Database Management Homework 5

Po-Yen Chu

1 Question 1

The implementation using Python and SQLAlchemy library is shown below:

```
from sqlalchemy import create_engine, Column, Integer, Numeric
from sqlalchemy.orm import sessionmaker, declarative_base

DATABASE_URL = "sqlite:///bank.db"
Base = declarative_base()

class Account(Base):
    __tablename__ = "accounts"
    account_id = Column(Integer, primary_key=True)
    balance = Column(Numeric, nullable=False)

engine = create_engine(DATABASE_URL)
Session = sessionmaker(bind=engine)

def transfer_funds(sender_id, receiver_id, amount):
    session = Session()
    try:
        sender = session.query(Account).filter_by(account_id=
           sender_id).one()
        sender.balance -= amount
        if sender.balance < 0:
            raise Exception("Insufficient funds")
        receiver = session.query(Account).filter_by(account_id
           =receiver_id).one()
        receiver.balance += amount
        session.commit()
        print("Transaction completed successfully!")
    except Exception as e:
        session.rollback()
        print(f"Transaction failed: {e}")
        raise
    finally:
```

```
        session.close()

if __name__ == "__main__":
    try:
        transfer_funds(sender_id=1, receiver_id=2, amount=100)
    except Exception as e:
        print(f"Error occurred: {e}")
```

The code manages a transactional workflow using SQLAlchemy. It performs the following steps:

1. Defines a database schema for an `accounts` table with `account_id` and `balance`.
2. Establishes a connection to a SQLite database.
3. Implements a function, `transfer_funds(sender_id, receiver_id, amount)`, which:
   - Deducts the specified `amount` from the sender's account.
   - Checks if the sender's balance is sufficient. If not, it raises an exception and rolls back the transaction.
   - Adds the `amount` to the receiver's account.
   - Commits the transaction if no errors occur.
4. Demonstrates a transfer operation by moving 100 units from account 1 to account 2.

The program ensures transactional integrity, handling exceptions with rollbacks and closing resources after execution.

2 Question 2

The implementation of two transactions performed separately by two clients (in this case is performed by two threads) is shown below:

```
import psycopg2
from threading import Thread
import time
from dotenv import load_dotenv
import os

load_dotenv()

DATABASE_CONFIG = {
    "dbname": os.getenv("DB_NAME"),
    "user": os.getenv("DB_USER"),
```

```
    "password": os.getenv("DB_PASSWORD"),
    "host": os.getenv("DB_HOST"),
    "port": os.getenv("DB_PORT")
}

def transaction_1():
    conn = psycopg2.connect(**DATABASE_CONFIG)
    cursor = conn.cursor()
    cursor.execute("BEGIN;")
    print("Transaction 1: Updating balance to 1000")
    cursor.execute("UPDATE accounts SET balance = 1000 WHERE
       account_id = 1;")
    time.sleep(10)
    print(f"Transaction 1 failed")
    conn.rollback()
    cursor.close()
    conn.close()

def transaction_2():
    conn = psycopg2.connect(**DATABASE_CONFIG)
    cursor = conn.cursor()
    time.sleep(3)
    cursor.execute("BEGIN;")
    cursor.execute("SET TRANSACTION ISOLATION LEVEL READ
       UNCOMMITTED;")
    print("Transaction 2: Reading balance")
    cursor.execute("SELECT balance FROM accounts WHERE
       account_id = 1;")
    result = cursor.fetchone()
    print(f"Transaction 2: Read balance = {result[0]}")

    # add 200 to the balance
    new_balance = result[0] + 200
    print(f"Transaction 2: Updating balance to {new_balance}")
    cursor.execute("UPDATE accounts SET balance = %s WHERE
       account_id = 1;", (new_balance,))
    print("Transaction 2: Committing")
    conn.commit()
    cursor.close()
    conn.close()

if __name__ == "__main__":
    t1 = Thread(target=transaction_1)
    t2 = Thread(target=transaction_2)
    t1.start()
    t2.start()
```

```
    t1.join()
    t2.join()
```

In this implementation, we simulate two clients executing separate transactions on the same database.

1. Transaction 1
    - Begins by updating the balance of `account_id = 1` to 1000, but the update is not committed immediately.
    - The transaction is paused for 10 seconds using `time.sleep(10)` to simulate an uncommitted state. After the delay, the transaction is rolled back, meaning no changes to the database are made.
2. Transaction 2
    - Starts 3 seconds after Transaction 1 begins to ensure that the update in Transaction 1 has been executed but not committed yet.
    - It explicitly sets the isolation level to `READ UNCOMMITTED`, which allows Transaction 2 to read the uncommitted data from Transaction 1.
    - Transaction 2 reads the balance of `account_id = 1` while it is still in the uncommitted state from Transaction 1. This is the point where we expect a dirty read to occur in systems that allow it, but in PostgreSQL, which uses the default `READ COMMITTED` isolation level, this will not happen.
    - Transaction 2 reads the balance (which would have been 1000 if PostgreSQL allowed dirty reads) and then updates the balance by adding 200. After that, the changes are committed.
3. Expected Behavior in PostgreSQL
    - PostgreSQL uses `READ COMMITTED` isolation level by default, which is equivalent to `READ UNCOMMITTED` in other databases. This means that even though Transaction 2 sets `READ UNCOMMITTED`, it will not be able to read the uncommitted data from Transaction 1.
    - PostgreSQL will prevent dirty reads, meaning Transaction 2 will not see the intermediate state of the data (i.e., the uncommitted balance of 1000) from Transaction 1.
    - Therefore, the balance read by Transaction 2 will remain the same as before Transaction 1's update, and no dirty reads will occur.

The result of the table before and after the transactions is shown in Figures 1 and 2. The first figure shows the state of the table before the transactions, while the second figure shows the state of the table after the transactions have been completed. The results shown in the two figures show that PostgreSQL does not allow `READ UNCOMMITTED`.

	account_id [PK] integer	balance integer
1	1	500
2	2	300

Figure 1: Table before transactions

	account_id [PK] integer	balance integer
1	1	700
2	2	300

Figure 2: Table after transactions

3 Question 3

(a) The transactions after removing lock operations is in Table 1

T_1	T_2
read_item(Y)	read_item(X)
read_item(X)	read_item(Y)
$X := X + Y$	$Y := Y + X$
write_item(X)	write_item(Y)

Table 1: Operations in Transactions 1 and 2 without Locks

(b) All conflict operations in the transactions are in Table 2.

T_1	T_2	Type of Conflict
read_item(Y)	write_item(Y)	Read-Write
write_item(X)	read_item(X)	Read-Write

Table 2: Conflicting operations between Transaction T_1 and T_2

(c) The serial schedule given T_1 happened before T_2 is in Table 3.

T_1	T_2
read_item(Y)	
read_item(X)	
$X := X + Y$	
write_item(X)	
	read_item(X)
	read_item(Y)
	$Y := Y + X$
	write_item(Y)

Table 3: Serial Schedule

(d) The non-serial schedule that is equivalent to the serial schedule in (c) does not exist, since the one operation in one of the pairs of the conflicting operations is the last operation in T_1 while the other is at the beginning of T_2.

(e) Table 4 shows the non-serial schedule that is not equivalent to Table 3. These two schedules are not equivalent because the first schedule ensures that all operations result in a serialized outcome, while the second schedule allows T_2 to read outdated data before T_1 completes its update, violating the serialized result.

T_1	T_2
read_item(Y)	
read_item(X)	
$X := X + Y$	
	read_item(X)
write_item(X)	
	read_item(Y)
	$Y := Y + X$
	write_item(Y)

Table 4: Non-Serial Schedule not Equivalent to Table 3

(f) Table 5 shows transactions implementing 2-phase locking.

T_1	T_2
read_lock(Y)	read_lock(X)
read_item(Y)	read_item(X)
read_lock(X)	read_lock(Y)
read_item(X)	read_item(Y)
$X := X + Y$	$Y := Y + X$
upgrade_lock(X)	upgrade_lock(Y)
write_item(X)	write_item(Y)
unlock(X)	unlock(Y)
unlock(Y)	unlock(X)

Table 5: Transactions with 2-Phase Locking

(g) Whether adding locks or not does not affect if it is serializable, thus no equivalent non-serial schedule.

(h) Table 6 shows the schedule with deadlock implementing 2-phase locking.

4 Question 4

(a) The followings are the four SQL query required:

```
-- Query for Step 1:
DO $$
BEGIN
    IF (SELECT remaining_ticket_qty FROM REMAINING_TICKET
```

T_1	T_2
read_lock(Y)	
read_item(Y)	
	read_lock(Y)
	read_item(Y)
	read_lock(X)
	read_item(X)
read_lock(X)	
read_item(X)	
$X := X + Y$	
upgrade_lock(X)	
	read_lock(Y)
	read_item(Y)
	$Y := Y + X$
	upgrade_lock(Y)
	write_item(Y)
	unlock(Y)
	unlock(X)
write_item(X)	
unlock(X)	
unlock(Y)	

Table 6: Schedule with Deadlock and 2-Phase Locking

```
        WHERE travel_date = '2024-12-10' AND train_id = '
           T123') <= 0 THEN
        RAISE EXCEPTION 'No tickets available';
    END IF;
END $$;

-- Query for Step 3:
DO $$
BEGIN
    IF (SELECT remaining_ticket_qty
        FROM REMAINING_TICKET
        WHERE travel_date = '2024-12-10'
          AND train_id = 'T123') > 0 THEN
        -- Update remaining tickets
        UPDATE REMAINING_TICKET
        SET remaining_ticket_qty = remaining_ticket_qty -
           1
        WHERE travel_date = '2024-12-10'
          AND train_id = 'T123';

        -- Insert purchase record
        INSERT INTO PURCHASE (customer_id, travel_date,
```

```
            train_id, purchase_datetime)
        VALUES ('C001', '2024-12-10', 'T123',
            CURRENT_TIMESTAMP);
    ELSE
        -- If no tickets remain, raise a notice
        RAISE NOTICE 'Sorry, tickets are sold out.';
    END IF;
END $$;
```

(b) There is no need to raise the level to unrepeatable read since after the user passed step 1, the system does not guarantee that the number of remaining tickets is correct until step 3. The design of the system aimed to only assure people who filled their payment information and pressed confirmed to have tickets.

(c) The only lock we needed to add is in the search query for step 3 by adding a `FOR UPDATE` to ensure that the customer get the ticket after the system checked the number of remaining tickets. The complete query is shown below:

```
BEGIN;

-- Query for Step 1:
DO $$
BEGIN
    IF (SELECT remaining_ticket_qty FROM REMAINING_TICKET
        WHERE travel_date = '2024-12-10' AND train_id = '
            T123') <= 0 THEN
        RAISE EXCEPTION 'No tickets available';
    END IF;
END $$;

-- Query for Step 3:
DO $$
BEGIN
    IF (SELECT remaining_ticket_qty
        FROM REMAINING_TICKET
        WHERE travel_date = '2024-12-10'
          AND train_id = 'T123'
        FOR UPDATE) > 0 THEN
        -- Update remaining tickets
        UPDATE REMAINING_TICKET
        SET remaining_ticket_qty = remaining_ticket_qty -
            1
        WHERE travel_date = '2024-12-10'
          AND train_id = 'T123';

        -- Insert purchase record
```

```
        INSERT INTO PURCHASE (customer_id, travel_date,
           train_id, purchase_datetime)
        VALUES ('C001', '2024-12-10', 'T123',
           CURRENT_TIMESTAMP);
    ELSE
        -- If no tickets remain, raise a notice
        RAISE NOTICE 'Sorry, tickets are sold out.';
    END IF;
END $$;

COMMIT;
```

(d) The only lock we needed to add is in step 1 by adding a `FOR UPDATE` to ensure that the customer get the ticket after the system checked the number of remaining tickets. The complete query is shown below:

```
BEGIN;

-- Query for Step 1:
SELECT remaining_ticket_qty
FROM REMAINING_TICKET
WHERE travel_date = '2024-12-10' AND train_id = 'T123'
FOR UPDATE;

Query for Step 2 & 3:
IF (remaining_ticket_qty > 0) THEN
    -- Here, the customer will enter their details and the
        transaction continues

    -- Update the ticket count and insert the purchase
       record
    UPDATE REMAINING_TICKET
    SET remaining_ticket_qty = remaining_ticket_qty - 1
    WHERE travel_date = '2024-12-10' AND train_id = 'T123'
       ;

    INSERT INTO PURCHASE (customer_id, travel_date,
       train_id, purchase_datetime)
    VALUES ('C001', '2024-12-10', 'T123',
       CURRENT_TIMESTAMP);

ELSE
    RAISE NOTICE 'Sorry, tickets are sold out.';
END IF;

COMMIT;
```

5 Question 5

Relational databases are not well-suited for distribution across multiple machines because they emphasize strong consistency and availability, aligning closely with the CA aspects of the CAP theorem. In distributed systems, ensuring consistency (e.g., ACID properties) across nodes introduces significant latency and complexity, particularly during transactions involving multiple tables and complex joins.

However, the CAP theorem states that in a distributed system, you cannot simultaneously achieve consistency, availability, and partition tolerance. Relational databases prioritize consistency and availability, often at the expense of partition tolerance. This makes them less efficient and more challenging to scale horizontally in large-scale, distributed environments where partition tolerance is critical. As a result, they struggle to match the performance and scalability of systems designed to prioritize partition tolerance and eventual consistency, such as many NoSQL databases.

6 Question 6

(a) Using NoSQL is reasonable for this company because it handles massive, unstructured data efficiently and scales horizontally to manage billions of articles. The flexible schema supports diverse article formats from different sources without rigid normalization, and NoSQL databases are optimized for high-speed searches and analytics, crucial for sentiment analysis and trend tracking. Additionally, built-in sharding simplifies distributing data by sources or time, ensuring efficient query performance. These features make NoSQL a better fit than relational databases for this large-scale, analytics-driven workload.

(b) The sharding criteria can be based on the clients' own brand and, if necessary for the sake of frequency, their competitors' brand. This is because most clients only look for analyses of their own brands and therefore sharding according to brands can increase efficiency. If needed, sharding based on hash can be the second layer to further enhance efficiency.

(c) For most tables related to `CLINIC` like `ROOMS`, `SCHEDULE` and `PERIOD`, they can be sharded according to clinic since an appointment or reservation operation simply related to one clinic. As for `PATIENT`, it can be sharded according to location. It is intuitive to imagine that patients have lower probability to visit clinics that is far from their home.

7 Question 7

(a) There are several pros and cons:

- Advantages
  - Reduced Operational Load: Analytical operations often involve heavy data querying, which can burden the operational database and impact its performance. A data warehouse offloads this load, allowing the operational system to focus on transaction processing.

  - Historical Data Integration: Data warehouses consolidate historical data from various sources, enabling cross-system analysis, whereas operational databases usually store only current data, limiting trend analysis.
  - Data Consistency: Data is cleaned before importing it into the warehouse, ensuring consistency and making it more suitable for analysis.

- Disadvantages
  - Latency: Data import into the warehouse is often batch-processed, which may not reflect the most recent operational data. Furthermore, different system may have different level of latency, incurring even more complicate issues.

(b) In a data warehouse environment for analysis, NoSQL can be advantageous due to its ability to handle massive datasets efficiently through distributed architectures, enabling seamless horizontal scaling. Additionally, NoSQL's flexible schema design is well-suited for analytical use cases where data fields often change or evolve, avoiding the need for costly schema migrations typical in relational databases. These features make NoSQL ideal for dynamic and large-scale analytical workloads.

(c) Advantages includes improved analytical performance, where column stores are particularly efficient for aggregate operations (e.g., SUM, AVG) as they only read relevant columns, while row stores need to read entire rows. Another benefit is that data in the same column is typically similar in type, making it easier to compress, further improving read efficiency.

However, column store also results in lower insert performance, inserting new data involves spreading it across multiple columns, making the process more complex.

Example: When analyzing sales data to calculate the total sales volume (SUM(sales)), a column store only reads the sales column, whereas a row store reads entire rows, incurring unnecessary I/O costs.