COMP2350/COMP6350 Database Systems

Tutorial (Sample Solution) – Week 12

The background knowledge for the tutorial questions is given in the textbook(s), lectures, any other components of the unit, and in the readings provided on iLearn. However, some questions cannot be answered without prior independent research and searching for other sources of information. Of the questions below, the (*) starred ones are core questions, and your tutor will try to discuss them before discussing other questions. Note that there might not be enough time to discuss all questions in the class.

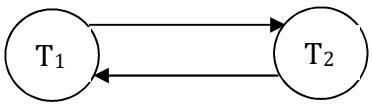
Question 1.

Draw a precedence graph for each of the schedules below, and determine if they are conflict serializable:

a) read(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), write(T_2, bal_x), commit(T_1), commit(T_2)
b) read(T_1, bal_x), read(T_2, bal_y), write(T_3, bal_x), read(T_2, bal_x), read(T_1, bal_y), commit(T_1), commit(T_2)
c) read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), abort(T_2), commit(T_1)
d) write(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), commit(T_2), abort(T_1)
e) read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), read(T_3, bal_x), commit(T_1), commit(T_2), commit(T_3)

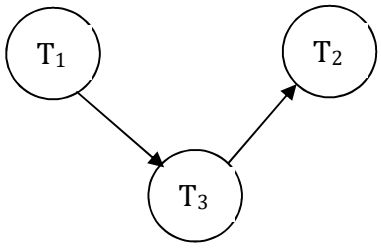
Answer:

(a) read(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), write(T_2, bal_x), commit(T_1), commit(T_2)



Cycle, so not serializable.

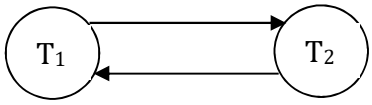
(b) read(T_1, bal_x), read(T_2, bal_y), write(T_3, bal_x), read(T_2, bal_x), read(T_1, bal_y), commit(T_1), commit(T_2)



No cycle, so serializable.

(c) read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), abort(T_2), commit(T_1)

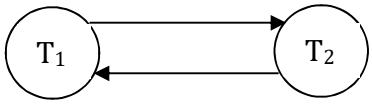
Get following:



cycle, so not serializable.

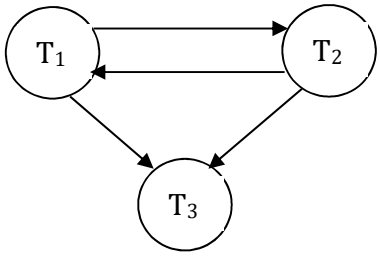
(d) write(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), commit(T_2), abort(T_1)

Get following:



cycle, so not serializable.

(e) read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), read(T_3, bal_x), commit(T_1), commit(T_2), commit(T_3)



Cycle, so not serializable.

Question 2.

Consider the following concurrent schedule (say S):

T1	time	T2
(begin)	t1	
read-lock(F)		
read(F)		
unlock(F)	t2	
	t3	(begin)
		read-lock(G)
		read(G)
		write-lock(F)
		F := G + 1
		write(F)
		unlock(G)
		unlock(F)
	t4	(commit)
write-lock(H)	t5	
H := F + 1		
write(H)		
unlock(H)		
(commit)	t6	

Suppose that the initial values of F, G, and H are all 0, just prior to the execution of S. Does S obey the two-phase locking protocol? Give reasons for your answer.

a) What are the final values of F, G, and H just after S has completed its execution?
b) What are the final values of F, G, and H for the serial execution "T1 then T2"?
c) What are the final values of F, G, and H for the serial execution "T2 then T1"?

Is S conflict serializable? Give reasons for your answer.

Answer

No. T1 releases a lock and then goes on to obtain another lock, hence T1 does not obey two phase locking protocol, so the schedule is not two-phase.

a) F = 1, G = 0, H = 1

b) F = 1, G = 0, H = 1

c) F = 1, G = 0, H = 2

S is conflict serializable. For the sake of simplicity, we only consider the read/write operations in the schedule (shown above). The only conflicting operations in S are read(F) in T1 and write(F) in T2. We can push read(G) and write(F) in T2 all the way past t6 (when T1 commits) because they do not conflict with write(H). (We cannot push them up past t1 because of the conflict.) Then S is conflict equivalent to the serial schedule "T1 then T2".

(*)Question 3.

The following diagram represents a hypothetical concurrent execution of two transactions (hypothetical in the sense that it assumes no specific concurrency control mechanism is in effect).

T1	time	T2
(begin)		t1
	t2	(begin)
lock-S(A)	t3	
	t4	lock-S(B)
lock-S(B)	t5	
	t6	lock-S(A)
upgrade(B)	t7	
write(B)		
	t8	upgrade(A)
		write(A)
	.	
	.	

What would happen if we used the following protocols/schemes:

a) two-phase locking
b) wait-die
c) wound-wait

Answer:

a) We have a deadlock at t8: T1 wants B which is locked by T2, and T2 wants A which is locked by T1.
b) T1 has the timestamp of t1, and T2 has the timestamp of t2 (T2 is younger than T1) If we used wait-die protocol, T2 dies at t8 and rolled back: T2 wants to upgrade its S-lock to X-lock on A which is S-locked by an older transaction T1, so T2 is not allowed to wait and dies.
c) T2 dies at t7 (wounded by T1) and rolled back: T1 wants to upgrade its S-lock to X-lock on B which is S-locked by a younger transaction T2, so T1 does not wait and T2 is wounded by T1.

Question 4.
What is starvation? Name a deadlock prevention protocol which is also starvation-free.

Answer:
Starvation means that some transactions may never get the resources they need to complete their tasks (hence they are starved). Both wait-die and wound-wait protocols avoid starvation. (Eventually any transaction has a chance to be the oldest one around in the case it is repeatedly rolled back and hence get its locks)

(*) Question 5.

What is a timestamp? How do timestamp-based protocols for concurrency control differ from locking based protocols?

Answer:

Timestamp is a unique identifier created by the DBMS that indicates the relative starting time of a transaction.

The use of locks, combined with the 2PL protocol, guarantees serializability of schedules. The order of transactions in the equivalent serial schedule is based on the order in which the transactions lock the items they require. If a transaction needs an item that is already locked, it may be forced to wait until the item is released.

Timestamp methods for concurrency control are quite different; no locks are involved, and therefore there can be no deadlock. Locking methods generally prevent conflicts by making transactions wait. With timestamp methods, there is no waiting; transactions involved in conflict are simply rolled back and restarted.

(See relevant Section in the text book, and learn about Thomas write rule among other things.)