

Séance 7

Consistance des données



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

- **Interface** entre le code client et le moteur NoSQL
 - Exemples d'APIs NoSQL
 - Le retour de SQL pour interroger des moteurs NoSQL
 - Fonctionnalités serveur et protocole d'accès aux données
- Le cas des moteurs **orienté-documents**
 - Lien avec la couche applicative et différence ORM/ODM
 - Exemple avec MongoDB et Mongoose.js

Objectifs

- Différents types de **consistance** des données
 - Consistance à la lecture et à l'écriture
 - Relaxation de la consistance et théorème CAP
 - Exemples de moteurs NoSQL existants
- **Techniques** pour assurer la consistance des données
 - Quorums pour systèmes peer-to-peer
 - Version stamps pour choisir la donnée la plus à jour

Consistance



Consistance des données

- Du relationnel centralisé au **NoSQL clusterisé**

Répartition des données de la DB sur plusieurs clusters

- Différence entre consistance forte et **consistance éventuelle**

Théorème CAP pour les bases de données NoSQL

- **Plusieurs formes** de consistance existantes

Consistance à la lecture, l'écriture...

Types de consistance

- Consistance avec les **autres utilisateurs**

Même donnée vue par deux utilisateurs faisant le même accès ?

- Consistance au sein d'une **même session**

Voit-on sa propre mise à jour d'une données dans une session ?

- Consistance au sein d'une seule **requête**

Donnée renvoyée par une seule requête cohérente en interne ?

- Consistance avec la **réalité**

Donnée reflète réalité que la DB tente de modéliser ?

Consistance de mise à jour (1)

- **Mise à jour** de la même donnée par plusieurs acteurs

Quelle est l'information qui sera effectivement stockée ?

- Conflit de type **write-write**

Même info mise à jour en même temps par plusieurs personnes

- **Sérialisation** des requêtes de mise à jour par le serveur

Choix d'appliquer les mises à jour l'une après l'autre

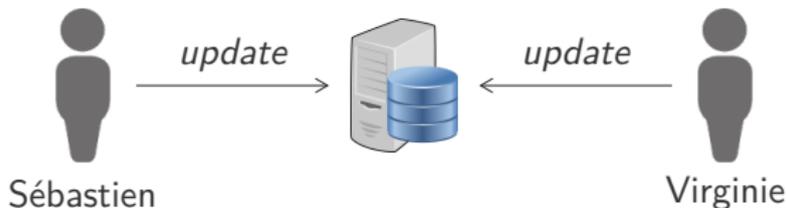
Consistance de mise à jour (2)

- Politique pour l'**ordre de traitement** des requêtes

Par exemple alphabétiquement par rapport aux noms des acteurs

- Dans cet exemple, la mise à jour de Virginie **écrase** Sébastien

Sans contrôle de concurrence, mise à jour de Sébastien perdue



Échec de mise à jour

- Plusieurs conséquences à l'**échec** d'une mise à jour
 - **Perte d'une mise à jour**
Pas forcément grave selon le contexte
 - **Échec de consistance**
Mise à jour sur des données pas à jour
- Importance de maintenir la **consistance des données**
 - Éviter que des conflits se produisent par **pessimisme**
 - Détection de conflits et résolution par **optimisme**

Exemple d'approche

- Approche **pessimiste**

Utilisation de lock pour l'écriture, à acquérir préalablement

- Approche **optimiste**

- Mise à jour conditionnelle par rapport à la dernière valeur lue
- Sauvegarder les deux mises à jour et signaler un conflit

- **Compromis** à trouver entre les deux approches

- Sécurité en évitant les conflits, mais lourdeur, deadlock...
- Liveness en répondant vite, mais conflits write-write

Serveur unique vs cluster

- Différence entre serveur unique et **cluster de plusieurs nœuds**
Complexité accrue lorsqu'utilisation de la réplication de données
- **Sérialisation consistante** des mises à jour sur serveur unique
Le serveur peut déterminer un ordre pour appliquer les opérations
- **Consistance séquentielle** dans un système distribué
L'ordre des opérations est préservée sur tous les nœuds

Consistance de lecture

- **Lecture** de la même donnée par plusieurs acteurs

Liront-ils tous la même information ?

- Conflit de type **read-write**

- Information composée de plusieurs entités élémentaires

- Lecture d'une information entre deux écritures la modifiant

- **Consistance logique** que des informations font sens ensemble

En relationnel, on utilise la notion de transaction

Transaction

- On entend que le NoSQL **ne supporte pas** les transactions
 - Et qu'il est donc impossible de garantir la consistance*
- Mais... ce n'est **pas complètement vrai**
 - N'affecte que certains types dont les orientées agrégats
 - DB orientées graphe supportent les transactions ACID
 - Support de mises à jour atomiques d'un seul agrégat
- Mise à jour plusieurs agrégats ouvre **fenêtre d'inconsistance**
 - Possibilité d'une lecture inconsistante dans cette fenêtre
 - Moins d'une seconde sur Amazon SimpleDB

Consistance de réplication (1)

- Problème de consistance lorsqu'on introduit de la **réplication**
Comment l'information est-elle diffusée aux différents nœuds ?
- Lectures donnent même valeur par **consistance de réplication**
Notamment lorsque les lectures se font sur des nœuds différents
- **Consistance « à la longue »** après diffusion
L'information finira par être mise à jour sur tous les nœuds

Consistance de réplication (2)

- Les données peuvent être **périmées** lorsque plus à jour (*stale*)

Le cache est d'ailleurs une forme de réplication

- Indépendance mais **renforcement** de l'inconsistance logique
 - Allongement de la durée de la fenêtre d'inconsistance
 - Ajout du délai réseau pour durée fenêtre sur slave vs master

- **Exemple** dans des applications répandues

Site de réservation d'hôtel utilisé depuis deux continents

Inconsistance avec soi-même

- **Ajout de commentaires** sur les articles d'un blog
 - Fenêtre d'inconsistance large ne pose pas de problèmes*
- Requêtes entrantes sur serveur **réparties sur plusieurs clusters**
 - Avec un système de load-balancing à l'entrée principale*
- Danger potentiel avec **conflit read-your-write**
 - Ajout d'un commentaire sur un post du blog
 - Rafraîchissement de la page et disparition du commentaire

Consistance de session

- Garantir la **consistance read-your-write** pour les utilisateurs
 - Une mise à jour faite par soi-même doit toujours être visible*
- **Consistance de session** pour assurer les read-your-write
 - Jusque la fin de la session volontaire ou non
 - Pas de garantie si connexions depuis deux machines différentes
- **Plusieurs approches** possibles
 - « *Sticky session* » en associant les sessions à des nœuds
 - Utilisation de version stamps sur les données

Sticky session

- Difficulté d'avoir la consistance de session avec **sticky session**
En particulier dans le cadre d'une réplication master-slave
- Normalement **lecture sur les slaves** et écriture sur le master
Pour accélérer les opérations de lecture
- Deux approches pour gérer **stockage/mise à jour** de la session
 - Délégation du write vers le slave de la session, qui le remonte
 - Transfert temporaire de la session vers le master

Relâchement de la consistance

- **Assurer la consistance** est généralement une bonne chose
Très difficile à obtenir sans faire d'autres sacrifices...
- **Différentes tolérances** par rapport au niveau de consistance
Selon le domaine d'application
- **Renoncement aux transactions** même en relationnel
Par exemple, pas de transactions aux débuts de MySQL



Théorème CAP

Théorème CAP

- Renoncement à la consistance justifié par **Théorème CAP**

Établi par Eric Brewer, prouvé par Seth Gilbert et Nancy Lynch

- On ne peut **garantir que deux** des trois propriétés suivantes

- **Consistency**

Plusieurs types de consistances des données existantes à garantir

- **Availability**

En discutant avec un nœud, on peut lire et écrire les données

- **Partition tolerance**

Le cluster survit à une séparation en plusieurs sous-clusters

Serveur unique

- Système avec **serveur unique** est de type CA
 - Pas de tolérance de partition, si le serveur crashe plus rien*
- Définition particulière de l'**availability** pour systèmes distribués
 - Requête reçue par un nœud pas crashé doit être répondue*
- Critère d'availability peut aussi être satisfait dans **un cluster**
 - Détection et fermeture de tous les nœuds d'une partition*

CAP en pratique

- En pratique, compromis entre **consistance et disponibilité**
Selon les besoins de l'application, on ajuste le compromis
- Réserveation de la dernière chambre d'un hôtel, **deux nœuds**
 - Distribution de type **peer-to-peer**
 - Vérification sur l'autre nœud avant réservation
 - Sacrifice de la disponibilité si un nœud crashe
 - Distribution **master-slave**, avec un master par hôtel
 - Réserveation faite uniquement sur le master
 - Inconsistance de mise à jour possible en cas de crash

Panier d'achats

- On peut toujours faire des write vers son **panier d'achats**
Même en cas de crash réseau, et on a plusieurs paniers d'achats
- Les panier sont **fusionnés** au moment du checkout
 - Union de tous les articles de tous les paniers d'achat existants
 - Le client a toujours le droit de vérifier son panier d'achats
- Relâchement de consistance de mise à jour pour **disponibilité**
Décision à faire dépend fortement du domaine

Relâchement de la durabilité

- **Consistance** est une propriété très demandée dans l'ACID

Former des unités de travail isolée et atomique

- **Durabilité** des données si le store ne perd pas d'information

Peut être relâchée pour gagner de la performance

- Plusieurs possibilités de **relâchement**

- Information stockée en mémoire, réécrite sur disque

Peut être fait pour stocker des sessions utilisateurs, par exemple

- Perte des mises à jour pas traitée par un master qui crashe

Ne pas dire ok au client tant que réplicats pas reçus mise à jour



Technique de consistance

Quorum (1)

- Combien de nœuds faut-il impliquer pour **forte consistance** ?
Chances d'éviter inconsistance \uparrow avec nombre de nœuds impliqués
- Assurer consistance avec données **répliquées sur trois nœuds**
Valider un write lorsque deux nœuds l'ont validé (majorité)
- **Write quorum** décrit sous la forme $W > N/2$
 - W nombre de nœuds impliqués dans l'écriture
 - N facteur de réplication

Quorum (2)

- Combien de nœuds pour lire l'information **la plus à jour**?

Dépend du nombre de nœuds impliqués dans l'écriture

- Données **répliquées sur trois nœuds**

- Avec $W = 2$, il suffit de lire deux nœuds
- Avec $W = 1$, il faut lire les trois nœuds

- **Read quorum** décrit sous la forme $R + W > N$

- R nombre de nœuds à contacter pour une lecture
- W nombre de nœuds impliqués dans l'écriture
- N facteur de réplication

Version stamp (1)

- Critiques envers le NoSQL pour l'**absence de transactions**

Outil important pour aider à garantir la consistance

- Généralement, toute **mise à jour** pas possible par transaction

Ne peut pas rester ouverte trop longtemps

- Utilisation de **version stamps**

Pratique surtout lorsqu'on quitte le modèle à serveur unique

Version stamp (2)

- Distinction entre transaction **business et applicative**

Point de vue utilisateur ou moteur de gestion de DB

- Utilisation de la **concurrence hors-ligne**

- Démarrage transaction à la fin de l'interaction avec utilisateur
- Mise à jour conditionnelle avec relecture des informations
- Vérification des version stamps des données

- Opération de type **Compare-and-Set (CAS)**

Comparaison des version stamps avant mise à jour effective

Version stamp (2)

- Plusieurs façons possibles d'avoir un **version stamp**
 - **Compteur** à incrémenter à chaque mise à jour
 - Génération du compteur stocké par un unique master*
 - Utilisation d'un **GUID** (Globally Unique Identifier)
 - Très large et pas possible de comparer pour trouver plus récent*
 - Calculer un **hash** des données
 - Déterministe, mais large et pas possible de trouver plus récent*
 - Utilisation d'un **timestamp**
 - Comparaison possible, mais nécessite de synchroniser les horloges*
- Création d'un **version stamp composite**
 - Par exemple, CouchDB combine compteur et hash du contenu*

Vector stamp

- Version stamps faciles lorsque **unique serveur ou master-slave**
Contrôle par le master et les slaves suivent
- Il faut pouvoir **détecter** les différentes versions des données
Utilisation d'un compteur permet de connaitre la plus à jour
- **Vector stamp** est un ensemble de compteurs
 - Un compteur par nœud qu'il y a dans le cluster
 - Comparaison des vecteurs pour décider synchronisation
 - Conflit write-write si pas possible d'ordonner deux vecteurs

■ Consistance

- Applicable uniquement pour les opérations sur une seule clé
- Write optimiste très cher à implémenter
- Consistance éventuelle pour les stores distribués (Riak)

■ Transaction

- Généralement pas de garanties sur les écritures
- Utilisation de quorum en demandant W (Riak)

■ Mise à l'échelle

- Généralement grâce au sharding sur la valeur de la clé
- Configuration des paramètres N et R (Riak)

■ Consistance

- Passage par memory log puis memtable pour succès d'écriture enfin stockage permanent dans SSTable (Cassandra)
- Configuration du niveau de consistance (ONE, QUORUM, ALL)

■ Transaction

- Write atomique sur les rangée (écriture d'une colonne)
- Utilisation d'outils externes pour transaction (ZooKeeper)

■ Mise à l'échelle

- Ajout dynamique de nouveau nœuds

■ Consistance

- Généralement pas de support de la distribution des nœuds
- Données toujours consistantes au sein d'un même serveur

■ Transaction

- Pas de relations branlantes : il faut nœuds et arêtes
- Possède souvent les propriétés ACID (Neo4J)

■ Mise à l'échelle

- Sharding très difficile à réaliser (sauf selon le domaine)
- Ajout d'esclaves accessibles en lecture seule (write-once, read)

■ Consistance

- Utilisation de l'ensemble de réplicats (écriture sur slaves)
- Autoriser la lecture sur un slave (MongoDB)
- Possibilité de demander un strong write (MongoDB)

■ Transaction

- Transaction atomique sur les documents
- Parfois support pour transactions multi-documents (RavenDB)
- Attente du write sur plusieurs nœuds avant succès (MongoDB)

■ Mise à l'échelle

- Ajout de slaves pour la lecture et sharding pour l'écriture

Crédits

- <https://www.flickr.com/photos/garrettc/2302193802>
- <https://openclipart.org/detail/163711/database-server>
- <https://openclipart.org/detail/177854/person-icon>
- <https://www.flickr.com/photos/amercer/5920432681>
- <https://www.flickr.com/photos/elwillo/33906854076>