Stockage et Transactions à large échelle

Hubert Naacke

Décembre 2017

Intro : Transaction

- Transaction SQL
  - Séquence d'instructions SQL manipulant des données
    - insert, select, update, delete
  - Propriétés des transactions
    - **A**tomicité d'une séquence d'opérations
    - **C**ohérence : données intègres
    - **I**solation : chaque utilisateur est isolé des autres
    - **D**urabilité : ne jamais perdre des données
- Application transactionnelle
  - Traitement de transactions en ligne
  - OnLine Transaction Processing (OLTP)
  - Les transactions à traiter ne sont pas connues à l'avance

    ≠ Traitement offline d'un lot de transactions prédéfinies

Intro : Large échelle

- Dynamique
  - De + en + d'utilisateurs
  - Possible fluctuation du nombre d'utilisateurs : ↗ ou ↘
- Volume des données
  - Augmente avec les nouveaux utilisateurs, et les nouvelles fonctionnalités
- Intensité du workload
  - Augmente avec le nombre d'utilisateurs
- Besoins
  - Performance élevée: 1K à 1M de transactions par minute (tpm)
  - Ceci à faible coût matériel et logiciel : rapport perf/prix élevé.

Problème

- Un SGBD centralisé n'est pas assez performant
  - Ne gère pas une quantité ↗ de données et de transactions
- Utiliser les ressources de plusieurs machines ?
  - Infrastructure répartie et redondante
- Traiter des transactions avec un système réparti et redondant est complexe :
  - Propriétés ACID pour des transactions réparties ?
  - Cohérence des répliques, i.e. répliques toujours identiques ?

→ Problème de performance

- Quelles propriétés garantir en priorité?
- Pour quel type de traitement ?

Approche

5

- **Compromis** : obtenir plus de performance en limitant les fonctionnalités supportées
- Priorité à la fonctionnalité la plus importante
  - Disponibilité à large échelle
- Nécessite la conception de **nouveaux** systèmes de stockage et gestion de données, différents des SGBD.
  - Meilleure disponibilité
  - Support limité des transactions
  - Répliques pas toujours identiques

→ Scalable datastore

Scalable datastore

6

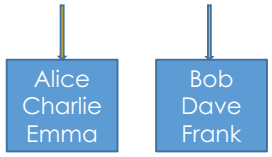
- Datastore
  - Système de stockage et de gestion de données
  - Modèles de Données
    - Données structurées (relationnelles)
    - ou Semi-structurées
- Scalable
  - Distribué sur plusieurs machines
  - Conçu pour facilement passer à l'échelle : scale-out

→la scalabilité permet une meilleure disponibilité des données

Scalabilité : Exemple de scale-out

7

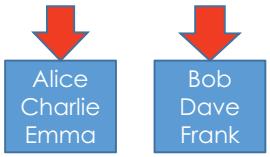
Répartition initiale des données sur 2 machines



Scale-out: transactions trop nombreuses

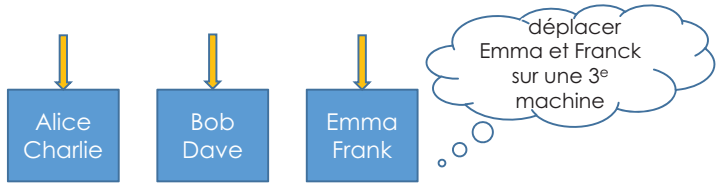
8

Problème : transactions trop lentes car demande > capacité de traitement



Scale-out: répartir les transactions

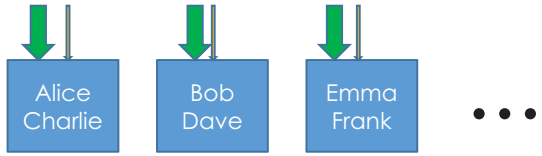
Ajouter des machines "horizontalement"



autre avantage: permet de gérer plus de données

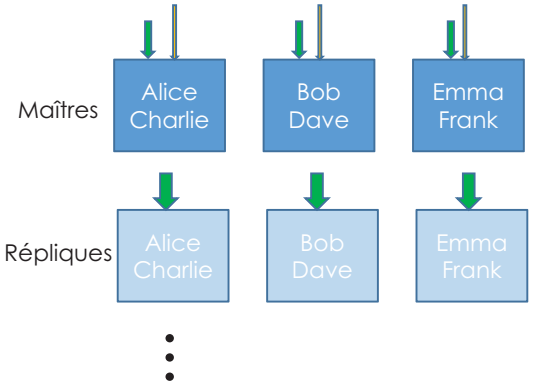
Scale-out : lectures trop nombreuses

Pb: requête trop lentes



Scale-out : répartir les lectures

Ajouter des machines "verticalement"

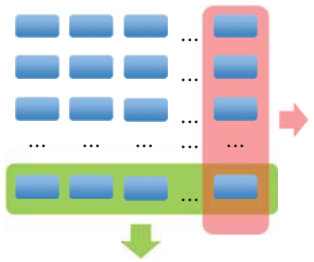


Cohérence des répliques
Solution mono maitre :
un seul maître par donnée

2e avantage :
tolère mieux les pannes

Défis de la scalabilité

- Maintenir le système disponible et/ou cohérent
  - Quand la quantité de données augmente
  - Quand le nombre de transactions augmente
- Distribuer les données
  - Ajouter/supprimer une partition
  - Scale-out horizontal
- Répliquer les données
  - Ajouter/supprimer une réplique
  - Scale-out vertical



Disponibilité d'un datastore

La disponibilité concerne deux aspects :

- Disponibilité en cas de **panne**
  - Pas d'interruption du service s'il manque des machines
- Disponibilité en fonctionnement **normal**
  - Réponse rapide, donc latence faible

Disponibilité en cas de panne Théorème CAP

- Fondement des systèmes distribués : le théorème CAP
  - Considère une donnée répliquée sur plusieurs machines
- Formalise le comportement du système (C ou A) en cas de panne (P)
  - P : network **P**artition = plusieurs machines sont injoignables
  - C : **C**onsistency = Cohérence des répliques
  - A : **A**vailability = disponibilité = réponse immédiate
  - Comportement PC : toujours garder les répliques cohérentes
    - Attendre qu'une majorité de machines soient accessibles
    - Risque de voir l'utilisateur s'impatienter
  - Comportement PA : toujours répondre rapidement
    - Lit/modifie une réplique sans connaitre l'état des autres répliques
    - Divergence possible : réconcilier les répliques divergentes

Disponibilité en fonctionnement normal

- Spécifier le compromis latence / cohérence
  - Selon la nature des opérations : écriture ou lecture
  - Selon la nature des données manipulées
- Quel compromis pour quel cas d'usage ?
- La latence et la cohérence dépendent du nombre de machines contactées
  - Rapide mais pas cohérent
    - Répondre le plus vite possible en contactant une seule machine
  - Moins rapide mais plus cohérent
    - Répondre moins vite après avoir contacté plus de machines

SGBD parallèle / NOSQL / NewSQL

Quelles sont les capacités transactionnelles des datastore ?
Classification en trois catégories :

- SGBD parallèles
  - Support complet des transactions ACID
- NOSQL (exple KVStore)
  - Pas de transactions, mais :
  - Seulement des opérations atomiques sur un seul granule de donnée.
- NewSQL (exple Calvin)
  - Support limité de certaines transactions paramétrées
    - Paramètres fixés avant le début de la transaction
  - Une transaction doit être courte (durée inférieure à un seuil fixé)

SGBD parallèle

17

- Données relationnelles (tables SQL)
- Supporte les transactions ACID, sans restriction
  - Entièrement conformes au standard SQL
- Nécessite du matériel dédié très coûteux
  - réseau, gros serveur (100+ cpu, TOs de RAM partagée), licence
- Disponibilité limitée
  - Protocoles de gestion complexes : latence accrue
    - Ex: l'isolation repose sur des techniques de verrouillage
  - Arrêt du service pendant le scale-out horizontal
- Exemple: Oracle RAC , Microsoft SQL Server

Etude de systèmes existants

18

- NOSQL
  - **KVStore**, Hbase, **DynamoDB**
- NewSQL
  - **Calvin**, VoltDB

KVStore

19

- Solution NOSQL proposée par Oracle
  - Open source, issue de Berkeley DB
- Données non relationnelles
  - semi-structurées
- Accès par clé
  - Pas de langage de requêtes déclaratif
  - Moins expressif que SQL

KVStore : plan

20

- Plan des diapos suivantes :
  - Stockage réparti
  - Modèle clé-valeur
    - lecture, écriture
  - Modèle table
    - lecture, écriture
  - Réplication et durabilité

Stockage réparti

- Principe
  - Stocker une quantité **illimitée** de données
  - Structure élémentaire simple : chaque valeur a une clé
    - Permet une gestion simplifiée visant à être plus performante
- Exple :

Livre

titre	prix	année
A	20	2018
B	10	1948
...	...	...

Jeu

nom	prix
B	30
C	10
...	...

  - /livre/A/prix → 20 /livre/A/année → 2018 /jeu/C/prix → 10

Stockage réparti

- Stockage réparti : partitionnement par hachage
  - h(donnée) = numéro de partition
  - Dictionnaire (numéro de partition → machine)
  - Exple avec 3 machines
    - Dictionnaire[1 à 5 → M1, 6 à 10 → M2, 11 à 15 sur M3]
    - h("/livre/A") = 4 sur M1 h("livre/B") = 13 sur M3
    - h("jeu/B") = 7 sur M2 h("jeu/C") = 2 sur M1

M1
/livre/A, 20
/jeu/C, 10

M2
/jeu/B, 30

M3
/livre/B, 10

Stockage réparti : dictionnaire

- N partitions
- Dictionnaire : vecteur de taille N. Une coordonnée par partition
  - D[i] = j signifie que la partition Pi est sur la machine Mj
  - N > nbre de machines pour faciliter la réorganisation du stockage
- Exemple pour 15 partitions sur 3 machines :

D

D[1]	1
	1
	1
	1
	1
	2
	2
	2
	2
	2
	3
	3
	3
	3
D[15]	3

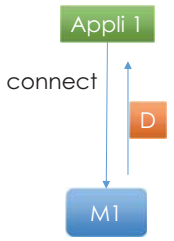
Partition, Machine

P_1	M_1
P_2	M_1
...	...
P_5	M_1
P_6	M_2
...	...
P_{10}	M_2
P_{11}	M_3
...	...
P_{15}	M_3

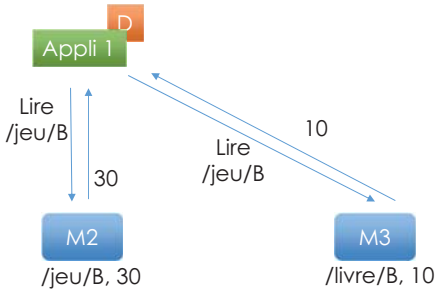
Stockage réparti : dictionnaire

- D est "petit". D est connu par les clients qui accèdent au store.

Etape 1 :
Connection

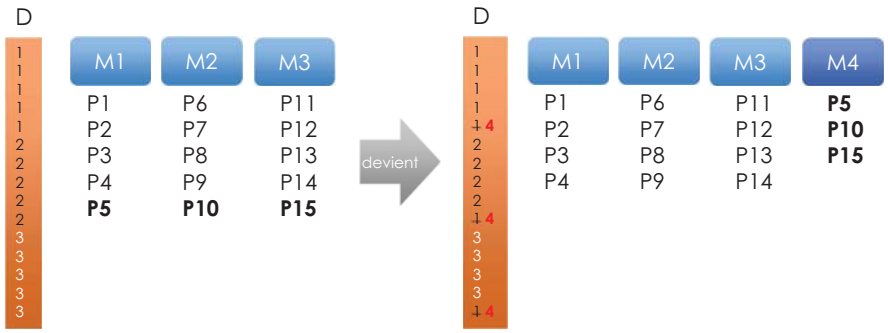


Etape 2 :
Accès direct aux machines



Disponibilité du stockage

- Volume croissant des données → ajout d'une machine
  - Déplace certaines partitions vers la nouvelle machine
  - Maintenir un stockage équilibré entre les machines
  - D est modifié et retransmis au client *à la demande*



Disponibilité du stockage

- Volume décroissant des données → retrait d'une machine
- Stockage "élastique"
  - augmenter/réduire automatiquement la capacité de stockage
- Mais: une partition n'est pas accessible pendant son déplacement
  - inconvénient moindre si déplacement rapide
- Amélioration du stockage élastique ?
  - Déplacer une partition sans interrompre l'accès
  - Cf Elastras (TODS 2013)

Kvstore : Modèles de données

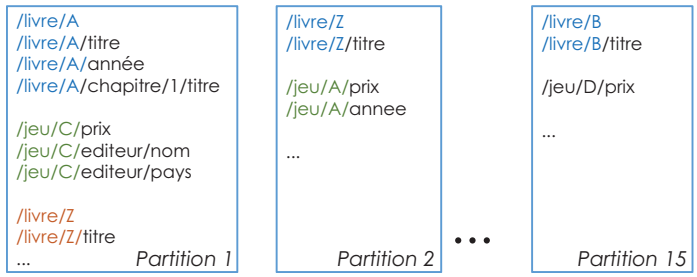
- KvStore propose 3 modèles de données
  - **Modèle clé-valeur** ou Key-Value ou KV
  - **Modèle table**
- Ces modèles sont implémentés dans 2 langages possible
  - Java API pour les modèles KV et table
  - "Dialecte" SQL simplifié pour le modèle table

Modèle KV clé-valeur

- Modèle KeyValue : paire (clé → valeur)
  - Clé formée d'une liste de termes
    - /livre/A/
  - Structure **hiérarchique** : des clés peuvent avoir le même préfixe
    - /livre/A /livre/A/titre /livre/A/année /livre/A/chapitre/1/titre
  - Clé structurée en 2 composantes (majeure, mineure)
    - Préfixe = Composante **majeure**
    - Suffixe = Composante mineure
    - Syntaxe : /livre/A/ - /chapitre/1/titre
- Version d'une donnée
  - Donnée immutable, i.e., non modifiable
  - Modifier une donnée = ajouter une nouvelle version de la donnée
  - Donnée: (Clé hiérarchique → (valeur, version))

KV : Localité des données

- La composante majeure détermine la partition
  - h(composante majeure) = numéro de partition
  - Les paires partageant la même composante majeure sont stockées dans la même partition
- Avantage: traitement **local** (centralisé) de toutes les clés ayant la même composante majeure



KV : Opérations de lecture

- Lecture ciblée d'une paire
  - (valeur, version) = get(clé)
- Lecture de plusieurs paires
  - Liste de (valeur, version) = multiget(K, intervalle, profond)
  - L'accès doit être local: toutes les paires doivent être dans la même partition
    - K doit avoir pour préfixe une composante majeure
  - L'intervalle et la profondeur spécifient les "sous-clés" à lire
- Les opérations get et multiget sont atomiques

KV : Opérations d'écriture

- Ecriture d'une paire (clé, valeur)
  - version = put(clé, valeur)
- Ecriture conditionnelle après lecture
  - Garantit que la séquence *Lit(A), Ecrit(A)* est atomique.
  - v' = putIfVersion(clé, valeur, v)
    - v: version avant l'écriture, v' version après l'écriture
- Ecriture atomique de plusieurs paires
  - Toutes les paires doivent avoir la même composante majeure
  - Liste d'opérations d'écriture
    - List<Operation>
    - Définir une opération : OperationFactory.createPut(.....)
  - Traitement atomique : execute(list)

Gestion des répliques

- Gestion flexible de la réplication
  - Plusieurs protocoles possibles pour gérer
    - Ecriture : la propagation vers les répliques
    - Lecture : la réplique à lire

→ Choisir le protocole le plus adapté aux exigences de l'application

Ecritures flexibles

33

Le protocole d'écriture est spécifié par :

- Le **mode de propagation** vers les répliques
  - Exprime le compromis : latence / cohérence

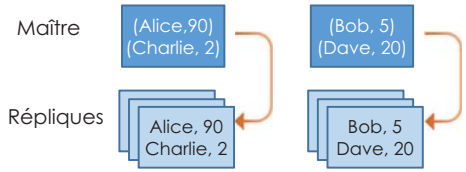
ET

- Le **niveau de durabilité** des écritures
  - le niveau dépend des pannes tolérées
  - Exprime le compromis : latence / durabilité

Réplication avec propagation synchrone ou asynchrone

34

- Donnée de référence
  - Une donnée a un seul maître
  - Les maîtres sont répartis entre plusieurs machines
- Propagation des écritures vers les répliques
  - découplée ou non de l'écriture sur la machine maître
  - Ecart possible entre les répliques
    - Si on lit une version antérieure d'une donnée
- Exemple avec 8 machines dont 2 maîtres :



Propagation vers les répliques

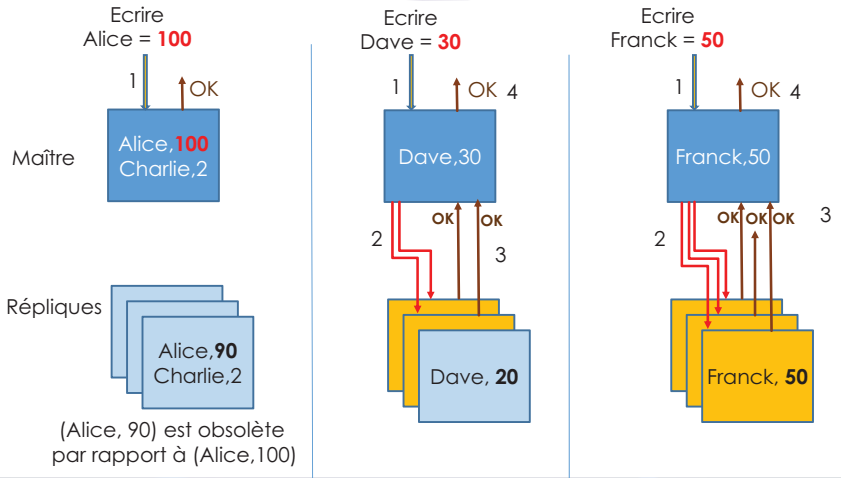
35

Trois modes de propagation des écritures :

- Le maître ne propage pas
  - Il valide l'écriture localement puis répond à l'application
- Propage vers la majorité des répliques (n/2 + 1)
  - Attendre qu'une majorité de répliques valident avant de répondre à l'application
- Propage vers toutes les répliques
  - Attendre que toutes les répliques valident avant de répondre à l'application

Les 3 modes de propagation

36



Niveaux de durabilité des écritures

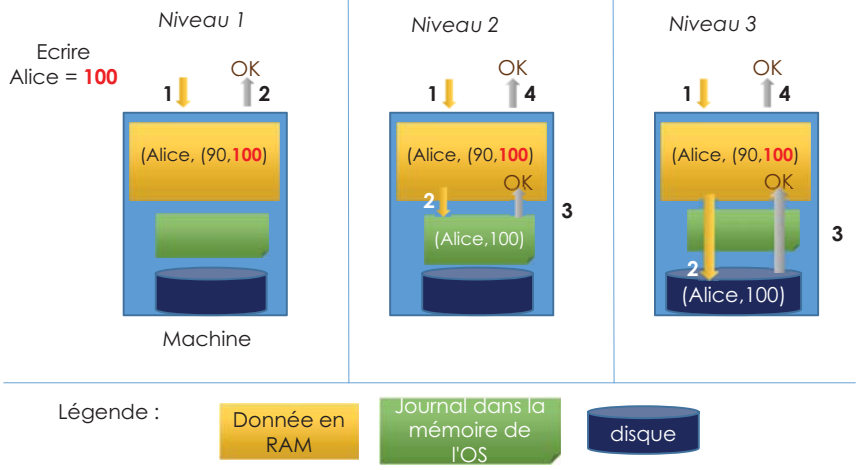
- Niveaux croissants de durabilité
  - Sur quel support écrire lors du commit ?
- Une écriture est durable si on dispose du journal pour restaurer la base
- Opération E = écrire 1 liste de paires
  - Compléter la base avec la nouvelle version des paires
  - Compléter le journal : ajout séquentiel (append)

Niveau 1 : Ne pas compléter le journal. Durabilité faible
- Ajouter la nouvelle version des paires dans la base en mémoire
  - Modifier la Map<K, (V, version)> en RAM
- Perte possible de E sauf si d'autres répliques ont traité E également

Niveau 2 : Compléter le journal. Durabilité moyenne
- Niveau 1 + écrire dans le tampon (buffered write) associé au fichier du journal
- Tolère une panne logicielle du store mais pas de l'OS

Niveau 3 : Ecriture durable. Durabilité forte
- Niveau 1 + Niveau 2 + forcer à écrire le journal sur disque (fsync)
- Tolère une panne de l'OS (reboot)

3 niveaux de durabilité



Syntaxe d'écriture flexible

- On peut préciser le protocole pour chaque écriture
  - put(K, V, protocole)
- Le protocole est spécifié par
  - Le mode de propagation
  - Le niveau de durabilité du maître
  - Le niveau de durabilité des répliques
- Syntaxe pour le mode de propagation
  - NONE
  - SIMPLE_MAJORITY
  - ALL
- Syntaxe pour le niveau de durabilité
  - NO_SYNC
  - WRITE_NO_SYNC
  - SYNC

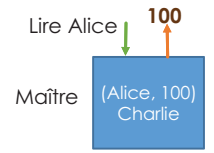
Lectures +/- cohérentes dans KVStore

- Lecture d'une donnée répliquée
- Possibilité de contrôler la cohérence entre les répliques
- Contrôle flexible de la cohérence :
  - plusieurs niveaux de cohérence au choix

Lecture cohérente

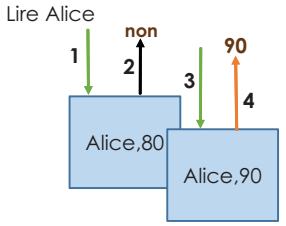
- Trois niveaux de cohérence:
  - Stricte
    - lire la dernière version: ABSOLUTE
  - Bornée
    - Lire une donnée dont la version >= N
      - Permet de garantir la cohérence "read your writes"
    - Lire une donnée sur une machine dont le retard < A
      - Retard = date courante – date dernière mise à jour
  - Relâchée
    - Quelconque : Lire n'importe quelle réplique ou le master: NONE_REQUIRED
    - Faible : Lire n'importe quelle réplique sauf le master: NONE_REQUIRED_NO_MASTER
- Durée tolérée
  - Permet d'attendre qu'une version satisfaisante soit générée
  - V = get(clé, niveau, timeout)

Les niveaux de cohérence des lectures



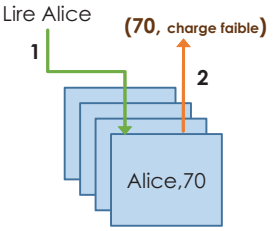
(pas d'accès au maître)

Plusieurs tentatives

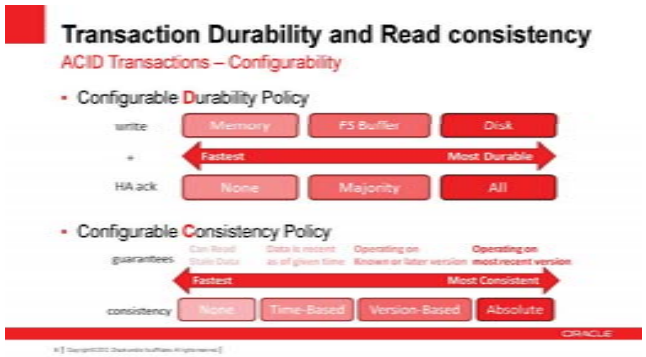


(pas d'accès au maître)

Lire la réplique la moins chargée
(charge "transmise" dans la réposne)



Résumé des écritures et lectures flexibles dans KVStore



Calvin

- Calvin

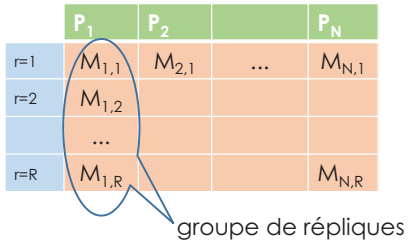
Calvin : principes

- Données SQL
- Transactions SQL courtes
  - durée bornée: nécessaire pour ordonner les transactions
- Système déterministe
  - Détermine l'ordre global des transactions **avant** de les traiter.
  - Traite les transactions dans l'ordre préalablement déterminé
- Nouveauté
  - Supporte les transaction globales (**multi-partitions**) sur des données réparties
    - Traitée par plusieurs transactions **locales** indépendantes
    - Sans nécessiter de validation globale

Architecture de Calvin

- Données partitionnées et répliquées sur un cluster de machines
  - N partitions: P_1 , ... P_i , ..., P_N
  - R répliques par partition

  → donc N*R machines
    - dénotées $M_{1,1}$ à $M_{N,R}$



Architecture de Calvin

Exemple avec 3 partitions (N=3) et un degré de réplication = 3

Cluster de 9 machines Mi,j

Coordination des transactions

- **Ordonner** les transactions en 3 étapes décentralisées
  1. Laisser des transaction arriver sur des machines
  2. Former des groupes de transactions (un groupe par partition)
  3. Compléter les groupes pour tenir compte des transactions **multi-partitions**
- Puis traitement décentralisé des transactions
- Pour **réaliser** cette coordination chaque machine $M_{i,j}$ a
  - un **séquenceur** : ordonner les transactions
  - un **scheduler :** traiter les transactions

Ordonner les transactions

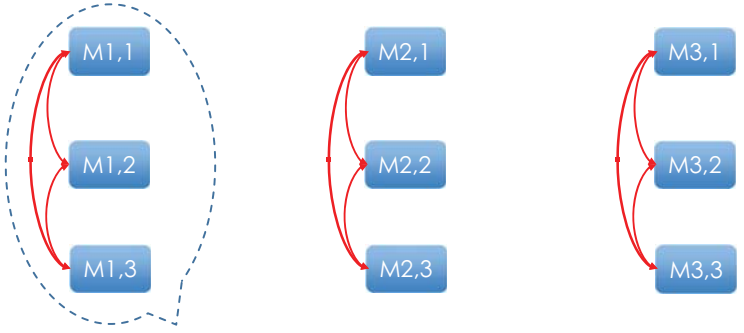
-

Séquenceur

- Découpe le temps en fenêtres de période fixe
- Reçoit des transactions
- En fin de période courante, **propage** les demandes aux autres séquenceurs du même groupe de répliques
  - Ainsi chaque séquenceur d'un groupe connait la liste des demandes du groupe
  - Avantages :
    - Plusieurs points d'entrée dans un groupe : disponibilité
    - Propagation interne à un groupe : plus rapide qu'une propagation globale entre toutes les machines.

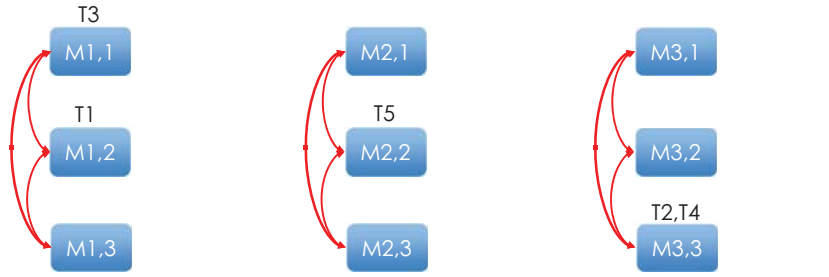
Séquenceur: exemple

- Séquenceur : coordination "verticale"



Séquenceur : exemple 1

- Les transactions T1 à T5 arrivent pendant la période courante
- Puis coordination verticale en fin de période



T3 → M1,2 et M1,3
T1 → M1,1 et M1,3

T5 → M2,1 et M2,3

(T2,T4) → M3,1 et M3,2

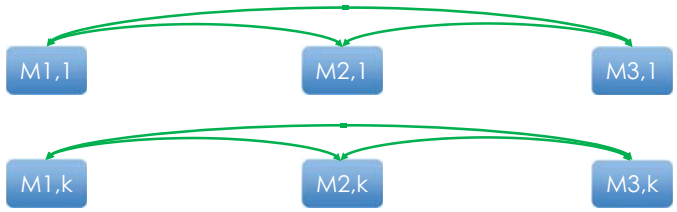
Séquenceur : exemple 1 (suite)

Transmission d'un séquenceur vers les schedulers

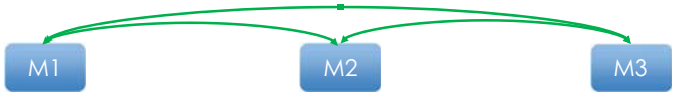
- Un séquenceur communique seulement avec les schedulers ayant le même indice r
  - Communication horizontale sur une même "ligne"
  - Du séquenceur $M_{i,r}$ vers les schedulers parmi $M_{1,r}$ à $M_{n,r}$
- Transmet les transactions aux seuls schedulers concernés
  - Chaque transaction est transmise sur chaque machine stockant au moins une donnée de la transaction.
- La réplication est "orthogonale"
  - Comportement identique sur chaque "ligne" de machines

Séquenceur → Scheduler

- Coordination "horizontale"



- Représentation simplifiée (quel que soit k)

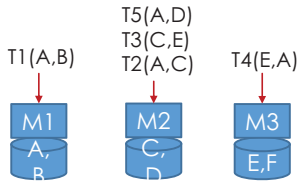


Ordre global des transactions

- L'ordre global des transactions est déterminé de manière **décentralisée**
  - En fixant un ordre global entre les séquenceurs à partir du numéro de machine : M1 < M2 < ... Mi < Mj .... < M_N
  - Pour tout i < j: une transaction posée sur Mi **précède** une transaction posée sur Mj
  - Si i = j, alors Ta précède Tb si a<b

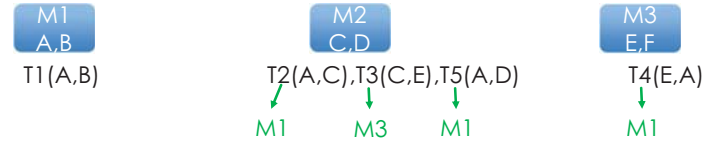
Séquenceur → Scheduler Exemple 2

- Exemple pour les machines M1 à M3
- Les données A à F sont réparties :
  - M_1 (A,B) M_2(C,D) M_3(E,F)
- T1 manipule (A,B) T2(A,C) T3(C,E) T4(E,A) T5(A,D)
- Les transactions arrivent sur :
  - M_1 : T1(A,B)
  - M_2 : T2(A,C), T3(C,E), T5(A,D)
  - M_3 : T4(E,A)

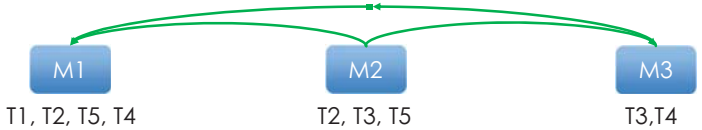


Séquenceur → Scheduler Exemple 2 (suite)

- Avant transmission

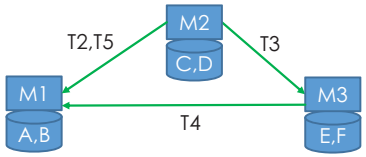


- **Après** transmission



Séquenceur → Scheduler Exemple 2 (suite)

- Transmission
  - M_1 : T1 reste locale pas d'envoi
  - M_2 : T2 et T5 vers M1, T3 vers M3
  - M_3 : T4 vers M1
- Les schedulers ordonnent seulement les transactions qu'ils reçoivent
  - M_1 : T1, T2, T5, T4
  - M_2 : T2, T3, T5
  - M_3 : T3, T4



Traiter les transactions

Scheduler : Traitement des transactions

- Planifie le traitement local des transactions
  - Ordonne les transactions reçues
- Détermine les données qu'une transaction lit ou écrit
  - Données lues = **R**ead set (R)
  - Données écrites = **W**rite set (W)
- Traitement local d'une transaction globale
  - Lit les données (∈ R) et les **envoie** aux machines concernées
  - **Reçoit** les données (∈ R) venant des autres machines
  - Traitement local
    - si la machine contient des données à écrire (∈ W)

Scheduler : Exemple

- Transactions : données lues (R) et écrites (W) pour T1(A,B) T2(A,C) T3(C,E) T4(E,A) T5(A,D) :
  - T1 : R = A, B W = A, B
  - T2 : R = A W = C
  - T3 : R = C, E W = E
  - T4 : R = E W = E, A
  - T5 : R = A, D W = A, D
- M1(A,B) : (T1, T2, T5, T4)
  - traiter T1
  - T2 : envoyer A vers M2
  - T5 : (A déjà envoyé à M2), recevoir D de M2, traiter T5
  - T4 : recevoir E de M3, traiter T4
- M2(C,D) : (T2, T3, T5)
  - T2: recevoir A de M1, traiter T2
  - T3: envoyer C vers M3
  - T5: envoyer D vers M1, recevoir A de M1, traiter T5
- M3(E,F) : (T3, T4)
  - T3: recevoir C de M2, traiter T3
  - Envoyer E vers M1, traiter T4

Scheduler : bilan

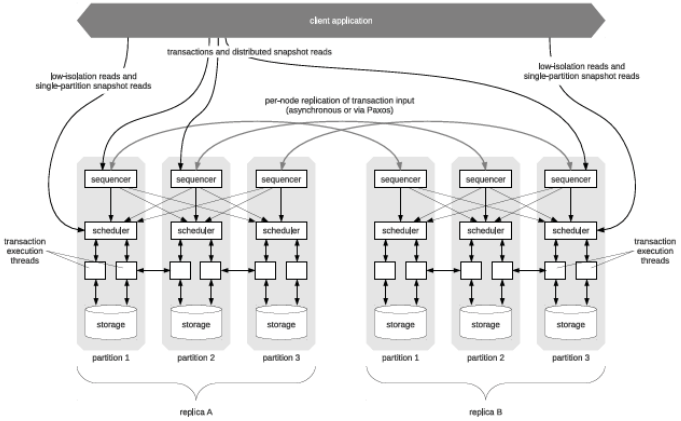
- Ne pas traiter une transaction dont les données à écrire ne sont pas locales (ex: T2 sur M1)
- Ne pas envoyer une donnée à un site qui ne fait que lire les données de la transaction sans la traiter
  - Ne pas envoyer C à M1 car M1 ne traite pas T2
- Ne pas renvoyer plusieurs fois une donnée lue si elle n'a pas été modifiée entre temps (ex: A sur M1)
- Efficacité
  - Dépend du nombre de transactions multi-partitions
  - Dépend de la taille des données lues par les transactions multi-partitions

Ref bibliographique

- Calvin
  - Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems
  - http://cs-www.cs.yale.edu/homes/dna/papers/calvin-tods14.pdf
  - http://cs-www.cs.yale.edu/homes/dna/papers/calvin-sigmod12.pdf
    - Lire la section: Scheduler and concurrency control

Ref bibliographique

Figure extraite de l'article Sigmod 2012
Scalable transaction layer over shared nothing storage system



Conclusion

- Grande variété de solutions scalable datastore spécifiques selon
  - La nature des données
  - La complexité des traitements
  - L'étendue géographique des utilisateurs
  - Le niveau de disponibilité souhaité
- Tendance : solution orientée service (cloud) pour le stockage et la gestion des données à l'échelle du web