CSCI 6333/6315 Database Systems

Spring 2020

ASSIGNMENT 5: Transaction Management

Sample solutions

Problem 1. Explain the distinction between the terms *serial schedule* and *serializable schedule*.

A serial schedule for a list of transactions is a schedule of instructions from these transactions such that (1) all instructions from any given transaction T will be listed together following the same order as they appear in the transaction; and (2) no instructions from are transactions are allowed between any two instructions of T.

Serializable schedule is a schedule of instructions from a list of transactions that is equivalent to a serial schedule.

The distinction between a serial schedule and a serializable schedule is that a serializable schedule allows concurrent execution of transactions, i.e., instructions of one transaction can occur between instructions of another transactions, while a serial schedule does not allow concurrency.

Problem 2. Consider the following two transactions:

T_1 : read(A);
read(B);
if $A = 0$ then $B = B + 1$;
write B;

T_2 : read(B);
read(A);
if $B = 0$ then $A = A + 1$;
write A;

Figure 1. Two transactions T_1 and T_2

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.

There are two possible serial schedules: T_1T_2 or T_2T_1. For the former, we have $A = 0 \vee B = 0$ before T_1T_2 After T_1T_2, we have A=0, B=1, hence the consistency $A = 0 \vee B = 0$ still holds. For the latter, we have $A = 0 \vee B = 0$ before T_2T_1 After T_2T_1, we have A=1, B=0, hence the consistency $A = 0 \vee B = 0$ still holds.

b. Show a concurrent execution of T_1 and T_2 that produces a nonserializable schedule.

T1	T2
Read(A)	
	Read(B)
Read(B)	
Write(B)	
	Read(A)
	Write(A)

c. Is there a concurrent execution of T_1 and T_2 that produces a serializable schedule?

No.
For T_1, the first instruction is Read(A) and the last is Write(B). For T_2,, the first is Read(B) and the last is Write(A). If Read(B) of T_2 is between Read(A) and Write(B) of T_1,, then Read(A) of T_1, cannot be moved past Write(A) of T_2or Write(B) of T_1, cannot be moved past the Read(B) of T_2. The same applies the case that Read(A) of T_1, is between Read(B) and Write(A) of T_2.

Problem 3. Consider the precedence graph in Figure 2. Is the corresponding schedule serializable?

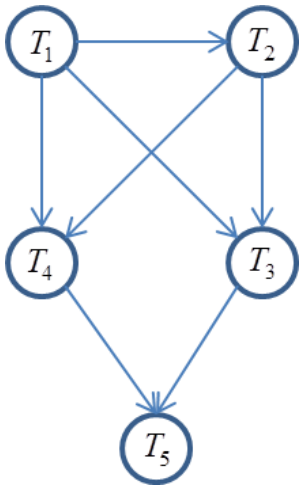

Figure 2. Precedence graph

Yes. The graph has no cycle. One equivalent serial schedule is T_1, T_2, T_3, T_4, T_5.

Problem 4. Show that two-phase locking protocol ensures conflict serializability, and that transaction can be serialized according to their lock points.

Let S be a 2PL schedule for transactions $T_1, T_2, \dots, T_n$. We construct a precedence graph G for S.

We claim that G has no cycle. This claim means that S is conflict serializable.

Proof of the claim. Assume by contradiction that G has a cycle. That is, we have a cycle $T_{i_1} \to T_{i_2} \to \cdots \to T_{i_j} \to T_{i_1}$. Let t(T) denote the lock point, which is the time transaction T requests its last lock. From the precedence graph definition, $T_{i_k} \to T_{i_{k+1}}$ implies two facts: (1) T_{i_k} accessed a data x that had already been accessed by $T_{i_{k+1}}$; and (2) the operation of T_{i_k} on x conflicts with the operation of $T_{i_{k+1}}$ on x. These two facts imply that $T_{i_{k+1}}$ held a lock on x but released its lock before T_{i_k} accessed x, otherwise T_{i_k} was not able to access x. These facts further imply that T_k gained a lock on x after $T_{i_{k+1}}$ released its lock on x. Since the schedule S is a 2PL schedule, we have $t(T_k) > t(T_{k+1})$. Applying this analysis to any two adjacent transactions on the cycle $T_{i_1} \to T_{i_2} \to \cdots \to T_{i_j} \to T_{i_1}$, we have

$$t(T_1) > t(T_2) > \cdots > t(T_j) > t(T_1).$$

Hence, we have a contradiction. Therefore, the graph G has no cycle.

Problem 5. Consider transactions T_1 and T_2 in Figure 1. Add lock and unlock instructions to them so that they observe the two-phase locking protocol. Can the execution of these two transactions result in a deadlock?

T1	T2
Lock-S(A)	
Read(A)	
	Lock-S(B)
	Read(B)
Lock-S(B)	
Read(B)	
	Lock-S(A)
	Read(A)
	Lock-X(A)
	Write(A)
Lock-X(B)	

Write(B)	
	Unlock(B)
	UnLock(A)
Unlock(B)	
UnLock(A)	

Yes, a deadlock occurs.

Problem 6. Show that there are schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa.

A 2PL protocol but not timestamp

T3	T4
Lock(A)	
W(A)	
	Lock(B)
	W(B)
	Unlock(B)
Lock(B)	
R(B)	
Unlock(B)	
Unlock(A)	

A timestamp protocol but not 2PL

T5	T6	
W(A)		
W(B)		
		← for 2PL, T5 must unlock B so that T6 lock B and then write B
	W(B)	
		← for 2PL, T5 must unlock A so that T6 can lock A and then read A
	R(A)	
		← for 2PL, T6 must unlock A so that T5 can lock A and then read A
		This contradicts to 2PL for

	T5
R(A)	

See justification included.

Problem 7. Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect

- System performance when no failure occurs
- The time it takes to recover from a system crash
- The time it takes to recover from a disk crash

If there is no checkpoint, when a failure occurs, the recovery system needs to search the log from the end to the beginning till all active transactions are found. This may, in the worst case reach the beginning of the log, and redo and undo may be done for all the transactions, which is a tedious and time-consuming process. With checkpoints, the system only needs to search to the first checkpoints from the end. At that points, the system knows all the active transactions.

A checkpoint needs to track active transactions. When there is no failure, frequent insertion of checkpoints will increase the overhead of the system performance.

A system crash occurs, the recovery needs to do only two passes of linear scan of the log: One pass from the end to the first checkpoint to undo and another pass from the checkpoint to the end to redo. If more checkpoints are inserted, then it takes less time to recover from a failure.

Checkpoint will not impact on the recovery time from a disk crash, as the latter is dealt with dumping approach.

Problem 8. When the system recovers from a crash, it constructs an undo-list and a redo-list. Explain why log records for transactions on the undo list must be processed in reverse order, while those log records for transactions on the redo-list are processed in a forward direction.

Consider the following log records:
<T1, x, 10, 20>
<T1, x, 20, 30>
<T1, x, 30, 40>

To undo, we'd like to restore the oldest value 10 to x. If we do so in forward order, the value of x will be 30. If we do so in backward order, x will be correctly set to 10.

To redo, if we follow the backward order, we'll set x to 20, which shall be 40. If we follow the forward order, we'll correctly set x to 40.