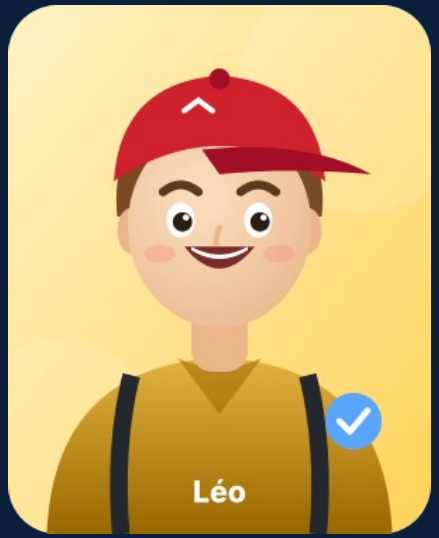
HEF-070 · MISSION 04 / 10

SQL d'écriture : modifier les données

INSERT, UPDATE, DELETE : trois verbes puissants, à utiliser avec une extrême prudence.





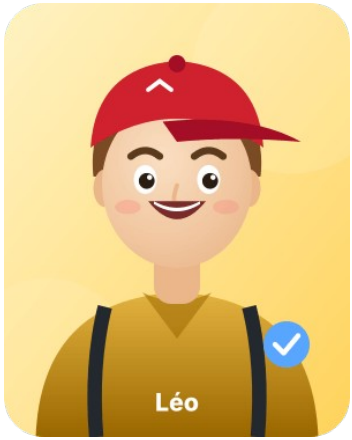
Badge : Éditeur de données

75 min · intermédiaire

L'histoire

Un vendredi, Kodi lance `DELETE FROM membres` pour supprimer un test. Il a oublié le `WHERE`. Tous les membres disparaissent. Heureusement, une sauvegarde existe.

> *« L'écriture est irréversible tant qu'on n'a pas de filet, dit Sam. Avant chaque modification : un SELECT avec la même condition, une transaction, et une sauvegarde. »*



Tu apprends à modifier les données proprement et sans te mettre en danger.

Ta mission

À la fin, je serai capable de…

1. utiliser INSERT, UPDATE et DELETE avec leurs clauses
2. insérer en masse avec des paramètres
3. utiliser l'upsert (INSERT … ON CONFLICT)
4. prévisualiser une modification avec un SELECT
5. utiliser une transaction pour pouvoir annuler

Défi d'échauffement

Défi d'échauffement

Tu dois supprimer tous les élèves qui ont quitté l'école d'un fichier de 3 000 lignes. Quelles précautions prends-tu **avant** de supprimer ? Comment vérifies-tu que tu supprimeras les bonnes lignes ? Comment pourrais-tu revenir en arrière si tu te trompes ?

Le concept

INSERT

```
INSERT INTO membres (nom, ville) VALUES ('Ada', 'Lyon');
INSERT INTO membres (nom, ville) VALUES ('Bob', 'Lille'), ('Cléo', 'Nice');
INSERT INTO archive SELECT * FROM membres WHERE actif = 0;       -- depuis une req
   uête
```

UPDATE

```
UPDATE membres SET ville = 'Paris' WHERE id = 3;
UPDATE ventes SET prix_centimes = prix_centimes * 110 / 100 WHERE produit = 'Sac';
```

DELETE

Le concept

DELETE (suite)

```
DELETE FROM membres WHERE id = 3;
```

⚠ ***Sans `WHERE`, UPDATE et DELETE concernent TOUTES les lignes.***

La règle d'or

1. Écrire un `SELECT` avec **la même condition** et vérifier les lignes concernées.
2. Ouvrir une **transaction**.
3. Exécuter la modification.
4. Vérifier le nombre de lignes affectées (`rowcount`).
5. **Valider** (`COMMIT`) ou **annuler** (`ROLLBACK`).

Le concept

Upsert : insérer ou mettre à jour

```
INSERT INTO scores (membre_id, points) VALUES (1, 10)
ON CONFLICT(membre_id) DO UPDATE SET points = points + excluded.points;
```

Si la clé existe déjà, on met à jour au lieu d'échouer. Utile pour les compteurs et les synchronisations.

Les paramètres et l'insertion en masse

```
bd.executemany("INSERT INTO membres (nom, ville) VALUES (?, ?)", [("Ada", "Lyon"),
    ("Bob", "Lille")])
```

C'est plus rapide et plus sûr qu'une boucle de requêtes construites à la main.

Le concept

Suppression logique

Au lieu de supprimer, on peut **marquer** : une colonne `supprime_le` (date) ou `actif = 0`. Avantages : récupération possible, historique. Inconvénient : toutes les requêtes doivent filtrer.

Valeurs générées

`lastrowid` donne l'identifiant de la ligne insérée. La clause `RETURNING` (SQLite ≥ 3.35, PostgreSQL) renvoie les colonnes modifiées.

```
INSERT INTO membres (nom) VALUES ('Dan') RETURNING id;
```

Exemple pas à pas

Problème : mettre à jour les scores en toute sécurité, avec prévisualisation et transaction.

```
import sqlite3

bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT, ville TEXT, score INTEGER
   DEFAULT 0, actif INTEGER DEFAULT 1);
INSERT INTO membres (nom, ville, score) VALUES ('Ada','Lyon',90), ('Bob','Lille',4
   5), ('Cléo','Lyon',60), ('Dan','Nice',10);
''')

# 1. Prévisualiser
condition = "ville = ? AND score < ?"
params = ("Lyon", 80)
print("à modifier :", bd.execute(f"SELECT nom, score FROM membres WHERE {condition
   }", params).fetchall())
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# 2. Modifier dans une transaction
try:
    with bd:                                          # BEGIN ... COMMIT (ROLLBAC
   K si exception)
        cur = bd.execute(f"UPDATE membres SET score = score + 5 WHERE {condition}"
   , params)
        print("lignes modifiées :", cur.rowcount)
        if cur.rowcount > 10:
            raise RuntimeError("trop de lignes, on annule")
except RuntimeError as e:
    print(e)

print(bd.execute("SELECT nom, score FROM membres ORDER BY id").fetchall())
```

Upsert pour un compteur :

Exemple pas à pas (suite)

```
bd.executescript('''
CREATE TABLE visites (page TEXT PRIMARY KEY, total INTEGER NOT NULL);
''')
for page in ("accueil", "cours", "accueil", "accueil"):
    bd.execute("INSERT INTO visites VALUES (?, 1) ON CONFLICT(page) DO UPDATE SET
   total = total + 1", (page,))
print(bd.execute("SELECT * FROM visites ORDER BY page").fetchall())      # [('accu
   eil', 3), ('cours', 1)]
```

Suppression logique :

```
bd.execute("ALTER TABLE membres ADD COLUMN supprime_le TEXT")
bd.execute("UPDATE membres SET supprime_le = '2025-03-01' WHERE nom = 'Dan'")
print(bd.execute("SELECT nom FROM membres WHERE supprime_le IS NULL ORDER BY id").
   fetchall())
```

Dan n'est plus visible, mais ses données existent toujours.

À toi, avec aide

Dans la table `membres(id, nom, ville, score, actif)` :

1. Prévisualise puis augmente de 10 % le score des membres de Lille.
2. Ajoute un membre « Eve » à Nice avec un score par défaut et récupère son `id`.
3. Désactive (suppression logique, `actif = 0`) les membres dont le score est inférieur à 20.
4. Montre ce que renverrait un `DELETE` sans `WHERE` **dans une transaction annulée** (on vérifie sans danger).

À toi, avec aide · réponse

Indices progressifs

Indice 1

`score * 110 / 100` pour +10 % (entiers).

Indice 2

`cursor.lastrowid` après un INSERT.

Indice 3

`bd.rollback()` annule la transaction en cours.

À toi, avec aide · réponse

Réponse

```
import sqlite3

bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT, ville TEXT, score INTEGER
   DEFAULT 0, actif INTEGER DEFAULT 1);
INSERT INTO membres (nom, ville, score) VALUES ('Ada','Lyon',90), ('Bob','Lille',4
   5), ('Cléo','Lille',60), ('Dan','Nice',10);
''')
print(bd.execute("SELECT nom, score FROM membres WHERE ville = 'Lille'").fetchall(
   ))
with bd:
    bd.execute("UPDATE membres SET score = score * 110 / 100 WHERE ville = 'Lille'
   ")
cur = bd.execute("INSERT INTO membres (nom, ville) VALUES ('Eve', 'Nice')")
bd.commit()
print("id d'Eve :", cur.lastrowid)
with bd:
    bd.execute("UPDATE membres SET actif = 0 WHERE score < 20")
```

À toi, avec aide · réponse

Réponse (suite)

```
print(bd.execute("SELECT nom, score, actif FROM membres ORDER BY id").fetchall())

bd.execute("BEGIN")
cur = bd.execute("DELETE FROM membres")
print("un DELETE sans WHERE aurait supprimé", cur.rowcount, "lignes")
bd.rollback()
print(bd.execute("SELECT COUNT(*) FROM membres").fetchone()[0], "lignes toujours l
   à")
```

À toi, en solo

Synchronisation de données. Tu reçois chaque nuit un fichier `nouveautes` (liste de couples `(email, nom)`) à fusionner dans la table `contacts(email PRIMARY KEY, nom, mis_a_jour_le)`.
Écris `synchroniser(bd, nouveautes, date)` qui :

1. insère les nouveaux e-mails ;
2. met à jour le nom des e-mails existants **seulement s'il a changé** ;
3. renvoie `(inseres, modifies, inchanges)` ;
4. est **atomique** : si une ligne est invalide (e-mail sans `@`), rien n'est modifié.

À toi, en solo · réponse

Pistes de correction

```
import sqlite3

def synchroniser(bd, nouveautes, date):
    for email, nom in nouveautes:
        if "@" not in email:
            raise ValueError(f"e-mail invalide : {email}")
    inseres = modifies = inchanges = 0
    with bd:                                    # tout ou rien
        for email, nom in nouveautes:
            ligne = bd.execute("SELECT nom FROM contacts WHERE email = ?", (email,
   )).fetchone()
            if ligne is None:
                bd.execute("INSERT INTO contacts VALUES (?, ?, ?)", (email, nom, d
   ate))
                inseres += 1
            elif ligne[0] != nom:
                bd.execute("UPDATE contacts SET nom = ?, mis_a_jour_le = ? WHERE e
   mail = ?", (nom, date, email))
                modifies += 1
```

À toi, en solo · réponse

Pistes de correction (suite)

```
            else:
                inchanges += 1
    return inseres, modifies, inchanges

bd = sqlite3.connect(":memory:")
bd.execute("CREATE TABLE contacts (email TEXT PRIMARY KEY, nom TEXT NOT NULL, mis_
   a_jour_le TEXT)")
print(synchroniser(bd, [("a@x.fr", "Ada"), ("b@x.fr", "Bob")], "2025-01-01"))
        # (2, 0, 0)
print(synchroniser(bd, [("a@x.fr", "Ada L."), ("b@x.fr", "Bob"), ("c@x.fr", "Cléo"
   )], "2025-01-02"))   # (1, 1, 1)
try:
    synchroniser(bd, [("d@x.fr", "Dan"), ("invalide", "Eve")], "2025-01-03")
except ValueError as e:
    print(e, bd.execute("SELECT COUNT(*) FROM contacts").fetchone()[0], "contacts"
   )     # 3 contacts
```

À toi, en solo · réponse

Pistes de correction (suite)

Variante en une seule requête : `INSERT ... ON CONFLICT(email) DO UPDATE SET nom = excluded.nom WHERE nom != excluded.nom`.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
UPDATE/DELETE sans WHERE	Toutes les lignes modifiées	SELECT de prévisualisation + transaction
Pas de transaction	Opération à moitié faite en cas d'erreur	`with bd:` ou BEGIN/COMMIT
Supprimer sans sauvegarde	Perte irréversible	Sauvegarde ou suppression logique
Coller des valeurs dans le SQL	Injection, erreurs de guillemets	Paramètres ?

Mini-boss

Mini-boss

Outil de migration sûr. Écris une fonction `appliquer(bd, requete, params, max_lignes)` qui exécute une modification en suivant la règle d'or :

1. prévisualise (transforme l'UPDATE/DELETE en SELECT de comptage) ;
2. refuse si le nombre de lignes dépasse `max_lignes` ;
3. exécute dans une transaction ;
4. vérifie que `rowcount` correspond à la prévisualisation ;
5. journalise (dans une table `journal`) la requête, les paramètres, le nombre de lignes et l'horodatage ;
6. propose `annuler_dernier()` qui restaure les lignes modifiées à partir d'une copie de sauvegarde prise avant l'opération.

Teste avec un UPDATE normal, un DELETE sans WHERE (refusé), et un rollback.

Mini-boss · réponse

Pistes de correction

Un bon travail : pour un `DELETE FROM t WHERE c`, la prévisualisation est `SELECT COUNT(*) FROM t WHERE c` ; un `DELETE`/`UPDATE` sans `WHERE` est détecté par analyse du texte (ou refusé d'office) ; la sauvegarde est faite avec `CREATE TABLE sauvegarde_n AS SELECT * FROM t WHERE c` avant l'opération ; `annuler_dernier` réinsère ou rétablit à partir de cette copie dans une transaction ; le journal garde l'opération dans la même transaction pour rester cohérent. Les tests montrent : modification acceptée (n lignes), tentative refusée (trop de lignes), rollback après erreur simulée.

Quiz éclair

Quiz éclair

1. Un UPDATE sans WHERE :

A. modifie toutes les lignes de la table
B. ne fait rien
C. provoque une erreur

2. Un upsert est :

A. supprimer puis insérer
B. insérer ou mettre à jour si la clé existe
C. un type d'index

3. Avant une modification risquée, on devrait :

A. exécuter directement

Quiz éclair

Quiz éclair (suite)

B. désactiver la base
C. prévisualiser avec un SELECT et utiliser une transaction

4. 🔁 *Révision mission 03* : Pour tester l'absence de valeur on écrit :

A. IS NULL
B. = NULL
C. == NULL

5. 🔁 *Révision mission 03* : Pour éviter l'injection SQL, on utilise :

A. des guillemets doubles
B. des requêtes paramétrées
C. des majuscules

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle action irréversible as-tu déjà regretté d'avoir faite trop vite ?
2. Quelles précautions prends-tu avant de modifier un fichier important ?

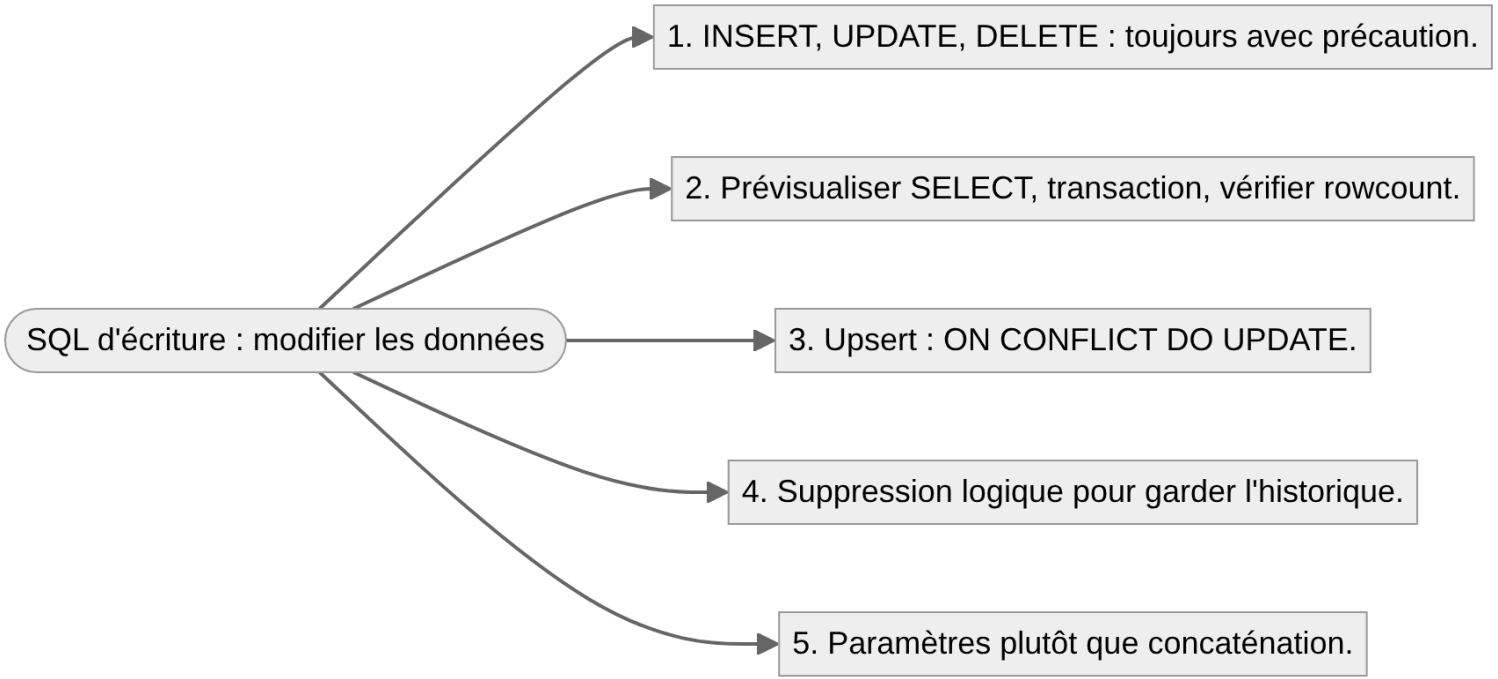
À retenir

À retenir

1. INSERT, UPDATE, DELETE : toujours avec précaution.
2. Prévisualiser (SELECT), transaction, vérifier `rowcount`.
3. Upsert : ON CONFLICT DO UPDATE.
4. Suppression logique pour garder l'historique.
5. Paramètres plutôt que concaténation.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Éditeur de données

Prochaine mission : Contraintes et intégrité

