YILDIZ TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

BLM3780 DBMS

HOMEWORK 2

FATIH ALTINCI 20011610

ILKER BAHAR 20011611

PostgreSQL Version: 15.1

The query entered to create the table:

Query | Query History

```
1  CREATE TABLE ACCOUNTS (
2      accno INTEGER PRIMARY KEY,
3      balance INTEGER
4  );
5
6  INSERT INTO ACCOUNTS (accno, balance) VALUES (0, 100);
7
8  INSERT INTO ACCOUNTS (accno, balance)
9  SELECT generate_series(1, 100), 0;
```

```
Query returned successfully in 44 msec.
```

Data Output | Messages | Notifications

	accno [PK] integer	balance integer
1	0	100
2	1	0
3	2	0
4	3	0
5	4	0
6	5	0
7	6	0
8	7	0
9	8	0
10	9	0
11	10	0
12	11	0
13	12	0
14	13	0
15	14	0
16	15	0
17	16	0
18	17	0
19	18	0
20	19	0

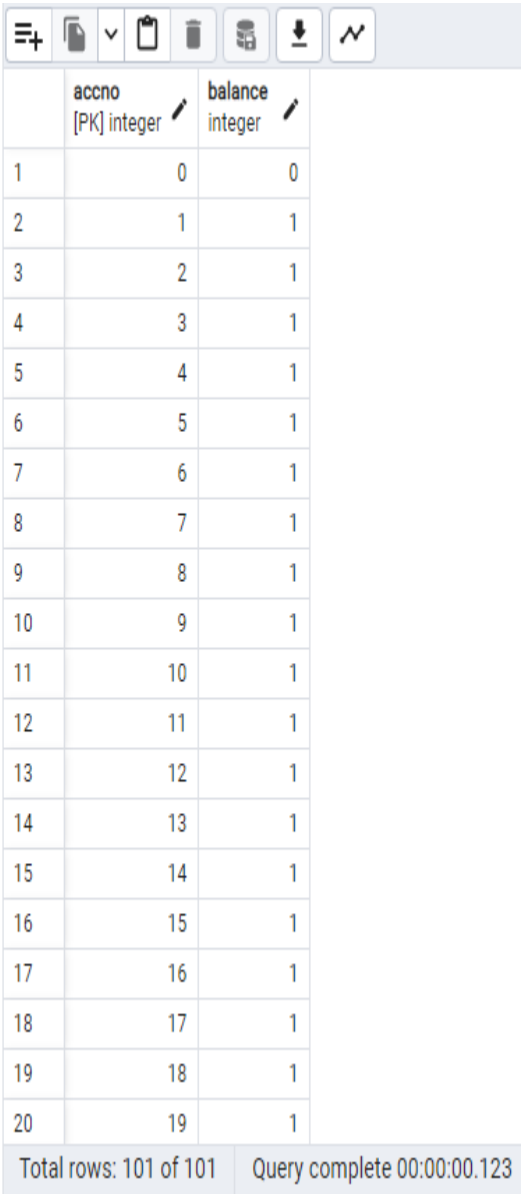
Total rows: 101 of 101 | Query complete 00:00:00.286

Connecting to the database in Python and performing the desired queries.

```python
import psycopg2

def run_tx_a(i):
    conn = psycopg2.connect(dbname="postgres", user="postgres", password="6054708Fatih", host="localhost")
    conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_READ_COMMITTED)
    cur = conn.cursor()

    cur.execute("SELECT balance FROM ACCOUNTS WHERE accno=%s", (i,))
    e = cur.fetchone()[0]
    cur.execute("UPDATE ACCOUNTS SET balance=%s WHERE accno=%s", (e+1, i))

    cur.execute("SELECT balance FROM ACCOUNTS WHERE accno=0")
    c = cur.fetchone()[0]
    cur.execute("UPDATE ACCOUNTS SET balance=%s WHERE accno=0", (c-1,))

    conn.commit()
    cur.close()
    conn.close()

for i in range(1,101):
    run_tx_a(i)
```

✓ 7.7s

1 TL transferred to other accounts:



	accno [PK] integer	balance integer
1	0	0
2	1	1
3	2	1
4	3	1
5	4	1
6	5	1
7	6	1
8	7	1
9	8	1
10	9	1
11	10	1
12	11	1
13	12	1
14	13	1
15	14	1
16	15	1
17	16	1
18	17	1
19	18	1
20	19	1

Total rows: 101 of 101 Query complete 00:00:00.123

Doing the same with threading:

```python
import threading
import psycopg2

def run_tx_a(i):
    conn = psycopg2.connect(dbname="postgres", user="postgres", password="6054708Fatih", host="localhost")
    conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_READ_COMMITTED)
    cur = conn.cursor()

    cur.execute("SELECT balance FROM ACCOUNTS WHERE accno=%s", (i,))
    e = cur.fetchone()[0]
    cur.execute("UPDATE ACCOUNTS SET balance=%s WHERE accno=%s", (e+1, i))

    cur.execute("SELECT balance FROM ACCOUNTS WHERE accno=0")
    c = cur.fetchone()[0]
    cur.execute("UPDATE ACCOUNTS SET balance=%s WHERE accno=0", (c-1,))

    conn.commit()
    cur.close()
    conn.close()

for concurrent_tx in range(1, 6):
    threads = []
    for i in range(1, 101):
        t = threading.Thread(target=run_tx_a, args=(i,))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
```

✓ 9.5s

OR

```python
import psycopg2
import threading

def transfer_funds(thread_id):
    conn = psycopg2.connect(dbname="postgres", user="postgres", password="6054708Fatih", host="localhost")
    cur = conn.cursor()

    cur.execute("SET TRANSACTION ISOLATION LEVEL READ COMMITTED")

    for i in range(1, 101):
        cur.execute("BEGIN;")
        cur.execute("UPDATE ACCOUNTS SET balance = balance + 1 WHERE accno = %s;",(i,))
        cur.execute("UPDATE ACCOUNTS SET balance = balance - 1 WHERE accno = 0;")
        cur.execute("COMMIT;")

    conn.close()
    print(f'Thread {thread_id} completed')

threads = []
for i in range(1, 6):
    thread = threading.Thread(target=transfer_funds, args=(i,))
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()
```

✓ 0.2s

```
Thread 1 completed
Thread 2 completed
Thread 3 completed
Thread 4 completed
Thread 5 completed
```

Incorrect Results with Read Committed:

	accno [PK] integer	balance integer
1	0	-400
2	1	5
3	2	5
4	3	5
5	4	5
6	5	5
7	6	5
8	7	5
9	8	5
10	9	5
11	10	5
12	11	5
13	12	5
14	13	5
15	14	5
16	15	5
17	16	5
18	17	5
19	18	5
20	19	5
Total rows: 101 of 101		Query complete 00:00:00.135

The relationship between the starting and ending balance of the first account after TX transactions and the desired values:

```python
import time
import threading
import psycopg2

start_time = time.time()

for concurrent_tx in range(1, 6):
    threads = []
    for i in range(1, 101):
        t = threading.Thread(target=run_tx_a, args=(i,))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()

end_time = time.time()

c1 = None
c2 = None
with psycopg2.connect(dbname="postgres", user="postgres", password="6054708Fatih", host="localhost") as conn:
    with conn.cursor() as cur:
        cur.execute("SELECT balance FROM ACCOUNTS WHERE accno=0")
        c1 = cur.fetchone()[0]
        for concurrent_tx in range(1, 6):
            threads = []
            for i in range(1, 101):
                t = threading.Thread(target=run_tx_a, args=(i,))
                threads.append(t)
                t.start()
            for t in threads:
                t.join()
        cur.execute("SELECT balance FROM ACCOUNTS WHERE accno=0")
        c2 = cur.fetchone()[0]

c_value = (c1 - c2) / 100
t = end_time - start_time
tps = 100 / t

print("Correctness: ", c_value)
print("Throughput: ", tps, " transactions per second")
```

✓ 19.7s

```
Correctness:  4.73
Throughput:  10.514631074002997  transactions per second
```

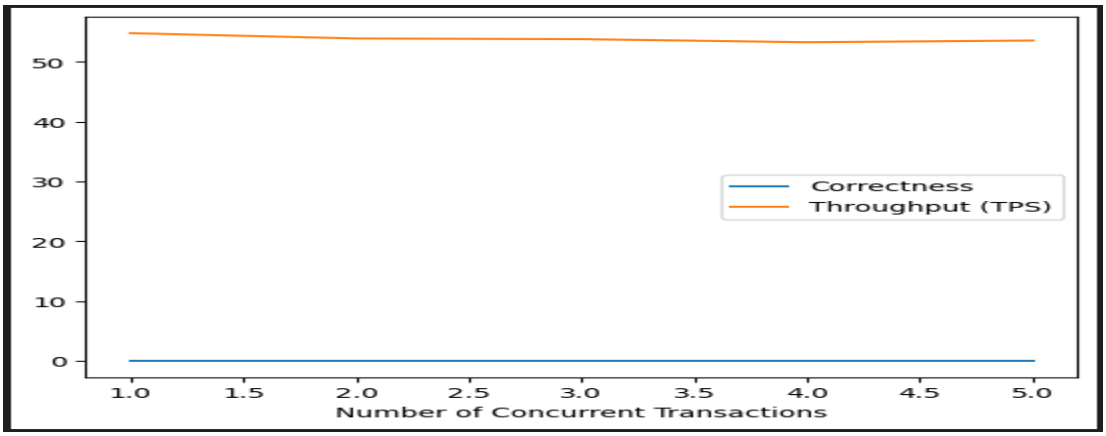
Table Current Status:

	accno [PK] integer	balance integer
1	0	-861
2	1	10
3	2	10
4	3	10
5	4	10
6	5	10
7	6	10
8	7	10
9	8	10
10	9	10
11	10	10
12	11	10
13	12	10
14	13	10
15	14	10
16	15	10
17	16	10
18	17	10
19	18	10
20	19	10

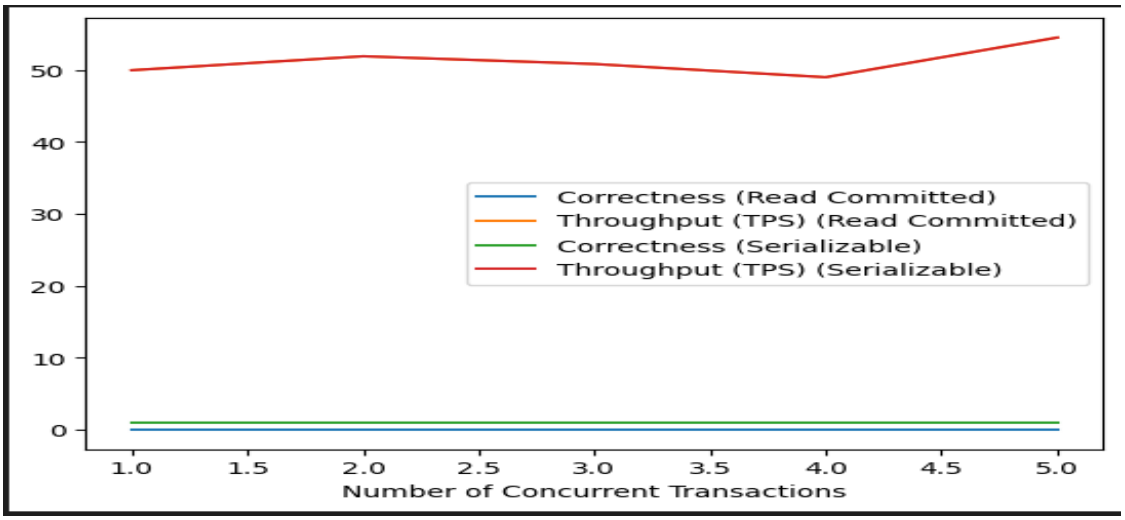
Total rows: 101 of 101 | Query complete 00:00:00.112

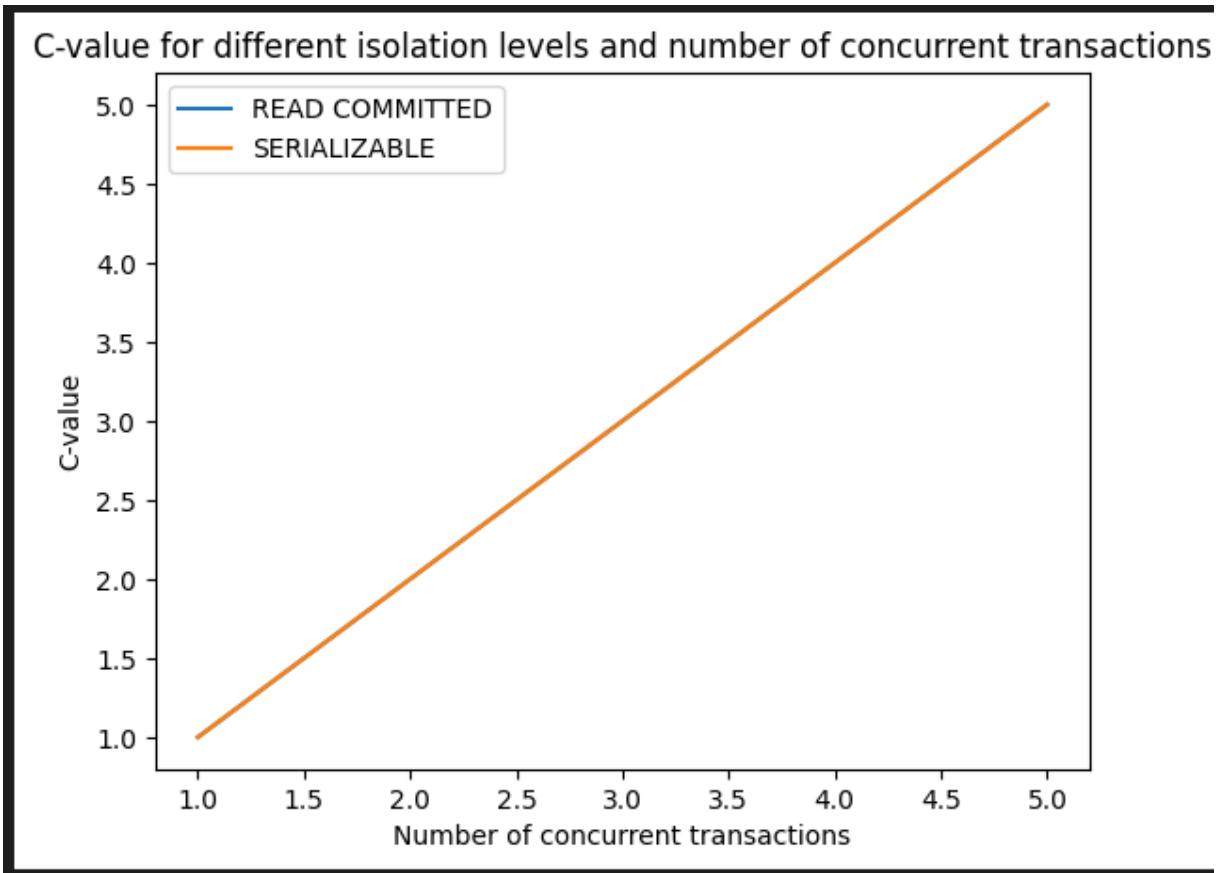
Graph Drawn for Different Experiments

```python
import matplotlib.pyplot as plt

c_values = []
tps_values = []

for concurrent_tx in range(1, 6):
    start_time = time.time()

    threads = []
    for i in range(1, 101):
        t = threading.Thread(target=run_tx_a, args=(i,))
        threads.append(t)
    for t in threads:
        t.start()
    for t in threads:
        t.join()

    end_time = time.time()
    with psycopg2.connect(dbname="postgres", user="postgres", password="6054708Fatih", host="localhost") as conn:
        with conn.cursor() as cur:
            cur.execute("SELECT balance FROM ACCOUNTS WHERE accno=0")
            c1 = cur.fetchone()[0]
            cur.execute("SELECT balance FROM ACCOUNTS WHERE accno=0")
            c2 = cur.fetchone()[0]
    c_value = (c1 - c2) / 100
    t = end_time - start_time
    tps = 100 / t
    c_values.append(c_value)
    tps_values.append(tps)

plt.plot(range(1, 6), c_values, label="Correctness")
plt.plot(range(1, 6), tps_values, label="Throughput (TPS)")
plt.xlabel("Number of Concurrent Transactions")
plt.legend()
plt.show()
```

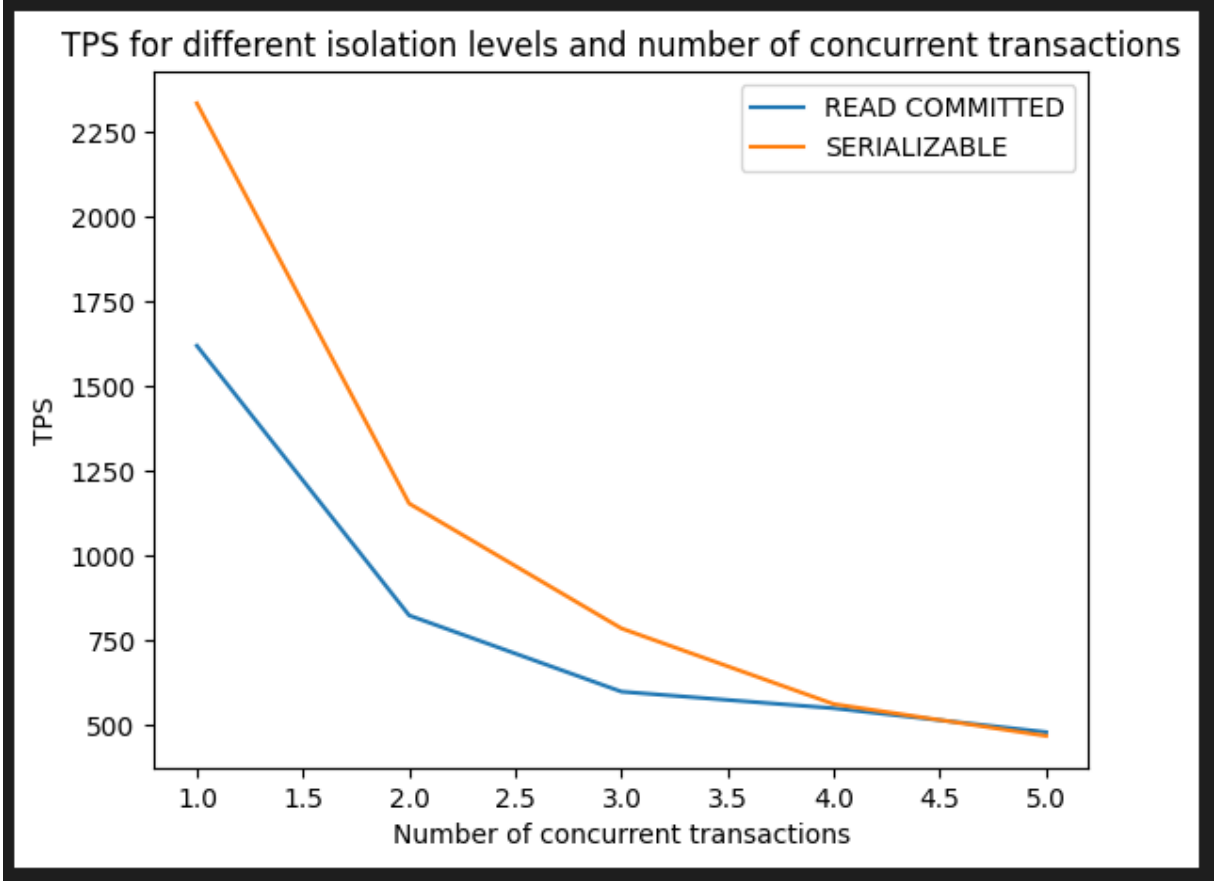


Read Committed and Serializable Comparison Charts:






```
<function matplotlib.pyplot.show(close=None, block=None)>
```



Comparing the Results

In general, the SERIALIZABLE isolation level will have a higher c value, indicating that the results of the transactions are more correct. This is because SERIALIZABLE isolation level provides a higher level of consistency, ensuring that all transactions are executed in a way that preserves the consistency of the data. However, this also comes at a cost of lower throughput or TPS values as SERIALIZABLE isolation level uses a higher level of locking in order to ensure consistency, which can lead to conflicts and delays.

On the other hand, the READ COMMITTED isolation level will have a lower c value, indicating that the results of the transactions may not be entirely correct. However, the READ COMMITTED isolation level will have a higher TPS value as it uses a lower level of locking and has less conflicts, allowing transactions to be executed faster.

In conclusion, depending on the use case, one isolation level may be more suitable than the other. If consistency is a higher priority, SERIALIZABLE isolation level may be more appropriate, while if performance is more important, READ COMMITTED isolation level may be a better option.