

# JDBC Transactions

---

adriencaubel.fr

# Ressources

- <https://blogs.oracle.com/maa/from-chaos-to-order-the-importance-of-concurrency-control-within-the-database-2-of-6>
- <https://vladmihalcea.com/a-beginners-guide-to-transaction-isolation-levels-in-enterprise-java/>
- <https://www.baeldung.com/jpa-optimistic-locking>
- <https://stackoverflow.com/questions/47441027/pessimistic-locking-vs-serializable-transaction-isolation-level>

# Table des matières

- 1. Introduction
  - 1. Pourquoi ce cours ?
- 2. Transaction physique
  - 1. Comportement par défaut
  - 2. autocommit=false
  - 3. Contrôle de l'accès concurrent : le problème
  - 4. Les solutions
  - 5. Niveaux d'isolation : Définition
  - 6. 4 niveaux d'isolation
  - 7. Anomalies autorisées
  - 8. Quel niveau d'isolation choisir ?
- 3. Transaction logique
  - 1. ACID n'est pas suffisant
  - 2. Limite des niveaux d'isolation
  - 3. Solution 1 : Pessimistic Locking
  - 4. Solution 2 : Optimistic Locking

# Introduction

# Pourquoi ce cours ?

Dans un système où plusieurs transactions s'exécutent simultanément, l'accès concurrent aux mêmes données peut entraîner des incohérences si ces accès ne sont pas correctement coordonnés. Deux transactions peuvent par exemple lire une valeur obsolète, écraser mutuellement leurs mises à jour (lost update), ou observer des états intermédiaires invalides.

## Les solutions

- Niveau physique : Les niveaux d'isolation transactionnelle (DB) protègent surtout via **des mécanismes "physiques"** côté base (locks, MVCC, etc.)
- Niveau logique : Pessimistic ou Optimistic locking protège via une **règle "logique" au niveau applicatif** (détection de conflit au moment du commit)

# Transaction physique

# Comportement par défaut

Par défaut, chaque instruction est faite dans une transaction indépendante `autocommit=true`

```
Connection conn = DriverManager.getConnection(url, user, password);
try {
    // Transaction 1
    PreparedStatement debit = conn.prepareStatement("UPDATE account SET balance = balance - 100 WHERE id = 1");
    debit.executeUpdate(); // COMMIT immédiat
    // Transaction 2
    PreparedStatement credit = conn.prepareStatement("UPDATE account SET balance = balance + 100 WHERE id = 2");
    credit.executeUpdate(); // COMMIT immédiat
} catch (Exception e) {
    // rollback inutile ici
    conn.rollback();
}
```

=> Les deux étapes doivent réussir ou échouer ensemble, une seule et unique transaction. Ici ce n'est pas le cas.

# autocommit=false

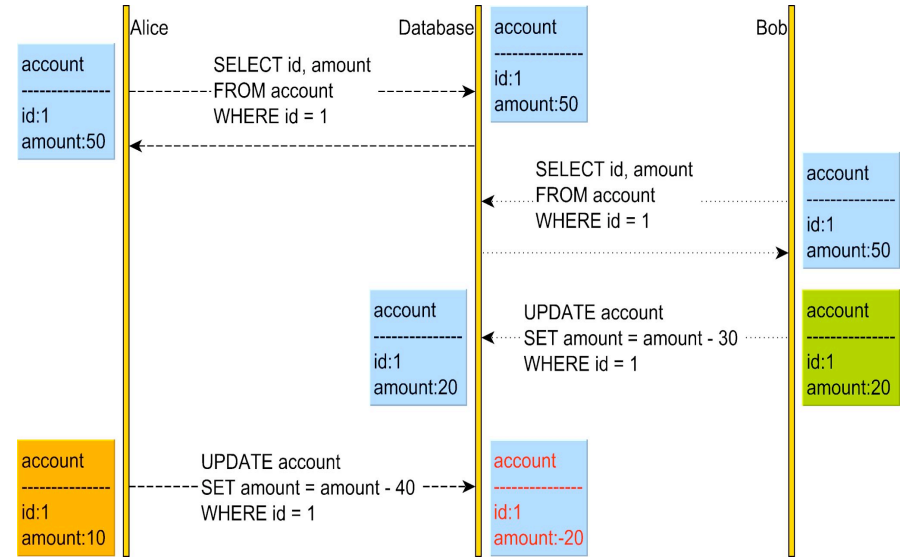
```
Connection conn = DriverManager.getConnection(url, user, password);
conn.setAutoCommit(false); // ● on dit qu'on gèrera nous même le .commit() et le .rollback()
try {
    // Pas de transaction
    PreparedStatement debit = conn.prepareStatement("UPDATE account SET balance = balance - 100 WHERE id = 1");
    debit.executeUpdate(); // COMMIT immédiat
    // Pas de transaction
    PreparedStatement credit = conn.prepareStatement("UPDATE account SET balance = balance + 100 WHERE id = 2");
    credit.executeUpdate(); // COMMIT immédiat

    conn.commit(); // commit atomique - 1 transaction
} catch (Exception e) {
    conn.rollback(); // rollback complet
}
```

# Contrôle de l'accès concurrent : le problème

Dans un système où plusieurs transactions s'exécutent simultanément, l'accès concurrent aux mêmes données peut entraîner des incohérences si ces accès ne sont pas correctement coordonnés. Deux transactions peuvent par exemple lire une valeur obsolète, écraser mutuellement leurs mises à jour (lost update), ou observer des états intermédiaires invalides.

1. Alice et Bob lisent (read) un compte
2. Bob le met à jour et le commit
3. Alice fait le même, mais ne réalise pas que Bob avait déjà changé la ligne. ⇒ Conflit



# Les solutions

Pour gérer les conflits de données, plusieurs mécanismes de contrôle de la concurrence ont été développés au fil des ans. Il existe essentiellement deux stratégies pour gérer les collisions de données :

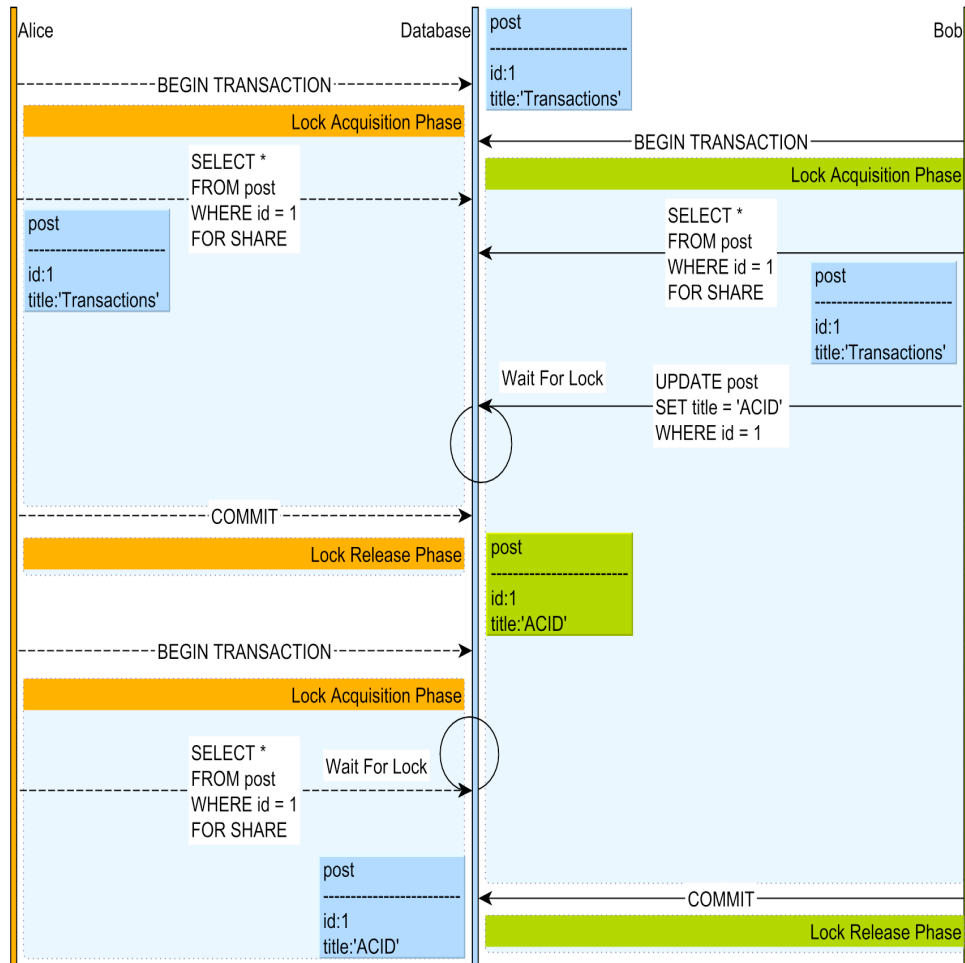
- **Eviter les conflits (Conflict Avoidance)** : par exemple, le verrouillage en deux phases, nécessite un verrouillage pour contrôler l'accès aux ressources partagées; cf Two-Phase Locking
- **Détecter les conflits (Conflict Detection)** : par exemple, le contrôle de concurrence multiversions, offre une meilleure concurrence, au prix d'un assouplissement de la sérialisabilité et de l'acceptation éventuelle de diverses anomalies de données. cf Multi-Version Concurrency Control

## Two-Phase Locking (2PL)

Chaque système de base de données possède sa propre hiérarchie de verrouillage, mais les types les plus courants restent les suivants :

- **shared (read) lock**, empêcher l'écriture d'un enregistrement tout en autorisant les lectures simultanées; le verrou est partagé entre les lecteurs
- **exclusive (write) lock**, interdit à la fois les opérations de lecture et d'écriture

1. Alice et Bob sélectionnent tous les deux un enregistrement de type post, acquérant chacun un verrou partagé (shared lock).
2. Bob tente de mettre à jour l'entrée post, son instruction est bloquée par le gestionnaire de verrous (Lock Manager), car Alice détient toujours un verrou partagé sur cette ligne.
3. Alice termine la transaction => libère le verrou partagé
4. La mise à jour de Bob provoque le remplacement du verrou partagé par un verrou exclusif
5. Alice veut SELECT mais bloqué par verrou exclusif de Bob
6. Après le commit de la transaction de Bob, tous les verrous sont libérés.



L'utilisation du verrouillage pour contrôler l'accès aux ressources partagées **est susceptible d'entraîner des deadlocks**, et le planificateur de transactions ne peut à lui seul empêcher leur apparition. Par exemple

- T1
  - lock(X) sur la ligne A
  - veut lock(X) sur la ligne B → bloquée
- T2
  - lock(X) sur la ligne B
  - veut lock(X) sur la ligne A → bloquée
- Résultat
  - T1 attend B détenu par T2
  - T2 attend A détenu par T1

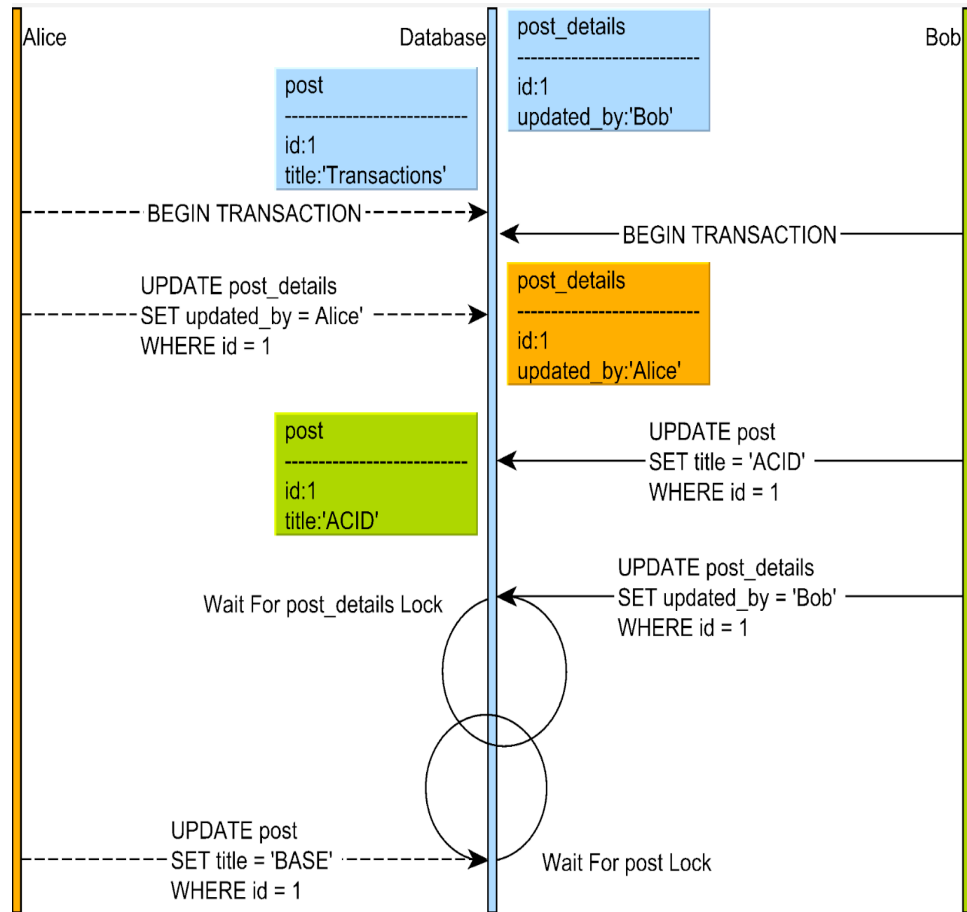


Figure 7.3: Dead lock

# Multi-Version Concurrency Control

Lorsque vous utilisez 2PL, chaque lecture nécessite l'acquisition d'un verrou partagé, tandis qu'une opération d'écriture nécessite l'acquisition d'un verrou exclusif.

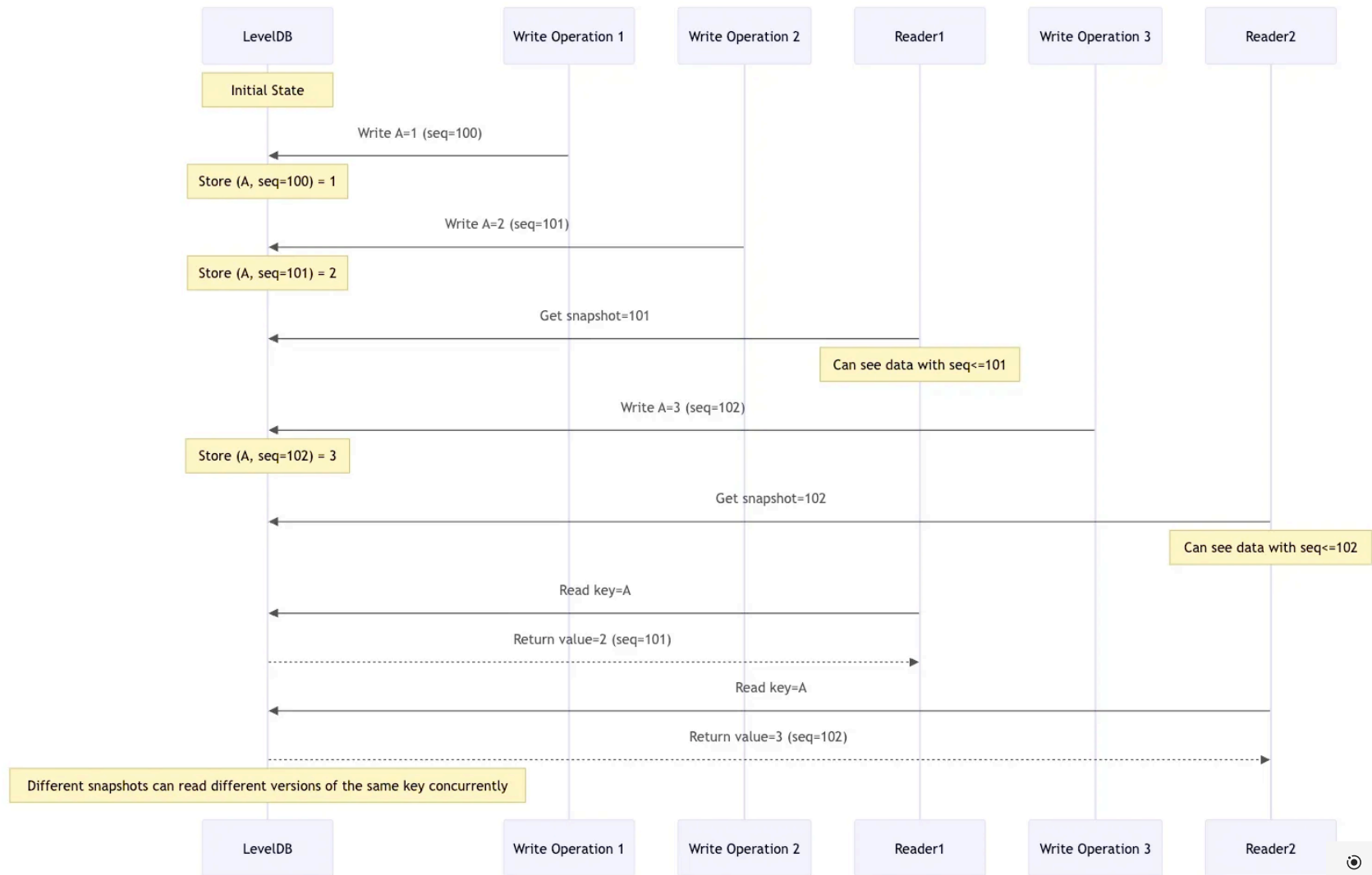
- shared lock bloque les écritures, mais permet à d'autres lecteurs d'acquérir le même verrou partagé
- exclusive lock bloque à la fois les lecteurs et les rédacteurs qui concourent pour le même verrou.

Bien que le verrouillage puisse fournir un plan de transactions, le coût des conflits de verrouillage peut nuire à la fois au temps de réponse des transactions et à l'évolutivité.

- Le temps de réponse peut augmenter car les transactions doivent attendre que les verrous soient libérés,
- et les transactions de longue durée peuvent également ralentir la progression des autres transactions simultanées.

⇒ Pour pallier ces lacunes, les fournisseurs de bases de données ont opté pour des mécanismes de contrôle de concurrence optimistes. Si le 2PL empêche les conflits, le contrôle de concurrence multiversion (MVCC) utilise plutôt une stratégie de détection des conflits.

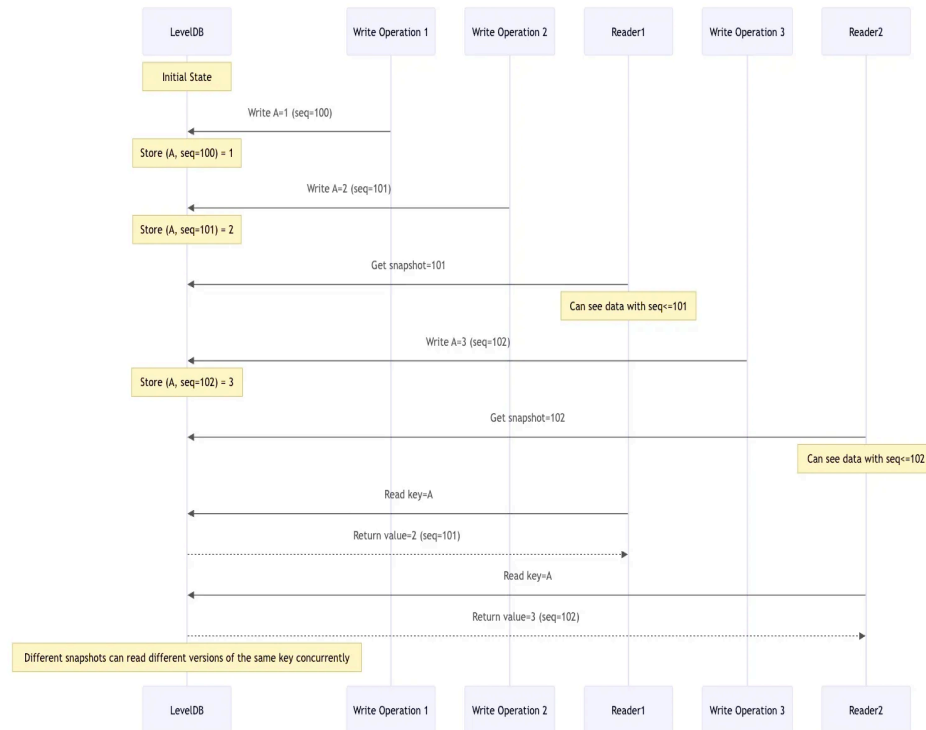
1. Chaque enregistrement de la base de données possède un numéro de version.
2. Les lectures simultanées s'effectuent sur l'enregistrement ayant le numéro de version le plus élevé.
3. Les opérations d'écriture s'effectuent sur une copie de l'enregistrement, et non sur l'enregistrement lui-même.
4. Les utilisateurs continuent à lire l'ancienne version pendant que la copie est mise à jour.
5. Une fois l'opération d'écriture réussie, l'identifiant de version est incrémenté.
6. Les lectures simultanées suivantes utilisent la version mise à jour.
7. Lorsqu'une nouvelle mise à jour a lieu, une nouvelle version est à nouveau créée, et le cycle se poursuit.



Que ce soit Reader1 ou Reader2 qui lit en premier,

- Reader1 lisant la clé=A obtiendra toujours la valeur=2 (séquence=101),
- tandis que Reader2 lisant la clé=A obtiendra la valeur=3 (séquence=102).

S'il y a des lectures ultérieures sans spécification d'instantané, elles obtiendront les données les plus récentes.



# Niveaux d'isolation : Définition

L'isolation de la base de données permet à une transaction de s'exécuter comme s'il n'y avait aucune autre transaction en cours d'exécution simultanément.

L'isolation est garantie par MVCC ou les Locks (2PL)

## 4 niveaux d'isolation

- **Sérialisable** : il s'agit du niveau d'isolation le plus élevé. Les transactions simultanées sont garanties d'être exécutées dans l'ordre (= 2PL strict).
- **Lecture répétable (Repeatable Read)** : les données lues pendant la transaction restent identiques à celles au début de la transaction. Si on lit une ligne 2 fois dans la même transaction, on verra la même valeur (même si une autre transaction l'a modifiée entre temps)
- **Lecture validée (Read Committed)** : les modifications apportées aux données ne peuvent être lues qu'après la validation de la transaction.
- **Lecture non validée (Read Uncommitted)** : les modifications apportées aux données peuvent être lues par d'autres transactions avant la validation d'une transaction.

# Read Committed : le problème

Lire une donnée qu'une autre transaction a modifiée mais pas encore commit.

## Transaction Alice

```
BEGIN;  
UPDATE stock SET qty = 6 WHERE id=1; -- pas encore COMMIT
```

## Transaction Bob

```
SELECT qty FROM stock WHERE id=1; -- verrait 6 ❌ (dirty read)
```

Si Alice fait ensuite :

```
ROLLBACK;
```

=> Bob lit une valeur qui n'a jamais existé officiellement.

=> Read Committed met en place un process pour ne lire que les valeurs réellement commit

# Repeatable Read : le problème

Le niveau d'isolation REPEATABLE READ sert principalement à résoudre le problème suivant : **Quand on lit deux fois la même ligne dans une même transaction on obtient des valeurs différentes**

Exemple (problème en READ COMMITTED)

Transaction Bob (READ COMMITTED)

```
BEGIN; -- début trx  
SELECT qty FROM stock WHERE id=1; -- retourne 7
```

Transaction Alice

```
UPDATE stock SET qty = 6 WHERE id=1;  
COMMIT;
```

Transaction Bob (relit)

```
SELECT qty FROM stock WHERE id=1; -- retourne 6 ❌  
COMMIT;
```

=> Dans la même transaction, Bob lit 7 puis 6 : c'est une lecture non répétable.

# Repeatable Read : les solutions

Avec REPEATABLE READ, Bob relira toujours la même version de la ligne pendant toute sa transaction :

```
BEGIN;  
SELECT qty FROM stock WHERE id=1;  -- retourne 7  
-- Alice change à 6 et commit...  
SELECT qty FROM stock WHERE id=1;  -- retourne encore 7 ✅  
COMMIT;
```

Deux grandes implémentations possibles selon la DB :

















## 1. MVCC snapshot

- on lit une "photo" (snapshot) de la base au début de ta transaction
- les lectures sont cohérentes
- mais ça ne bloque pas forcément les autres

## 2. Locks la DB pose des verrous partagés (*shared lock*) qui empêchent certains changements concurrents

# Anomalies autorisées

Suivant le niveau d'isolation on autorise ou pas certaines anomalies

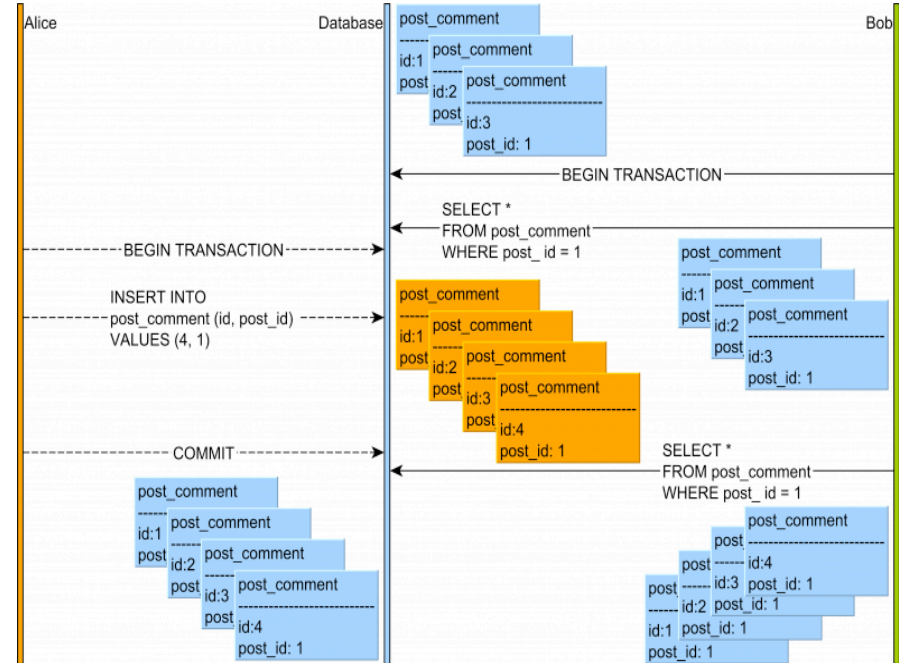
Isolation level	Dirty read	Non-repeatable read	Phantom	Lost update
READ UNCOMMITTED	 allowed	 allowed	 allowed	 allowed
READ COMMITTED	 prevented	 allowed	 allowed	 allowed
REPEATABLE READ	 prevented	 prevented	 allowed	 prevented
SERIALIZABLE	 prevented	 prevented	 prevented	 prevented

Seul le niveau SERIALIZABLE garantit un niveau "parfait" de cohérence (mais au détriment des performances; attente des locks)

# Phantom Read

1. Alice et Bob lancent deux transactions de base de données.
2. Bob lit tous les enregistrements `post_comment` associés à la ligne `post` dont la valeur d'identifiant est 1.
3. Alice ajoute un nouvel enregistrement `post_comment` associé à la ligne `post` dont la valeur d'identifiant est 1. Et valide la trx
4. Si Bob relit les enregistrements `post_comment` dont la valeur de la colonne `post_id` est égale à 1, il observera une version différente de cet ensemble de résultats.

**Ce phénomène pose problème lorsque la transaction en cours prend une décision commerciale basée sur la première version de l'ensemble de résultats donné.**



# Phantom Read : exemple

## Transaction Bob

```
BEGIN;  
SELECT * FROM stock WHERE qty > 0; -- retourne 5 lignes
```

## Transaction Alice

```
INSERT INTO stock(id, qty) VALUES (99, 10);  
COMMIT;
```

## Transaction Bob (refait la même requête)

```
SELECT * FROM stock WHERE qty > 0; -- retourne 6 lignes ❌  
COMMIT;
```

=> Un phantom read concerne une requête qui retourne un ensemble de lignes, et au second read tu vois une ligne en plus / en moins car quelqu'un a fait un INSERT / DELETE (ou parfois UPDATE qui fait passer une ligne dans le filtre).

## Phantom Read - solution ?

The SQL standard says that Phantom Read occurs if two consecutive query executions render different results because a concurrent transaction has modified the range of records in between the two calls.

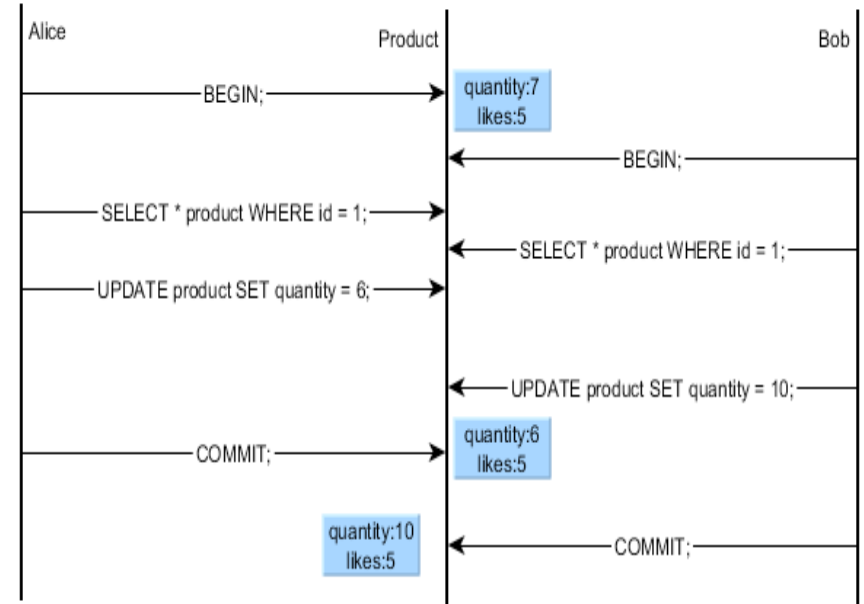
- 2PL avec SERIALIZABLE : un verrou exclusif est déposé lors de la lecture (au lieu d'un verrou shared) => Alice ne peut ni lire ni ajouter un post.

<https://vladmihalcea.com/phantom-read/>

# Lost Update

Dans cet exemple, Bob n'est pas au courant qu'Alice vient de modifier la quantité de 7 à 6, donc sa mise à jour est écrasée par la modification de Bob.

1. SELECT qty FROM stock WHERE id=1
2. On calcule en app
3. UPDATE stock SET qty = ... alors une autre transaction peut passer entre les deux → et on écrase sa modification.



## Lost Update - solutions

- Monter l'isolation a REPEATABLE READ
  - un verrou partagé (*shared lock*) est posé => les autres trx peuvent lire mais ne peuvent pas update la ligne
- Verrou pessimiste : `SELECT ... FOR UPDATE` (voir après)
- Optimistic Locking (colonne `version` ) (voir après)

# Quel niveau d'isolation choisir ?

Par défaut :

- MySQL : REPEATABLE READ
- PostgreSQL et Oracle : READ COMMITTED

=> Ca dépend quel phénomène on souhaite éviter

```
Connection connection = DriverManager.getConnection(  
    "jdbc:postgresql://localhost:5432/mydb",  
    "user",  
    "password"  
);  
  
// Disable auto-commit to start a transaction  
connection.setAutoCommit(false);  
  
// Set isolation level  
connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
```

# 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.

# ACID n'est pas suffisant

1. Alice demande l'affichage d'un produit.
2. Le produit est récupéré dans la base de données et renvoyé au navigateur.
3. Alice demande une modification du produit.
4. Comme Alice n'a pas conservé de copie de l'objet précédemment affiché, elle doit le recharger une nouvelle fois.
5. Le produit est mis à jour et enregistré dans la base de données.
6. La mise à jour du traitement par lots a été perdue et Alice ne s'en rendra jamais compte.

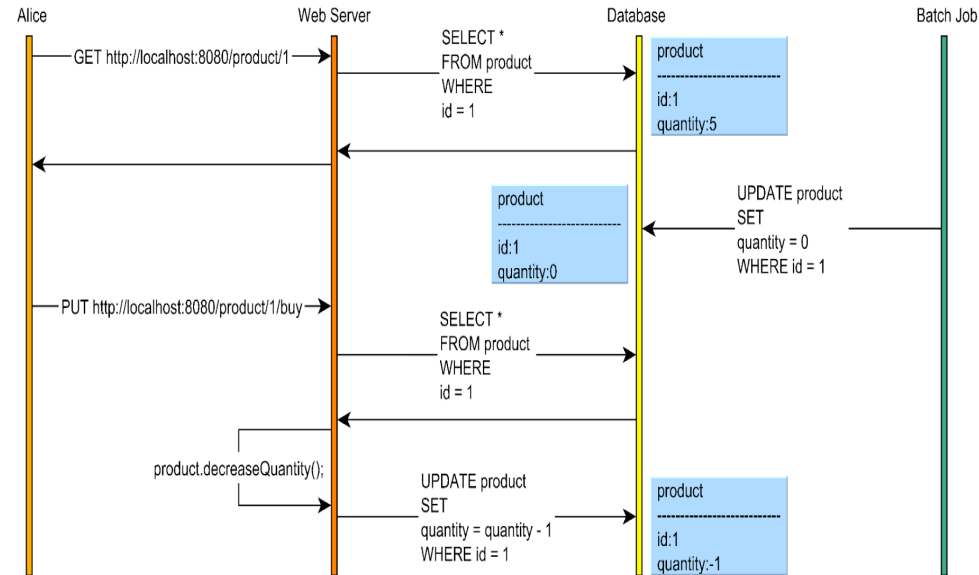


Figure 7.14: Stateless conversation losing updates

# 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

# Solution 1 : Pessimistic Locking

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.

=> Un lock exclusif est acquis pour éviter qu'une autre transaction acquière elle aussi un verrou shared/exclusif

## Ne fonctionne que dans un environnement stateful

```
GET /order/42 → lock row  
(wait for user decision); le lock est rendu ...  
PUT /order/42 → update
```

```
POST /transfer
```

```
BEGIN;  
SELECT * FROM accounts WHERE id IN (A, B) FOR UPDATE; ici ca fonctionne  
-- business rules  
UPDATE accounts ...  
COMMIT;
```

## Solution 2 : Optimistic Locking

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.

## Solution 2: Optimistic Locking

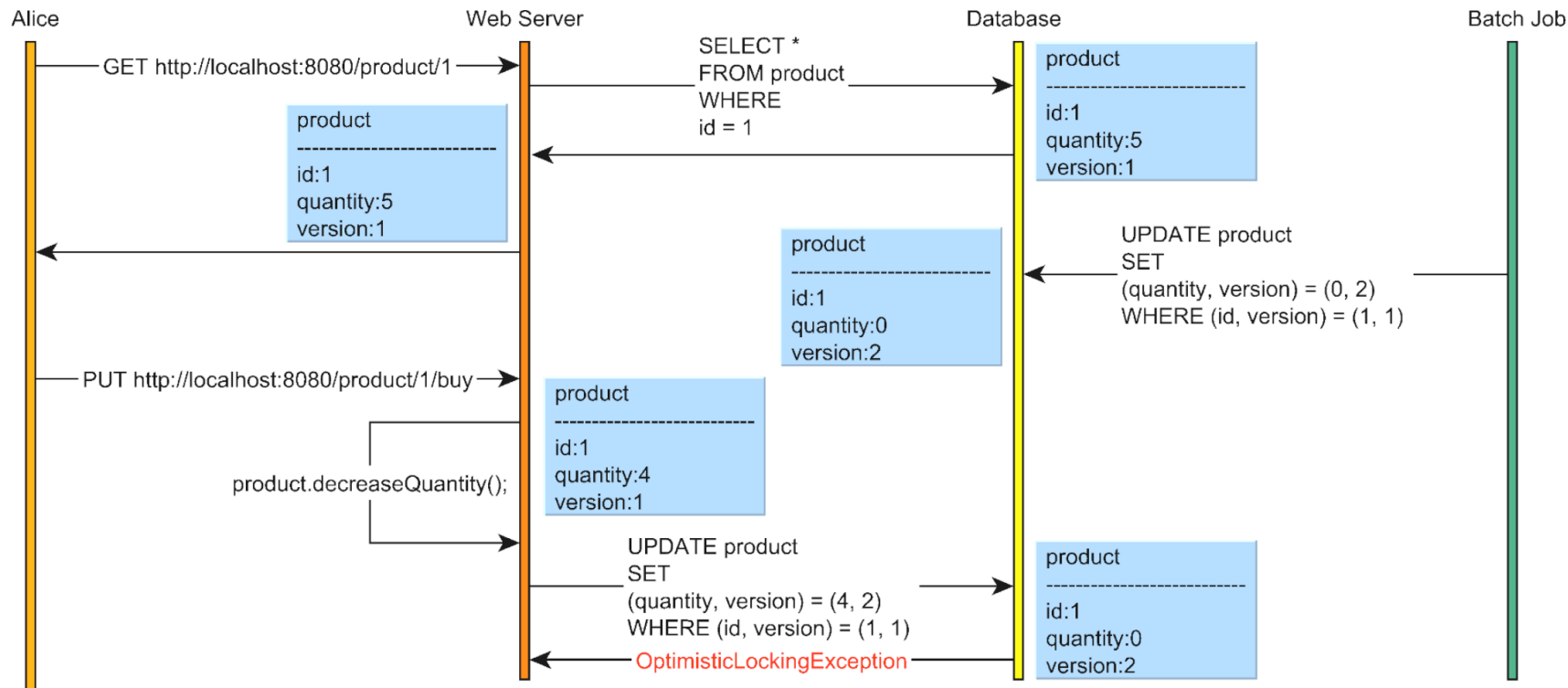


Figure 7.16: Stateful conversation preventing lost updates

# Quand utiliser Optimistic Locking

## Deux applications

Lorsque deux applications distinctes accèdent à la même base de données

## Deux onglets du navigateur

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

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

```
Onglet A valide -> status = VALIDATED
```

```
Onglet B annule -> status = CANCELED
```

## Conclusion

En cas d'accès concurrent, alors une exception est levée et doit être transmise à l'utilisateur en lui demandant de rafraichir sa page par exemple

# Isolation vs Pessimistic Lock vs Optimistic Lock

Lorsqu'on a un accès concurrent à la donnée, nous avons plusieurs solution pour le régler :

- physique : utiliser un niveau d'isolation plus strict (cas une seule trx physique)
- logique : utiliser les locks optimistes ou pessimistes (cas plusieurs trx physique)

## Note

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