ACID Insuffisant

adriencaubel.fr

Ressources

- https://vladmihalcea.com/optimistic-vs-pessimistic-locking/
- https://www.baeldung.com/jpa-optimistic-locking
- https://stackoverflow.com/questions/47441027/pessimistic-locking-vs-serializable-transaction-isolation-level

Table des matières

Rappels

Rappels

Dans le cours précédent nous nous sommes intéressé à l'Atomicité. Puis nous avons détaillé l'Isolation :

- comment garantir l'isolation au niveau physique : Two-Phase Locking et MVCC
- les différents niveaux d'isolation

Niveau d'isolation	Lecture sale *(Dirty read)*	Lecture non répétable *(Non-repeatable read)*	Fantôme *(Phantom)*	Mise à jour perdue *(Lost update)*
READ UNCOMMITTED	❌ Autorisée	❌ Autorisée	❌ Autorisée	❌ Autorisée
READ COMMITTED	✅ Empêchée	❌ Autorisée	❌ Autorisée	❌ Autorisée
REPEATABLE READ	✅ Empêchée	✅ Empêchée	❌ Autorisée	✅ Empêchée
SERIALIZABLE	✅ Empêchée	✅ Empêchée	✅ Empêchée	✅ Empêchée

Pourquoi ce cours ?

Au final, à la fin du cours précédent nous pouvions garantir un certain niveau d'isolation physique. Mais, souvent cela ne suffit pas. D'un point de vue applicatif, une logique métier peut s'étendre sur plusieurs transaction physique de base de données.

Exemple

```
Onglet A charge la commande -> status = OPEN
Onglet B charge la commande -> status = OPEN

Onglet A valide -> status = VALIDATED
Onglet B annule -> status = CANCELED
```

Nous validons et annulons en même temps la commande, quel doit être le résultat ?

- du point de vue du SGBD, tout est techniquement correct — chaque action est dans une transaction bien isolée.

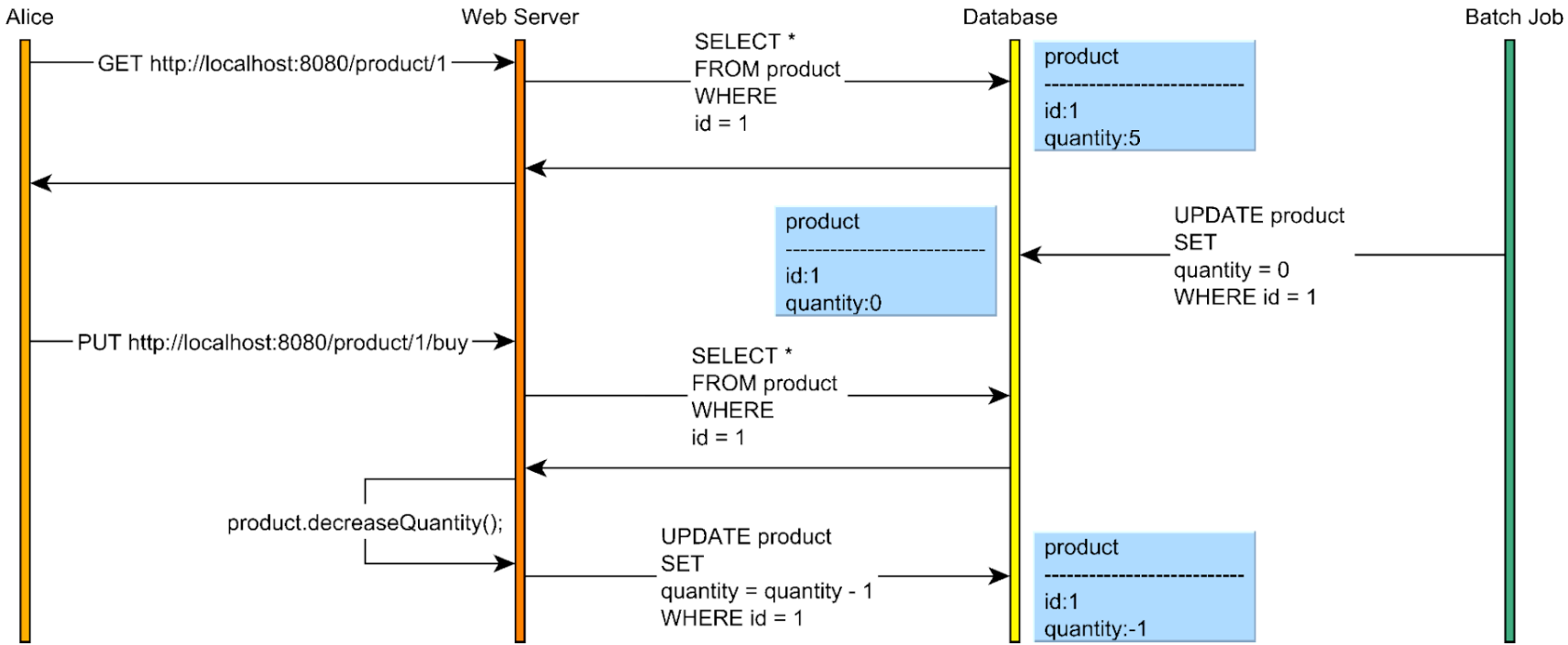


Figure 7.14: Stateless conversation loosing updates

Étant donné que la transaction logique d'Alice englobe deux requêtes Web distinctes, chacune étant associée à une transaction de base de données distincte, sans mécanisme de contrôle de concurrence supplémentaire, même le niveau d'isolation le plus élevé ne peut empêcher le phénomène de perte de mise à jour (lost update).

Transaction logique

ACID n’est pas suffisant

ACID garantit la cohérence technique des transactions au niveau de la base de données, mais cela ne suffit plus dès que l’on raisonne en transactions logiques métier, souvent réparties sur plusieurs interactions.

1. Une première transaction lit des données et les expose à l’utilisateur (⇒ une transaction)
2. L’utilisateur modifie ces données côté frontend, puis les renvoie au backend (⇒ une seconde transaction)

=> Ces deux étapes font partie d’une même intention métier, mais sont exécutées dans deux transactions techniques séparées.

Limite des niveaux d’isolation

Le niveau d’isolation — y compris SERIALIZABLE — **ne garantit la cohérence que à l’intérieur d’une transaction unique**. Dès lors qu’une logique métier s’étend sur plusieurs transactions

- L’isolation ne peut plus empêcher les modifications concurrentes
- Les hypothèses faites lors de la première lecture peuvent devenir invalides
- => La cohérence métier n’est plus garantie automatiquement

Limite des niveaux d'isolation

1. `GET product/1`

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT stock FROM produit WHERE id = 1;
COMMIT; -- fermeture de la transaction
```

2. Un batch `UPDATE` , mais comme la transaction 1 est finie il n'y a pas de verrou. Le batch update et commit sans aucun problème

3. POST `GET product/1/buy`

- ouverture d'une nouvelle transaction

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

UPDATE produit SET stock = stock - 1 WHERE id = 1;
COMMIT;
```

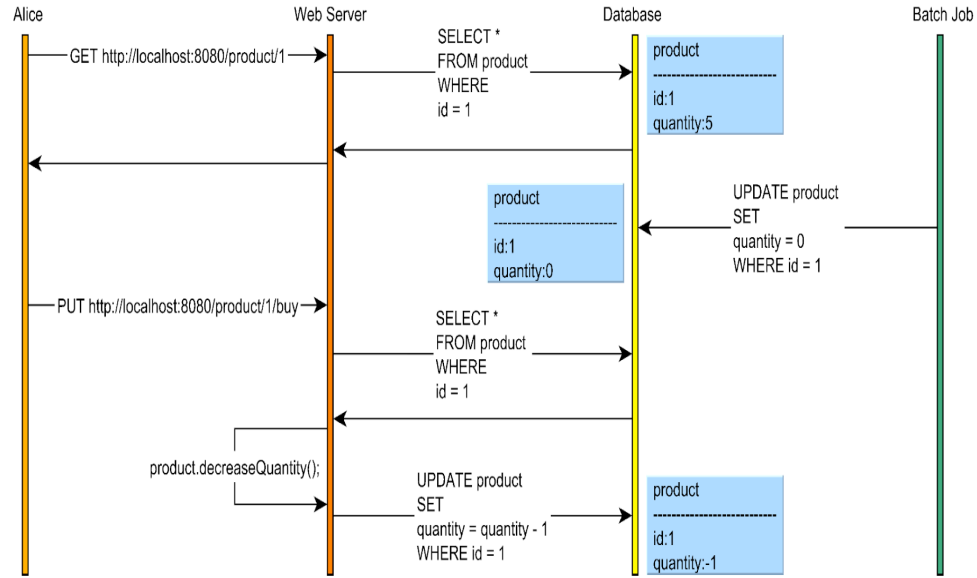


Figure 7.14: Stateless conversation loosing updates

Limite des niveaux d’isolation : exemple 2

Alice

```
// GET /commande/123
Commande commande = commandeRepository.findById(123);
// commande.status = "OPEN"



// Alice réfléchie



// POST /commande/123/valider
commande.setStatus("VALIDATED");
commandeRepository.save(commande);
// COMMIT
```

Bob

```
// GET /commande/123
Commande commande = commandeRepository.findById(123);
// commande.status = "OPEN"

// POST /commande/123/annuler
commande.setStatus("CANCELED");
commandeRepository.save(commande);
// COMMIT
```

- Les deux utilisateurs ont vu le même état initial "OPEN", mais ont pris deux décisions opposées.
- Ils agissent dans deux transactions indépendantes, peut-être même en SERIALIZABLE.
- Comme chaque UPDATE est valide individuellement, la base les accepte.
- **Le dernier commit gagne** → l’autre modification est écrasée sans alerte.

Pourquoi l’isolation physique ne suffit pas ici

> **Note**
> Le SGBD n’a aucune idée que ces deux actions sont mutuellement exclusives fonctionnellement. ⇒ Il ne peut donc pas prévoir de verrou !

Il voit juste `UPDATE commande SET status = 'VALIDATED' WHERE id = 123;`

Puis `UPDATE commande SET status = 'CANCELED' WHERE id = 123;`

Solutions

Nous avons deux solutions pour résoudre ce problème

- **Pessimistic Locking** : part du principe que des conflits sont susceptibles de se produire, il verrouille donc les données de manière préventive avant toute mise à jour.

- **Optimistic Locking**: part du principe que « les conflits sont rares ». Au lieu de verrouiller les données à l'avance, il permet à plusieurs utilisateurs d'accéder et même de modifier les mêmes données simultanément, et ne vérifie les conflits qu'au moment de la validation.

Juste avant d'étudier ces deux notions, regardons une solution naïve et fausse !

Solution naïve et fausse

Se dire qu'il suffit de faire un controle applicatif (Java)

```
decreaseStock() {
  int qty = createQuery("Select qty FROM article where id=1")
  if(qty <= 0) { throw new Exception("stock insuffisant") }
  createQuery("UPDATE article SET stock = stock - 1") }
}
```

Alice (/api/buy/1)

```
BEGIN;

SELECT stock FROM produit
WHERE id = 1

-- stock = 1 → OK

UPDATE produit
SET stock = stock - 1
WHERE id = 1;

COMMIT; -- stock = 0
```

Bob (/api/buy/1 en même temps)

```
BEGIN;

SELECT stock FROM produit
WHERE id = 1

-- stock = 1 → OK

UPDATE produit
SET stock = stock - 1
WHERE id = 1;

COMMIT;  -- stock = - 1
```

=> même si on vérifie `if(qty<0)` en cas d'accès concurrent, alors la valeur de `qty` sera à `1`

Pessimistic Locking

Comment le mettre en place ?

Définitionn
Il part du principe que des conflits sont susceptibles de se produire, il verrouille donc les données de manière préventive avant toute mise à jour.

Lors de chaque `SELECT` nous allons verrouiller les données durant toute la transaction : `SELECT ... FOR UPDATE` .

Avec Pessimistic Locking

```
decreaseStock() {
  int qty = createQuery("Select qty FROM article where id=1 FOR UPDATE") // SELECT ... FOR UPDATE
  if(qty <= 0) { throw new Exception("stock insuffisant") }
  createQuery("UPDATE article SET stock = stock - 1") }
}
```

Alice (/api/buy/1)

```
BEGIN;

SELECT stock FROM produit
WHERE id = 1
FOR UPDATE;  -- 🔒 Alice pose un verrou exclusif

-- stock = 1 → OK

UPDATE produit
SET stock = stock - 1
WHERE id = 1;

COMMIT;  -- 🔓 Libère le verrou
```

Bob (/api/buy/1 en même temps)

```
BEGIN;

SELECT stock FROM produit
WHERE id = 1
FOR UPDATE;
-- ⏳ Bob est bloqué → attend que le verrou soit libéré




-- Quand Alice commit, Bob reprend...
-- stock = 0 ❌ → détecté par if(qty<0) => Exception

ROLLBACK;  -- Aucun achat effectué
```

Est-ce la bonne approche ?

Si on met systématique un verrou, alors toute autre transaction voulant lire en mode FOR UPDATE ou écrire sur cette même ligne sera :

- bloquée jusqu'à la fin de la transaction (commit/rollback)
- ou rejetée si un timeout/détection de deadlock survient

En effet, si pour un besoin de statistique nous avons besoin de `SELECT stock FROM produit WHERE id = 1` alors nous devrons attendre que le verrou soit levé, même si le traitement tier (ici statistique) est non critique.

> Pessimistic locking would not help us in this case since Alice's read and the write happen in **different HTTP requests** and database transactions. (https://vladmihalcea.com/optimistic-vs-pessimistic-locking/)

⇒ Trouver une autre solution que les verrous : optimistic locking

Optimistic Locking

Définition

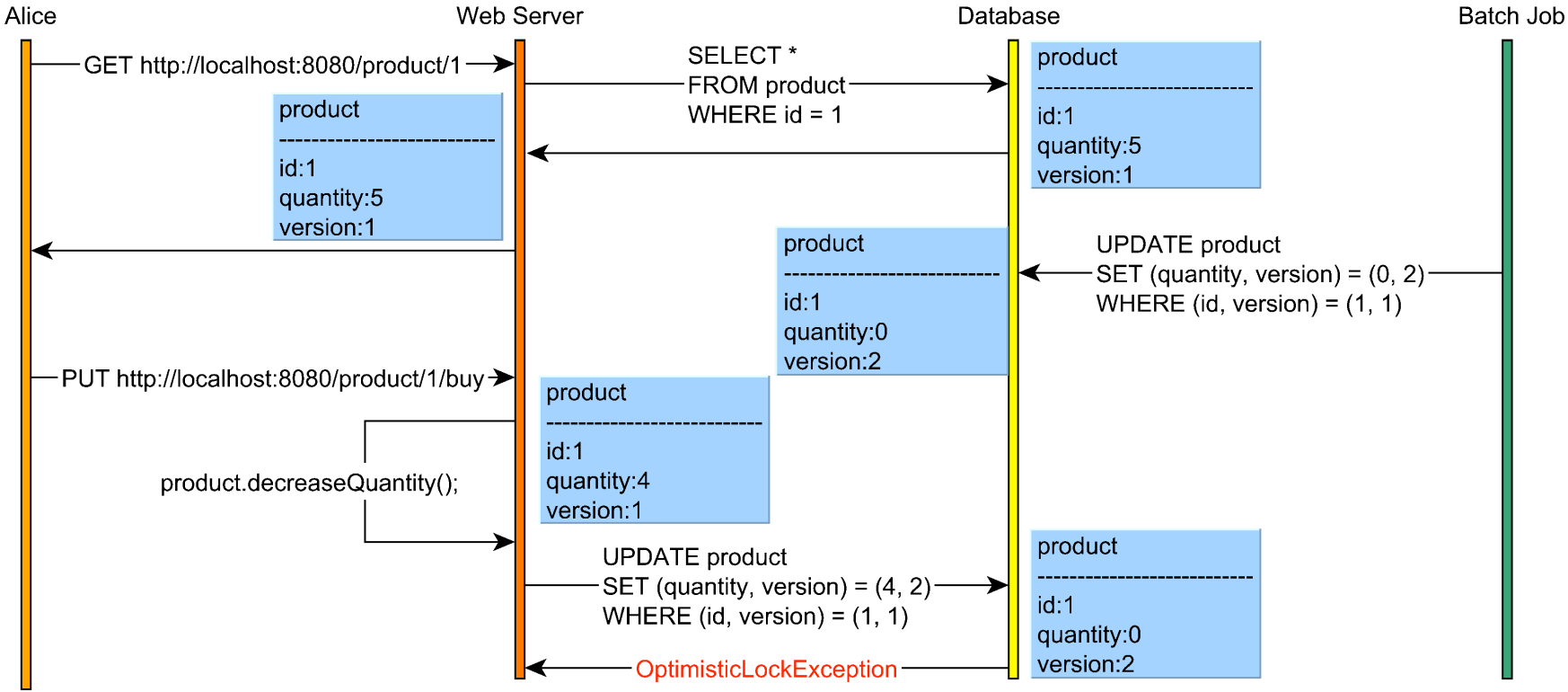
Définition
Il part du principe que « les conflits sont rares ». Au lieu de verrouiller les données à l'avance, il permet à plusieurs utilisateurs d'accéder et même de modifier les mêmes données simultanément, et ne vérifie les conflits qu'au moment de la validation.

L'algorithme de verrouillage optimiste fonctionne comme suit :

1. Lorsqu'un client lit une ligne particulière, sa version accompagne les autres champs
2. lors de la mise à jour d'une ligne, le client filtre l'enregistrement actuel en fonction de la version qu'il a précédemment chargée.

```
UPDATE produit
SET (quantité, version) = (4, 2)
WHERE id = 1 AND version = 1; -- ajout de "version =" dans la clause WHERE
```

3. Si le résultat de l'instruction est égal à zéro, cela signifie que la version a été incrémentée entre-temps. Donc la transaction actuelle opère désormais sur une version obsolète de l'enregistrement.



Par conséquent, la méthode executeUpdate de l'instruction UPDATE PreapredStatement va renvoyer une valeur de 0, ce qui signifie qu'aucun enregistrement n'a été modifié, et le framework d'accès aux données sous-jacent va lever une exception OptimisticLockException qui provoquera le rollback de la transaction d'Alice.

La table `produit` contient une colonne `version`

```
id | stock | version
 1     1       4
```

```sql
BEGIN;

-- Étape 1 : lecture avec version
SELECT stock, version FROM produit
WHERE id = 1;
-- Résultat : {stock = 1, version = 4}

-- Côté logique java: stock>0 on continue

-- Étape 2 : tentative de mise à jour conditionnée
UPDATE produit
SET stock = stock - 1,
    version = version + 1
WHERE id = 1 AND version = 4;

-- Si 1 ligne modifiée → OK
COMMIT;
```

```sql
BEGIN;

-- Étape 1 : lecture initiale (quasi simultanée avec Alice)
SELECT stock, version FROM produit
WHERE id = 1;
-- Résultat : {stock = 1, version = 4}

-- Côté logique java: stock>0 on continue

-- Étape 2 : tentative de mise à jour conditionnée
UPDATE produit
SET stock = stock - 1,
    version = version + 1
WHERE id = 1 AND version = 4;

-- Version est passé à 5, donc rien à UPDATE;
--    WHERE id = 1 AND version = 4 donne 0 ligne

ROLLBACK;
```

Avantages Optimistic Locking

- Pas de verrou bloquant (`SELECT ... FOR UPDATE`)
- => Donc plusieurs traitement peuvent accéder à la donnée de manière simultanée
- => Si conflit, conflit détecté au moment du commit, `OptimisticLockException`

Note

Dans le cas d'une API REST on utilisera généralement des locks optimistes (`@Version`)

Conclusion

Conclusion

Dans cette seconde partie sur les transactions nous avons évoqué le problème au niveau applicatif

> *Comment garantir la cohérence métier dans un système où la logique applicative s'étend au-delà d'une seule transaction physique ?*

- Même avec des niveaux d'isolation élevés (REPEATABLE READ, SERIALIZABLE), certaines anomalies métier peuvent survenir.
- Il est donc nécessaire de mettre en place des mécanismes de contrôle de concurrence applicatifs.

Deux approches

Approche	Principe	Avantage principal
Pessimistic Locking	Bloquer l'accès dès la lecture	Sécurisé, immédiat
Optimistic Locking	Détecter les conflits au moment du commit (`@Version`)	Léger, adapté au web

Conclusion sur les transaction

Dans cette longue section dédiée aux transactions nous avons

- revue la notion d'atomicité en SQL et en JAVA avec l'instruction `setAutoCommit(true/false)`
- puis étudier en détail la notion d'isolation

Cette notion d'isolation nous a permis de comprendre comment on résout l'accès concurrent à la donnée

- tout d'abord les SGBD implémente des mécanismes de Two-Phase Locking et de MVCC
- mais, pour des raisons de performance nous ne souhaitons pas forcément les mettre en oeuvre complètement
- ainsi, plusieurs niveaux d'isolation sont définis autorisant certaines anomalies

Mais malgré ce mécanisme, il nous reste le problème des transaction logiques, où ACID n'est pas suffisant

- une solution naive (et fausse) de contrôler uniquement dans le code ne suffit pas
- nous devons soit implémenter une approche pessimiste, mais qui va provoquer des latences
- soit implémenter une approche optimiste qui consiste à détecter les conflits via @version