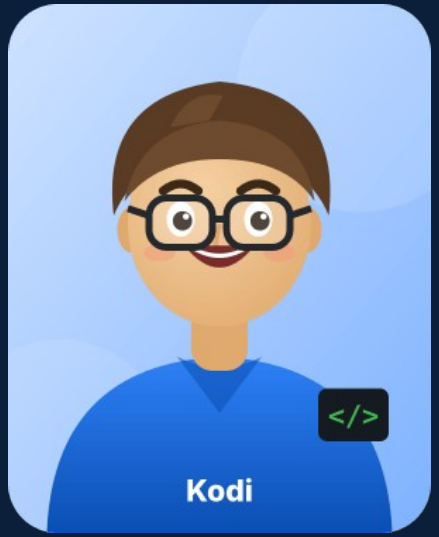
HEF-070 · MISSION 09 / 10

Isolation, verrouillage et concurrence

Quand plusieurs utilisateurs écrivent en même temps, la base doit décider qui voit quoi, et qui attend qui.



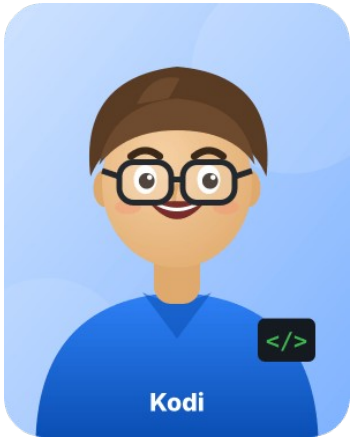


Badge : Chef de la concurrence

90 min · intermédiaire

L'histoire

Deux membres de HashBox achètent la dernière place d'une formation **au même instant**. Les deux voient « 1 place disponible », les deux cliquent, les deux reçoivent une confirmation. Il y a maintenant 2 inscrits pour 1 place.



> *« Chaque transaction était correcte seule, dit Sam. Ensemble, elles ont produit un résultat impossible. C'est un problème d'isolation. »*

Tu apprends les anomalies de concurrence, les niveaux d'isolation, les verrous et MVCC.

Ta mission

À la fin, je serai capable de…

1. décrire les anomalies de concurrence (lecture sale, non répétable, fantôme, mise à jour perdue)
2. énoncer les niveaux d'isolation et ce qu'ils empêchent
3. expliquer le principe de MVCC
4. utiliser verrouillage pessimiste et optimiste
5. reconnaître et traiter un interblocage (deadlock)

Défi d'échauffement

Défi d'échauffement

Deux personnes réservent des places de concert sur le même site. Chacune voit « il reste 1 place ». Elles cliquent à quelques millisecondes d'écart. Que veux-tu qu'il arrive ? Imagine **trois** façons de résoudre ce conflit : (a) faire attendre la deuxième, (b) refuser la deuxième en lui disant de réessayer, (c) vendre quand même et gérer le surplus plus tard. Quels avantages et inconvénients ?

Le concept

Les anomalies de concurrence

Anomalie	Description	Exemple
Lecture sale (*dirty read*)	on lit une donnée modifiée par une transaction **pas encore validée**	on voit un solde qui sera annulé
Lecture non répétable	en relisant la même ligne, on obtient une autre valeur (modifiée et validée entre-temps)	deux lectures du solde donnent deux valeurs
Lecture fantôme	en relisant avec la même condition, de **nouvelles lignes** apparaissent	`COUNT(*)` change entre deux lectures
Mise à jour perdue (*lost update*)	deux transactions lisent la même valeur et l'écrasent chacune	deux ventes enregistrées comme une

Le concept

Les anomalies de concurrence (suite)

Anomalie	Description	Exemple
Anomalie d'écriture (*write skew*)	deux transactions lisent un état, modifient des lignes différentes, et ensemble violent une règle	deux médecins de garde partent simultanément

Les niveaux d'isolation (norme SQL)

Niveau	Lecture sale	Non répétable	Fantôme
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	**empêchée**	possible	possible
REPEATABLE READ	empêchée	**empêchée**	possible (selon le SGBD)
SERIALIZABLE	empêchée	empêchée	**empêchée**

Le concept

Les niveaux d'isolation (norme SQL) (suite)

Plus le niveau est fort, plus les garanties sont grandes, plus le coût (attentes, conflits à réessayer) est élevé. **PostgreSQL** utilise `READ COMMITTED` par défaut ; **SQLite** est `SERIALIZABLE` (un seul écrivain à la fois).

MVCC : plusieurs versions

MVCC (*Multi-Version Concurrency Control*) : au lieu de bloquer les lecteurs, la base garde **plusieurs versions** d'une ligne. Chaque transaction voit un **instantané** (*snapshot*) cohérent.

- les **lecteurs ne bloquent pas les écrivains**, et inversement ;
- les anciennes versions sont nettoyées plus tard (`VACUUM` dans PostgreSQL).

Verrouillage

Le concept

Verrouillage (suite)

Approche	Principe	Quand
Pessimiste	verrouiller dès la lecture : `SELECT ... FOR UPDATE`	conflits fréquents, opérations courtes
Optimiste	ne pas verrouiller ; à l'écriture vérifier que personne n'a modifié (colonne `version`) et **réessayer** sinon	conflits rares

Verrouillage optimiste :

```
UPDATE places SET reservee = 1, version = version + 1
WHERE id = 7 AND version = 3;        -- 0 ligne modifiée = quelqu'un d'autre est p
   assé avant
```

Ne pas lire puis écrire : préfère l'opération atomique `UPDATE stock SET n = n - 1 WHERE n > 0` à « lire n, puis écrire n−1 ».

Le concept

L'interblocage (deadlock)

Deux transactions s'attendent mutuellement :

```
T1 : verrouille A, puis demande B
T2 : verrouille B, puis demande A
```

La base détecte le cycle et **annule** l'une des deux. Prévention : verrouiller **toujours dans le même ordre**, garder les transactions courtes, réessayer en cas d'échec.

Simuler une mise à jour perdue

Le concept

Simuler une mise à jour perdue (suite)

```
import threading, time

stock = {"n": 1}
def acheter_naif(nom, resultats):
    n = stock["n"]                      # lecture
    time.sleep(0.01)                    # un autre acheteur passe ici
    if n > 0:
        stock["n"] = n - 1              # écriture basée sur une valeur périmée
        resultats.append(nom)

res = []
fils = [threading.Thread(target=acheter_naif, args=(nom, res)) for nom in ("Ada",
   "Bob")]
for f in fils: f.start()
for f in fils: f.join()
print(res, stock)    # ['Ada', 'Bob'] {'n': 0}  : deux acheteurs pour une place
```

Exemple pas à pas

Problème : corriger la double réservation avec trois techniques.

```
import sqlite3

def nouvelle_base():
    bd = sqlite3.connect(":memory:")
    bd.executescript('''
    CREATE TABLE places (id INTEGER PRIMARY KEY, client TEXT, version INTEGER NOT
   NULL DEFAULT 0);
    INSERT INTO places (id) VALUES (1);
    ''')
    bd.commit()
    return bd

# 1. Opération atomique conditionnelle : un seul UPDATE qui vérifie ET modifie
def reserver_atomique(bd, client):
    cur = bd.execute("UPDATE places SET client = ? WHERE id = 1 AND client IS NULL
   ", (client,))
```

Exemple pas à pas (suite)

```
    bd.commit()
    return cur.rowcount == 1

bd = nouvelle_base()
print(reserver_atomique(bd, "Ada"), reserver_atomique(bd, "Bob"))       # True Fa
   lse

# 2. Verrouillage optimiste : on lit la version, puis on écrit seulement si elle n
   'a pas changé
def reserver_optimiste(bd, client, version_lue):
    cur = bd.execute("UPDATE places SET client = ?, version = version + 1 WHERE id
    = 1 AND client IS NULL AND version = ?",
                     (client, version_lue))
    bd.commit()
    return cur.rowcount == 1

bd = nouvelle_base()
v = bd.execute("SELECT version FROM places WHERE id = 1").fetchone()[0]      # les
    deux lisent version 0
```

Exemple pas à pas (suite)

```
print(reserver_optimiste(bd, "Ada", v), reserver_optimiste(bd, "Bob", v))    # Tru
   e False

# 3. Transaction exclusive (verrouillage pessimiste, version SQLite)
def reserver_pessimiste(bd, client):
    bd.execute("BEGIN IMMEDIATE")              # verrou d'écriture pris dès le dé
   but
    try:
        libre = bd.execute("SELECT client IS NULL FROM places WHERE id = 1").fetch
   one()[0]
        if libre:
            bd.execute("UPDATE places SET client = ? WHERE id = 1", (client,))
        bd.execute("COMMIT")
        return bool(libre)
    except Exception:
        bd.execute("ROLLBACK")
        raise

bd = nouvelle_base()
```

Exemple pas à pas (suite)

```
print(reserver_pessimiste(bd, "Ada"), reserver_pessimiste(bd, "Bob"))       # Tru
   e False
```

Lecture :

1. **Atomique** : la condition et la modification sont dans la **même requête**. Le plus simple, à préférer quand c'est possible.
2. **Optimiste** : le deuxième `UPDATE` ne trouve plus la version attendue : `rowcount = 0`, le client doit **réessayer**.
3. **Pessimiste** : le verrou empêche tout autre écrivain pendant la lecture suivie de l'écriture.

Deux connexions qui se bloquent (SQLite) :

Exemple pas à pas (suite)

```
import os, sqlite3, tempfile

with tempfile.TemporaryDirectory() as d:
    chemin = os.path.join(d, "t.db")
    a = sqlite3.connect(chemin, timeout=0.2, isolation_level=None)
    b = sqlite3.connect(chemin, timeout=0.2, isolation_level=None)
    a.execute("CREATE TABLE t (x INTEGER)")
    a.execute("BEGIN IMMEDIATE")                # a prend le verrou d'écriture
    try:
        b.execute("BEGIN IMMEDIATE")            # b doit attendre, puis abandonn
   e
    except sqlite3.OperationalError as e:
        print("b bloqué :", e)                  # database is locked
    a.execute("COMMIT")
```

À toi, avec aide

Pour chaque scénario, dis quelle anomalie se produit et quelle solution simple appliquer :

1. Deux administrateurs ouvrent la fiche d'un membre, modifient chacun un champ différent, et enregistrent : le second écrase le premier.
2. Un rapport lit deux fois `SELECT SUM(solde)` pendant qu'un virement s'exécute et obtient des totaux différents.
3. Un compteur de visites est incrémenté par 20 processus avec « lire n, écrire n+1 ».
4. Deux transactions verrouillent les comptes A puis B, et B puis A.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Le 1 est une mise à jour perdue : colonne `version` ou ne mettre à jour que le champ modifié.

Indice 2

Le 2 est une lecture non répétable : un seul SELECT ou un niveau d'isolation plus fort.

Indice 3

Le 4 est un interblocage : ordre de verrouillage fixe.

Réponse

1. **Mise à jour perdue** → verrouillage optimiste (`version`) ou `UPDATE` ne portant que sur la colonne modifiée.
2. **Lecture non répétable** → faire un seul `SELECT`, ou une transaction en `REPEATABLE READ`/snapshot.
3. **Mise à jour perdue** → `UPDATE compteur SET n = n + 1` (atomique) au lieu de lire puis écrire.
4. **Interblocage** → toujours verrouiller dans le même ordre (par identifiant croissant), transactions courtes, réessai en cas d'annulation par la base.

À toi, en solo

Simulation de MVCC. Implémente une mini-base clé-valeur multi-versions :

- chaque écriture crée une nouvelle version avec un numéro de transaction ;
- `debut()` renvoie un instantané (numéro de version actuel) ;
- `lire(snapshot, cle)` renvoie la dernière version **visible** par ce snapshot ;
- `ecrire(snapshot, cle, valeur)` échoue (conflit) si la clé a été modifiée depuis le snapshot (premier validateur gagne).

Teste : une transaction longue lit toujours la même valeur malgré des écritures concurrentes, et deux écritures concurrentes sur la même clé produisent un conflit pour la seconde.

À toi, en solo · réponse

Pistes de correction

```
class MVCC:
    def __init__(self):
        self.versions = {}       # cle -> liste de (numero, valeur)
        self.horloge = 0

    def debut(self):
        return self.horloge

    def lire(self, snapshot, cle):
        visibles = [(n, v) for n, v in self.versions.get(cle, []) if n <= snapshot
   ]
        return visibles[-1][1] if visibles else None

    def ecrire(self, snapshot, cle, valeur):
        dernieres = self.versions.get(cle, [])
        if dernieres and dernieres[-1][0] > snapshot:
            raise RuntimeError("conflit d'écriture : la clé a changé depuis le sna
   pshot")
        self.horloge += 1
```

Pistes de correction (suite)

```
        self.versions.setdefault(cle, []).append((self.horloge, valeur))

bd = MVCC()
s0 = bd.debut(); bd.ecrire(s0, "solde", 100)

long = bd.debut()                                  # transaction longue
print(bd.lire(long, "solde"))                      # 100

t1 = bd.debut(); t2 = bd.debut()
bd.ecrire(t1, "solde", 150)                        # t1 valide en premier
print(bd.lire(long, "solde"))                      # toujours 100 : instantané coh
   érent
try:
    bd.ecrire(t2, "solde", 80)                     # t2 avait lu l'ancien état
except RuntimeError as e:
    print(e)
print(bd.lire(bd.debut(), "solde"))                # 150
```

Pistes de correction (suite)

Les lecteurs ne bloquent personne et voient un état cohérent ; seul le second écrivain d'une même clé est refusé et doit réessayer. C'est le principe du niveau REPEATABLE READ/SNAPSHOT de PostgreSQL.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Lire puis écrire (read-modify-write)	Mises à jour perdues	Opération atomique en une requête
Verrous pris dans un ordre différent	Interblocage	Ordre fixe (par exemple par identifiant)
Pas de réessai en cas de conflit	Erreurs rares mais réelles en production	Boucle de réessai avec limite
Isolation maximale partout	Beaucoup d'abandons, performances en baisse	Choisir le niveau selon le besoin

Mini-boss

Mini-boss

Banque concurrente. Simule 8 threads qui font 500 virements aléatoires chacun entre 10 comptes (SQLite en fichier temporaire, une connexion par thread, `timeout` suffisant).

1. Version naïve : lire les soldes puis écrire. Montre que la **somme totale** change (argent créé ou détruit).
2. Version correcte : `UPDATE ... SET solde = solde - ? WHERE id = ? AND solde >= ?` dans une transaction `BEGIN IMMEDIATE`, avec réessai en cas de `database is locked`.
3. Vérifie l'invariant : la somme totale reste constante et aucun solde n'est négatif.
4. Ajoute un détecteur de deadlock simulé avec deux verrous Python pris dans des ordres opposés, et montre la correction (ordre global).
5. Mesure le débit (virements/seconde) des deux versions.

Mini-boss · réponse

Pistes de correction

Un bon travail : la version naïve utilise `SELECT solde ...` puis `UPDATE ... SET solde = ?` avec la valeur lue ; on observe que la somme finale s'écarte de 10 × solde initial. La version correcte utilise `isolation_level=None`, `BEGIN IMMEDIATE`, deux `UPDATE` atomiques avec condition `solde >= montant` (et on annule si le premier ne modifie aucune ligne), et réessaie sur `OperationalError`. L'invariant est vérifié par `SELECT SUM(solde)` et `SELECT COUNT(*) WHERE solde < 0`. Le deadlock est reproduit avec `threading.Lock` (`acquire(timeout=...)` pour le détecter) puis corrigé par `sorted((a, b))` avant d'acquérir. Le débit de la version correcte est plus faible (écrivains sérialisés) mais le résultat est juste.

Quiz éclair

Quiz éclair

1. Une lecture non répétable est :

A. une lecture trop lente
B. une donnée qui change entre deux lectures dans la même transaction
C. une lecture interdite

2. MVCC permet :

A. de supprimer les index
B. de chiffrer la base
C. aux lecteurs de ne pas bloquer les écrivains

3. Pour éviter une mise à jour perdue sur un compteur, on utilise :

A. UPDATE compteur SET n = n + 1

Quiz éclair

Quiz éclair (suite)

B. lire n puis écrire n+1
C. un fichier texte

4. 🔁 *Révision mission 08* : L'atomicité signifie :

A. des transactions rapides
B. tout ou rien
C. des données chiffrées

5. 🔁 *Révision mission 08* : Un ROLLBACK :

A. valide la transaction
B. supprime la table
C. annule les modifications de la transaction en cours

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

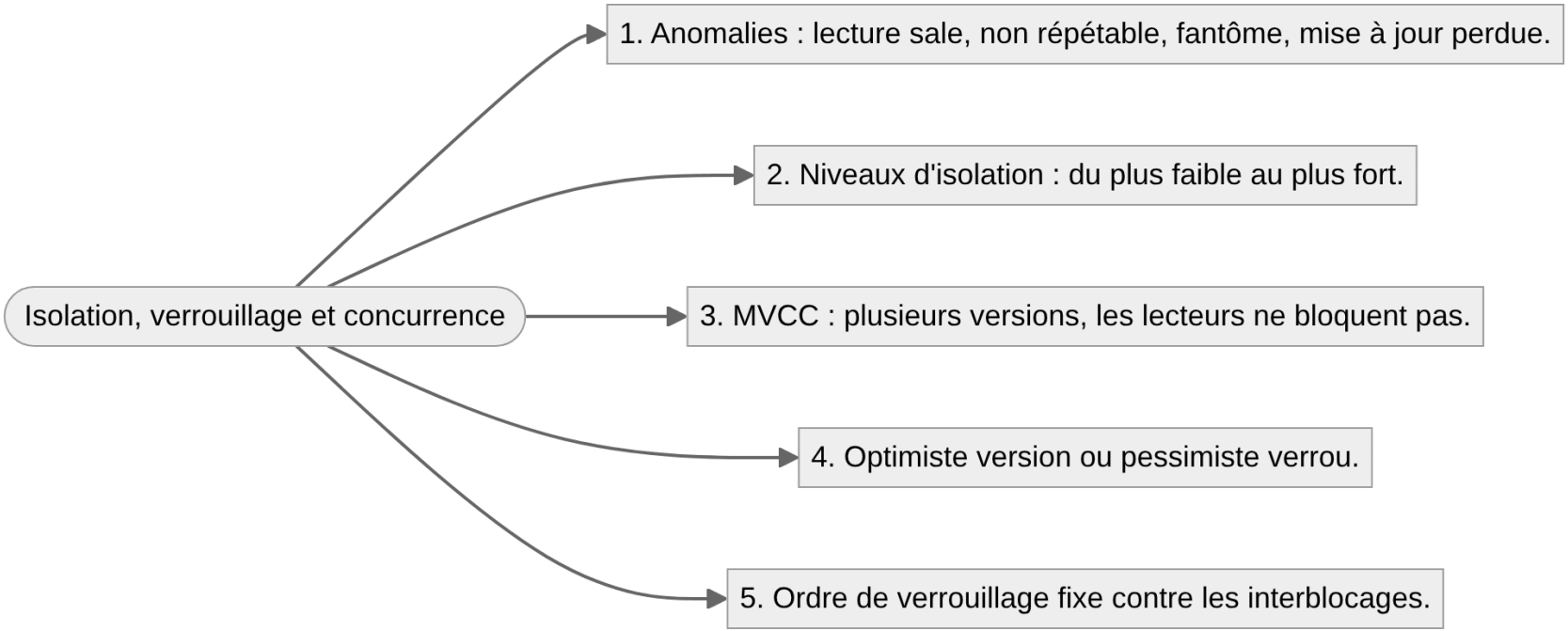
1. Dans quelles situations de ta vie « lis-tu puis écris-tu » sans savoir si l'information a changé entre-temps ?
2. Entre attendre (pessimiste) et réessayer (optimiste), laquelle préfères-tu et pourquoi ?

À retenir

1. Anomalies : lecture sale, non répétable, fantôme, mise à jour perdue.
2. Niveaux d'isolation : du plus faible au plus fort.
3. MVCC : plusieurs versions, les lecteurs ne bloquent pas.
4. Optimiste (version) ou pessimiste (verrou).
5. Ordre de verrouillage fixe contre les interblocages.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Chef de la concurrence

Prochaine mission : PostgreSQL en pratique

