

Tutoriel

LLM · SQL · Data analysis

Créer un copilote SQL sécurisé

Générer une requête SQL à partir d'une question, l'examiner avec SQLite et bloquer les instructions d'écriture.

Auteur **Youness Boutyour**

**DURÉE**

60 minutes

NIVEAU

Intermédiaire

FORMAT

Tutoriel pratique

LLM · SQL · Data analysis · publié le 2026-06-22 · mock

Objectifs et livrables

Une équipe souhaite connaître les catégories les plus fréquentes dans ses données sans écrire elle-même la requête SQL. Nous allons construire un copilote qui traduit cette question en requête, puis soumet sa proposition à des contrôles avant de lire une base SQLite.

Vous préparerez la base et le schéma présenté au modèle, puis vous suivrez la génération d'une instruction `SELECT`. Nous examinerons les vérifications locales, le plan SQLite et le résultat de la lecture. Vous pourrez voir à quel moment le programme accepte une requête et à quel moment il la refuse.

Des demandes interdites, dont `DROP TABLE`, permettront de vérifier le périmètre de cette démonstration. Le modèle propose du texte ; le programme décide de son exécution. Les contrôles du tutoriel constituent une première barrière à compléter pour un service partagé, notamment par des droits de lecture et des limites de ressources.

À la fin, vous retrouverez la requête dans `outputs/query.sql`, la réponse dans `outputs/result.json` et un test de refus. Ces éléments permettront de relier une question métier à la requête réellement exécutée et d'expliquer les décisions du garde-fou.

Ne jamais exécuter directement un SQL généré

Schéma du copilote SQL qui sépare question, requête candidate, contrôles en lecture seule, exécution SQLite et journal.



La requête générée reste une proposition jusqu'à son analyse et son autorisation explicite.

Lire le flux avant de coder

Le schéma place un garde-fou entre le texte généré et la base. La question métier devient une proposition SQL, puis le programme vérifie que l'instruction est une lecture autorisée avant de l'exécuter avec un compte ou une connexion adaptée. Le modèle ne possède jamais le dernier mot.

Le test DROP TABLE est utile parce qu'il vérifie une propriété négative : une instruction dangereuse doit être refusée avant SQLite. L'acceptation d'une requête SELECT ne prouve pas qu'elle est pertinente ; le plan, le résultat et la trace doivent encore être inspectés.

1. Préparer l'environnement

SQLite est accessible depuis la bibliothèque standard de Python ; `pytest` sert à exécuter les tests du projet. Travaillez sur la base de démonstration dans `data/`, jamais sur une base de production. La création et l'insertion des données ont besoin d'écrire dans ce fichier ; cette phase de préparation est distincte de la connexion en lecture seule utilisée par le copilote.

Le premier extrait illustre une initialisation sur une base neuve. Pour recréer les données, utilisez ensuite `src/seed.py`, dont la création de table tolère sa présence ; ne relancez pas aveuglément un `CREATE TABLE` simple sur une table existante. Les commandes du tutoriel partent de la racine du projet afin que le même fichier SQLite soit utilisé partout.

Terminal · Bash

```
mkdir -p sql-copilot-securise/{data,src,outputs}
cd sql-copilot-securise
python3 -m venv .venv && source .venv/bin/activate
python -m pip install pytest
```

Sortie obtenue

Sortie / exemple

Successfully installed pytest

Commande de vérification

Terminal · Bash

```
python -c "import sqlite3; print(sqlite3.sqlite_version)"
```

Code de l'étape

Python

```
import sqlite3

schema = """
CREATE TABLE orders (
    id INTEGER, category TEXT, amount REAL, created_at TEXT
)
"""

with sqlite3.connect("data/sales.db") as db:
    db.executescript(schema)
    db.execute("DELETE FROM orders")
    categories = ["data", "ai", "governance", "security"]
    db.executemany(
        "INSERT INTO orders VALUES (?, ?, ?, ?)",
        [(i, categories[(i - 1) % len(categories)], 10.0 + i * 3.5, f"2026-10-{(i % 10) + 1:02d}")]
        for i in range(1, 21)],
    )
```

2. Créer la base et son contrat

Le schéma transmis au modèle décrit les noms des colonnes et leurs types sans exposer les commandes individuelles. C'est le contrat qui lui permet de formuler une requête sur `category` ou `amount`. Il ne remplace pas les définitions métier : ici, une ligne représente une commande, et `COUNT(*)` compte ces lignes plutôt que des clients ou des articles.

Les vingt lignes sont réparties uniformément entre quatre catégories : plusieurs catégories auront donc le même effectif. Un top-3 trié uniquement sur le compte peut varier entre ex æquo, sans que l'agrégation soit fautive. Vérifiez les totaux et les montants indépendamment de cet ordre. La réinitialisation supprime les lignes de la table de démonstration avant de les recréer ; elle n'est pas destinée à conserver un historique.

Créez `data/sales.db` avec `orders(id, category, amount, created_at)` et insérez vingt commandes réparties entre quatre catégories et plusieurs dates. Exportez le schéma dans `data/schema.sql` ; il sera le seul contexte envoyé au modèle.

Code de l'étape

Python

```
import sqlite3
from pathlib import Path

db = sqlite3.connect("data/sales.db")
```

```
db.execute("CREATE TABLE IF NOT EXISTS orders (id INTEGER, category TEXT, amount REAL, created_at TEXT)")
db.execute("DELETE FROM orders")
categories = ["data", "ai", "governance", "security"]
rows = [
    (i, categories[(i - 1) % len(categories)], 10.0 + i * 3.5, f"2026-10-{(i % 10) + 1:02d}")
    for i in range(1, 21)
]
db.executemany("INSERT INTO orders VALUES (?, ?, ?, ?)", rows)
db.commit()
schema = db.execute("SELECT sql FROM sqlite_master WHERE type='table'").fetchone()[0]
Path("data/schema.sql").write_text(schema + "\n", encoding="utf-8")
assert db.execute("SELECT COUNT(*) FROM orders").fetchone()[0] == 20
db.close()
```

Sortie obtenue

La base contient vingt commandes dans quatre catégories et le schéma ne contient aucun secret.

Sortie / exemple

```
table=orders rows=20 categories=4 columns=4 secrets=0
```

Commande de vérification

Terminal · Bash

```
python src/seed.py
sqlite3 data/sales.db '.schema orders'
```

3. Générer puis contrôler le SQL

Le programme contrôle la proposition avant de la transmettre à SQLite, quel que soit son origine : fixture ou modèle. `safe_select` retire les espaces de bord pour son analyse, exige un début `SELECT` et refuse les points-virgules intermédiaires. La première définition présente la fonction ; l'extrait complet la reprend avec le générateur mock, il ne faut pas empiler deux implémentations différentes.

Ce filtre textuel est volontairement simple : il ne constitue ni un analyseur SQL ni une autorisation complète sur les tables et fonctions. La connexion en lecture seule de l'étape suivante apporte une barrière distincte contre les écritures. Une requête admise peut encore être coûteuse ou répondre à la mauvaise question ; le test de refus doit être accompagné de contrôles sur les résultats des requêtes autorisées.

Implémentez le garde-fou avant l'exécution :

● ● ● Python

```
def safe_select(sql: str) → str:
    clean = sql.strip().lower()
    if not clean.startswith("select") or ";" in clean[:-1]:
        raise ValueError("Seules les requêtes SELECT simples sont autorisées")
    return sql
```

Ajoutez un mode mock qui fournit deux requêtes SELECT de référence.

Code de l'étape

● ● ● Python

```
def safe_select(sql: str) → str:
    clean = sql.strip().lower()
    if not clean.startswith("select") or ";" in clean[:-1]:
        raise ValueError("Seules les requêtes SELECT simples sont autorisées")
    return sql

def mock_query(question: str) → str:
    if "catég" in question.casefold():
        return "SELECT category, COUNT(*) AS n FROM orders GROUP BY category ORDER BY n DESC LIMIT 3"
    return "SELECT COUNT(*) AS n, ROUND(AVG(amount), 2) AS average_amount FROM orders"

sql = safe_select(mock_query("top catégories"))
print(sql)
```

Sortie obtenue

Les deux questions analytiques donnent une requête contrôlée ; `DROP TABLE orders` est refusé avant SQLite.

■ Sortie / exemple

```
query=SELECT category, COUNT(*) AS n FROM orders GROUP BY category ...
attack=DROP TABLE orders result=refused
```

Ajoutez `--real --question "quelles sont les trois catégories les plus fréquentes ?"` pour demander la requête à un endpoint compatible. Le garde-fou s'exécute dans tous les modes.

Commande de vérification

Terminal · Bash

```
python src/copilot.py --mock --question "quelles sont les trois catégories les plus fréquentes ?"  
python src/copilot.py --mock --attack "DROP TABLE orders"; test $? -ne 0
```

4. Exécuter et conserver la trace

La chaîne `file:data/sales.db?mode=ro`, associée à `uri=True`, ouvre le fichier existant en lecture seule. Si le chemin est incorrect, corrigez le dossier de travail plutôt que de créer une autre base vide. `row_factory` transforme les colonnes en objets nommés, ce qui rend les fichiers JSON lisibles sans devoir mémoriser la position de chaque valeur.

`EXPLAIN QUERY PLAN` décrit la manière dont SQLite prévoit d'accéder aux données ; ce n'est ni une estimation complète du temps ni une validation de l'intention métier. Comparez ensuite les lignes retournées à un calcul direct sur les vingt commandes. Conservez la requête avec les exports : le plan et le résultat seuls ne suffisent pas à expliquer quelle question le copilote a effectivement exécutée.

Lancez `EXPLAIN QUERY PLAN`, exécutez la requête dans une connexion en lecture seule et sauvegardez la requête, le schéma utilisé et le résultat.

Code de l'étape

Python

```
import json  
import sqlite3  
from pathlib import Path  
  
Path("outputs").mkdir(exist_ok=True)  
with sqlite3.connect("file:data/sales.db?mode=ro", uri=True) as db:  
    db.row_factory = sqlite3.Row  
    plan = [dict(row) for row in db.execute("EXPLAIN QUERY PLAN " + sql)]  
    rows = [dict(row) for row in db.execute(sql)]  
(Path("outputs/plan.json")).write_text(json.dumps(plan, indent=2))  
(Path("outputs/result.json")).write_text(json.dumps(rows, indent=2))
```

Sortie obtenue

Le résultat JSON contient les lignes retournées et la requête exacte exécutée.

Sortie / exemple

```
plan_rows=1 result_rows=3 readonly=true outputs=result.json
```

Commande de vérification

Terminal · Bash

```
python -m json.tool outputs/plan.json >/dev/null
python -m json.tool outputs/result.json >/dev/null
pytest -q
```

Exemples de sorties

Le mode mock répond aux deux questions, la requête d'écriture est refusée et les artefacts de contrôle sont créés.

Dépannage et diagnostics

Ne donnez pas au modèle un compte d'écriture. Vérifiez les noms de tables, limitez le nombre de lignes et journalisez les refus autant que les succès.

Vérification finale

- ☐ schéma minimal transmis ;
- ☐ SELECT uniquement ;
- ☐ attaque d'écriture refusée ;
- ☐ plan et résultat sauvegardés.

Limites et prolongements

Une requête syntaxiquement correcte peut être sémantiquement fausse. Ajoutez des questions de référence et une validation par un analyste avant usage métier.

Références

- [SQLite — SQL language](#)
- [Python — sqlite3](#)