

Проектування високонавантажених систем

Лабораторна робота 3

Торгало Ігнатій ФБ-51мп

1. Створення таблиці

```
postgres=# \c counterdb
psql (18.1 (Debian 18.1-1), server 17.4 (Debian 17.4-1))
You are now connected to database "counterdb" as user "postgres".
counterdb=# CREATE TABLE user_counter (
    user_id INTEGER PRIMARY KEY,
    counter INTEGER NOT NULL,
    version INTEGER NOT NULL
);
CREATE TABLE
counterdb=# INSERT INTO user_counter VALUES (1, 0, 0);
INSERT 0 1
counterdb=# \q
```

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.

```

1 import psycopg2
2 import threading
3 import time
4
5
6 DB = {
7     "host": "localhost",
8     "dbname": "counterdb",
9     "user": "counteruser",
10    "password": "pass123"
11 }
12
13 def get_conn():
14     return psycopg2.connect(**DB)
15
16 def lost_update_worker():
17     conn = get_conn()
18     cur = conn.cursor()
19
20     for _ in range(10000):
21         cur.execute("SELECT counter FROM user_counter WHERE user_id = 1")
22         counter = cur.fetchone()[0]
23         counter += 1
24         cur.execute("UPDATE user_counter SET counter=%s WHERE user_id=1", (counter,))
25         conn.commit()
26
27     cur.close()
28     conn.close()
29
30 def test_lost_update():
31     reset()
32
33     threads = []
34     start = time.time()
35
36     for _ in range(10):
37         t = threading.Thread(target=lost_update_worker)
38         t.start()
39         threads.append(t)
40
41     for t in threads:
42         t.join()
43
44     end = time.time()
45
46     print("LOST UPDATE RESULT:", get_result())
47     print("TIME:", end - start)
48     print("-" * 50)
49

```

```

50 def serializable_worker():
51     conn = get_conn()
52     conn.set_session(isolation_level="SERIALIZABLE")
53     cur = conn.cursor()
54
55     for _ in range(10000):
56         ok = False
57         while not ok:
58             try:
59                 cur.execute("SELECT counter FROM user_counter WHERE user_id = 1")
60                 counter = cur.fetchone()[0] + 1
61                 cur.execute("UPDATE user_counter SET counter=%s WHERE user_id=1", (counter,))
62                 conn.commit()
63                 ok = True
64             except psycopg2.errors.SerializationFailure:
65                 conn.rollback()
66
67     cur.close()
68     conn.close()
69
70 def test_serializable():
71     reset()
72
73     threads = []
74     start = time.time()
75
76     for _ in range(10):
77         t = threading.Thread(target=serializable_worker)
78         t.start()
79         threads.append(t)
80
81     for t in threads:
82         t.join()
83
84     end = time.time()
85     print("SERIALIZABLE RESULT:", get_result())
86     print("TIME:", end - start)
87     print("-" * 50)
88
89 def inplace_worker():
90     conn = get_conn()
91     cur = conn.cursor()
92
93     for _ in range(10000):
94         cur.execute("UPDATE user_counter SET counter = counter + 1 WHERE user_id = 1")
95         conn.commit()
96
97     cur.close()
98     conn.close()

```

```

100 def test_inplace():
101     reset()
102
103     threads = []
104     start = time.time()
105
106     for _ in range(10):
107         t = threading.Thread(target=inplace_worker)
108         t.start()
109         threads.append(t)
110
111     for t in threads:
112         t.join()
113
114     end = time.time()
115     print("IN-PLACE UPDATE RESULT:", get_result())
116     print("TIME:", end - start)
117     print("-" * 50)
118
119 def rowlock_worker():
120     conn = get_conn()
121     conn.set_session(isolation_level="READ COMMITTED")
122     cur = conn.cursor()
123
124     for _ in range(10000):
125         cur.execute("SELECT counter FROM user_counter WHERE user_id=1 FOR UPDATE")
126         counter = cur.fetchone()[0] + 1
127         cur.execute("UPDATE user_counter SET counter=%s WHERE user_id=1", (counter,))
128         conn.commit()
129
130     cur.close()
131     conn.close()
132
133 def test_rowlock():
134     reset()
135
136     threads = []
137     start = time.time()
138
139     for _ in range(10):
140         t = threading.Thread(target=rowlock_worker)
141         t.start()
142         threads.append(t)
143
144     for t in threads:
145         t.join()
146
147     end = time.time()
148     print("ROW LOCK RESULT:", get_result())
149     print("TIME:", end - start)
150     print("-" * 50)

```

```

152 def optimistic_worker():
153     conn = get_conn()
154     cur = conn.cursor()
155
156     for _ in range(10000):
157         while True:
158             cur.execute("SELECT counter, version FROM user_counter WHERE user_id=1")
159             counter, version = cur.fetchone()
160
161             counter += 1
162
163             cur.execute(
164                 "UPDATE user_counter SET counter=%s, version=%s "
165                 "WHERE user_id=1 AND version=%s",
166                 (counter, version + 1, version)
167             )
168             conn.commit()
169
170             if cur.rowcount > 0:
171                 break
172
173     cur.close()
174     conn.close()
175
176 def test_optimistic():
177     reset()
178
179     threads = []
180     start = time.time()
181
182     for _ in range(10):
183         t = threading.Thread(target=optimistic_worker)
184         t.start()
185         threads.append(t)
186
187     for t in threads:
188         t.join()
189
190     end = time.time()
191     print("OPTIMISTIC RESULT:", get_result())
192     print("TIME:", end - start)
193     print("-" * 50)
194
195 def reset():
196     conn = get_conn()
197     cur = conn.cursor()
198     cur.execute("UPDATE user_counter SET counter=0, version=0 WHERE user_id=1")
199     conn.commit()
200     cur.close()
201     conn.close()

```

```

204 def get_result():
205     conn = get_conn()
206     cur = conn.cursor()
207     cur.execute("SELECT counter FROM user_counter WHERE user_id=1")
208     v = cur.fetchone()[0]
209     cur.close()
210     conn.close()
211     return v
212
213 print("\n==LOST UPDATE==")
214 test_lost_update()
215
216 print("\n==SERIALIZABLE==")
217 test_serializable()
218
219 print("\n==IN-PLACE UPDATE==")
220 test_inplace()
221
222 print("\n==ROW LOCK==")
223 test_rowlock()
224
225 print("\n==OPTIMISTIC LOCK==")
226 test_optimistic()

```

3. Результат тестування

```
~/Desktop/KPI/ПБС/Lab3
python3 client.py

==LOST UPDATE==
LOST UPDATE RESULT: 11418
TIME: 51.926154375076294
-----

==SERIALIZABLE==
SERIALIZABLE RESULT: 100000
TIME: 101.71843099594116
-----

==IN-PLACE UPDATE==
IN-PLACE UPDATE RESULT: 100000
TIME: 57.25981378555298
-----

==ROW LOCK==
ROW LOCK RESULT: 100000
TIME: 93.98470306396484
-----

==OPTIMISTIC LOCK==
OPTIMISTIC RESULT: 100000
TIME: 342.30898571014404
-----
```

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.