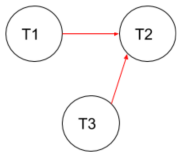
CS W186 Introduction to Database Systems
Spring 2020 Josh Hug, Michael Ball

DIS 9

1 Conflict Serializability

T1		R(A)	W(A)	R(B)					
T2					W(B)	R(C)	W(C)	W(A)	
T3	R(C)								W(D)

(a) Draw the dependency graph (precedence graph) for the schedule.

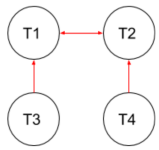


Solution:

(b) Is this schedule conflict serializable? If so, what are all the conflict equivalent serial schedules? If not, why not?
Solution: Yes. T3, T1, T2 and T1, T3, T2. Topologically sorting the above graph gives these schedules.

T1	R(A)		R(B)				W(A)	
T2		R(A)		R(B)				W(B)
T3					R(A)			
T4						R(B)		

(c) Draw the dependency graph (precedence graph) for the schedule.



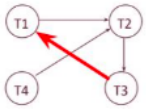
Solution:

(d) Is this schedule conflict serializable? If so, what are all the conflict equivalent serial schedules? If not, why not?
Solution: No, there's a cycle between T1 and T2: T1 must come before T2 and T2, before T1.

2 Deadlock

T1	S(A)	S(D)		S(B)					
T2			X(B)				X(C)		
T3					S(D)	S(C)			X(A)
T4								X(B)	

(a) Draw a "waits-for" graph and state whether or not there is a deadlock
Solution: Yes there is deadlock. There is a cycle between T1, T2, and T3.



3 Locking

T1	T2
Lock_X(B)	
Read(B)	
B := B * 10	
Write(B)	
Lock_X(F)	
Unlock(B)	
	Lock_S(F)
F:= B * 100	
Write(F)	
Commit	
Unlock(F)	
	Read(F)
	Unlock(F)
	Lock_S(B)
	Read(B)
	Print(F + B)
	Commit
	Unlock(B)

(a) What is printed, assuming we initially have B = 3 and F = 300?
Solution: 3030

(b) Does the execution use 2PL, strict 2PL, or neither?
Solution: Neither - T2 unlocks S(F) before it acquires S(B)

(c) Would moving Unlock(F) in the second transaction to any point after Lock_S(B) change this (or keep it) in 2PL?
Solution: Yes - all locks would be acquired (for T2) before any are released.

(d) Would moving Unlock(F) in the first transaction and Unlock(F) in the second transaction to the end of their respective transactions change this (or keep it) in strict 2PL?
Solution: No - T1 still unlocks B before the end of the transaction

(e) Would moving Unlock(B) in the first transaction and Unlock(F) in the second transaction to the end of their respective transactions change this (or keep it) in strict 2PL?
Solution: Yes - all unlocks would only happen when the respective transactions end

4 Multigranularity Locking

(a) Suppose a transaction T1 wants to scan a table R and update a few of its tuples. What kinds of locks should T1 have on R, the pages of R, and the updated tuples?
Solution:

(a) Obtain SIX on R
(b) Obtain IX on Page [We don't obtain a SIX because there is already an S lock on R (from the SIX). Obtaining another S on the Page is redundant.]
(c) Obtain X on Tuples being modified

(b) Is an S lock compatible with an IX lock?
Solution: Suppose T1 wants an S lock on an object, O, and T2 wants an IX lock on the same object O. An S lock implies that T1 will read the entire object (all of its sub-objects). An IX lock implies that T2 will write some of the sub-objects of the object. This means that there is some sub-object of O that T1 will read and T2 will write. This is not valid, so the S and IX locks must be incompatible.

(c) Consider a table which contains two pages with three tuples each, with Page 1 containing Tuples 1, 2, and 3, and Page 2 containing Tuples 4, 5, and 6.

(a) Given that a transaction T1 has an IX lock on the table, an IX lock on Page 1, and an X lock on Tuple 1, which locks could be granted to a second transaction T2 for Tuple 2?
Solution: X, S

(b) Given that a transaction T1 has an IS lock on the table and an S lock on Page 1, what locks could be granted to a second transaction T2 for Page 1?
Solution: S, IS

5 Project Prep

The LockManager in the project will have the following 4 functions:

- `acquire(transaction, resourceName, lockType)`: this method is the standard acquire method of a lock manager. It allows a transaction to request one lock, and grants the request if there is no queue and the request is compatible with existing locks. Otherwise, it should queue the request (at the back) and block the transaction
- `release(transaction, resourceName)`: this method is the standard release method of a lock manager. It allows a transaction to release one lock that it holds.
- `acquireAndRelease(transaction, resourceName, lockType, releaseLocks)`: this method atomically acquires one lock and releases zero or more locks. This method has priority over any queued requests (it should proceed even if there is a queue, and it is placed in the front of the queue if it cannot proceed).