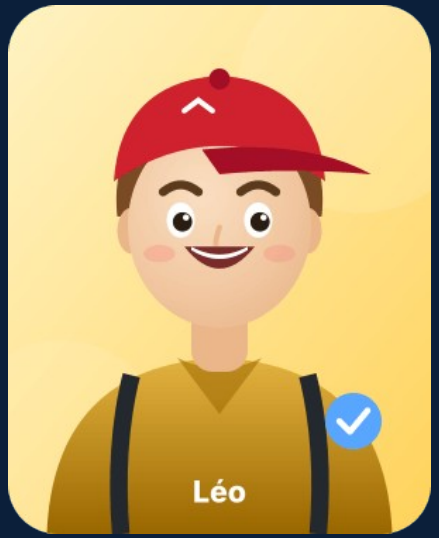
HEF-070 · MISSION 08 / 10

Transactions et ACID

Un virement, c'est deux opérations. Elles doivent réussir ensemble, ou pas du tout.





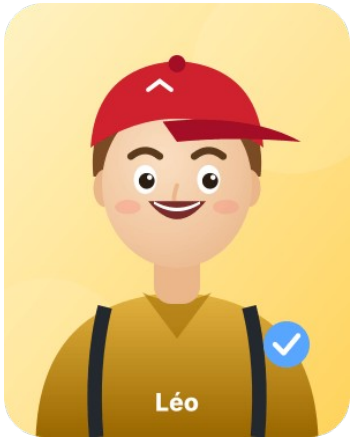
Badge : Garant de la fiabilité

85 min · intermédiaire

L'histoire

Kodi implémente un transfert de points entre deux membres : retirer 50 points à Ada, ajouter 50 à Bob. Un jour, le serveur plante entre les deux opérations : 50 points ont disparu.

> *« Tu viens de rencontrer la raison d'être des transactions, dit Sam. Elles regroupent plusieurs opérations en un tout indivisible. »*



Tu apprends ce que garantit une transaction, et comment l'utiliser correctement.

Ta mission

À la fin, je serai capable de…

1. définir une transaction et son cycle de vie (BEGIN, COMMIT, ROLLBACK)
2. expliquer les quatre propriétés ACID
3. écrire un transfert atomique avec gestion d'erreur
4. utiliser des points de sauvegarde (SAVEPOINT)
5. expliquer la durabilité (journal, WAL)

Défi d'échauffement

Défi d'échauffement

Tu retires 100 € au distributeur et la machine plante exactement après avoir débité ton compte mais avant de te donner les billets. Que se passe-t-il pour toi ? Pour la banque ? Quelles garanties veux-tu : tout se passe, ou rien ne se passe ? Que se passe-t-il si deux personnes retirent de l'argent du **même compte** en même temps ?

Le concept

Qu'est-ce qu'une transaction ?

Une **transaction** est une suite d'opérations traitée comme **une seule unité** : tout est validé, ou rien.

```
BEGIN;
UPDATE comptes SET solde = solde - 50 WHERE id = 1;
UPDATE comptes SET solde = solde + 50 WHERE id = 2;
COMMIT;        -- valide tout    (ou ROLLBACK pour tout annuler)
```

Les propriétés ACID

Lettre	Propriété	Garantie	Exemple
A	**Atomicité**	tout ou rien	si le crédit échoue, le débit est annulé
C	**Cohérence**	la base passe d'un état valide à un état valide	les contraintes (`CHECK solde >= 0`) restent vraies

Le concept

Les propriétés ACID (suite)

Lettre	Propriété	Garantie	Exemple
I	**Isolation**	les transactions simultanées ne se perturbent pas	un virement en cours n'est pas vu à moitié
D	**Durabilité**	ce qui est validé survit à une panne	après COMMIT, même une coupure de courant ne le perd pas

Comment c'est réalisé ?

- **Atomicité et durabilité** : un **journal** (*write-ahead log*, WAL). On écrit d'abord l'intention dans un journal sur disque, puis on modifie les données. En cas de panne, on **rejoue** ou on **annule** à partir du journal.
- **Isolation** : verrous et/ou **MVCC** (versions multiples) : mission suivante.
- **Cohérence** : contraintes + application correcte.

Le concept

Python et les transactions

```
import sqlite3

bd = sqlite3.connect(":memory:")
bd.execute("CREATE TABLE comptes (id INTEGER PRIMARY KEY, solde INTEGER NOT NULL C
   HECK (solde >= 0))")
bd.executemany("INSERT INTO comptes VALUES (?, ?)", [(1, 100), (2, 20)])
bd.commit()

def virer(bd, source, destination, montant):
    try:
        with bd:                                    # BEGIN ... COMMIT / ROLLBACK
   si exception
            bd.execute("UPDATE comptes SET solde = solde - ? WHERE id = ?", (monta
   nt, source))
            bd.execute("UPDATE comptes SET solde = solde + ? WHERE id = ?", (monta
   nt, destination))
```

Le concept

Python et les transactions (suite)

```
        return "ok"
    except sqlite3.IntegrityError as e:
        return f"annulé : {e}"

print(virer(bd, 1, 2, 30), bd.execute("SELECT * FROM comptes").fetchall())    # ok
   , [(1, 70), (2, 50)]
print(virer(bd, 1, 2, 500), bd.execute("SELECT * FROM comptes").fetchall())   # an
   nulé, soldes inchangés
```

Points de sauvegarde

Un **SAVEPOINT** permet d'annuler une partie de la transaction sans tout perdre.

```
BEGIN;
INSERT ...;
SAVEPOINT etape2;
INSERT ...;            -- erreur
```

Le concept

Points de sauvegarde (suite)

```
ROLLBACK TO etape2;    -- annule seulement après le point de sauvegarde
COMMIT;
```

Bonnes pratiques

- Transactions **courtes** : elles retiennent des verrous / des ressources.
- Ne jamais laisser une transaction **ouverte** en attendant une action de l'utilisateur.
- Faire les opérations qui ne sont pas transactionnelles (envoi d'e-mail, appel externe) **après** le COMMIT.
- Gérer les **échecs** : réessayer en cas de conflit, ne pas avaler l'exception.

Durabilité en pratique

Le concept

Durabilité en pratique (suite)

Après `COMMIT`, la base s'assure que le journal est écrit sur disque (`fsync`). Désactiver cette garantie (réglages de performance) peut faire **perdre** des transactions validées en cas de panne.

Exemple pas à pas

Problème : prouver que la transaction protège contre une panne au milieu d'un transfert.

```
import sqlite3

class PanneSimulee(Exception):
    pass

bd = sqlite3.connect(":memory:")
bd.execute("CREATE TABLE comptes (id INTEGER PRIMARY KEY, solde INTEGER NOT NULL C
   HECK (solde >= 0))")
bd.executemany("INSERT INTO comptes VALUES (?, ?)", [(1, 100), (2, 20)])
bd.commit()

def transfert(montant, panne_au_milieu=False, transactionnel=True):
    if transactionnel:
        try:
            with bd:
                bd.execute("UPDATE comptes SET solde = solde - ? WHERE id = 1", (m
```

Exemple pas à pas

Exemple pas à pas (suite)

```
   ontant,))
                if panne_au_milieu:
                    raise PanneSimulee()
                bd.execute("UPDATE comptes SET solde = solde + ? WHERE id = 2", (m
   ontant,))
            except PanneSimulee:
                pass
        else:
            # sans transaction explicite : chaque opération validée séparément
            bd.execute("UPDATE comptes SET solde = solde - ? WHERE id = 1", (montant,)
   ); bd.commit()
            if panne_au_milieu:
                return
            bd.execute("UPDATE comptes SET solde = solde + ? WHERE id = 2", (montant,)
   ); bd.commit()

total = lambda: bd.execute("SELECT SUM(solde) FROM comptes").fetchone()[0]
print("total initial :", total())                                    # 120
transfert(30, panne_au_milieu=True, transactionnel=True)
```

Exemple pas à pas (suite)

```
print("après panne avec transaction :", bd.execute("SELECT * FROM comptes").fetcha
   ll(), total())    # inchangé
transfert(30, panne_au_milieu=True, transactionnel=False)
print("après panne sans transaction :", bd.execute("SELECT * FROM comptes").fetcha
   ll(), total())    # 30 points perdus
```

Lecture : avec la transaction, la panne annule le débit : le total reste **120**. Sans transaction, le débit est déjà validé quand la panne survient : **30 points ont disparu**, le total devient 90. C'est l'**atomicité**.

Durabilité : une base sur fichier survit à la fermeture.

```
import os, sqlite3, tempfile

with tempfile.TemporaryDirectory() as d:
    chemin = os.path.join(d, "test.db")
    bd1 = sqlite3.connect(chemin)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    bd1.execute("CREATE TABLE t (x INTEGER)")
    with bd1:
        bd1.execute("INSERT INTO t VALUES (42)")
    bd1.close()                                  # simule l'arrêt du programme
    bd2 = sqlite3.connect(chemin)
    print(bd2.execute("SELECT x FROM t").fetchall())     # [(42,)] : validé = cons
ervé
    bd2.close()
```

À toi, avec aide

À toi, avec aide

Écris `acheter(bd, membre_id, produit_id, quantite)` qui, **dans une seule transaction** :

1. vérifie le stock suffisant ;
2. décrémente le stock ;
3. crée la commande ;
4. débite le solde du membre.

Si une étape échoue (stock insuffisant, solde insuffisant), **rien** ne doit être modifié. Teste les trois cas : succès, stock insuffisant, solde insuffisant.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise des contraintes `CHECK (stock >= 0)` et `CHECK (solde >= 0)` : l'erreur déclenche le ROLLBACK.

Indice 2

`with bd:` s'occupe du COMMIT / ROLLBACK.

Indice 3

Après chaque test, compare l'état de la base à l'état attendu.

À toi, avec aide · réponse

Réponse

```
import sqlite3

bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE produits (id INTEGER PRIMARY KEY, nom TEXT, prix INTEGER NOT NULL, st
   ock INTEGER NOT NULL CHECK (stock >= 0));
CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT, solde INTEGER NOT NULL CHE
   CK (solde >= 0));
CREATE TABLE commandes (id INTEGER PRIMARY KEY, membre_id INTEGER, produit_id INTE
   GER, quantite INTEGER);
INSERT INTO produits VALUES (1, 'Stylo', 10, 5);
INSERT INTO membres VALUES (1, 'Ada', 100);
''')
bd.commit()

def acheter(bd, membre_id, produit_id, quantite):
    try:
        with bd:
            prix = bd.execute("SELECT prix FROM produits WHERE id = ?", (produit_i
```

Réponse (suite)

```
    d,)).fetchone()[0]
            bd.execute("UPDATE produits SET stock = stock - ? WHERE id = ?", (quan
    tite, produit_id))
            bd.execute("INSERT INTO commandes (membre_id, produit_id, quantite) VA
    LUES (?, ?, ?)", (membre_id, produit_id, quantite))
            bd.execute("UPDATE membres SET solde = solde - ? WHERE id = ?", (prix
    * quantite, membre_id))
        return "ok"
     except sqlite3.IntegrityError as e:
        return f"annulé ({e})"

etat = lambda: (bd.execute("SELECT stock FROM produits").fetchone()[0], bd.execute
    ("SELECT solde FROM membres").fetchone()[0],
                bd.execute("SELECT COUNT(*) FROM commandes").fetchone()[0])
print(acheter(bd, 1, 1, 3), etat())        # ok, (2, 70, 1)
print(acheter(bd, 1, 1, 4), etat())        # stock insuffisant : annulé, (2, 70, 1
    )
bd.execute("UPDATE produits SET stock = 100"); bd.commit()
print(acheter(bd, 1, 1, 8), etat())        # solde insuffisant (80 > 70) : annulé,
```

À toi, avec aide · réponse

Réponse (suite)

```
    (100, 70, 1)
```

À toi, en solo

Journal de transactions (write-ahead log) miniature. Implémente une petite base clé-valeur en mémoire avec :

- `begin()`, `set(cle, valeur)`, `commit()`, `rollback()` ;
- un **journal** (liste) où chaque `commit` écrit d'abord les opérations puis un marqueur `COMMIT` ;
- `recuperer(journal)` qui reconstruit l'état après un **crash simulé** : seules les transactions terminées par `COMMIT` sont rejouées, les autres sont ignorées.

Teste : deux transactions validées, une en cours au moment du crash.

À toi, en solo · réponse

Pistes de correction

```
class BaseKV:
    def __init__(self):
        self.donnees = {}
        self.journal = []            # simule un fichier sur disque
        self._tx = None
        self._id = 0

    def begin(self):
        self._id += 1
        self._tx = {"id": self._id, "ops": []}

    def set(self, cle, valeur):
        self._tx["ops"].append((cle, valeur))

    def commit(self):
        for cle, valeur in self._tx["ops"]:
            self.journal.append(("SET", self._tx["id"], cle, valeur))     # d'abor
   d le journal
        self.journal.append(("COMMIT", self._tx["id"]))
```

Pistes de correction (suite)

```
        for cle, valeur in self._tx["ops"]:                                  # puis
   les données
            self.donnees[cle] = valeur
        self._tx = None

    def rollback(self):
        self._tx = None

def recuperer(journal):
    valides = {e[1] for e in journal if e[0] == "COMMIT"}
    etat = {}
    for e in journal:
        if e[0] == "SET" and e[1] in valides:
            etat[e[2]] = e[3]
    return etat

bd = BaseKV()
bd.begin(); bd.set("a", 1); bd.set("b", 2); bd.commit()
bd.begin(); bd.set("a", 10); bd.commit()
```

Pistes de correction (suite)

```
bd.begin(); bd.set("b", 999)
# crash ici : la 3e transaction n'a jamais été validée ; on ne perd que la mémoire
   , pas le journal
journal_sur_disque = list(bd.journal)
print(recuperer(journal_sur_disque))      # {'a': 10, 'b': 2}
```

Même si le crash survient **pendant** `commit` (après quelques lignes du journal mais avant le marqueur `COMMIT`), la transaction est ignorée : c'est l'atomicité par le journal. C'est le principe du WAL de PostgreSQL et de SQLite.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Plusieurs opérations liées sans transaction	Un état incohérent en cas d'erreur	Les mettre dans une même transaction
Transaction trop longue	Verrous gardés, concurrence dégradée	Garder les transactions courtes
Appel externe dans la transaction	Impossible à annuler (e-mail envoyé)	Le faire après le COMMIT
Ignorer les échecs de COMMIT	Données perdues sans le savoir	Gérer les exceptions, réessayer

Mini-boss

Mini-boss

Système de réservation de billets. Une salle a 100 places. Écris `reserver(bd, client, nb_places)` qui garantit :

1. jamais plus de 100 places vendues ;
2. une réservation de 3 places est atomique : soit 3 sont attribuées, soit aucune ;
3. un journal d'audit enregistre chaque tentative (réussie ou refusée) **dans la même transaction** pour la réussite, et **hors transaction** pour le refus (afin qu'il ne soit pas annulé).

Teste : 40 réservations de 3 places (séquentielles), puis des cas limites (exactement 100, 101), et une panne simulée entre l'écriture des places et l'écriture de la facture.

Mini-boss · réponse

Pistes de correction

Un bon travail : tables `places(numero PRIMARY KEY, client)`, `factures`, `audit` ; `reserver` ouvre une transaction (`BEGIN IMMEDIATE`), sélectionne N places libres (`WHERE client IS NULL LIMIT N`), refuse si moins de N sont trouvées, les attribue, écrit la facture, puis COMMIT. Le refus écrit l'audit **après** le ROLLBACK (autre connexion ou après `with bd:`). La panne simulée est une exception levée entre l'attribution des places et la facture : après ROLLBACK, aucune place n'est attribuée et aucune facture n'existe. Le test final vérifie que `COUNT(places attribuées) = SUM(places des factures)` et ≤ 100.

Quiz éclair

Quiz éclair

1. L'atomicité signifie :

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

2. Après un COMMIT, la durabilité garantit que :

A. les données survivent à une panne
B. les données sont dupliquées
C. les données sont chiffrées

3. Un ROLLBACK :

A. valide la transaction

Quiz éclair

Quiz éclair (suite)

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

4. 🔁 *Révision mission 07* : Un index composite `(a, b)` sert pour :

A. WHERE b = ? seul
B. aucune requête
C. WHERE a = ? (préfixe gauche)

5. 🔁 *Révision mission 07* : `SCAN` dans un plan d'exécution signifie :

A. balayage complet de la table
B. recherche par index
C. jointure

Quiz éclair · réponse

Réponses

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

Je réfléchis

Je réfléchis

1. Quelles actions de ta vie doivent se faire « toutes ensemble ou pas du tout » ?
2. Pourquoi faut-il attendre le COMMIT avant d'envoyer un e-mail de confirmation ?

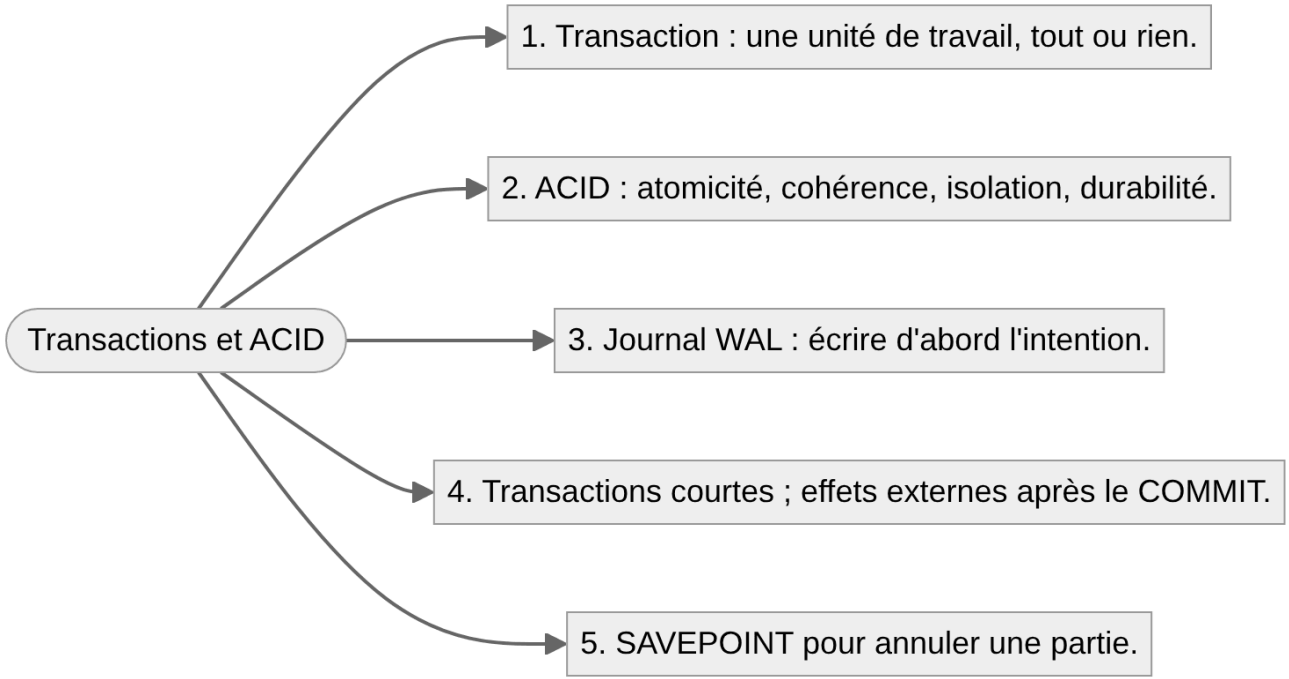
À retenir

À retenir

1. Transaction : une unité de travail, tout ou rien.
2. ACID : atomicité, cohérence, isolation, durabilité.
3. Journal (WAL) : écrire d'abord l'intention.
4. Transactions courtes ; effets externes après le COMMIT.
5. SAVEPOINT pour annuler une partie.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Garant de la fiabilité

Prochaine mission : Isolation, verrouillage et concurrence

