Connor Brown

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 T1 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, purhcaseID, 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 cutomerID, 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 will never have conflicts with itself as each transaction would be done by a different customer and as a result read uncommitted is the best choice.

T2 reads data based on data inserted from T1, and if it begins to read from the purchase table before the customers card information is verified it is possible to get a dirty read if it has to rollback the data from a customer using an invalid card. However if it is the case that we make T2 read committed transaction level we can avoid the dirty read because it will require that the data is committed first.

Why did not you select a less restrictive isolation level?

I could not pick a lower level for T1 as it is the least restrictive level. T2 cannot be at a lower level because we need to be able to stop dirty reads and read uncommitted is the least restrictive level that stops them from happening.

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:

T1:

SET TRANSACTION ISOLATION LEVEL Repeatable Read
Start transaction;

//Create report preBonus

preBonus = SELECT * FROM salesperson;

//Update bonus

For Each salesperson

UPDATE salesperson SET bonus = totalSale * .1;

//Create report monthlyBonus

monthlyBonus = SELECT * FROM salesperson;

Commit;

Why did you select the isolation level?

I selected repeatable read for the isolation level. This is because if we are to assume that other transactions may also be working on the table and we want to ensure that T1 generates consistent reports then we want to make sure that data is not modified while it is working. So we need it to be repeatable read level in order to lock the table.

Why did not you select a less restrictive isolation level?

While a read committed could be used in order to just do the updating for each salesperson it would only lock that row. When it comes to creating a report, we need to be able to get all the information at once and so we need to lock the entire table in order to ensure that the information is all correct when the report is generated. The only way we can lock the entire table is by using repeated read or serializable, and so repeated read is the least restrictive isolation level that can be selected.

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:

T1:

SET TRANSACTION ISOLATION LEVEL Read Committed

Start transaction;

SenderBalance = SELECT balance FROM bitCoins WHERE accountNum = senderAccNum;

RecieverBalance = SELECT balance FROM bitCoins WHERE accountNum = receiverAccNum;

//Check sender balance

If(senderBalance >= amount) {

    //Deduct money from sender balance

UPDATE bitCoins SET senderBalance = senderBalance – amount;

//Add money to receiver

UPDATE bitCoins SET receiverBalance = receiverBalance + amount;

}

Commit;

Why did you select the isolation level?

I selected read committed as the isolation level because the transaction is only dealing with updating individual rows. This allows it to lock the rows it is updating and prevent dirty reads from occurring. Since each transaction is checking if the sender balance is high enough to send the amount it is possible for there to be a dirty read where two transactions use the same sender and both check the sender balance before it has been committed it is possible for them to send more money than they have available and result in a negative balance. To avoid this issue we need an isolation level that prevents dirty reads and so read committed is the best choice.

Why did not you select a less restrictive isolation level?

I selected read committed over read uncommitted because we need to be able to prevent dirty reads and so read committed is the least restrictive level that prevents them.

Graph Databases (neo4J).

Question 5 to 13) (60 points)

Please check Questions5to13.txt file for the neo4j questions.