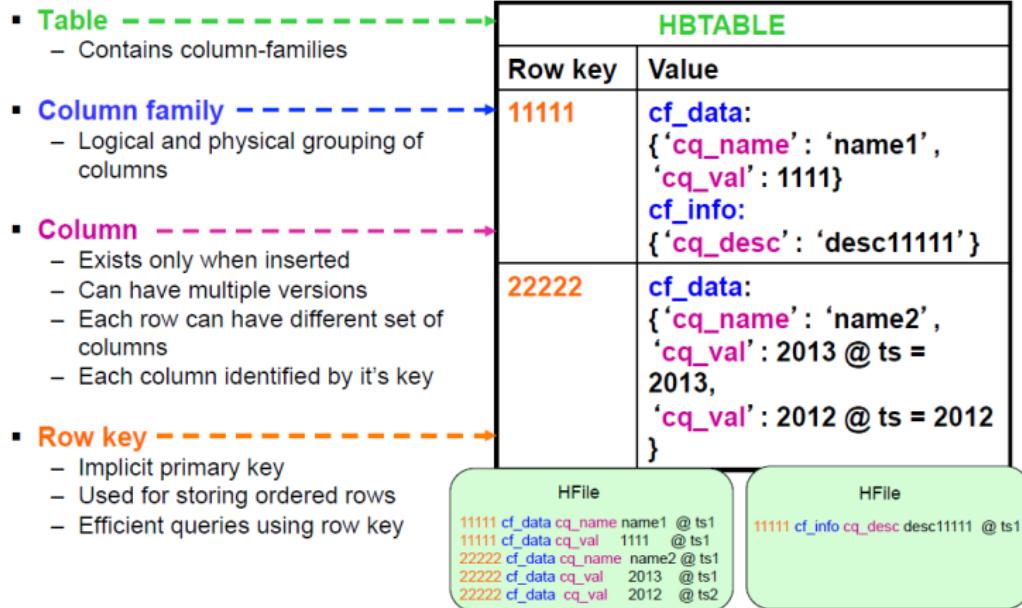


Figure reprise de "IBM Big Data Specialist"

Architecture de HBase

- **HBase Master** : Coordonne les opérations sur l'ensemble du cluster, gère la distribution des régions et la répartition des charges.
- **Régions** : HBase divise les tables en régions. Chaque région contient une gamme de lignes triées par clé de ligne.
- **Region Servers** : Les régions sont gérées par des serveurs régionaux qui gèrent les lectures/écritures pour les lignes de la région.
- **Zookeeper** : Utilisé pour la coordination et la gestion du cluster, y compris la détection des pannes des nœuds.

Lectures et Écritures dans HBase

• Écritures

- Les écritures dans HBase suivent un processus asynchrone pour améliorer les performances :
- Les données sont d'abord enregistrées dans le **WAL (Write Ahead Log)** pour garantir la durabilité.
- Ensuite, elles sont stockées dans la mémoire (MemStore) avant d'être finalement écrites dans un fichier HFile sur HDFS.
- Cela permet une tolérance aux pannes, car les données peuvent être récupérées à partir du journal en cas de défaillance.

• Lectures

- Les lectures suivent un chemin d'accès spécifique :
- Les données sont d'abord recherchées dans le **MemStore** (mémoire) pour des accès rapides.
- Si elles ne sont pas trouvées, elles sont lues à partir des **HFiles** sur HDFS.
- Utilisation de caches (BlockCache) pour optimiser la lecture répétée des mêmes blocs de données.

Chapitre 6 - Au-delà des document-stores

Apache Cassandra, une base peer-to-peer avec hachage cohérent

Apache HBase

Quiz

Quiz

- ① Qu'est-ce que le hachage cohérent et pourquoi est-il utilisé dans Cassandra?
- ② Comment Cassandra assure-t-il la tolérance aux pannes et la haute disponibilité?
- ③ Quelle est le principe et l'utilité du protocole Gossip?
- ④ Comment les opérations d'écriture sont-elles gérées dans HBase?
- ⑤ Quelles sont les principales caractéristiques du modèle de données d'HBase?

Exercice

Un système documentaire réparti utilise le hachage cohérent (*Consistent Hashing*) pour la répartition d'une collection de documents sur une grappe de serveurs.

La grappe est composée de 2 serveurs S_1 et S_2 , identifiés respectivement par 12007 et 12011. La collection contient 4 documents d_1, d_2, d_3 et d_4 , identifiés respectivement par 10, 14, 15 et 17. Les identifiants des serveurs et des documents sont récapitulés dans les tableaux ci-après. Pour simplifier nous supposons que le système en question utilise la fonction de hachage : $h(k) = k \bmod 6$ (le reste de la division par 6).

Serveur	Identifiant
S_1	12007
S_2	12010

Document	Identifiant
d_1	10
d_2	14
d_3	15
d_4	17

1. Représentez les serveurs et les documents sur l'anneau qui correspond à la table de hachage. En déduire le répertoire (la répartition des documents sur les différents serveurs).
2. Dites ce qui se passe lors de l'ajout d'un serveur S_3 identifié par 12015 à la grappe.
3. Expliquez (en une phrase) ce qu'il faudrait faire pour obtenir un meilleur équilibrage de la charge (*Load Balancing*) entre les serveurs de la grappe.
4. Expliquez (en une phrase) ce qu'il faudrait faire pour gérer les pannes qui peuvent survenir sur un des serveurs.

Exercice

