MAC - Méthode d'accès aux données

Modèle de Distribution et Cohérence

09 November 2025

Table des matières

1 Réplication

1.1 Introduction

Ce chapitre explore les défis majeurs posés par la **distribution des données sur plusieurs machines** et les principales approches pour y faire face.

1.2 Pourquoi distribuer les données?

Trois raisons principales:

- **Évolutivité**: répartir la charge au-delà des capacités d'une machine
- **Haute disponibilité**: continuer à fonctionner malgré les pannes
- **Latence réduite**: servir les utilisateurs depuis des centres proches

1.3 Architectures de distribution

1.3.1 Architecture à mémoire partagée (Scale Up)

Plusieurs processeurs, RAM et disques réunis sous un seul système avec une mémoire unique (SMP).

Désavantages: coûts élevés, croissance limitée, tolérance aux pannes limitée.

1.3.2 Architecture de disques partagés

Processeurs et RAM indépendants, mais disques partagés via réseau rapide. La complexité de gestion des conflits limite l'évolutivité.

1.3.3 Architectures sans partage (Scale Out)

Chaque nœud gère indépendamment ses ressources. Coordination au niveau logiciel via réseau.

Avantages: matériel standard, distribution géographique, accessible aux petites entreprises.

Problèmes: synchronisation, cohérence, gestion des pannes.

1.4 Modèles de distribution

Deux approches génériques:

- **Partitionnement (sharding)**: diviser la base en sous-ensembles attribués à différents nœuds
- **Réplication**: maintenir des copies des mêmes données sur plusieurs nœuds

Ces mécanismes sont souvent utilisés ensemble.

1.5 Réplication

1.5.1 Raisons principales

- Conserver les données géographiquement proches (latence)
- Continuer à fonctionner malgré les pannes (disponibilité)
- Augmenter le débit de lecture

Le défi: **gestion des modifications** apportées aux données répliquées.

1.5.2 Algorithmes de réplication

Trois catégories:

- Réplication à **leader unique**
- Réplication **multi-leader**
- Réplication **sans leader**

1.5.3 Compromis

Questions principales:

- Répliquer de manière **synchrone ou asynchrone**?
- Comment **gérer les répliques qui ont échoué**?

1.6 Réplication à leader unique

Un **leader** traite les écritures, les **followers** les répliquent. Lectures possibles depuis le leader ou les followers.

Utilisée par: PostgreSQL, MySQL, Oracle, MongoDB, RethinkDB, etc.

1.6.1 Réplication synchrone vs asynchrone

Synchrone: le leader attend la confirmation des followers

- Avantage: données à jour garanties
- Inconvénient: peut bloquer si un follower est lent

Asynchrone: le leader n'attend pas

- Avantage: rapide, pas de blocage
- Inconvénient: risque de perte de données

Semi-synchrone: un seul follower synchrone, les autres asynchrones (compromis courant).

1.6.2 Ajout de nouveaux followers

Processus en 4 étapes:

1. Prendre un **snapshot cohérent** du leader
2. Copier le snapshot sur le nouveau follower
3. Demander toutes les modifications après le snapshot (via **log sequence number**)
4. Rattraper le retard (**catch up**)

1.6.3 Gestion des pannes

1.6.3.1 Panne de follower: Catch-up recovery

Le follower maintient un log local. Après redémarrage, il demande les modifications manquées.

1.6.3.2 Panne de leader: Failover

Processus automatique en 3 étapes:

1. **Déterminer l'échec**: via timeout (ex. 30 secondes)
2. **Choisir un nouveau leader**: via controller node ou processus électoral (algorithme de consensus: Raft, Paxos)
3. **Reconfigurer le système**: clients redirigés, ancien leader devient follower

1.6.4 Journaux de réplication

1.6.4.1 Statement-based replication

Le leader enregistre les requêtes SQL (INSERT, UPDATE, DELETE) et les envoie aux followers.

Problèmes: fonctions non-déterministes (`NOW()` , `RAND()`), dépendances aux données existantes, effets secondaires.

1.6.4.2 Write-ahead log (WAL) shipping

Le log contient les octets modifiés dans les blocs de disque. Utilisé par PostgreSQL et Oracle.

Avantage: copie exacte des données.

Inconvénient: couplage fort au moteur de stockage, incompatibilité entre versions.

1.6.4.3 Logical (row-based) log

Le log décrit les modifications au niveau des lignes (valeurs insérées, supprimées, mises à jour).

Avantages:

- Découplage du format de stockage (rétrocompatibilité)
- Facile à analyser par des applications externes

1.6.4.4 Trigger-based replication

Utilise des déclencheurs pour enregistrer les modifications dans une table distincte, lue par un processus externe.

Avantages: très flexible, logique applicative personnalisée.

Inconvénients: coûts plus élevés, plus de bugs potentiels.

1.6.5 Problèmes de délai de réplication

Le **délai de réplication** (**replication lag**) est le temps entre une écriture sur le leader et sa réflexion sur un follower.

Cohérence éventuelle: état temporaire d'incohérence qui se résout avec le temps.

1.6.5.1 Reading Your Own Writes

Problème: l'utilisateur peut ne pas voir ses propres modifications si elles n'ont pas encore atteint le follower.

Solution: **cohérence lecture-écriture**

- Lire depuis le leader pour les données modifiables par l'utilisateur
- Suivre le timestamp de la dernière écriture
- Utiliser le **log sequence number** pour garantir la cohérence

1.6.5.2 Lectures monotones

Problème: deux lectures successives peuvent retourner des résultats incohérents (régression).

Solution: s'assurer que chaque utilisateur lit toujours depuis la même réplique (ex. via hachage de l'ID utilisateur).

1.6.5.3 Solutions générales

- Accepter le délai si tolérable
- Implémenter des garanties au niveau application (complexe)
- Utiliser des **transactions** pour garanties plus fortes

1.7 Réplication multi-leader

1.7.1 Principe

Plusieurs nœuds acceptent les écritures. Chaque leader agit comme follower des autres leaders.

1.7.2 Problèmes principaux

1.7.2.1 Conflits d'écriture

Deux leaders peuvent accepter des écritures conflictuelles sur la même donnée. Nécessite une **résolution des conflits**.

1.7.2.2 Ordre incorrect des écritures

Une réplique peut recevoir les écritures dans un ordre différent (ex. UPDATE avant INSERT).

Solution: utiliser des **vecteurs de version** pour déterminer l'ordre causal correct.

> ⚠ **Warning**
>
> Les techniques de détection de conflits ne sont pas parfaites. Il faut bien lire la documentation et tester intensivement.

1.7.3 Cas d'usage

La réplication multi-leader est rarement adaptée (complexité > avantages). Situations utiles:

- **Opérations offline**: chaque appareil a une base locale (ex. calendrier mobile/desktop)
- **Collaborative editing**: chaque utilisateur modifie localement, réplication asynchrone (ex. Google Docs)
- **Plusieurs centres de données**: un leader par région géographique (ex. système de réservation)

1.8 Réplication sans leader

1.8.1 Principe

Abandonner le concept de leader: **n'importe quelle réplique peut accepter directement les écritures** des clients.

Architecture **Dynamo-style** (Amazon Dynamo), utilisée par Riak, Cassandra, Voldemort.

1.8.2 Gestion des pannes

Pas de failover possible (pas de leader). Les clients envoient les écritures aux répliques en parallèle.

Exemple: écriture envoyée à 3 répliques

- 2 répliques disponibles acceptent l'écriture
- 1 réplique en panne la manque
- Si 2 confirmations suffisent → écriture réussie

1.8.3 Lecture avec nœud en panne

Problème: un nœud qui revient après panne peut avoir des données obsolètes.

Solution: envoyer les requêtes de lecture à plusieurs nœuds en parallèle. Utiliser les **numéros de version** pour déterminer la valeur la plus récente.

1.8.4 Quorums de lecture et d'écriture

S'il y a **n répliques**:

- Chaque écriture doit être confirmée par **w nœuds**
- On doit interroger au moins **r nœuds** pour chaque lecture
- Tant que $\mathbf{w + r > n}$, au moins un des nœuds lus sera à jour

Configuration courante: n impair (3 ou 5), w = r = (n + 1) / 2

1.8.4.1 Exemple

Avec n = 5, w = 3, r = 3, on peut tolérer 2 nœuds indisponibles.

1.8.4.2 Ajustements possibles

Pour une charge avec peu d'écritures et nombreuses lectures:

- w = n (toutes les répliques), r = 1 (une seule réplique)
- Avantage: lectures très rapides
- Inconvénient: un seul nœud en panne bloque toutes les écritures

> ⚠ **Warning**
>
> Les lectures et écritures sont envoyées aux n répliques en parallèle. Les paramètres w et r déterminent combien doivent signaler le succès.

2 Partitionnement et Cohérence

2.1 Introduction au Partitionnement

Le **partitionnement** (aussi appelé *sharding*) consiste à diviser les données en partitions lorsque la réplication seule ne suffit pas face à des données très volumineuses ou un débit de requêtes très élevé.

Les terminologies varient selon les systèmes:

- **shard** dans MongoDB, Elasticsearch et SolrCloud
- **region** dans HBase
- **tablet** dans Bigtable
- **vnode** dans Cassandra et Riak
- **vBucket** dans Couchbase

2.1.1 Partitionnement et Réplication

Le partitionnement est souvent combiné avec la réplication pour garantir la tolérance aux pannes. Chaque partition a un leader attribué à un nœud, avec des followers attribués à d'autres nœuds. Les nœuds peuvent servir de leaders pour certaines partitions et de followers pour d'autres.

2.1.2 Objectifs du Partitionnement

Partitionnement équitable: Chaque nœud gère sa part de données et de requêtes de manière proportionnelle, ce qui augmente potentiellement le débit global.

Problème à éviter - Partitionnement asymétrique: Certaines partitions contiennent plus de données ou reçoivent plus de requêtes, ce qui compromet l'efficacité et peut conduire à des **points chauds** (*hot spots*), où une seule partition supporte une charge disproportionnée.

2.2 Stratégies de Partitionnement

2.2.1 Partitionnement des données clé-valeur

L'approche la plus simple pour éviter les points chauds serait d'attribuer des enregistrements aux nœuds de manière aléatoire, mais cela rend impossible la localisation d'un enregistrement lors de la lecture.

Une approche plus efficace implique un modèle de données clé-valeur simple, dans lequel les enregistrements sont accessibles par leurs clés primaires, ce qui permet une récupération efficace et fournit un moyen pour localiser les enregistrements.

2.2.2 Partitionnement par intervalle de clé

On attribue une plage continue de clés (d'un minimum à un maximum) à chaque partition. En connaissant les limites entre les plages, on peut facilement déterminer quelle partition contient une clé donnée.

Les plages de clés ne sont pas nécessairement espacées de manière égale. Les limites de partition peuvent être choisies manuellement par un administrateur, ou la base de données peut les choisir automatiquement.

Avantages:

- À l'intérieur de chaque partition, les données peuvent être gardées triées
- Facilite les **requêtes de plage** (*range queries*)

Exemple d'utilisation: Application stockant les données d'un réseau de capteurs, où la clé est le timestamp de la mesure. Récupérer toutes les lectures d'un mois particulier devient efficace.

Inconvénients:

- Risque de points chauds selon les motifs d'utilisation
- Si on écrit les données des capteurs au fur et à mesure, toutes les écritures finissent par aller vers la même partition (celle d'aujourd'hui), créant un point chaud

2.2.3 Partitionnement par hachage de clé

On utilise une fonction de hachage pour déterminer la partition pour chaque clé. On attribue à chaque partition une plage de hachages (plutôt qu'une plage de clés), et chaque clé dont le hachage se situe dans la plage d'une partition sera stockée dans cette partition.

Avantages:

- Répartition équitable des clés entre les partitions
- Les frontières des partitions peuvent être régulièrement espacées ou choisies de manière pseudo-aléatoire

Inconvénients:

- Comme les clés ne sont plus adjacentes, on perd la possibilité d'effectuer des **requêtes de plage efficaces**
- Dans MongoDB avec hachage, toute requête de plage doit être envoyée à toutes les partitions
- Les requêtes de plage sur la clé primaire ne sont pas offertes par Riak et Couchbase

2.3 Rééquilibrage des Partitions

Le rééquilibrage est le processus de déplacement de la charge d'un nœud du cluster à un autre, nécessaire lorsque:

- Le débit des requêtes augmente
- La taille de l'ensemble de données augmente
- Une machine tombe en panne et d'autres machines doivent assumer ses responsabilités

2.3.1 Approche à ne pas adopter: hash mod N

Utiliser `hash(key) mod N` semble simple mais pose problème: si le nombre de nœuds N change, la plupart des clés doivent être déplacées d'un nœud à un autre.

Exemple: Avec `hash(key) = 123456`:

- 10 nœuds: clé sur nœud 6 ($123456 \bmod 10 = 6$)
- 11 nœuds: clé sur nœud 3 ($123456 \bmod 11 = 3$)
- 12 nœuds: clé sur nœud 0 ($123456 \bmod 12 = 0$)

Des mouvements aussi fréquents seraient excessivement coûteux.

2.3.2 Partitions à nombre fixe

Principe: Créer beaucoup plus de partitions qu'il n'y a de nœuds et attribuer plusieurs partitions à chaque nœud.

Fonctionnement:

- Si un nœud est ajouté, le nouveau nœud peut « voler » quelques partitions de chaque nœud existant jusqu'à ce que les partitions soient à nouveau équitablement réparties
- Si un nœud est supprimé, la même chose se produit en sens inverse
- Seules des partitions entières sont déplacées entre les nœuds
- Le nombre de partitions ne change pas, ni l'attribution des clés aux partitions
- La seule chose qui change est l'affectation des partitions aux nœuds

Cette approche de rééquilibrage est utilisée dans Riak, Elasticsearch, Couchbase et Voldemort.

2.3.3 Automatisation du Rééquilibrage

Rééquilibrage entièrement automatique: Le système décide automatiquement quand déplacer des partitions d'un nœud à un autre, sans aucune intervention de l'administrateur.

Rééquilibrage entièrement manuel: L'attribution des partitions aux nœuds est configurée explicitement par un administrateur et ne change que lorsque l'administrateur la reconfigure explicitement.

Il existe un spectre de solutions possibles entre ces extrêmes. Par exemple, Couchbase, Riak et Voldemort génèrent une attribution de partition suggérée automatiquement, mais nécessitent l'approbation d'un administrateur.

2.4 Request Routing (Service Discovery)

Une fois qu'on a partitionné les données sur plusieurs nœuds, une question reste: « Quand un client veut faire une requête, comment sait-il à quel nœud il faut se connecter? »

De plus, avec le rééquilibrage des partitions, leurs affectations aux nœuds changent. Il faut rester au courant de ces changements pour répondre à cette question.

Ce problème, connu comme **service discovery**, existe pour tous les logiciels accessibles via réseau qui visent une haute disponibilité dans des configurations redondantes.

2.4.1 Trois approches

2.4.1.1 Approche 1: Pilotée par le client

Les clients peuvent contacter n'importe quel nœud (ex. via un répartiteur de charge en mode round-robin). Si le nœud contacté détient la partition concernée, il gère directement la demande; sinon, il la transmet au nœud approprié.

2.4.1.2 Approche 2: Couche de routage

Les demandes des clients sont routées via une couche de routage dédiée, qui détermine le nœud responsable de chaque demande et la transmet en conséquence. Elle ne traite pas directement les demandes mais agit comme un répartiteur de charge conscient des partitions.

2.4.1.3 Approche 3: Conscience du client

Les clients sont conscients du partitionnement et des affectations de nœuds, ce qui leur permet de se connecter directement au nœud approprié sans intermédiaire.

2.4.2 Le défi principal

Quelle que soit l'approche, le principal défi est: « Comment celui qui décide du routage apprend les changements des affectations de partitions aux nœuds? »

Pour y parvenir, les participants doivent se mettre d'accord. Il existe des protocoles de consensus pour des systèmes distribués, mais ils sont difficiles à implémenter correctement.

2.4.3 Service de coordination

On peut utiliser un service séparé de coordination comme **ZooKeeper** pour suivre les métadonnées des nœuds:

- Chaque nœud s'enregistre dans ZooKeeper
- ZooKeeper fait l'autorité de mappage des partitions aux nœuds
- Les acteurs, tels que la couche de routage ou le client conscient, peuvent s'abonner à ces informations dans ZooKeeper
- Chaque fois qu'une partition change de propriétaire ou qu'un nœud est ajouté ou supprimé, ZooKeeper informe la couche de routage afin qu'elle puisse maintenir ses informations de routage à jour

2.5 Cohérence dans les Bases de Données

La cohérence est l'**absence de contradiction** dans la base de données.

Principes de base:

- Toute transaction de base de données doit modifier les données affectées uniquement de manière autorisée
- Toutes les données écrites dans la base de données doivent être valides selon les règles d'intégrité

Un SGBDR centralisé garantit une **forte cohérence**.

Les bases de données NoSQL distribuées assouplissent souvent la cohérence:

- De cohérence forte vers **cohérence à terme** (*eventual consistency*)
- Théorème CAP
- Compromis entre cohérence et disponibilité

2.5.1 Problèmes de cohérence dans un SGBD centralisé

2.5.1.1 Conflit d'écriture (Dirty Write)

Problème: Deux utilisateurs souhaitent mettre à jour le même enregistrement. Si deux transactions tentent simultanément de mettre à jour le même objet, l'écriture ultérieure peut écraser une valeur non validée.

Exemple: Alice et Bob essaient simultanément d'acheter la même voiture. Cela nécessite deux écritures dans la base de données. La vente est attribuée à Bob (car il effectue la mise à jour réussie dans la table `listings`), mais la facture est envoyée à Alice (car elle effectue la mise à jour réussie dans la table des `invoices`).

Solutions:

- **Approche pessimiste**: éviter les conflits en acquérant des verrous en écriture avant la mise à jour (se fait automatiquement avec des transactions)
- **Approche optimiste**: laisser les conflits se produire, mais les détecter et prendre des mesures pour les résoudre

2.5.1.2 Conflit de lecture (Dirty Read)

Problème: Un utilisateur lit au milieu des écritures d'un autre utilisateur. Une transaction lit les écritures non-validées d'une autre transaction.

Exemple: L'utilisateur 1 insère un message non-lu et incrémente le compteur. Pour l'utilisateur 2, la boîte de réception affiche un message non lu, mais le compteur n'affiche aucun message non lu car l'incrémentation du compteur n'a pas encore eu lieu.

Solution idéale: Utiliser des transactions (ACID) pour garantir une **cohérence forte** (*strong consistency*).

2.5.2 Niveaux d'isolation des transactions

Différents niveaux d'isolation sont possibles:

- **read uncommitted** - permet dirty reads
- **read committed** - empêche dirty reads
- **repeatable reads** - empêche dirty reads et nonrepeatable reads
- **serializable** - isolation complète

2.5.3 Traitement des transactions dans NoSQL

2.5.3.1 Système centralisé

Si la base de données est centralisée, les transactions ACID peuvent être facilement mises en œuvre et leur gestion peut être réalisée en utilisant la **journalisation** (*logging*):

En termes simples: quand un client modifie une valeur, le gestionnaire de transaction enregistre cette modification dans le journal des transactions. Si nécessaire d'annuler la transaction ou si une panne arrive, le log contient les informations nécessaires.

2.5.3.2 Système distribué

Lorsque les données sont réparties sur des serveurs distribués (c'est-à-dire quand chaque serveur est une entité indépendante avec des enregistrements de journal séparés), le processus peut devenir plus délicat.

2.5.4 Cohérence de la réplication

Dans un système distribué où les données sont répliquées, il faut aussi assurer la cohérence entre répliques:

- S'assurer que le même élément de données a la même valeur lors de la lecture à partir de répliques différentes
- Après un certain temps, l'écriture se propage partout: **cohérence à terme** (*eventual consistency*)
- En attendant: données périmées (*stale data*)

2.5.4.1 Read-your-writes (cohérence de session)

Est violé si un utilisateur écrit et lit sur différentes répliques.

Solution: *sticky session* (affinité de session), c'est-à-dire une session liée à un nœud.

2.5.5 Exemple d'incohérence de réplication

Martin et Pramod veulent réserver la dernière chambre d'un hôtel. L'un est aux États-Unis, l'autre en Inde. Ils envoient leurs requêtes aux nœuds de traitement locaux.

Les nœuds de traitement doivent communiquer pour que le système dans son ensemble aboutisse à une conclusion.

Si le réseau est partitionné, il y a deux choix pour la base de données de réservation:

1. **Faire double réservation** de la chambre (incohérent mais disponible)
2. **Annoncer que le système est indisponible** (personne ne peut réserver pour le moment) → cohérent mais non disponible

2.6 Théorème CAP

Le théorème CAP définit trois propriétés pour les systèmes distribués:

C - Consistency (Cohérence): Après une mise à jour, tous les lecteurs d'un système distribué (en supposant la réplication) voient les mêmes données.

A - Availability (Disponibilité): Si un nœud (serveur) fonctionne, il peut lire et écrire des données. Chaque demande doit donner lieu à une réponse.

P - Partition Tolerance (Tolérance au partitionnement): Le système continue à fonctionner, même si deux ensembles de serveurs sont isolés. Un échec de connexion ne devrait pas arrêter le système.

2.6.1 Formulation du théorème

Théorème CAP: Un système de « données distribuées » ne peut pas avoir toutes les trois propriétés CAP.

Ou: seules **deux des trois** propriétés CAP sont possibles.

Initialement proposé par Eric Brewer en 2000. A reçu une preuve formelle par Seth Gilbert et Nancy Lynch quelques années plus tard.

2.6.2 Le théorème reformulé

Si vous avez un système où le réseau peut être partitionné, vous avez le choix entre:

- Cohérence?
- Disponibilité?

Face au partitionnement de réseau, les concepteurs d'application doivent choisir entre la cohérence parfaite ou la disponibilité parfaite.

Être cohérent signifie ne pas être en mesure de répondre à la requête d'un client car le système ne peut garantir de retourner l'écriture la plus récente. Ceci **sacrifie la disponibilité**. Le partitionnement de réseau oblige les nœuds non défaillants à rejeter les demandes des clients car ces nœuds ne peuvent pas garantir la cohérence des données.

Être disponible signifie être capable de répondre à la demande d'un client, mais le système ne peut garantir la cohérence, c'est-à-dire la valeur la plus récente écrite. Les systèmes disponibles fournissent la meilleure réponse possible dans les circonstances données.

2.6.3 Le compromis

Ce n'est pas vraiment un choix binaire dans l'ensemble du système:

- Un compromis entre les deux niveaux (cohérence et disponibilité)
- Il peut varier selon les opérations en cours d'utilisation
- Certaines opérations sont très cohérentes
- D'autres sont hautement disponibles
- La plupart du temps, c'est une **négociation de la cohérence contre le temps de réponse**!

3 Conclusion

La distribution des données nécessite des compromis fondamentaux entre cohérence, disponibilité et performances.

Le choix du modèle de réplication dépend des besoins spécifiques:

- **Leader unique**: simple, mais point de défaillance unique
- **Multi-leader**: complexe, pour cas spécifiques (offline, multi-datacenter)
- **Sans leader**: flexible, bonne tolérance aux pannes, mais cohérence éventuelle

Le partitionnement permet de gérer de grandes quantités de données et un débit élevé, mais introduit des défis de routage et de rééquilibrage.

Le théorème CAP impose des compromis fondamentaux dans les systèmes distribués, obligeant les concepteurs à choisir entre cohérence et disponibilité en cas de partitionnement réseau. Les systèmes NoSQL modernes offrent différentes stratégies pour gérer ces compromis selon les besoins de l'application.