ЗВІТ

Лабораторна робота №3

Реалізація каунтера з використанням PostgreSQL

ПІБ	Одінцов Валерій Вячеславович
Група	ФБ-51мп
ОС / Середовище	Kali Linux, Docker, Python 3.x (venv)
СУБД	PostgreSQL (Docker image)
Параметри тесту	threads=10, perThread=10000, expected=100000

1. Мета роботи

Реалізувати декілька способів конкурентного оновлення значення лічильника (каунтера) у PostgreSQL та порівняти їх час виконання. Перевірити, чи отримується коректний результат (100 000 інкрементів) для кожного сценарію, окрім навмисно некоректного Lost-update.

2. Вимоги та умови

• Мова реалізації: Python.

• ORM не використовується.

• Для кожного потоку створюється окреме підключення до БД.

• Окрема транзакція на кожен інкремент (BEGIN/COMMIT на кожну ітерацію циклу).

• Кількість потоків: 10; інкрементів на потік: 10 000; очікуване значення: 100 000.

• DSN: postgresql://postgres:postgres@127.0.0.1:5432/counterdb

3. Структура таблиці

Використовується таблиця user_counter:

Поле	Тип	Опис
user_id	INTEGER (PK)	Ідентифікатор користувача
counter	INTEGER	Значення каунтера
version	INTEGER	Версія рядка для оптимістичного контролю

4. Підготовка середовища (Kali + Docker + PostgreSQL)

4.1. Файл compose.yaml для запуску PostgreSQL:

```
services:
  db:
```

```
    image: postgres:16-alpine
    container_name: pg-counter-db
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: counterdb
    ports:
      - "5432:5432"
    volumes:
      - pg_counter_data:/var/lib/postgresql/data
volumes:
  pg_counter_data:
```

4.2. Команди запуску контейнера:

```
docker compose up -d
docker ps --filter "name=pg-counter-db"
```

```
CONTAINER ID   IMAGE                COMMAND                  CREATED         STATUS         PORTS                                       NAMES
791c277c174b   postgres:16-alpine   "docker-entrypoint.s…"   7 seconds ago   Up 6 seconds   0.0.0.0:5432→5432/tcp, :::5432→5432/tcp   pg-counter-db
```

4.3. Ініціалізація таблиці (psql):

```
DROP TABLE IF EXISTS user_counter;

CREATE TABLE user_counter (
  user_id  INTEGER PRIMARY KEY,
  counter  INTEGER NOT NULL,
  version  INTEGER NOT NULL
);

INSERT INTO user_counter(user_id, counter, version)
VALUES (1, 0, 0);

SELECT * FROM user_counter;
```

```
 user_id | counter | version
---------+---------+---------
       1 |       0 |       0
(1 row)
```

4.4. Підготовка Python-оточення та встановлення psycopg3:

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install -U pip
pip install "psycopg[binary]"
```

5. Варіанти реалізації

5.1. Lost-update (втрата оновлень)

Схема read-modify-write без синхронізації: спочатку читаємо counter, потім збільшуємо на 1, потім записуємо нове значення. При одночасному доступі кількох потоків частина інкрементів перезаписується.

```
BEGIN;
SELECT counter FROM user_counter WHERE user_id=1;
UPDATE user_counter SET counter = counter + 1 WHERE user_id=1; -- логічно, але тут саме
read-modify-write на клієнті
COMMIT;
```

5.2. Serializable update (SERIALIZABLE)

Той самий алгоритм read-modify-write, але транзакції виконуються з рівнем ізоляції SERIALIZABLE. PostgreSQL може перервати транзакцію з помилкою serialization_failure при конфліктах, тому потрібні ретраї.

5.3. In-place update

Один SQL-оператор: UPDATE ... SET counter = counter + 1.

```
BEGIN;
UPDATE user_counter SET counter = counter + 1 WHERE user_id=1;
COMMIT;
```

5.4. Row-level locking (SELECT ... FOR UPDATE)

Блокуємо рядок через SELECT ... FOR UPDATE перед оновленням.

```
BEGIN;
SELECT counter FROM user_counter WHERE user_id=1 FOR UPDATE;
UPDATE user_counter SET counter = counter + 1 WHERE user_id=1; -- після читання
COMMIT;
```

5.5. Optimistic concurrency control (version CAS)

UPDATE виконується лише якщо version не змінився; інакше повтор.

```
BEGIN;
SELECT counter, version FROM user_counter WHERE user_id=1;
UPDATE user_counter SET counter = $counter, version = $version+1 WHERE user_id=1 AND
version=$version;
COMMIT;
```

6. Запуск тестування та докази виконання

Команда запуску скрипта:

```
source .venv/bin/activate
python -u pg_counter_bench.py --threads 10 --perThread 10000
```

Консольний вивід одного з прогонів:

```
Config: threads=10, perThread=10000, expected=100000
DSN: postgresql://postgres:postgres@127.0.0.1:5432/counterdb

--- Running: 1) Lost-update (SELECT then UPDATE) ---
--- Done: 1) Lost-update (SELECT then UPDATE) | time=178.236s | value=11026 |
expected=100000 => BAD

--- Running: 2a) SERIALIZABLE naive (no retry) ---
--- Done: 2a) SERIALIZABLE naive (no retry) | time=141.512s | value=15853 |
expected=100000 => BAD | errors=84147
```

```
--- Running: 2b) SERIALIZABLE with retry (correct) ---
--- Done: 2b) SERIALIZABLE with retry (correct) | time=493.052s | value=100000 |
expected=100000 => OK | errors=147146, retries=147146

--- Running: 3) In-place UPDATE counter=counter+1 ---
--- Done: 3) In-place UPDATE counter=counter+1 | time=151.392s | value=100000 |
expected=100000 => OK

--- Running: 4) Row-level locking SELECT FOR UPDATE ---
--- Done: 4) Row-level locking SELECT FOR UPDATE | time=234.962s | value=100000 |
expected=100000 => OK

--- Running: 5) Optimistic (counter+version CAS) ---
--- Done: 5) Optimistic (counter+version CAS) | time=411.652s | value=100000 |
expected=100000 => OK | retries=129478

=== SUMMARY ===
Scenario                                    time(s)      value   expected   status
1) Lost-update (SELECT then UPDATE)         178.236      11026     100000      BAD
2a) SERIALIZABLE naive (no retry)           141.512      15853     100000      BAD
2b) SERIALIZABLE with retry (correct)       493.052     100000     100000       OK
3) In-place UPDATE counter=counter+1        151.392     100000     100000       OK
4) Row-level locking SELECT FOR UPDATE      234.962     100000     100000       OK
5) Optimistic (counter+version CAS)         411.652     100000     100000       OK
```

```
Config: threads=10, perThread=10000, expected=100000
DSN: postgresql://postgres:postgres@127.0.0.1:5432/counterdb

── Running: 1) Lost-update (SELECT then UPDATE) ──
── Done: 1) Lost-update (SELECT then UPDATE) | time=178.236s | value=11026 | expected=100000 ⇒ BAD

── Running: 2a) SERIALIZABLE naive (no retry) ──
── Done: 2a) SERIALIZABLE naive (no retry) | time=141.512s | value=15853 | expected=100000 ⇒ BAD | errors=84147

── Running: 2b) SERIALIZABLE with retry (correct) ──
── Done: 2b) SERIALIZABLE with retry (correct) | time=493.052s | value=100000 | expected=100000 ⇒ OK | errors=147146, retries=147146

── Running: 3) In-place UPDATE counter=counter+1 ──
── Done: 3) In-place UPDATE counter=counter+1 | time=151.392s | value=100000 | expected=100000 ⇒ OK

── Running: 4) Row-level locking SELECT FOR UPDATE ──
── Done: 4) Row-level locking SELECT FOR UPDATE | time=234.962s | value=100000 | expected=100000 ⇒ OK

── Running: 5) Optimistic (counter+version CAS) ──
── Done: 5) Optimistic (counter+version CAS) | time=411.652s | value=100000 | expected=100000 ⇒ OK | retries=129478

═══ SUMMARY ═══
Scenario                                time(s)      value   expected   status
1) Lost-update (SELECT then UPDATE)     178.236      11026     100000      BAD
2a) SERIALIZABLE naive (no retry)       141.512      15853     100000      BAD
2b) SERIALIZABLE with retry (correct)   493.052     100000     100000       OK
3) In-place UPDATE counter=counter+1    151.392     100000     100000       OK
4) Row-level locking SELECT FOR UPDATE  234.962     100000     100000       OK
5) Optimistic (counter+version CAS)     411.652     100000     100000       OK
```

7. Результати та порівняння

У таблиці наведено час виконання кожного сценарію та перевірка коректності кінцевого значення. У всіх сценаріях, крім Lost-update, очікується значення 100 000.

Сценарій	Час, с	Значення	Очікуване	OK/BAD	Помилки	Ретраї/повтори
1) Lost-update (SELECT then UPDATE)	178.236	11026	100000	BAD	0	0

2a) SERIALIZABLE naive (no retry)	141.512	15853	100000	BAD	84147	0
2b) SERIALIZABLE with retry (correct)	493.052	100000	100000	OK	147146	147146
3) In-place UPDATE counter=counter+1	151.392	100000	100000	OK	0	0
4) Row-level locking SELECT FOR UPDATE	234.962	100000	100000	OK	0	0
5) Optimistic (counter+version CAS)	411.652	100000	100000	OK	0	129478

8. Аналіз результатів

Lost-update показав значну втрату інкрементів через race condition у read-modify-write. SERIALIZABLE без ретраїв також дав некоректний результат через відміну транзакцій (serialization_failure). SERIALIZABLE з ретраями забезпечив коректність, але значно збільшив час через дуже велику кількість повторів. In-place UPDATE виявився найшвидшим коректним сценарієм у цьому прогоні. SELECT ... FOR UPDATE коректний, але повільніший через блокування рядка. Optimistic CAS коректний, але має багато повторів у умовах високої конкуренції.

9. Відповіді на питання по SERIALIZABLE

Чи буде втрата значень? PostgreSQL забезпечує серіалізовність, але застосунок має повторювати транзакції, які завершились з serialization_failure, інакше частина інкрементів не буде зафіксована.

Чи будуть помилки? Так, serialization_failure. У цьому прогоні: 84 147 (naive) та 147 146 (з ретраями).

Як модернізувати код? Реалізувати retry loop з перехопленням serialization_failure (SQLSTATE 40001) та повторювати транзакцію до успішного commit; бажано додати backoff/jitter.

10. Висновки

За результатами експерименту найкращим способом оновлення каунтера в PostgreSQL під навантаженням є in-place UPDATE (counter = counter + 1), оскільки він забезпечує коректність і має мінімальні накладні витрати. SERIALIZABLE підходить для високих гарантій, але потребує ретраїв і може бути повільним при конфліктах. SELECT ... FOR UPDATE гарантує коректність через блокування, але знижує паралелізм. Оптимістичний контроль версій працює, але при високій конкуренції породжує багато повторів.