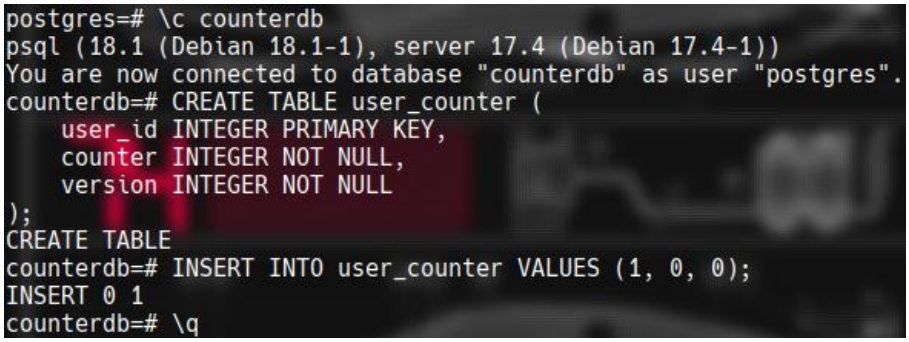
Проектування високонавантажених систем

Лабораторна робота 3

Торгало Ігнатій ФБ-51мп

1. Створення таблиці



2. Розробка клієнтського скрипта

Для проведення експериментів був розроблений Python-скрипт client.py, який реалізує всі необхідні сценарії оновлення лічильника в PostgreSQL. У програмі використовується модуль psycopg2 для прямої роботи з БД без ORM та механізм багатопоточності threading, що дозволяє емулювати конкурентний доступ 10 паралельних клієнтів.

У скрипті реалізовано п’ять окремих функцій, кожна з яких відповідає певному механізму оновлення даних: lost update, serializable, in-place update, row-level locking та optimistic concurrency control. Для кожного сценарію вимірюється час виконання та кінцеве значення лічильника, що дозволяє оцінити коректність та продуктивність різних підходів.

```python
import psycopg2
import threading
import time


DB = {
    "host": "localhost",
    "dbname": "counterdb",
    "user": "counteruser",
    "password": "pass123"
}

def get_conn():
    return psycopg2.connect(**DB)

def lost_update_worker():
    conn = get_conn()
    cur = conn.cursor()

    for _ in range(10000):
        cur.execute("SELECT counter FROM user_counter WHERE user_id = 1")
        counter = cur.fetchone()[0]
        counter += 1
        cur.execute("UPDATE user_counter SET counter=%s WHERE user_id=1", (counter,))
        conn.commit()

    cur.close()
    conn.close()

def test_lost_update():
    reset()

    threads = []
    start = time.time()

    for _ in range(10):
        t = threading.Thread(target=lost_update_worker)
        t.start()
        threads.append(t)

    for t in threads:
        t.join()

    end = time.time()

    print("LOST UPDATE RESULT:", get_result())
    print("TIME:", end - start)
    print("-" * 50)
```

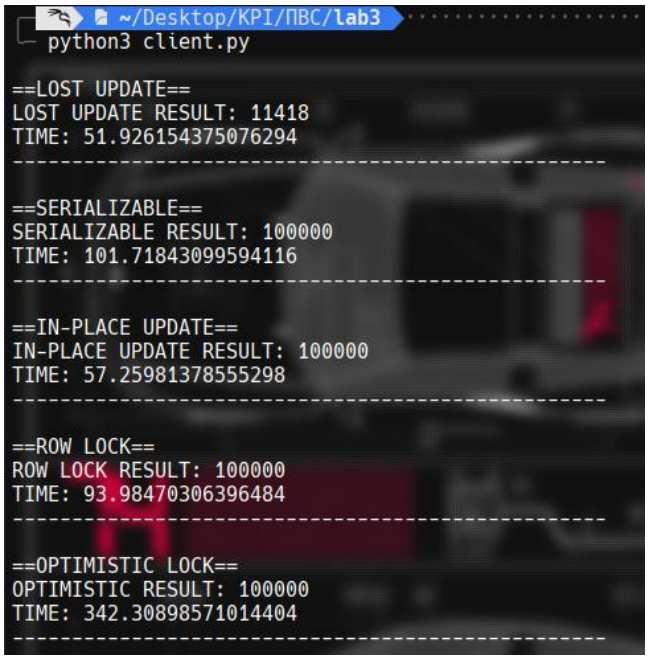
```python
def serializable_worker():
    conn = get_conn()
    conn.set_session(isolation_level="SERIALIZABLE")
    cur = conn.cursor()

    for _ in range(10000):
        ok = False
        while not ok:
            try:
                cur.execute("SELECT counter FROM user_counter WHERE user_id = 1")
                counter = cur.fetchone()[0] + 1
                cur.execute("UPDATE user_counter SET counter=%s WHERE user_id=1", (counter,))
                conn.commit()
                ok = True
            except psycopg2.errors.SerializationFailure:
                conn.rollback()

    cur.close()
    conn.close()

def test_serializable():
    reset()

    threads = []
    start = time.time()

    for _ in range(10):
        t = threading.Thread(target=serializable_worker)
        t.start()
        threads.append(t)

    for t in threads:
        t.join()

    end = time.time()
    print("SERIALIZABLE RESULT:", get_result())
    print("TIME:", end - start)
    print("-" * 50)

def inplace_worker():
    conn = get_conn()
    cur = conn.cursor()

    for _ in range(10000):
        cur.execute("UPDATE user_counter SET counter = counter + 1 WHERE user_id = 1")
        conn.commit()

    cur.close()
    conn.close()
```

```python
def test_inplace():
    reset()

    threads = []
    start = time.time()

    for _ in range(10):
        t = threading.Thread(target=inplace_worker)
        t.start()
        threads.append(t)

    for t in threads:
        t.join()

    end = time.time()
    print("IN-PLACE UPDATE RESULT:", get_result())
    print("TIME:", end - start)
    print("-" * 50)

def rowlock_worker():
    conn = get_conn()
    conn.set_session(isolation_level="READ COMMITTED")
    cur = conn.cursor()

    for _ in range(10000):
        cur.execute("SELECT counter FROM user_counter WHERE user_id=1 FOR UPDATE")
        counter = cur.fetchone()[0] + 1
        cur.execute("UPDATE user_counter SET counter=%s WHERE user_id=1", (counter,))
        conn.commit()

    cur.close()
    conn.close()

def test_rowlock():
    reset()

    threads = []
    start = time.time()

    for _ in range(10):
        t = threading.Thread(target=rowlock_worker)
        t.start()
        threads.append(t)

    for t in threads:
        t.join()

    end = time.time()
    print("ROW LOCK RESULT:", get_result())
    print("TIME:", end - start)
    print("-" * 50)
```

```python
def optimistic_worker():
    conn = get_conn()
    cur = conn.cursor()

    for _ in range(10000):
        while True:
            cur.execute("SELECT counter, version FROM user_counter WHERE user_id=1")
            counter, version = cur.fetchone()

            counter += 1

            cur.execute(
                "UPDATE user_counter SET counter=%s, version=%s "
                "WHERE user_id=1 AND version=%s",
                (counter, version + 1, version)
            )
            conn.commit()

            if cur.rowcount > 0:
                break

    cur.close()
    conn.close()

def test_optimistic():
    reset()

    threads = []
    start = time.time()

    for _ in range(10):
        t = threading.Thread(target=optimistic_worker)
        t.start()
        threads.append(t)

    for t in threads:
        t.join()

    end = time.time()
    print("OPTIMISTIC RESULT:", get_result())
    print("TIME:", end - start)
    print("-" * 50)

def reset():
    conn = get_conn()
    cur = conn.cursor()
    cur.execute("UPDATE user_counter SET counter=0, version=0 WHERE user_id=1")
    conn.commit()
    cur.close()
    conn.close()
```

```python
def get_result():
    conn = get_conn()
    cur = conn.cursor()
    cur.execute("SELECT counter FROM user_counter WHERE user_id=1")
    v = cur.fetchone()[0]
    cur.close()
    conn.close()
    return v

print("\n==LOST UPDATE==")
test_lost_update()

print("\n==SERIALIZABLE==")
test_serializable()

print("\n==IN-PLACE UPDATE==")
test_inplace()

print("\n==ROW LOCK==")
test_rowlock()

print("\n==OPTIMISTIC LOCK==")
test_optimistic()
```

3. Результат тестування



4. Висновки

Тестування показало суттєву різницю між підходами до конкурентного оновлення лічильника в PostgreSQL. У варіанті Lost Update кінцеве значення (11418 замість 100000) підтвердило наявність класичної проблеми «втрати оновлень», коли паралельні транзакції перезаписують одна одну. У режимі SERIALIZABLE база гарантує коректність, але ціною зростання часу виконання — через повтори транзакцій та конфлікти. In-place update працює найефективніше серед коректних методів, оскільки виконує атомарне оновлення на стороні сервера без необхідності читати попереднє значення. Підхід із Row-level locking також дає точний результат, але є повільнішим через блокування рядка на кожній операції. Метод Optimistic Concurrency Control забезпечує правильний кінцевий результат, однак має найбільший час виконання через часті конфлікти та повторні спроби оновлення при зростанні кількості потоків. Таким чином, найкращий компроміс між швидкістю та коректністю демонструє in-place update.