# NoSQL avec KVStore

Administration des bases de données réparties

**NI401 - ABDR**

**Hubert Naacke**

6 janvier 2014

Robin Keunen
3303515
Clément Barbier
3061254

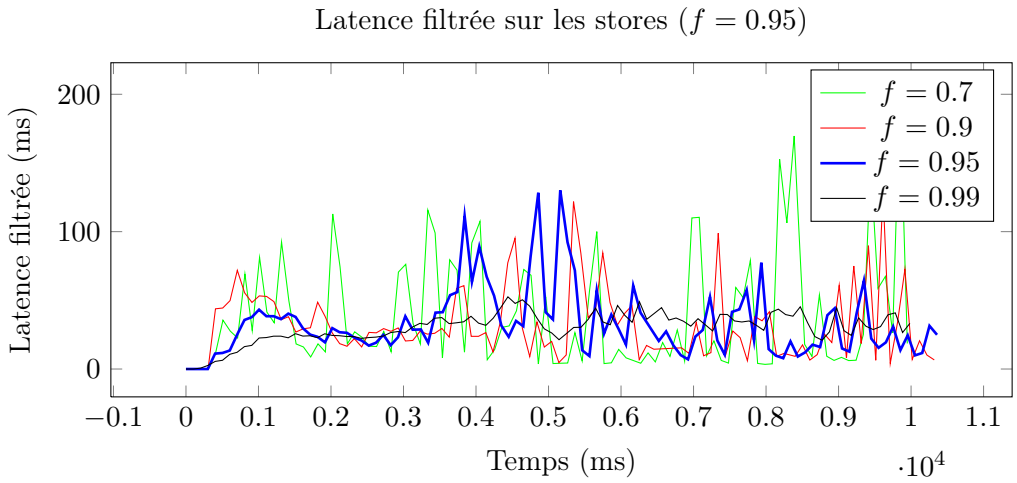


FIGURE 1 – Latence filtrée sur les stores 1, 3 et 5. Les stores 1 à 4 subissent une charge de 15 applications. Le store 1 subit la charge additionnelle de 5 applications. Pas de balance de la charge.

# 1 Transaction et concurrence

Cette section présente la solution du tme KVStore. Le code se trouve dans le projet kvstore dans les packages `tme1.ex1` et `tme1.ex2`.

## Exercice 1

La classe `init` permet d'initialiser le store afin d'obtenir des résultats reproductibles.

**A1** Dans cet exercice, nous ne devions pas tenir compte des accès concurrents au données. Le programme lit simplement la valeur stockée à la clé $P1$, incrémente ce qu'il a lu et l'écrit à la clé $P1$. Cette solution n'est plus viable dès que deux programmes manipulent la même donnée simultanément. En effet, si les programmes $a$ et $b$ lisent $n$ simultanément, ils écriront chacun $n + 1$ en base alors que la valeur aurait du être incrémentée deux fois. La valeur finale devrait être $n + 2$.

Ce résultat est montré par l'expérience : en lançant deux programmes `A1` qui lisent et incrémentent 1000 fois la valeur stockée à P1, on obtient une valeur finale en $P1$ de 1261, 1269 et 1289 (3 exécutions consécutives) au lieu de 2000.

**A2** Dans `A2`, le programme vérifie si la valeur stockée en base est fraiche avant d'écrire. On vérifie la fraicheur grace à la fonction `putIfVersion`. Cette fonction n'écrit en base que si la version de la donnée en base correspond à la version de la donnée que nous y avions lue. Si la version est périmée, le programme relit la valeur et tente à nouveau de l'incrémenter et de la réécrire.

Cette version retourne les résultats attendus : deux programmes `A2` exécutés simultanément incrémentent effectivement 2000 fois la valeur de $P1$.

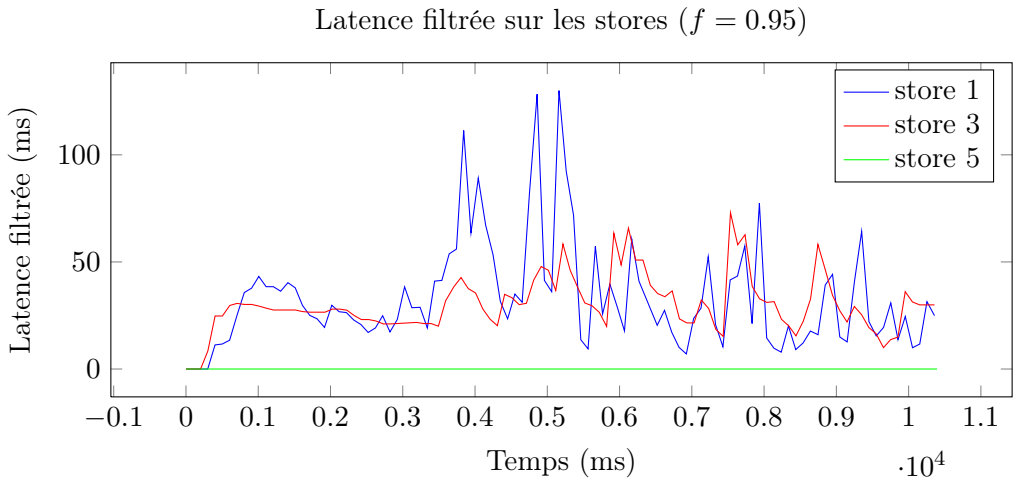


FIGURE 2 – Latence filtrée sur les stores 1, 3 et 5. Les stores 1 à 4 subissent une charge de 10 applications. Le store 1 subit la charge additionnelle de 10 applications. Pas de balance de la charge.

### Exercice 2

**M1** Ce programme est identique à `A2`, il lit les valeurs de $P0$ à $P4$ en base, les incrémente et les écrit en base. Si on vérifie la fraicheur des données avant de réécrire, il n'y aura pas de problème de cohérence de données. En effet, vu que les données écrites (de $P0$ à $P4$) sont indépendantes les unes des autres, l'ordre d'écriture de deux programmes n'affecte pas la valeur des données finales.

**M2** En exécutant deux programmes `M2` en série, on obtient une valeur finale de 2000. En exécutant deux programmes `M2` en parallèle, les résultats divergent, on obtient par exemple lors de 3 expériences consécutives : 2054, 2056 et 2044.

Cette erreur est provoquée par la non-atomicité des écritures en base. L'écriture de $max+1$ sur P0..5 peut être interrompue par l'écriture d'un programme concurrent. Quand l'écriture est interrompue en $P3$ par exemple, la valeur $max + 1$ a déjà été écrite en $P0$, $P1$ et $P2$. Le programme recommence cette itération (lecture et écriture des 5 produits). Au final, les valeurs auront été incrémentées deux fois lors de cette itération. Voir la table 2.5 en annexe pour un exemple d'exécution.

**M3** Pour que les résultats soient cohérents, il faut que les opérations de mises à jours soient atomiques. Les transactions atomiques sont implémentées dans la classe `Transaction`. Les opérations ne peuvent être exécutées de façon atomiques que si les clés ciblées partagent la même clé majeur. C'est le cas ici, nous ne manipulons que des éléments de la même catégorie. La clé des produits est composée d'une partie majeure correspondant à la catégorie et d'une clé mineure correspondant au produit.

## 2 Equilibrage de charges sur plusieurs KVStores

### 2.1 Structure de notre solution

Nous avons essayé de rajouter une couche d'abstraction au dessus des KVStores : les applications clientes s'adressent à un élément unique, le maitre (*Singleton*) : le

`StoreMaster`. Le maitre redistribue la charge aux différents stores. Puisque l'objet est unique et centralisé, il est possible que les opérations subissent un effet de goulot d'étranglement. Cependant, un maitre décentralisé aurait été trop long à implémenter pour ce projet.

Cette section détaille les éléments logiciels de notre solution.

**project**

**Item**
`Item` modélise l'objet à ajouter à la base. Il contient cinq champs numériques et cinq champs `String`.

**ServerParameters**
`ServerParameters` encapsule les paramètres nécessaires pour accéder à un KVStore (nom, IP et port).

**ConfigsServer**
`ConfigsServer` mémorise les paramètres correspondant aux serveurs lancés par le script fourni avec le projet.

**project.master**

**StoreMaster**
`StoreMaster` est l'interface fournie aux applications clientes. Il maintient une liste de `StoreController`s (un par KVStore concret). Ces `StoreController`s fournissent une interface aux KVStore. `StoreMaster` délègue les opérations du client à un des contrôleurs. La délégation est faite par un `StoreDispatcher` au moyen du numéro du profil manipulé.

**MissingConfigurationException**
Avant d'être utilisé, le `StoreMaster`doit être configuré. La configuration consiste à lui passer une liste d'instances de `oracle.kv.KVStore`. Si cette configuration n'a pas été faite avant la instantiation, cette exception est levée.

**StoreSupervisor**
`StoreSupervisor` est un Runnable lancé au moment de l'instantiation du maitre. Cet objet consulte les statistiques de chaque contrôleur et déclenche un déplacement de profil pour équilibrer la charge.

**project.master.dispatchers**

**StoreDispatcher**
Les `StoreDispatcher` servent à attribuer un contrôleur à chaque profil. La fonction *getStoreIndexForKey* retourne un entier qui servira d'index dans la liste de contrôleurs du `StoreMaster`.

**SingleStoreDispatcher**
Utilisé dans l'étape 1 de la résolution. Retourne 0 quelque soit le profil.

**TwoStoreDispatcher**
Utilisé dans l'étape 2 de la résolution. Retourne 0 pour les profils pairs et 1 pour les profils impairs.

**MultipleStoreDispatcher**
*Dispatcher* final. Il retourne $index = (profil \mod n)$ où $n$ est le nombre de stores. `MultipleStoreDispatcher` supporte en outre la fonction *manualMap*. Cette fonction permet d'assigner manuellement un store à un

profil. Lorsque le `StoreMaster` demande un index pour un profil, `MultipleStoreDispatcher` fait un lookup dans sa table de profil. Si le profil est dans la table, il retourne l'index associé, sinon il retourne l'index selon la fonction indiquée plus haut.

**project.store**

**StoreController**
`StoreController` est un *wrapper* pour les KVStore concrets (`oracle.kv.KVStore`). Il implémente les opérations du KVStore dont nous avons besoin.

**ProfileTransaction**
`ProfileTransaction` est créé par le controleur pour créer une transaction atomique. Cette classe est instanciée pour un profil donné, le contrôlleur lui ajoute des opérations et lance l'exécution.

**StoreMonitor**
`StoreMonitor`est un `Runnable` permettant de récolter des statistiques sur les opérations. C'est aussi cette classe qui gère l'attribution des indexes aux `Item`s insérés.

**TransactionMetrics**
Implémente `oracle.kv.stats.OperationMetrics`. Cette classe sert à collecter les statistiques sur les transactions effectuées. Les statistiques sont mises à jour pour chaque transaction. Nous avons ajouté le champ *filteredLatency* dont nous parlerons plus tard.

**project.application**

**ClientApplication**
The first item

**ClientApplication**
The second item

**tests**

**First**
The first item

**Second**
The second item

**Third**
The third etc ...

## 2.2 Politique d'équilibrage

filtre exponentiel
facteur d'oubli
mean et standard deviation

## 2.3 étape 1 a

## 2.4 étape 1 b

75695802

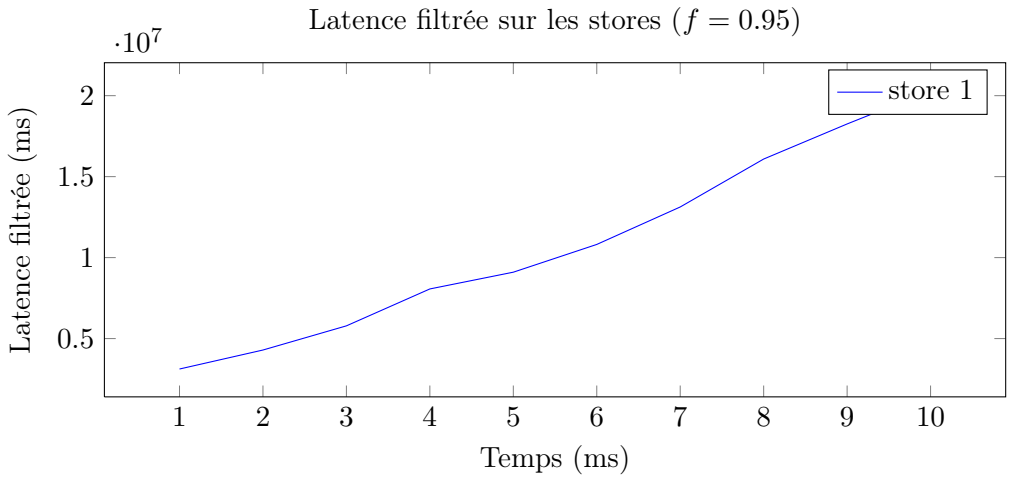


FIGURE 3 – Latence filtrée sur les stores 1, 3 et 5. Les stores 1 à 4 subissent une charge de 15 applications. Le store 1 subit la charge additionnelle de 5 applications. Pas de balance de la charge.

## 2.5 Etape 2

Le déplacement d'un profil Px d'un store Si à un store Sj n'est pas une opération atomique par définition.

Imaginons un fonction moveProfil prenant en paramêtre un store source (Si), un store cible (Sj) et l'identifiant du profil à déplacer (Px). Cette fonction est une transaction qui fait éxécute une suite d'opérations : - Lire le profil Px sur Si. - Copier le profil Px sur Sj. - Supprimer le profil Px sur Si.

On ne peut pas se permettre de vérouiller l'accès à un profil durant le déplacement (généralement d'une longue durée) car celà altérerait trop les performances.

Celà implique que le profil Px, en cours de déplacement, peut être sollicité par des applications clientes pour des lectures, modifications, ajouts d'objets au profil ce qui peut occassioner des problêmes de consistence des données.

Il faut à tout moment (même pendant le déplacement) être capable de : - Accéder à la dernière version du Profil Px et ses objets. - Modifier les profils sans que ces modifications soient perdues durant le déplacement. - Insérer de nouveaux objets dans le profil Px tout en assurant la validité et l'unicité des clés crées.

TABLE 1 – Exemple d'exécution de M2 sans le mécanisme des transactions. Le programme *a* est interrompu avant sa dernière écriture. *a* recommence l'itération lecture/écriture. Au final, les valeurs sont incrémentées deux fois lors d'une seule itération.

| programme a | programme b |
|---|---|
| read(P4) = 73 | |
| write(P0, 74) | |
| write(P1, 74) | |
| write(P2, 74) | |
| write(P3, 74) | |
| | read(P0) = 74 |
| | read(P1) = 74 |
| | read(P2) = 74 |
| | read(P3) = 74 |
| | read(P4) = 73 |
| | write(P0, 75) |
| | write(P1, 75) |
| | write(P2, 75) |
| | write(P3, 75) |
| | write(P4, 75) |
| write(P4, 74)!! | |
| read(P0..4) = 75 | |
| write(P0..4, 76) | |