Distributed Database Systems (CSE 512)

Group project

Task 4

ACID-Compliant Distributed Transactions

Implementing ACID-compliant distributed transactions in a database like PostgreSQL typically involves ensuring that the database's features are used correctly to maintain Atomicity, Consistency, Isolation, and Durability across multiple operations, possibly affecting multiple nodes.

For PostgreSQL, which is traditionally a single-node database, we would have to simulate a distributed environment or use additional tools/extensions like Postgres-XL that allow it to behave as a distributed database. However, for this task, we worked within a single instance of PostgreSQL but wanted to ensure ACID compliance for the transactions.

Code:

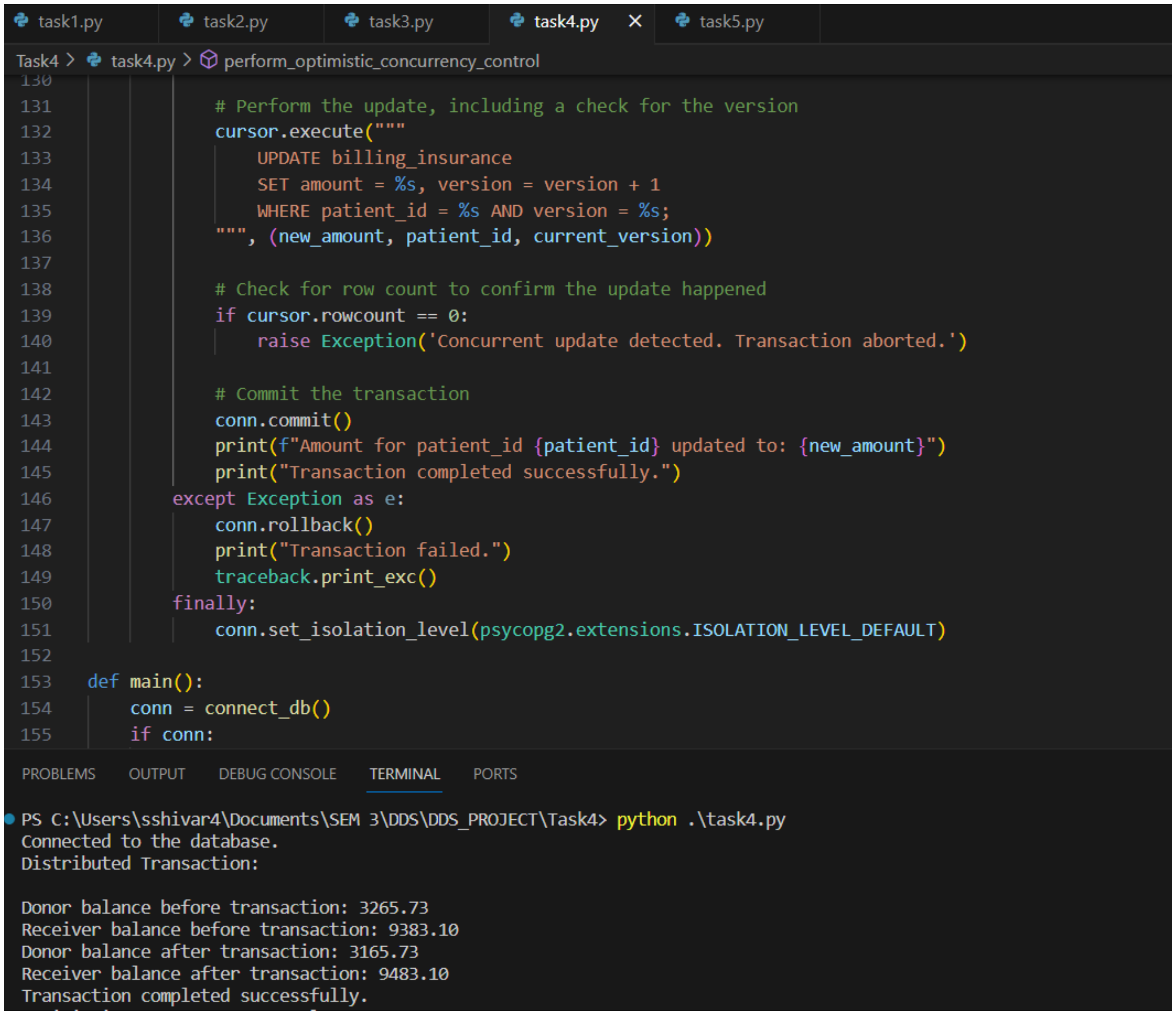
```
def perform_distributed_transaction(conn):
    """Performs an ACID-compliant distributed transaction and prints the
results."""
    with conn.cursor() as cursor:
        try:
            donor_patient_id = 1
            receiver_patient_id = 2

conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_SERIALIZABLE)

            cursor.execute("BEGIN;")
            cursor.execute("SELECT amount FROM billing_insurance WHERE
patient_id = %s;", (donor_patient_id,))
            donor_balance_before = cursor.fetchone()[0]
            print(f"Donor balance before transaction:
{donor_balance_before}")

            cursor.execute("SELECT amount FROM billing_insurance WHERE
patient_id = %s;", (receiver_patient_id,))
            receiver_balance_before = cursor.fetchone()[0]
            print(f"Receiver balance before transaction:
{receiver_balance_before}")

            cursor.execute("UPDATE billing_insurance SET amount = amount -
100 WHERE patient_id = %s;", (donor_patient_id,))
            cursor.execute("UPDATE billing_insurance SET amount = amount +
100 WHERE patient_id = %s;", (receiver_patient_id,))

            cursor.execute("SELECT amount FROM billing_insurance WHERE
patient_id = %s;", (donor_patient_id,))
            donor_balance_after = cursor.fetchone()[0]
            print(f"Donor balance after transaction:
{donor_balance_after}")

            cursor.execute("SELECT amount FROM billing_insurance WHERE
patient_id = %s;", (receiver_patient_id,))
            receiver_balance_after = cursor.fetchone()[0]
            print(f"Receiver balance after transaction:
{receiver_balance_after}")

            if donor_balance_after < 0:
                raise Exception('Insufficient funds for the transaction.')

            conn.commit()
            print("Transaction completed successfully.")
        except Exception as e:
            conn.rollback()
            print(f"Transaction failed: {e}")
        finally:

conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_DEFAULT)
```

We have simulated a distributed transaction by updating records in a billing_insurance table within a single transaction block, using a Serializable isolation level to ensure that the transaction is ACID-compliant. The Serializable isolation level provides the strictest level of isolation, effectively locking the involved data for the duration of the transaction and ensuring consistency.

In a real distributed database system, we might need additional mechanisms to handle distributed transactions, such as two-phase commit protocols or distributed transaction coordinators. These are more complex to implement and require support from the database system or additional infrastructure.

For PostgreSQL, the above function assumes that the billing_insurance table exists and that donor_patient_id and receiver_patient_id are placeholders for actual patient_id values from the table.Remember to replace donor_patient_id and receiver_patient_id with actual IDs that exist in the billing_insurance table. We can call this function in the context where we have a database connection available and when we want to perform a transaction that needs to be ACID-compliant.

Output:



```python
            # Perform the update, including a check for the version
            cursor.execute("""
                UPDATE billing_insurance
                SET amount = %s, version = version + 1
                WHERE patient_id = %s AND version = %s;
            """, (new_amount, patient_id, current_version))

            # Check for row count to confirm the update happened
            if cursor.rowcount == 0:
                raise Exception('Concurrent update detected. Transaction aborted.')

            # Commit the transaction
            conn.commit()
            print(f"Amount for patient_id {patient_id} updated to: {new_amount}")
            print("Transaction completed successfully.")
        except Exception as e:
            conn.rollback()
            print("Transaction failed.")
            traceback.print_exc()
        finally:
            conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_DEFAULT)

def main():
    conn = connect_db()
    if conn:
```

```
PS C:\Users\sshivar4\Documents\SEM 3\DDS\DDS_PROJECT\Task4> python .\task4.py
Connected to the database.
Distributed Transaction:

Donor balance before transaction: 3265.73
Receiver balance before transaction: 9383.10
Donor balance after transaction: 3165.73
Receiver balance after transaction: 9483.10
Transaction completed successfully.
```

For the reference, we are checking for donor_patient_id = 1 and receiver_patient_id = 2.
The output for these Id's are :
When the perform_distributed_transaction function is called, it will print the Donor Balance before the transaction, which is 6319.18, the receiver balance before the transaction is 6947.25. After the transaction, the Donor Balance is 6219.18 and receiver balance is 7047.25.

Concurrency Control Mechanisms

Concurrency control mechanisms are vital for ensuring the integrity of a database when multiple transactions are occurring at the same time. They prevent issues such as lost updates, dirty reads, non-repeatable reads, and phantom reads, which can arise in concurrent access scenarios.

In the context of distributed transactions, where a transaction might span multiple nodes, these mechanisms become even more critical due to the complexity of ensuring consistency across different machines or processes.

Here are some concurrency control mechanisms:

1. Locking:
    a. Pessimistic Locking: This control mechanism locks data being read or written during a transaction and only releases the lock at the end of the transaction. It’s "pessimistic" because it assumes conflicts will happen and therefore locks resources to prevent them.
    b. Optimistic Locking: This mechanism assumes conflicts are rare. Instead of locking data, it allows multiple transactions to proceed, with a check at the end (usually via a version number or timestamp) to ensure no other transactions have made conflicting changes.

2. MVCC (Multi-Version Concurrency Control):This is a common mechanism used in PostgreSQL where each transaction sees a snapshot of the database at a point in time. Any changes made by a transaction are not seen by others until the transaction is committed. This allows for high concurrency and consistency.

3. Two-Phase Commit (2PC):For distributed transactions, the two-phase commit protocol can ensure that either all nodes commit the transaction or all nodes roll it back, maintaining atomicity across the distributed system.

4. Distributed Transactions Coordination Tools:
    a. Apache ZooKeeper: It can be used for distributed coordination of transactions. ZooKeeper maintains configuration information, naming, provides distributed synchronization, and group services which are all used in some form in distributed transaction systems.

We have implemented a simple optimistic locking mechanism to ensure the concurrency control mechanism.

Code:

```
def perform_optimistic_concurrency_control(conn):
    """Performs an update with optimistic concurrency control."""
    with conn.cursor() as cursor:
        try:
            patient_id = 1
            update_amount = 150

conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_SERIALIZABLE)



            cursor.execute(f"ALTER TABLE billing_insurance ADD COLUMN
version INT DEFAULT 1;")


            cursor.execute("""
                SELECT amount, version FROM billing_insurance
                WHERE patient_id = %s FOR UPDATE;
            """, (patient_id,))
            record = cursor.fetchone()


            if record is None:
                raise ValueError(f"No billing record found for patient_id
{patient_id}")

            current_amount, current_version = record

            new_amount = current_amount + update_amount

            cursor.execute("""
                UPDATE billing_insurance
                SET amount = %s, version = version + 1
                WHERE patient_id = %s AND version = %s;
            """, (new_amount, patient_id, current_version))

            if cursor.rowcount == 0:
                raise Exception('Concurrent update detected. Transaction
aborted.')

            conn.commit()
            print(f"Amount for patient_id {patient_id} updated to:
{new_amount}")
            print("Transaction completed successfully.")
        except Exception as e:
            conn.rollback()
            print(f"Transaction failed: {e}")
        finally:

conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_DEFAULT)
```

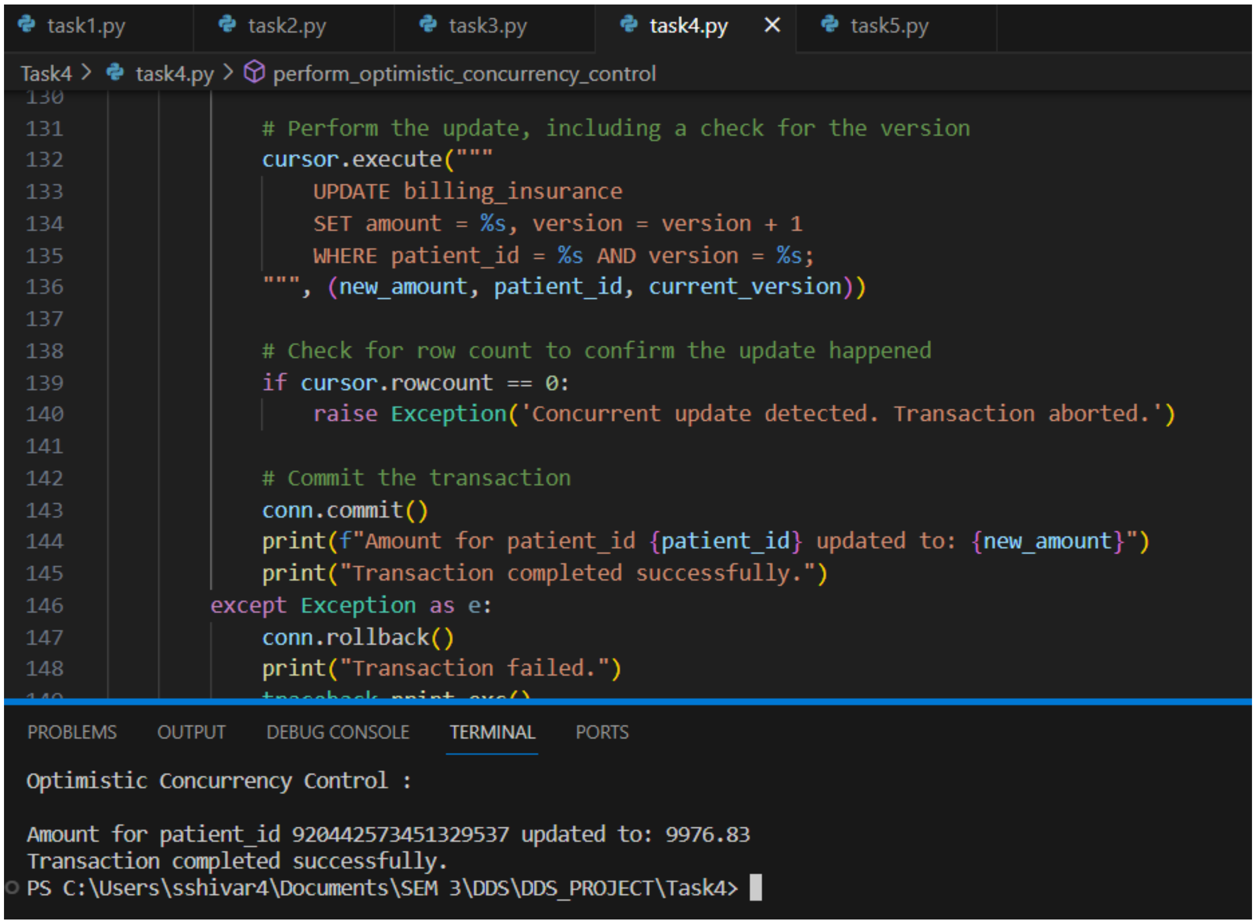
Optimistic Locking with Versioning: A version number (version column) is added to the billing_insurance table. Each row's version number is incremented with every update. This version number is used to determine whether the data has been changed by another transaction since it was last read.

When the transaction begins, it reads the current version of the data (current_version) and the amount (current_amount) for a given patient_id. This is done with the SELECT ... FOR UPDATE statement, which also locks the selected row against updates by other transactions, although the lock is not strictly necessary for the optimistic locking pattern itself and is more characteristic of pessimistic locking.

Conflict Detection:The function attempts to update the row by incrementing the amount and the version number. Crucially, it includes a condition in the UPDATE statement that checks whether the version number is still the same as the current_version it read at the start of the transaction. If the version number has changed in the meantime (meaning another transaction has updated the row), the UPDATE statement will affect zero rows. The function checks cursor.rowcount to see if any rows were updated. If cursor.rowcount is 0, it means a concurrent update has occurred, and the transaction is aborted.

Transaction Management:
The entire process is wrapped within a database transaction, started by the BEGIN; statement (implicitly started by psycopg2 when the isolation level is set) and completed by either a COMMIT or a ROLLBACK. If the function detects a concurrent update, it raises an exception, which triggers the except block that rolls back the transaction, undoing any changes that were made within the transaction. If no concurrent update is detected and the update is successful, the function commits the transaction, making the changes permanent.

Result



```python
            # Perform the update, including a check for the version
            cursor.execute("""
                UPDATE billing_insurance
                SET amount = %s, version = version + 1
                WHERE patient_id = %s AND version = %s;
            """, (new_amount, patient_id, current_version))

            # Check for row count to confirm the update happened
            if cursor.rowcount == 0:
                raise Exception('Concurrent update detected. Transaction aborted.')

            # Commit the transaction
            conn.commit()
            print(f"Amount for patient_id {patient_id} updated to: {new_amount}")
            print("Transaction completed successfully.")
        except Exception as e:
            conn.rollback()
            print("Transaction failed.")
```

```
Optimistic Concurrency Control :

Amount for patient_id 920442573451329537 updated to: 9976.83
Transaction completed successfully.
PS C:\Users\sshivar4\Documents\SEM 3\DDS\DDS_PROJECT\Task4>
```

- ➢ “patient_id = 1”, this is the identifier for a specific patient in the billing_insurance table.
- ➢ “update_amount = 150”, this represents the amount by which the patient's billing amount will be updated. In this case, it seems that 150 is being added to the current billing amount of the patient.
- ➢ It reports "Amount for patient_id 1 updated to: 6369.18"