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

Introduction

Pourquoi ce cours ?

Objectif
- Définir la notion de transaction
- Étudier en détail le terme Isolation de ACID

Dans ce premier cours, nous allons nous concentrer exclusivement sur la notion de transaction au sens du langage SQL.

Nous allons approfondir en particulier le concept d'isolation, en examinant les différents problèmes qui peuvent survenir lorsque plusieurs transactions s'exécutent simultanément.

Définir une transaction ?

ACID

- Atomicity guarantees that multiple operations in a transaction act as a single unit–either all succeed or all fail.
- Consistency ensures that the database remains in a valid state that adheres to defined rules and constraints.
- Isolation prevents concurrent transactions from interfering with each other.
- Durability makes sure that once a transaction commits, the results are permanent, even after system failures.

Focus sur le A et le I

La *consistance* et la *durabilité* sont gérés par la base de données. Le développeur lui peut intervenir sur l'atomicité et l'isolation.

- A : **à réviser**
- C : invariant BDD
- I : **à étudier**
- D : composants internes BDD

=> Donc le focus de cette leçon est sur l'isolation

L’atomicité

Définition

Définition
L'atomicité garantit que chaque transaction est traitée comme une seule "unité", qui réussit complètement ou échoue complètement

Atomicité SQL

```sql
BEGIN TRANSACTION;

-- Étape 1 : Débiter le compte source
UPDATE account
SET balance = balance - 100
WHERE id = 1;

-- Étape 2 : Créditer le compte destinataire
UPDATE account
SET balance = balance + 100
WHERE id = 2;

-- COMMIT Si tout s'est bien passé OU ROLLBACK on annule tout
COMMIT ou ROLLBACK;
```

Atomicité JDBC

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, si échec alors transaction 1 non rollback

} catch (Exception e) {
    // rollback inutile ici
    conn.rollback();
}
```

Comportement attendu : les deux étapes doivent réussir ou échouer ensemble, une seule et unique transaction. Ici ce n'est pas le cas.

Atomicité JDBC

autocommit=false + commit manuel

```
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();  // pas de commit immédiat
    // Pas de transaction
    PreparedStatement credit = conn.prepareStatement("UPDATE account SET balance = balance + 100 WHERE id = 2");
    credit.executeUpdate(); // pas de commit immédiat

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

L’Isolation

Définition

Définition
L'isolation garantit que l'exécution simultanée des transactions laisse la base de données dans le même état que celui qui aurait été obtenu si les transactions avaient été exécutées séquentiellement.

Pour comprendre cette définition, regardons **le problème d'accès concurrent** à la base de données.

Le problème d’accès concurrent : exemple 1

```sql
BEGIN; -- Trx 1

SELECT solde FROM compte WHERE id = 1;
-- Résultat : 50

-- T1 calcule côté appli : 100 - 30 = 70

UPDATE compte
SET solde = 70
WHERE id = 1;

COMMIT;
```

```sql
BEGIN; -- Trx 2


SELECT solde FROM compte WHERE id = 1;
-- Résultat : 100 (car T1 n’a pas encore commit au moment d

-- T2 calcule côté appli : 100 - 50 = 50

UPDATE compte
SET solde = 50
WHERE id = 1;

COMMIT;
```

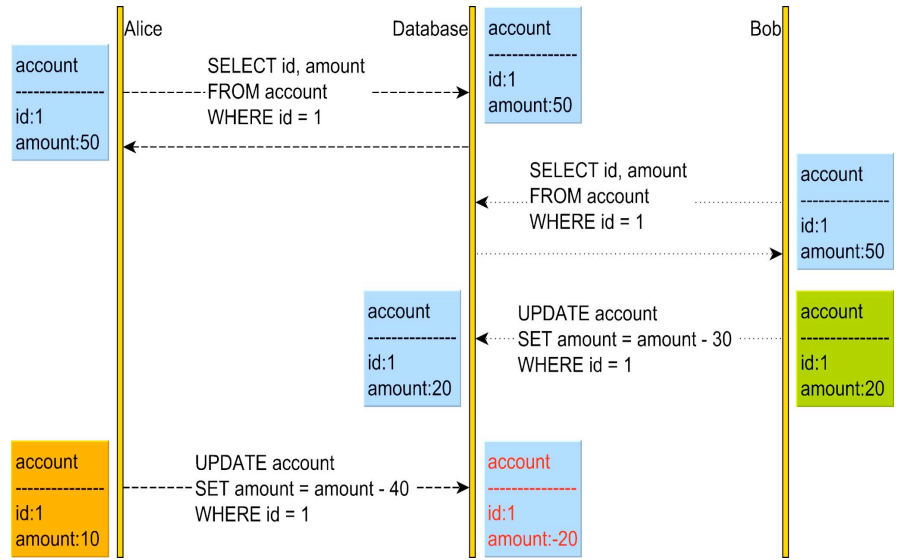
Résultat final

- T1 et T2 lisent 100
- T1 écrit 70
- T2 écrit 50, il écrase l’update de T1 => On a un *Lost Update*

Le problème d'accès concurrent : exemple 2

Étape	Action	solde
Début	Valeur initiale	50
Alice	lit 50	—
Bob	lit 50	—
Bob	écrit 20	20
T2	écrit 20-40	**-20**

Dans l'exemple précédent le `UPDATE` inséré directement la valeur, ici on fait un calcul `amount - valeur`. Également *LOST UPDATE*



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

Eviter les conflits (Conflict Avoidance)

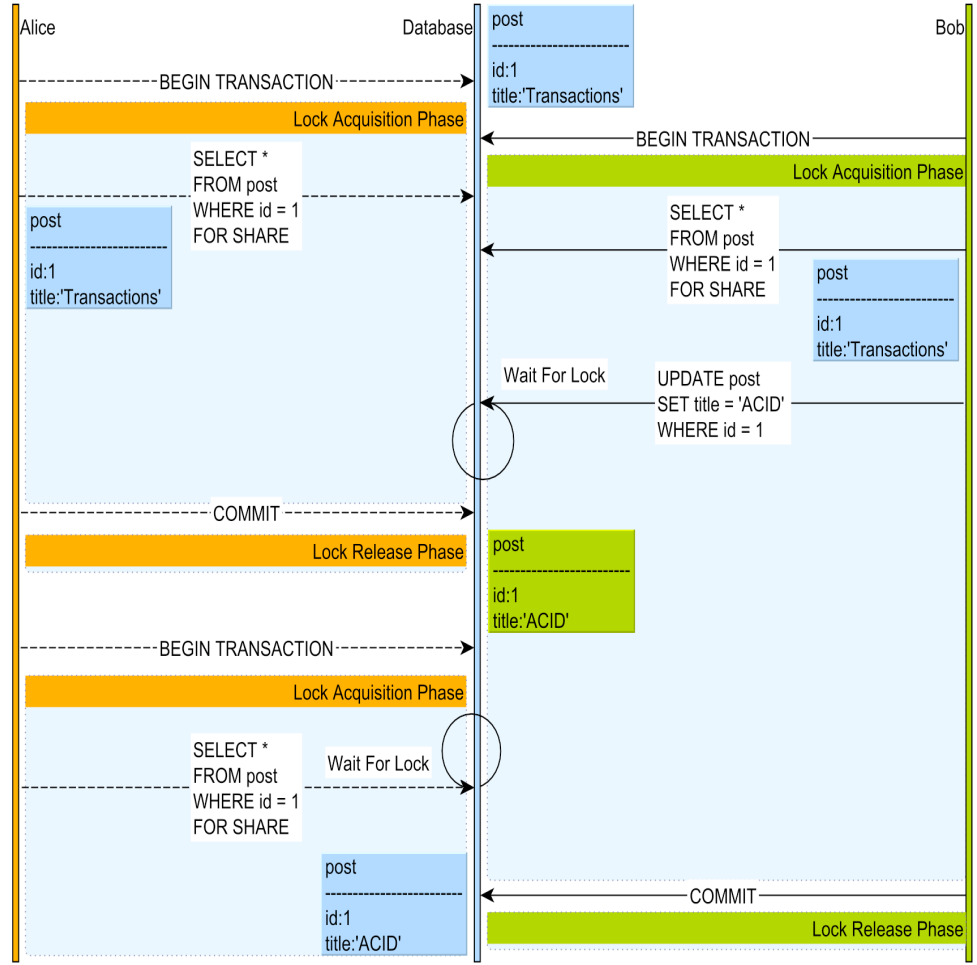
Two-Phase Locking (2PL)

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.

⇒ pas d'anomalie, mais crée de l'attente



```sql
BEGIN; -- T1

SELECT solde FROM compte
WHERE id = 1
FOR UPDATE;   -- 🔒 verrou exclusif (X-lock) sur id=1

-- Résultat : 100
-- calcul appli : 100 - 30 = 70

UPDATE compte
SET solde = 70
WHERE id = 1;

COMMIT; -- 🔒 libération du verrou
```

- T1 : met un verrou lors de l'acquisition
- *T2 : doit attendre que le verrou soit libéré*
- T1 : fait la mise à jour **et commit**
- T2 : le verrou est libéré, le SELECT renvoie 70
- T2 : calcule est fait l'insertion => 70-50 = 20

Sans verrou on avait un *Lost Update* où on ignoré la valeur de T1 (on avait 50).

```sql
BEGIN; -- T2

SELECT solde FROM compte
WHERE id = 1
FOR UPDATE;  -- ⏳ BLOQUÉ tant que T1 n'a pas commit




-- Maintenant T2 obtient le verrou
-- Résultat du SELECT : 70 (car T1 a commit)

-- calcul appli : 70 - 50 = 20

UPDATE compte
SET solde = 20
WHERE id = 1;

COMMIT;
```

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

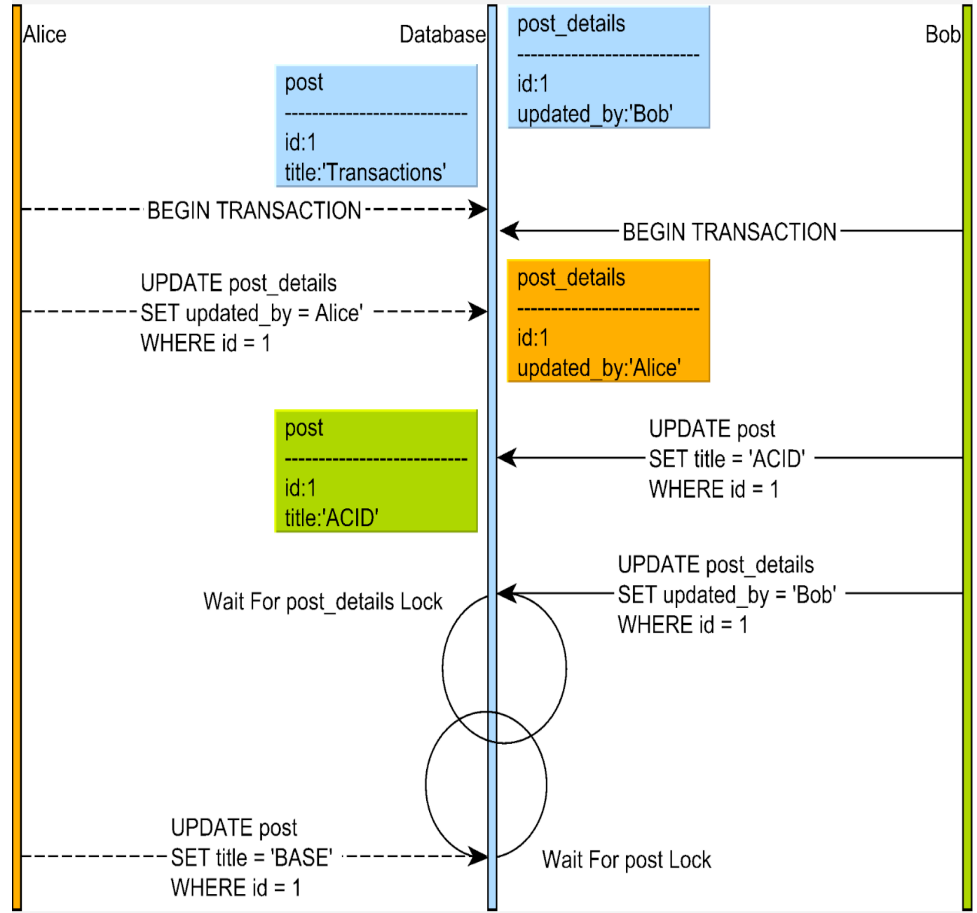


Figure 7.3: Dead lock

```sql
BEGIN; -- T1

-- Étape 1 : lock sur la ligne A (id = 1)
SELECT * FROM compte
WHERE id = 1 FOR UPDATE;

-- Attente... ensuite veut verrouiller B
SELECT * FROM compte
WHERE id = 2 FOR UPDATE; -- ⏳ bloquée par T2
```

```sql
BEGIN; -- T2

-- Étape 1 : lock sur la ligne B (id = 2)
SELECT * FROM compte
WHERE id = 2 FOR UPDATE;

-- Ensuite veut verrouiller A
SELECT * FROM compte
WHERE id = 1 FOR UPDATE; -- ⏳ bloquée par T1
```

Transaction	Action	Statut
T1	Lock sur id=1	🔒 obtenu
T2	Lock sur id=2	🔒 obtenu
T1	veut lock id=2	⏳ en attente
T2	veut lock id=1	⏳ en attente

Le planificateur du SGBD (ex: PostgreSQL, MySQL InnoDB) détecte le cycle d'attente et choisit une transaction à annuler automatiquement avec un message comme : `ERROR: deadlock detected`

Conclusion Two Phase Locking

Le 2PL est une solution pour gérer les problèmes d'accès concurrent. Mais à des limitations :

- création de latence (attendre que le verrou soit libéré)
- création de deadlock

Le 2Pl est donc une stratégie pour **éviter les conflits**, regardons maintenant une stratégie pour les détecter.

Détecter les conflits (Conflict Detection)

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.

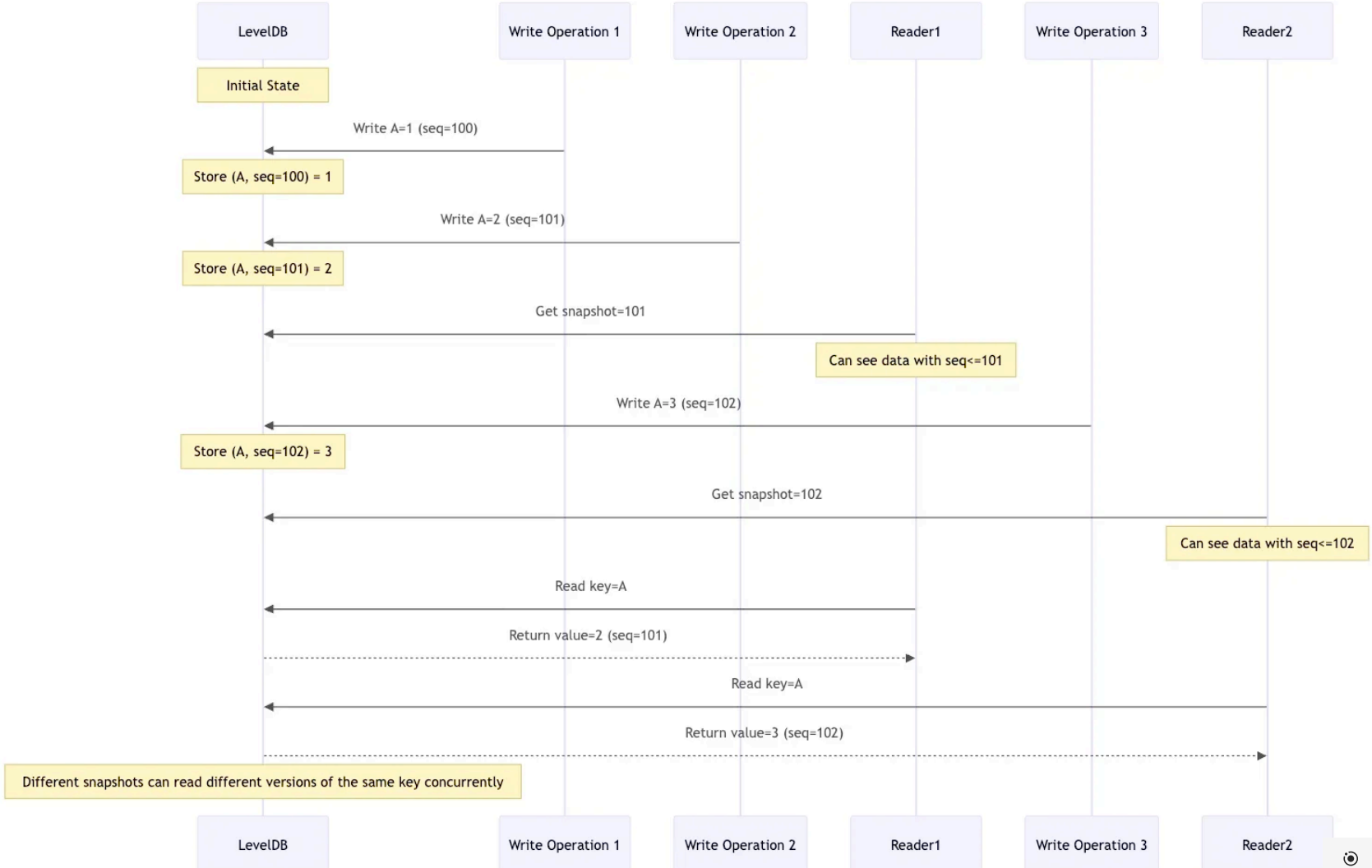
- shared lock bloque les écritures, mais permet à d'autres lecteurs d'acquérir le même verrou partagé
- exclusive lock bloque à la fois les lecteurs et les rédacteurs qui concourent pour le même verrou.

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.

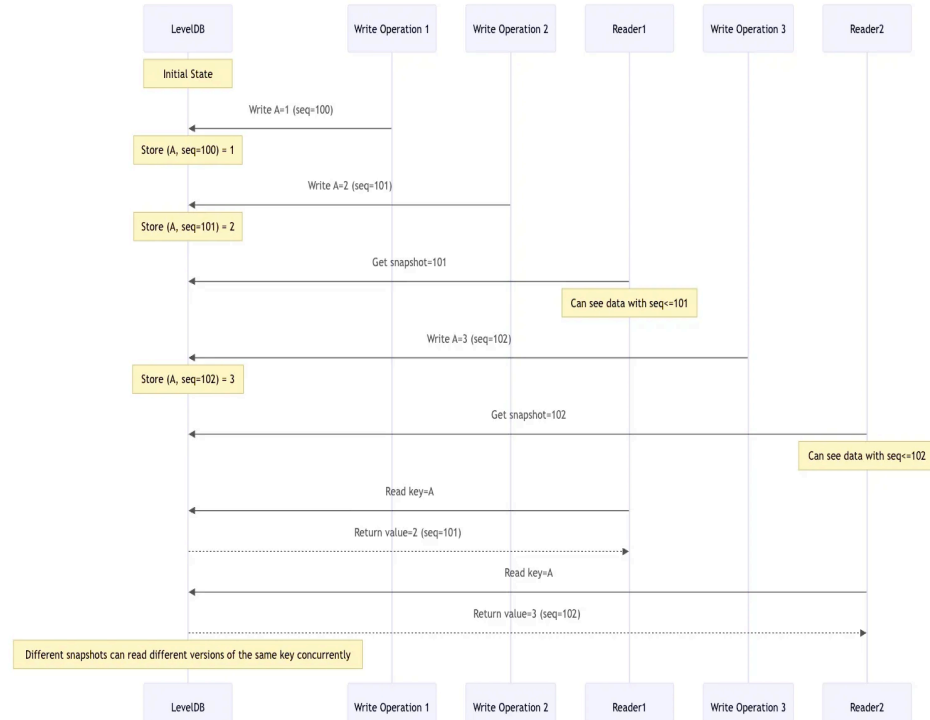
1. Chaque enregistrement de la base de données possède un numéro de version.
2. Les lectures simultanées s'effectuent sur l'enregistrement ayant le numéro de version le plus élevé.
3. Les opérations d'écriture s'effectuent sur une copie de l'enregistrement, et non sur l'enregistrement lui-même.
4. Les 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.


LevelDB
Write Operation 1
Write Operation 2
Reader1
Write Operation 3
Reader2
Initial State
Write A=1 (seq=100)
Store (A, seq=100) = 1
Write A=2 (seq=101)
Store (A, seq=101) = 2
Get snapshot=101
Can see data with seq<=101
Write A=3 (seq=102)
Store (A, seq=102) = 3
Get snapshot=102
Can see data with seq<=102
Read key=A
Return value=2 (seq=101)
Read key=A
Return value=3 (seq=102)
Different snapshots can read different versions of the same key concurrently
LevelDB
Write Operation 1
Write Operation 2
Reader1
Write Operation 3
Reader2

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.



Étape	Action	solde vu	Verrou	Résultat
Début	Valeur initiale	100	—	—
T1	`BEGIN` → `SELECT WHERE id=1`	**100**	🔓 pas de verrou	lecture snapshot
T2	`BEGIN` → `SELECT WHERE id=1`	**100**	🔓 pas de verrou	lecture snapshot
T1	calcule 100 - 30 = 70	—	—	—
T2	calcule 100 - 50 = 50	—	—	—
T1	`UPDATE ... SET solde = 70`	🔒 écrit une nouvelle version	tentative de mise à jour	
T2	`UPDATE ... SET solde = 50`	🔒 tentative concurrente	❌ conflit	

Étape	Action	solde vu	Verrou	Résultat
T1	`COMMIT`	OK	✅ validé	`solde = 70`
T2	`COMMIT`	❌ **échec : tuple modifié**	❌ annulé	erreur de conflit de mise à jour

MVCC permet aux deux transactions de lire sans se bloquer, mais :

- Au moment où T2 veut modifier, elle se rend compte que la ligne a changé depuis sa lecture initiale
- Cela provoque une erreur de concurrence, `ERROR: could not serialize access due to concurrent update`

Conclusion MVCC

MVCC permet aux deux transactions de lire sans se bloquer

Conclusion 2PL et MVCC

Les protocoles Two-Phase Locking (2PL) et Multiversion Concurrency Control (MVCC) sont deux mécanismes fondamentaux pour garantir l'isolation, l'une des quatre propriétés ACID des transactions.

Ces deux approches sont des stratégies concrètes d'implémentation des niveaux d'isolation définis par SQL (READ COMMITTED, REPEATABLE READ, SERIALIZABLE), chacune avec ses forces et ses compromis.

Introduction niveaux d’isolation et anomalies

Introduction

Objectifs des niveaux d’isolation

> ⚠ **Objectif des niveaux d'isolation**
> - 2PL ou MVCC sont des outils utilisés par le SGBD pour mettre en œuvre les niveaux d’isolation définis par le standard SQL.
> - ⇒ Ce sont les niveaux d’isolation qui définissent si une transaction bénéficie d'une isolation totale ou partielle.

Exemple (avant de rentrer dans les explications)

Les niveaux d'isolation SQL (ANSI) :

- `READ UNCOMMITTED`
- `READ COMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`

=> Ils décrivent ce qui est possible/interdit : dirty reads, non-repeatable reads, phantom reads, etc.

Exemple

Niveau d'isolation	Mécanismes utilisés (exemples)	Garanties
`READ COMMITTED`	MVCC ou 2PL léger	dirty read évité, mais pas le lost update
`REPEATABLE READ`	MVCC ou 2PL	lecture stable, mais phantom possible
`SERIALIZABLE`	2PL strict ou MVCC détection de conflits	aucune anomalie

Pourquoi permettre des anomalies ?

=> Pour la performance

Les niveaux d'isolation stricts (comme SERIALIZABLE) :

- bloquent plus souvent,
- augmentent le risque de deadlocks,
- et limitent le parallélisme.

En permettant certains types d'anomalies contrôlées, on peut :

- exécuter plus de transactions en parallèle,
- améliorer la latence et le débit global du système.

> **Note**
> Les SGBD permettent certaines anomalies par choix stratégique, pour améliorer la performance, éviter les blocages, et laisser au développeur le contrôle du compromis entre cohérence et efficacité.

Les niveaux d’isolation

Définition niveaux d'isolation
Définit un contrat SQL pour préciser quelles anomalies sont autorisées ou non

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

Les anomalies

Les différentes anomalies

- Lecture sale (dirty read) : lire une donnée non encore validée par une autre transaction.
- Lecture non répétable : lire deux fois la même donnée avec un résultat différent.
- Lecture fantôme (phantom) : une nouvelle ligne apparaît entre deux lectures avec la même requête.
- Mise à jour perdue (lost update) : une modification écrase une autre sans le savoir.

Ici, nous n'allons étudier que deux anomalies en détail

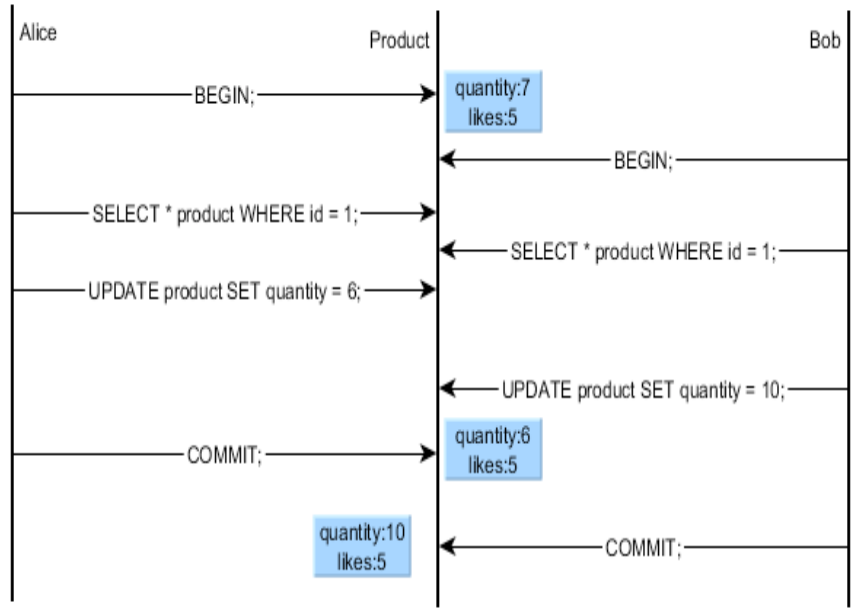
- le `LOST UPDATE` (que nous avons déjà évoqué au début des slides)
- le `PHANTOM READ`

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.

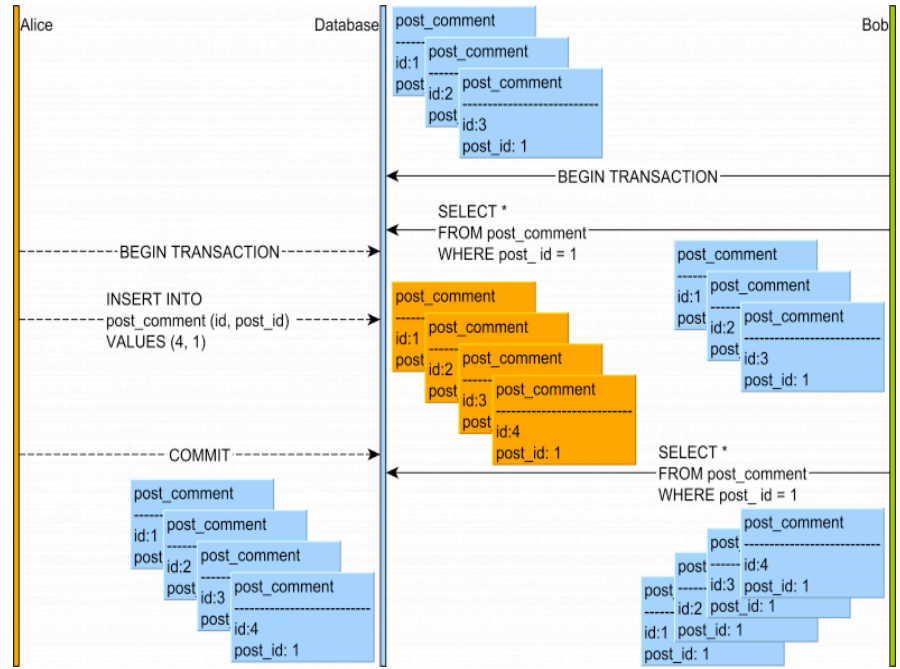
Si les deux transactions visent à modifier les mêmes colonnes, la deuxième transaction écrasera la première, entraînant ainsi la perte de la mise à jour de la première transaction.



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




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

Transaction Alice

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

=> Dans une même transaction, nous n'avons pas le même nombre de lignes retournées

Les niveaux d’isolation

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.

Quelles anomalies autorisées ?

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

Nous avons étudié *Phantom* et *Lost Update*. Nous allons voir maintenant comment les niveaux d'isolation permettent de garantir de ne pas avoir ces deux anomalies

Read Uncommitted

Définition
Avec READ UNCOMMITED on peut lire une donnée qu'une autre transaction a modifiée mais pas encore commit.

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	❌ Autorisé

Transaction Alice

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN;

UPDATE stock SET qty = 6 WHERE id = 1;
-- modification non commitée
-- par exemple un UPDATE qui mettrais plusieurs minutes

-- Alice ne valide pas encore




ROLLBACK;  -- annulation de la modification
```

Transaction Bob

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN;

SELECT qty FROM stock WHERE id = 1;
-- Résultat : 6 ❌ (valeur lue alors qu'elle n'est pas enco
```

Problème

`READ UNCOMMITTED` permet de lire une valeur qui n'a jamais officiellement existé. On a une **Lecture sale (*dirty read*)**

Read Committed

Définition
Avec READ COMMITTED on ne lit que les valeurs commit

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

Pourquoi c’est un problème ?

Parce que Bob peut faire des calculs ou des décisions en supposant que les données ne changent pas :

- calculs de stock
- facturation
- logique métier (ex : “si qty >= 10 alors…”)

et obtenir un résultat incohérent car la donnée change “en plein milieu”.

Read Committed : solution

Pour éviter ça :

- soit passer à REPEATABLE READ (lecture stable)
- (soit utiliser verrouillage pessimiste : SELECT ... FOR UPDATE)

Repeatable Read

Définition
Avec REPEATABLE READ quand on lit deux fois la même ligne dans une même transaction on obtient la même valeur

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

Problème 1

On commence avec une quantité en **stock = 10**

Transaction Alice

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

UPDATE stock SET qty = 6 WHERE id = 1;
-- Alice ne fait pas encore COMMIT




COMMIT;  -- commit de la modification
```

Transaction Bob

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

SELECT qty FROM stock WHERE id = 1;
-- Résultat : 10 ✅ (ancienne valeur, dernière version comm


SELECT qty FROM stock WHERE id = 1;
-- Résultat : 10 ✅ (même transaction => même valeur)
-- même si Alice à commit

UPDATE stock SET qty = 3 WHERE id = 1;

COMMIT -- ❌ LOST UPDATE de la valeur d'Alice
```

Problème

1. résout le problème des lectures sales + lecture non répétable
2. MAIS, Bob à écrasée silencieusement la valeur d'Alice (**lost update**)

Note importante

Dans la slide précédente nous disons que `READ COMMITTED` ne prévient pas du LOST UPDATE. Mais dans le tableau nous marquons que si.

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

Explication

Ca dépend du SGBD,

- si on ne met que `READ COMMITTED` sans rien d'autre alors Lost Update possible
- si on met `READ COMMITTED` + `SELECT ... FOR UPDATE` alors Lost Update impossible

Note importante : explication (suite)

Transaction Alice

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

SELECT qty FROM stock WHERE id = 1 FOR UPDATE;
-- ALICE DOIT POSER UN VERROU POUR QUE CA FONCTIONNE
-- 🔒 verrou exclusif posé sur la ligne id = 1

UPDATE stock SET qty = 6 WHERE id = 1;
-- Alice ne fait pas encore COMMIT


COMMIT;  -- impossible car Bob a SELECT ... FOR UPDATE
```

Transaction Bob

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE COMMITTED;
BEGIN;

SELECT qty FROM stock WHERE id = 1 FOR UPDATE;
-- ⏳ Bloqué ! ❌ Attend qu'Alice libère le verrou


-- reprise, maintenant la quantité est de 6

UPDATE stock SET qty = 3 WHERE id = 1;

COMMIT -- c'est comme si on l'avait fait de manière séquent
```

Solution

Il faut que toutes les transaction pose un `FOR UDPATE` avant de faire l'update

Problème 2

On vient de voir qu'il garantit la lecture non répétable et parfois le lost update. Mais il reste quand même un problème

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

SELECT * FROM stock WHERE qty > 5;
-- Résultat : 1 ligne (ex: id = 1, qty = 10)

-- attend pendant que Bob travaille...


SELECT * FROM stock WHERE qty > 5;
-- Résultat : ! 2 lignes (la nouvelle ligne de Bob appa
-- → PHANTOM READ détecté

COMMIT;
```

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE COMMITTED;
BEGIN;



INSERT INTO stock (id, qty) VALUES (2, 8);
-- La nouvelle ligne correspond à WHERE qty > 5

COMMIT;  -- ✅ validé
```

Problème

- Alice a lu deux fois la même requête `SELECT ... WHERE qty > 5` dans la même transaction et pourtant, elle a obtenu deux jeux de résultats différents (**phantom read**)

Repeatable Read : solution

De base Repeatable Read, permet d'éviter les lectures non répétable

- mais suivant le SGBD il permet aussi de prévenir du LOST UPDATE en utilisant des verrous `SELECT ... FOR UPDATE`

Cependant, il ne permet pas d'éviter les **phantom read** Pour éviter ça :

- passer à SERIALIZABLE (lecture stable)

Serializable

Définition
Niveau le plus élevé. Il garantit que l'exécution concurrente des transactions produit un résultat équivalent à celui qu'on obtiendrait si les transactions avaient été exécutées séquentiellement, l'une après l'autre (c'est-à-dire de façon sérialisée).

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

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;

SELECT * FROM stock WHERE qty > 5;
-- Résultat : 1 ligne (ex: id = 1, qty = 10)
-- Alice positionne un RANGE LOCK


SELECT * FROM stock WHERE qty > 5;
-- Résultat : 1 lignes

COMMIT;
```

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;



INSERT INTO stock (id, qty) VALUES (2, 8);
-- ⏳ BLOQUÉ car 8 ∈ (qty > 5)









COMMIT;
```

- Un range lock (verrou de plage) est un verrou qui protège non seulement les lignes existantes, mais aussi l'espace entre ces lignes,
- => Un range lock sur qty > 5 empêche d'insérer ou de modifier toute ligne ayant un qty > 5, tant que la transaction est active.

Conclusion

Nous avons passé les niveaux d'isolation un par un pour comprendre ce qu'ils permettaient.

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

Résumé en 4 slides des problèmes

Transaction Alice

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN;

UPDATE stock SET qty = 6 WHERE id = 1;
-- modification non commitée
-- par exemple un UPDATE qui mettrais plusieurs minutes

-- Alice ne valide pas encore




ROLLBACK;  -- annulation de la modification
```

Transaction Bob

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN;

SELECT qty FROM stock WHERE id = 1;
-- Résultat : 6 ❌ (valeur lue alors qu'elle n'est pas enco
```

Problème

`READ UNCOMMITTED` permet de lire une valeur qui n'a jamais officiellement existé. On a une **Lecture sale (*dirty read*)**

Problème 1

On commence avec une quantité en **stock = 10**

Transaction Alice

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

UPDATE stock SET qty = 6 WHERE id = 1;
-- Alice ne fait pas encore COMMIT




COMMIT;  -- commit de la modification
```

Transaction Bob

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

SELECT qty FROM stock WHERE id = 1;
-- Résultat : 10 ✅ (ancienne valeur, dernière version comm


SELECT qty FROM stock WHERE id = 1;
-- Résultat : 10 ✅ (même transaction => même valeur)
-- même si Alice à commit

UPDATE stock SET qty = 3 WHERE id = 1;

COMMIT -- ❌ LOST UPDATE de la valeur d'Alice
```

Problème

1. résout le problème des lectures sales + lecture non répétable
2. MAIS, Bob à écrasée silencieusement la valeur d'Alice (**lost update**)

Problème 2

On vient de voir qu'il garantit la lecture non répétable et parfois le lost update. Mais il reste quand même un problème

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

SELECT * FROM stock WHERE qty > 5;
-- Résultat : 1 ligne (ex: id = 1, qty = 10)

-- attend pendant que Bob travaille...


SELECT * FROM stock WHERE qty > 5;
-- Résultat : ! 2 lignes (la nouvelle ligne de Bob appa
-- → PHANTOM READ détecté

COMMIT;
```

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE COMMITTED;
BEGIN;



INSERT INTO stock (id, qty) VALUES (2, 8);
-- La nouvelle ligne correspond à WHERE qty > 5

COMMIT;  -- ✅ validé
```

Problème

- Alice a lu deux fois la même requête `SELECT ... WHERE qty > 5` dans la même transaction et pourtant, elle a obtenu deux jeux de résultats différents (**phantom read**)

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;

SELECT * FROM stock WHERE qty > 5;
-- Résultat : 1 ligne (ex: id = 1, qty = 10)
-- Alice positionne un RANGE LOCK


SELECT * FROM stock WHERE qty > 5;
-- Résultat : 1 lignes

COMMIT;
```

- Un range lock (verrou de plage) est un verrou qui protège non seulement les lignes existantes, mais aussi l'espace entre ces lignes,
- => Un range lock sur qty > 5 empêche d'insérer ou de modifier toute ligne ayant un qty > 5, tant que la transaction est active.

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;



INSERT INTO stock (id, qty) VALUES (2, 8);
-- ⏳ BLOQUÉ car 8 ∈ (qty > 5)




COMMIT;
```

Changer le niveaux d’isolation avec JDBC

Par défaut, le SGBD définit un niveau d'isolation. Mais vous avez la possibilité de la changer (très très rare de le faire)

```
Connection conn = DriverManager.getConnection(url, user, password);

// 1. Désactiver l'autocommit (pour contrôler la transaction)
conn.setAutoCommit(false);

// 2. Définir le niveau d'isolation souhaité
conn.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);

// 3. Commencer à exécuter des requêtes
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM stock WHERE qty > 5");
ResultSet rs = stmt.executeQuery();

// Thread bloqué 1 minutes
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM stock WHERE qty > 5");
ResultSet rs = stmt.executeQuery(); // TJRS la même ligne

// 4. Commit ou rollback à la fin
conn.commit();
```

- Si une autre transaction change la quantité en stock, alors ici on ne verra pas la modification
- MAIS si quelqu'un insère une ligne en plus, alors on aura un *phantom read* => une ligne en plus dans le select