CS 348 - Homework 5: Transactions, concurrency, and graph databases (Neo4J).

(90 Points)

Fall 2020
Due on: **11/16/2020 at 11:59 pm**
This assignment is to be completed by individuals. You should only talk to the instructor, and the TAs about this assignment. You may also post questions (and not answers) to Campuswire.

There will be a 10% penalty if the homework is submitted 24 hours after the due date, a 15% penalty if the homework is submitted 48 hours after the due date, or a 20% penalty if the homework is submitted 72 hours after the due date. The homework will not be accepted after 72 hours, as a solution will be posted by then.

Submission Instruction: For questions 2, 3, and 4, write your answers on this word document and generate a pdf file. **Upload the pdf file to Gradescope**.

For questions 5 to 13, write your answers (queries) in the HW5_Neo4j.py file. Upload the file to Brightspace.

Transactions and Concurrency.

Write transactions for each of the following application features. Each transaction should specify the appropriate isolation level that maximize concurrency and reduce overhead (locking overhead) but still provides the required level of consistency the application requires. Explain why you choose each isolation level. If your selected isolation level is not read uncommited (the least restrictive), explain why the less restrictive isolation level is not appropriate for the application. You may include a schedule where the less restrictive schedule violates the level of consistency the application requires.

Question 1) (zero points, solution is provided)

Biker Performance Application: Consider a speed sensor attached to each bike in a bike race. The sensor sends data periodically (e.g., every 2 seconds) about the current speed of a biker to the application backend. The backend has transaction T1 that inserts the data into the following table: bikerStats(bikerID, raceID, timestamp, speed). Another transaction T2 runs every 10 seconds to update a dashboard with the approximate average speed of each biker since the beginning of the trip. Note that multiple instances of transaction T1 run simultaneously to

insert data for different bikers. On the other hand, there is only one instance of T2 that updates the dashboard of all bikers.

Solution:
Transactions:

T1:
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
Start Transaction;
Insert Into bikerStats(bikerID, tournamentID, timestamp, speed) VALUES (v1, v2, v3, v4);
// v1 to v4 are data sent from a speed sensor
Commit;

T2:
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
Start Transaction;
SELECT bikerID, AVG(Speed)
Group By BikerID;

// Send query results to the dashboard module/function
commit;

Why READ UNCOMMITTED was chosen:

Different instances of T1 will never have conflicts since they touch different data (each transaction inserts data for a different biker).

T1 and T2 accessing the same biker data can have interleaving operations. The most interesting schedule of operations is for T1 to insert a new row in bikeStats, T2 computes the average, then T1 commits/aborts. Since T2 does not require a read lock (read uncommitted), T2 reads the uncommited new row that T2 inserted. If T2 aborts for some reason (e.g., system crash), T2 will report an average speed that is not consistent with the data in bikerStats. However, since T2 needs only to report an approximate average, read uncommited is the appropriate level that provides an acceptable consistency for the application and reduces the overhead of obtaining a read lock(s) over the rows of a certain biker.

Why less restrictive isolation levels are not appropriate:
We choose read uncommitted, which is the least restrictive isolation level.

Question 2) (10 points)

Consider two transactions T1 and T2. T1 inserts a customer-purchase data in a large grocery store application. T2 on the other hand runs periodically (e.g., every 10 minutes) to look for customers whose total purchases is greater than $3000. For those customers, T2 adds the customers to the coupon table. Another transaction/feature in the application uses the coupon table data to send valuable coupons to those frequent customers. Coupons should not be sent to customers who do not qualify. For simplicity, you only need to worry about concurrency when accessing the purchase table (you can ignore the coupon table)

For this question, the transactions are included. You only need to specify the isolation level for each transaction.

Transactions

T1:

```
SET TRANSACTION ISOLATION LEVEL Read Uncommitted
Start transaction;
total = 0
For each item the customer has purchased
    Insert into PurchaseItems (purchaseID, productID, quantity, price) VALUES (v1, v2, v3, v4);
    total = total + (quantity * price)

Insert Into purchase(customerID, purchaseID, date, total) VALUES (v5, v1, v6, total)

Communicate with bank to verify customer debit/credit info. and available balance
if card is verified
    commit;
else
    rollback;
```

T2:

```
SET TRANSACTION ISOLATION LEVEL Read Committed
Start Transaction;

Select customerID, sum(total) From purchase
where customerID NOT IN (select customerID from coupons)
Group By CustomerID
Having sum(total) > 3000;

Iterate over the rows from previous query
     Insert into coupons(customerID, Date) Values (customerID, todayDate)

Commit;
```

Why did you select the isolation level?

T1 does not read. Therefore, there is no difference among the isolation levels. We choose read uncommitted since it has the lowest overhead.

T2 needs to be Read Committed. This isolation level ensures that all purchases included in a customer's total are committed (i.e., verified by the bank) before granting coupons to the customer. We avoid the scenario (explained in the next paragraph) where we include a purchase p1 in a customer's total where p1 is aborted later by T1.

Why did not you select a less restrictive isolation level?

The less restrictive isolation level for T2 is read uncommitted. Because this isolation level does not require read locks, it permits dirty reads (reading uncommitted data). Consider a schedule using read uncommitted where T1 inserts purchase p1, T2 computes the total of the customer (including p1), then T1 aborts because of bank verification issue. In this case, if p1 helped the customer to reach the minimum total (3000), then T2 has granted a coupon for a customer who does not qualify.

Question 3) (10 points)

Bonus Calculator: Given the relation salesperson(ssn, name, salary, bonus, totalSale), update the sales bonus for all salespersons. Sales bonus is 10% of the total sales of the salesperson. The transaction should 1) create a pre-bonus report (preBonus) for all salespersons (SFW query), 2) update the bonus (UPDATE command), and 3) create a bonus report (monthlyBonus) for all salespersons (SFW query). Assume other transactions may also be working on the table by updating, deleting, and adding new rows. T1 should generate consistent reports.

Transaction:

```
SET TRANSACTION ISOLATION LEVEL Serializable
Start transaction;
select * from salesperson;

//create report preBonus

UPDATE salesperson SET bonus = bonus + (totalSale * 0.1);

select * from salesperson;

//create report monthlyBonus

commit;
```

Why did you select the isolation level?

The two reports access all the rows in the table. As the reports need to be consistent, we should avoid new salaesperson(s) (phantom data) added between the reports. If the phantom data is allowed, the second report will contain more salesperson(s) than the second one. The serializable isolation level prevents phantom data from happening by locking the whole table/range.

Why did not you select a less restrictive isolation level?

The less restrictive isolation level (repeatable read) allows phantom data as it does not lock the whole table. In this case, we risk having inconsistent reports.

Question 4) (10 points)

Bitcoin PayBuddy: Given the relation bitCoins(customerName, accountNum, secretKey, balance), write a transaction for transferring bitcoins between two customers. The transaction should 1) check the sender has enough balance, 2) deduct money from the sender's balance, 3) add money to the receiver's balance. Assume that your transaction already received the sender and receiver account numbers (senderAccNum, receiverAccNum), and the bitcoins amount to be transferred (amount). Assume multiple instances of the transaction doing multiple money transfer operations can work simultaneously touching the same customer information.

Transaction:

```
SET TRANSACTION ISOLATION LEVEL Repeatable Read
Start transaction;
senderBalance = select balance from bitCoins where accountNum = senderAccNum;

if senderBalance >= amount:

        UPDATE bitCoins SET balance = senderBalance - amount where accountNum =
senderAccNum;

        UPDATE bitcoins SET balance =  balance + amount where accountNum =
receiverAccNum;

commit;
```

Why did you select the isolation level?

The application works with sensitive data and we should use the strongest isolation level. As T1 only works on individual rows (compare with the previous question), there is no phantom data to worry about. Therefore, serializable is not needed and repeatable read is sufficient.

Why did not you select a less restrictive isolation level?

The less restrictive isolation level is read committed, which may allow schedules that violate the consistency of the database. For example, consider the following schedule of two instances of

T1: T1a and T1b. The two transactions try to transfer money from the same customer c1 to two other customers c2 and c3. We only show part of the transactions with B is the balance of c1 and the values read or written to it. Note that T1a releases the shared lock on B after the read (read committed releases shared locks immediately). Therefore, T1b is able to get an exclusive lock on B and update B to 400 (transfer 100 to the other customer). T1a in turn will update B to 200 (based on B = 500, which is no longer true). The final balance of customer c1 will include 200 while it should have been only 100.

T1a: S(B) R(B=500) Rel(S(B)) W(B=500-300)

T1b: S(B) R(B=500) X(B) W(B=500-100) …… Commit

Graph Databases (neo4J).

Question 5 to 13) (60 points)

Please check Questions5to13.txt file for the neo4j questions.