Assignment 3
Group 5: Qi Lin, Xiaoying Li, Ziying Yu
5/12/2022

Question 1

A schedule would be allowed by the 2PL protocol has a growing phase in which it acquires all the locks it needs; it then has a shrinking phase in which it releases all locks. Once a transaction releases any lock it acquired, it does not acquire locks on any data item.

A schedule would be allowed by the validation protocol if Ti is validated before Tj, then Ti is serialized before Tj. Thus, transactions are serialized in the order they are validated.

A schedule that would be allowed by the 2PL protocol but would not be generated by the validation protocol:

w2(y) w1(x) w2(x)

The above schedule can be generated using 2PL since it can be associated with the following lock and unlock operations:

X-L2(y) w2(y) X-L1(x) w1(x) U1(x) X-L2(x) w2(x) U2(x) U2(y)

Both transactions T1 and T2 hold an appropriate lock during the execution of the operation. Both transactions' lock requests are granted and not conflicted with each other. And both transactions don't acquire locks on any data item once they release any lock they acquired. The growing phase of the above schedule is L2(y) L1(x) L2(x), in which it acquires all the locks it needs to w2(y) w1(x) w2(x). It then has a shrinking phase U2(x) U2(y) in which it releases all locks.

But the above schedule cannot be generated by the validation protocol since the write phase must overlap but the write sets intersect:

validate T1, validate T2, w2(y) w1(x) w2(x)

We cannot identify such a schedule that a validation protocol can generate but would not be allowed by 2PL protocol. Below is the explanation:

Consider a validation schedule S, each transaction T consists of three phases:

- Read phase: transactions read data from database and update local copy.
- Validation phase: check if transactions can apply its updates to database without causing inconsistency.
- Write phase: copy its changes to database.

For each transaction T in S insert locks and unlocks to get S':

- Read phase (when T starts): request S-lock for all read data items of T.
- Validation phase (when T validates): request X-lock for all write data items of T; release S-locks for read-only data items.
- Write phase (when T ends): release all X-locks.

In order to show such a schedule that a validation protocol can generate but would not be allowed by 2PL protocol doesn't exist, i.e. every schedule that a validation protocol can generate is allowed by 2PL protocol, we need to show that the above schedule S' must be allowed by 2PL protocol. Let's prove it by contradiction.

Suppose S' is not allowed by 2PL protocol due to write-read conflict:

S': … L1(x) w2(x) r1(x) validate1 U1(x) …

→ At validation 1: T2 has finished write phase, T1 can observe the change made by T2.

→ T1 cannot validate, since T1 read from some data items written by T2.

→ Contradict to the assumption that schedule S is validation schedule.

Suppose S' is not allowed by 2PL protocol due to write-write conflict:

S': … validate1 L1(x) w2(x) w1(x) U1(x) …

→ Suppose T2 validates first.

→ At validation 1: T2 is also at validation phase.

→ T2 has not finished yet, T1 cannot validate, since T1 overwrites some data items written by T2.

→ Contradict to the assumption that schedule S is validation schedule.

Therefore, we proved that every schedule that a validation protocol can generate is allowed by 2PL protocol, and we cannot identify such a schedule that a validation protocol can generate but would not be allowed by 2PL protocol.

Reference: https://web.stanford.edu/class/cs245/win2020/slides/13-Concurrency-p3.pdf

Question 2

Conservative SGT scheduler:

Same as the basic SGT scheduler, the concurrency control mechanism maintains a graph SSG in which nodes correspond to transactions.

- Step 1:

When a transaction begins execution, the transaction predeclares its read and write operations. And the scheduler maintains a list of its read and write operations that the transaction may request.

- Step 2:

When a transaction Ti requests an operation oi, same as the basic SGT scheduler, before the operation oi belonging to Ti executes, an edge Tj to Ti is added if there exists a conflict operation oj belonging to Tj in the schedule that executed before oi.

- Step 3:

Since the conservative SGT scheduler already knows what operations other transactions will request in the future, an edge Ti to Tk is temporarily added if there exists a conflict operation ok belonging to Tk in the schedule that has not been executed yet.

- Step 4:

If the addition of the temporary edge to the graph SSG results in a cycle, the operation oi is delayed.
Else, if the addition of the temporary edge does not result in the cycle in SSG, the operation is scheduled.

- Step 5:

When all the predeclared read and write operations of a transaction maintained in the scheduler's list are scheduled, the transaction commits.

Question 3

Thrashing is a phenomenon in which the throughput of a system does not increase linearly with the number of users; both data contention and resource contention could cause thrashing, and during thrashing, increasing the number of transactions or MPL causes throughput to reduce.

Since we have infinite resources, the increased data contention is the only reason for thrashing as the MPL is increased beyond a certain level. Due to the mechanism of the dynamic 2PL protocol, we need to block some transactions until they can get the data resource. Such blocking or locking mechanism will lead to the increase of the data contention, as more transactions come up, the data contention will become more severe, which reduces the throughput of the system. That is the reason why dynamic 2PL scheme thrashes with increased MPL.

However, this is not the case in an optimistic scheme, there is no blocking or locking. In each transaction read phase, we can apply infinite resource to establish this phase, there is no data contention, every transactions work on their own data copy, when enter validation phase, we just need to check the conflict, we can commit or abort transactions, but it will not lead any data contention, in the write phase, there is no conflict in this phase. Therefore, as more transactions come up, this protocol will be able to complete more transactions, the throughput of the optimistic scheme keeps increasing.

Question 4

Assume that the protocol uses 2 types of locks but releases them once the operation is done.
Let's consider the following schedule:

w1(a1) r1(a2) w2(a1) r2(a2) w2(a3) w1(a3)

T1	T2
X-lock(a1)	
w(a1)	
Release X-lock(a1)	
S-lock(a2)	
r(a2)	
Release S-lock(a2)	
	X-lock(a1)
	w(a1)
	Release X-lock(a1)
	S-lock(a2)
	r(a2)
	Release S-lock(a2)
	X-lock(a3)
	w(a3)
	Release X-lock(a3)
X-lock(a3)	
w(a3)	
Release X-lock(a3)	

The above schedule satisfied the questions' requirement that transactions acquired an S-lock to read and an X-lock to update data items, but it is not serializable since it forms the cyclic graph and therefore not CSR.

A new protocol that can ensure that serializability is following: we simply only release the lock on a data item ai after I have acquired an lock on the next data item which I wish to lock on at a later time, so the transaction that starts later would not go past the original transaction. The transaction must acquire the lock on next data item before release the lock on current data, which make sure that all transaction cannot execute same data item at same time, thus make the schedule serializable.

Question 5

Similar to the E mode lock for decrementing the data item, we can support another mode lock for incrementing the data item. Let's say a transaction wishing to increment the data item needs to acquire an I mode lock on the hot spot.

Similar to the E mode lock, two I mode locks do not conflict. But I mode lock conflicts with other S and X lock modes.

If we have both E and I mode lock, whether it is the E mode lock followed by the I mode lock or another way around. The second transaction followed by the first transaction can only start if the first transaction successfully tests the escrow logic and then releases the lock. That is to test if the return value by the first lock will succeed for any value of the hot spot in the range[a,b]. If that is the case, the transaction proceeds with its first operation and then starts the second one.
After successfully finishing the first transaction, we test the escrow technique on the second transaction. It tests if the return value by the second lock will succeed for any value of the hot spot in the range[a,b]. If that is the case, the second transaction proceeds with its operation. Else, it can either delay or abort.

Question 6

Based on the U mode lock, we can support another mode lock U' which is the refinement of the update mode lock.
Similar to the U mode lock, U' mode lock does not conflict and is compatible with the S lock. A transaction that needs to update x can acquire a U' lock if a transaction else has S-lock(x).
However, different from the U mode lock, the S lock does conflict and is not compatible with U' lock. A transaction that needs to read x cannot acquire an S lock if a transaction else has U'-lock(x).
Now the modified lock compatibility matrix and the new process is the following:

Granted Mode / Requested Mode	None	S	U	U'	X
S	✓	✓	✓	✓	
U	✓	✓			
U'	✓				
X	✓				

A transaction that needs to read x but may later require to write x, requests a U-lock(x). If this transaction is sure to write x later on, it requests a U'-lock(x) instead of X-lock(x). A U' lock will be granted if other readers are actively reading x. But no new U-lock(x)/S-lock(x)/X-lock(x) once we have requested U'-lock(x), since they all are not compatible with U' mode. Similar to question 4, the U' mode lock acts as a barrier. After successfully acquiring U'-lock(x), the transaction can then request lock conversion to X-lock(x). When all the transactions that had acquired S lock before requesting U' lock finished and released the lock, the conversion request for X-lock can then be granted. In the end, this added U' lock mode can prevent starvation.

Question 7

- Summary

In this paper, the author proposed a new optimistic concurrency control algorithm to improve the scalability of the system. By leveraging the data-driven timestamp management protocol, this control algorithm can avoid some drawbacks in the traditional timestamp ordering system, and thus achieve higher throughput performance without violating serializability.

This new timestamp management protocol assigns read and write timestamps to data items, not to the transaction, then uses these timestamps to compute a valid commit timestamp for every transaction. By using this valid commit timestamp for the transaction, this system can find a logical-time order that enforces serializability even among transactions that would be aborted in the traditional timestamp ordering system. Thus, this system can uncover more concurrency than other timestamp ordering systems.

The core of this protocol is how to determine the commit timestamp by assigning the timestamp to the data item. TicToc calculates the commit timestamp using the version of the data tuple actually read/write rather than the latest version in the database right now. This commit timestamp is bound by the write timestamp (wts) and read timestamp (rts). Here the author defines two types of validation, read validation and write validation. A read by a transaction is valid if and only if the commit timestamp is between the wts and rts. A write by a transaction is valid if and only if the commit timestamp is greater than the rts. We can find these two validations cannot overlap, that is the reason why this method still keeps serializability, the author proves the serializability by using this disjoint property in the later chapter. This protocol follows the OCC algorithm, like read phase, validation phase, and write phase.

When in the validation phase, we need to compute the commit timestamp, it should be the maximum of the read tuple's wts and write tuple rts+1, since we need to check these two validation rules and set the commit timestamp in the correct value. When determining the commit timestamp, we should check whether all version of data it read is valid or not, if some version of data it read (rts) is greater than or equal to the commit timestamp, the read validation is held, thus no further action is required, we can successfully commit this transaction. However, if the rts is smaller than the commit timestamp, here we need to divide it into two cases. If the local wts of the same data is different from the latest wts of this data, which means another transaction has already modified this data, the rts of this data cannot be

extended to commit timestamp, the validation fails, and this transaction will be aborted. If wts matches, but the data is already locked by another transaction, the rts cannot be extended, this transaction will be aborted. Otherwise, this rts can be extended to commit timestamp, which leads to a successful validation. After the system design chapter, the author introduces some detail about the implementation and proves the correctness of this protocol, then describes some optimization works for this protocol, like no-wait locking, preemptive abort, and timestamp history. In the end, the author presents some evaluation works for this protocol and shows that this TicToc protocol can achieve better throughput while reducing the abort rate.

a. What is the principal novelty of the paper?

The novelty of this paper is the timestamp management, how determine the commit timestamp is the most interesting work in this paper, it is similar to dynamic timestamp allocation (DTA), but TicToc assigned timestamp to data tuple not to transaction, which can help this protocol to uncover more concurrency than other timestamp ordering system and reduce the abort rate. The timestamp assignment or computation can be deferred, even if the transactions overlap in physical time, we still can achieve serializability by using logical ordering.

b. Describe the serializable guarantee provided by TicToc. Compare this to Conflict serializability.

TicToc protocol is able to correctly enforce serializability, but is not order-preserving serializable, since the commit order may not be the serial order. The commit timestamp is the transaction's logical commit time, and thus two transactions may commit with the same commit timestamp, therefore, the author defined the serial order in this protocol. Transaction A is ordered before transaction B if A has a smaller commit timestamp or if they have the same commit timestamp, but A commits before B in physical time. If A and B both have the same logical and physical commit time, then they can have arbitrary serial order. The author proved any schedule in TicToc is equivalent to this serial schedule by using three lemmas, which are mentioned in chapter 4.2. For each transaction executed in the actual schedule, all values it observes are identical to the value it would be observed in the serial schedule.

However, in conflict serializability, a schedule is conflict serializable if it is conflict equivalent to some serial schedule. Two schedules are conflict equivalent if and only if they contain the same transactions and the order of conflicting operations is the same. We care about the same order of conflicting operations, if not the same, we can claim that these two schedules are not conflict equivalent, and if there is no serial schedule is conflict equivalent to the schedule, this schedule is not conflict serializable. But in TicToc, we don't care about the order of operations, we care about the timestamp, although it is like a kind of order, the commit timestamp can be the same, since it is a logical commit time. In this protocol, some order of conflict operations may not be the same, but if these transactions can compute the valid commit timestamp and pass the validation phase, this schedule is still serializable.

c. Describe the difference between Timestamp ordering in TicToc and the Multi-version timestamp ordering discussed in class.

In multi-version timestamp ordering, each data item has a sequence of versions, each version contains a read, and write timestamp. When a transaction issues a read operation, it is very easy to return the latest version of data, but in a write operation, we need to compare the timestamp of the transaction and the read/write timestamp of data, then abort or commit. The timestamp ordering in TicToc is different from the multi-version timestamp ordering.

First, we don't have a centralized timestamp allocator in TicToc, since the commit timestamp is lazily computed by the read and write timestamp of the data item. But in multi-version timestamp ordering, we need a timestamp allocator to assign the timestamp, since the protocol needs a unique and monotonically increasing timestamp to assure the serial order.

Second, the commit timestamp in TicToc is a logical time, not a physical commit time. However, in multi-version timestamp ordering, we need the unique timestamp to guarantee serializable, which means the order of timestamp is restricted to its value. But in TicToc, the commit timestamp is computed, even if two transactions have the same commit timestamp, we can still get a serializable schedule if these transactions passed the validation phase.

d. Present a valid critique of the paper (based on experiments or design discussion in the paper).

In the evaluation part of this paper, the author compared five approaches by using TPC-C and YCSB benchmarks, the result showed that this protocol can achieve better performance and lower abort rate. But in the high contention case of the YCSB benchmark, this protocol cannot reduce the abort rate obviously, the author explains this is because the workload is too write-intensive, so I think this protocol cannot reduce the abort rate in a more intensive workload. Although this protocol still achieved higher throughput, we cannot claim that this is because of the new design of timestamp ordering. From figure 10, the performance of the original TicToc is the same as the SILO, a baseline protocol. By adding some optimizations, this TicToc protocol achieved better performance, but we also can add some optimizations in SILO to make it achieve better performance. It's hard to draw a conclusion that the design of timestamp ordering in this protocol is better than the baseline protocol.