CMPT-454 Fall 2009
Instructor: Martin Ester
TA: Yi Liu

Solution Assignment 5

Total marks: 200 (20 % of the assignments)
Due date: December 2, 2009

Problem 5.1 (50 marks)

For each of the following schedules, state whether they are serializable and whether they are conflict-serializable. For each schedule, draw the corresponding precedence graph. If the schedule is conflict-serializable, show all the conflict-equivalent serial schedules. If the schedule is not serializable, provide an initial DB state and some semantics (pseudo-code) for the transactions for which no serial schedule has the same net effect.

a) S = r1(A) r2(A) w1(A) w2(A)

P(S) = T2 → T1 → T2

The schedule is not conflict-serializable, since there is a cycle in the precedence graph.

To see that the schedule is also not serializable, consider the following example:
A = 10, T1: A:=A*10, T2: A:=A*10
The above interleaved schedule results in A = 100, but both serial schedules result in A = 1000.

b) S = r1(A) r2(B) w3(A) r2(A) r1(B)

P(S) = T1 → T3 → T2

The schedule is conflict-serializable, since its precedence graph does not contain any cycle.

The only conflict-equivalent serial schedule is T1, T3, T2.

The schedule is also serializable, since conflict-serializability implies serializability.

c) S = r1(A) w2(A) w1(A) r3(A)

P(S) = T1 → T2 → T1 → T3

The schedule is not conflict-serializable, because its precedence graph is cyclic.

To see that the schedule is also not serializable, consider the following example:

A = 10, T1: A:= A * 10, T2: A:= 1000, T3: PRINT(A)

S results in A = 100 and PRINT 100. All possible serial schedules have a different net effect as follows:

T1, T2, T3: A = 1000, PRINT 1000
T1, T3, T2: A = 1000, PRINT 100
T2, T1, T3: A = 10000, PRINT 10000
T2, T3, T1: A = 10000, PRINT 1000
T3, T1, T2: A = 1000, PRINT 10
T3, T2, T1: A = 10000, PRINT 10.

Problem 5.2 (50 marks)

Show all the conflict-serializable schedules of the following transactions T1 and T2. Explain why these are all such schedules.

a) T1: r1(A) w1(A) r1(B) w1(B)
T2: r2(B) w2(B) r2(A) w2(A).

conflict-equivalent to T1, T2
r1(A) w1(A) r1(B) **w1(B) r2(B)** w2(B) r2(A) w2(A)

This is the serial schedule T1, T2. There are no other conflict-equivalent schedules, since there is no non-conflicting swap of adjacent actions in this schedule: we cannot swap actions within a transaction, and w1(B) r2(B) cannot be swapped, because they are both manipulating the same database element and one of them is a write.

conflict-equivalent to T2, T1
r2(B) w2(B) r2(A) **w2(A) r1(A)** w1(A) r1(B) w1(B)

This is the serial schedule T2, T1. There are no other conflict-equivalent schedules, since there is no non-conflicting swap of adjacent actions in this schedule: we cannot swap actions within a transaction, and w2(A) r1(A) cannot be swapped, because they are both manipulating the same database element and one of them is a write.

b) T1: r1(A) w1(A) r1(B) w1(B)
T2: r2(A) w2(A) r2(B) w2(B)

Note that the action in **bold** font is the one that has been moved forward in the last swap.

conflict-equivalent to T1, T2
r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)

r1(A) w1(A) r1(B) **r2(A)** w1(B) w2(A) r2(B) w2(B)
r1(A) w1(A) r1(B) r2(A) **w2(A)** w1(B) r2(B) w2(B)
r1(A) w1(A) **r2(A)** r1(B) w1(B) w2(A) r2(B) w2(B)
r1(A) w1(A) r2(A) r1(B) **w2(A)** w1(B) r2(B) w2(B)
r1(A) w1(A) r2(A) **w2(A)** r1(B) w1(B) r2(B) w2(B)

No further swap is possible, since (1) we cannot swap actions of the same transaction and (2) w1(A) r2(A) and w1(B) r2(B) are both manipulating the same database element and one of them is a write.

conflict-equivalent to T2, T1

r2(A) w2(A) r2(B) w2(B) r1(A) w1(A) r1(B) w1(B)
r2(A) w2(A) r2(B) **r1(A)** w2(B) w1(A) r1(B) w1(B)
r2(A) w2(A) r2(B) r1(A) **w1(A)** w2(B) r1(B) w1(B)
r2(A) w2(A) **r1(A)** r2(B) w2(B) w1(A) r1(B) w1(B)
r2(A) w2(A) r1(A) r2(B) **w1(A)** w2(B) r1(B) w1(B)
r2(A) w2(A) r1(A) **w1(A)** r2(B) w2(B) r1(B) w1(B)

No further swap is possible, since (1) we cannot swap actions of the same transaction and (2) w2(A) r1(A) and w2(B) r1(B) are both manipulating the same database element and one of them is a write.

Problem 5.3 (40 marks)

Consider the following schedules of transactions T1, T2 and T3. Insert shared and exclusive lock actions as well as unlock actions and modify the schedule where necessary. Place a shared or exclusive lock immediately before the read or write action that requires it. Place the necessary unlocks at the end of a transaction. If a deadlock occurs, abort the participating transaction that started last, release all its locks and re-start that transaction after all the other participating transactions have committed. Show the complete schedules satisfying the 2PL protocol.

a) r1(A) r2(B) r3(C) r1(B) r2(C) r3(D) w1(A) w2(B) w3(C) w1(A)

T1	T2	T3	
sl-1(A)			
r1(A)			
	sl-2(B)		
	r2(B)		
		sl-3(C)	
		r3(C)	
sl-1(B)			
r1(B)			
	sl-2(C)		
	r2(C)		T2 has to wait until T1 returns the S lock on B
		sl-3(D)	
		r3(D)	T3 has to wait until T2 returns the S lock on C

xl-1(A)
w1(A)
w1(A)
u1(A)
u1(B)

xl-2(B)
w2(B)
u2(B)
u2(C)

xl-3(C)
w3(C)
u3(C)
u3(D)

b) r1(A) r2(B) r3(C) r1(B) r2(C) r3(A) w1(A) w2(B) w3(C) w1(A)

T1	T2	T3	
sl-1(A)			
r1(A)			
	sl-2(B)		
	r2(B)		
		sl-3(C)	
		r3(C)	
sl-1(B)			
r1(B)			
	sl-2(C)		
	r2(C)		
		sl-3(A)	
		r3(A)	
xl-1(A)			T1 has to wait until T3 releases the S lock on A
	xl-2(B)		T2 has to wait until T1 releases the S lock on B
		xl-3(C)	T3 has to wait until T2 releases the S lock on C
		abort	
		u3(A)	
		u3(C)	
xl-1(A)			
w1(A)			
w1(A)			
u1(A)			
u1(B)			
	w2(B)		
	u2(B)		
	u2(C)		
		sl-3(C)	restart T3
		r3(C)	
		sl-3(A)	

r3(A)
xl-3(C)
w3(C)
u3(A)
u3(C)

Note that the lock requests in **bold font** cannot be granted right away, and the requesting transactions have to wait.

Problem 5.4 (60 marks)

Provide one example schedule (including the commits) of two transactions (that both commit, i.e. no aborts involved) with each of the following properties. In concurrency control by validation, suppose that validation is performed at the time of the commit. To show that a schedule validates or does not validate, show the read and write sets for the two transactions and show that both validation rules are satisfies or list the validation rule violated. To show that a schedule satisfies or does not satisfy the 2PL protocol, show the modified schedule after inserting shared and exclusive lock actions right before the corresponding read / write action and unlock actions right before the commit.

a) The schedule is allowed by 2PL, but not by concurrency control by validation.

T1: r1(A) w1(A) commit
T2: r2(B) w2(B) r2(A) w2(A) commit

To see that the schedule is allowed by 2PL:

T1	T2
sl-1(A)	
r1(A)	
	sl-2(B)
	r2(B)
	xl-2(B)
	w2(B)
xl-1(A)	
w1(A)	
u1(A)	
commit	
	sl-2(A)
	r2(A)
	xl-2(A)
	w2(A)
	u2(A)

u2(B)
commit

The schedule does not validate, since it violates the first validation rule:
RS(T1) = {A}, WS(T1) = {A}
RS(T2) = {A, B}, WS(T2) = {A, B}
FIN(T1) > START(T2) and $RS(T2) \cap WS(T1) = \{A\}$.
Note that FIN(T1) > VAL(T1) = COMMIT(T1) > START(T2).

b) The schedule is allowed by concurrency control by validation, but not by 2PL.

T1:	r1(A) w1(B)	r1(C) commit	
T2:	w2(A)		r2(C) w2(C) commit

To see that the schedule validates:
RS(T1) = {A, C}, WS(T1) = {B}
RS(T2) = {C}, WS(T2) = {A, C}
T1 trivially validates, because there is no other transaction that is validated earlier.
FIN(T1) > START(T2) holds, but $RS(T2) \cap WS(T1) = \varnothing$. We do not know whether FIN(T1) > VAL(T2), but anyway $WS(T2) \cap WS(T1) = \varnothing$. Therefore, T2 also validates.

To see that the schedule is not allowed by 2PL:

T1	T2	
sl-1(A)		
r1(A)		
xl-1(B)		
w1(B)		
sl-1(C)		
r1(C)		
u1(A)		
u1(B)	xl-2(A)	T2 has to wait until T1 releases the S lock on A
u1(C)	w2(A)	
commit	sl-2(C)	
	r2(C)	
	xl-2(C)	
	w2(C)	
	commit	