1 Question 1

Given the following concurrent transaction schedule, draw the corresponding precedence graph and give a serial order of execution based on a topological sort order.

1.1 Solution

We should have edges: {(T1, T2), (T1, T3), (T1, T4), (T2, T4), (T3, T4)}.

We find one of the serial orders:

- T5, T1, T2, T3, T4
- T5, T1, T3, T2, T4
- Note: T5 can be placed anywhere

2 Question 2

Consider the following schedule, S:

T1	T2	T3
R(X)		
R(Y)		
W(X)		
	R(Y)	
		W(Y)
W(X)		
	R(Y)	

1. Is this schedule conflict-serializable? Why or why not?
2. Is this schedule view-serializable? Why or why not?
3. Determine the precedence graph for this schedule.

For each of the following, modify S to create a complete schedule that satisfies the stated condition. You may move around read or write operations in any way you'd like, but you may not add or remove any read or write operations. You may add commits or aborts.

4. Resulting schedule is recoverable.
5. Resulting schedule avoids cascading aborts but is not recoverable.
6. Resulting schedule is conflict-serializable.

2.1 Solution

1. Is this schedule conflict-serializable? Why or why not?

No. It is not conflict-serializable because a cycle exists in the precedence graph.

T1	T2	T3	T4	
	read(A)			
read(B)				
read(C)				
	read(B)			
	write(B)			
		write(C)		
read(G)				
			read(B)	
			write(A)	
			read(C)	
			write(C)	
read(D)				
write(G)				

Figure 1: Q1

2. Is this schedule view-serializable? Why or why not?

Yes. Even thought it is not conflict-serializable, it is view-equivalent. View-equivalence requires three conditions be met:

i. T2 and T3 read the same initial value of Y.
ii. T2 will read T3's written value of Y.
iii. T3 performs the final write in all possible schedules.

Since it is view-equivalent to the serial schedule T1, T3, T2, it is view-serializable.

3. Determine the precedence graph for this schedule.

Should have edges {(T1, T3), (T2, T3), (T3, T2)}. Cycle exists.

4. Resulting schedule is recoverable.

T1	T2	T3
R(X)		
R(Y)		
W(X)		
W(X)		
Comm		
		W(Y)
		Comm
	R(Y)	
	R(Y)	
	Comm	

T3 should commit before T2.

5. Resulting schedule avoids cascading aborts but is not recoverable.

Not possible; requires each pair Ti, Tj to have both of the following: If Tj read an object previously written by Ti, Tj commits after Ti commits (recoverable), AND If Tj reads an object previously written by Ti, Ti commits before the read operation of Tj.

6. Resulting schedule is conflict-serializable.

T1	T2	T3
R(X)		
R(Y)		
W(X)		
		W(Y)

T1	T2	T3
		Comm
	R(Y)	
W(X)		
Comm		
	R(Y)	

There are many possible conflict-serializable schedules.

3 Question 3

Consider the following transactions:

```
T1:
read(B);
read(A);
if(A > 0) then A := -B;
write(A);

T2:
read(A);
read(B);
if(B > 0) then B := -A;
write(B);
```

Assume that the database is consistent if A = 1 or B = 1, with A = B = 1 as initial values.

1. Show that every serial execution involving these two transactions preserves the consistency of the database.
2. Show a concurrent execution of T1 and T2 that produces a non-serializable schedule.
3. Is there a concurrent execution of T1 and T2 that produces a serializable schedule? If such execution exists, show an example, otherwise prove why it doesn't exist. (Hint: can we swap?)
4. Add locking instructions to the above transactions so that they are compatible with 2-Phase Locking.
5. Can your new transaction code result in a deadlock? Explain your answer.

3.1 Solution

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

We notice that there are two serial schedules, both of which end up with A = 1 or B = 1.

T1	T2
R(B)	
R(A)	
if A>0: A := -B	
W(A)	
	R(A)
	R(B)
	if B>0: B := -A
	W(B)

T1	T2
	R(A)
	R(B)
	if B>0: B := -A
	W(B)
R(B)	
R(A)	
if A>0: A := -B	
W(A)	

2. Show a concurrent execution of T1 and T2 that produces a non-serializable schedule.

Consider the following execution plan:

T1	T2
R(B)	
R(A)	
	R(A)
	R(B)
if A>0: A := -B	
W(A)	
	if B>0: B := -A
	W(B)

The final state is A = -1 and B = -1, which fails our consistency requirement. We notice that this is non-serializable, since we will never be able to swap write(A) in T1 above read(A) in T2. If we tried putting all of T1 below T2, we would fail, since we cannot swap write(B) in T2 above read(B) in T1. Thus, this in non-serializable.

3. Is there a concurrent execution of T1 and T2 that produces a serializable schedule? If such execution exists, show an example, otherwise prove why it doesn't exist.

No. Since the last operation in T1 conflicts with the first operation of T2, they cannot be swapped. And, since the first operation in T1 conflicts with the last operation of T2, they also cannot be swapped. Thus, since we cannot swap ourselves into a concurrent state from a serial state, we cannot swap ourselves into a serial state from a concurrent state, so there is no serializable schedule.

4. Add locking instructions to the above transactions so that they are compatible with 2-Phase Locking.

```
T1:
RLock(B)
WLock(A)
read(B);
read(A);
if(A > 0) then A := -B;
write(A);
Unlock(A)
Unlock(B)

T2:
RLock(A)
WLock(B)
```

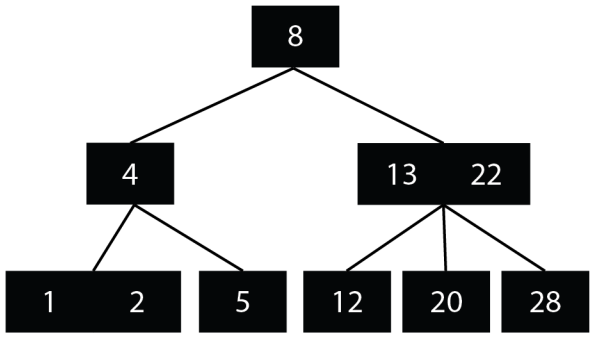

Figure 2: Q4

```
read(A);
read(B);
if(B > 0) then B := -A;
write(B);
Unlock(B)
Unlock(A)
```

5. Can your new transaction code result in a deadlock? Explain your answer.

If we execute these transactions in a serial schedule, no, unless there's a system failure and something fails to unlock. Since we always unlock something after locking, and we know that we have to run these transactions as a serial schedule we should be fine.

4 Question 4

Given the following BTree (max degree 3), explain the steps that a database system that uses lock crabbing would go through to do the following operations:

1. Read "28"
2. Write "23"

Assume that all operations are done using a single pointer `curr`. You should include locking, unlocking, and moving in your steps. An example of a response would be: `Lock {8}. Unlock {8}. Lock {13, 22}` and so on. Assume that the system will never abort.

3. Now, suppose we want to write "6" to the B-tree. Explain the lock steps in this situation. How many locks in total will we have to acquire using crabbing?
4. Write "6" using optimistic locking. How many locks did we acquire?
5. In our "optimistic" scheme, we don't have locks on ancestor nodes. Suppose we go to read "6", but we concurrently delete "4" from the tree. How can we handle this? When should we use crabbing and optimistic locking?

4.1 Solution

1. Read “28”

Lock {8}. Lock {13, 22} Unlock {8}. Lock {28}. Unlock {13, 22}. Read. Done.

2. Write “23”

Lock {8}. Lock {13, 22}. Lock {28}. Unlock {8} (we wait because we technically could have split up until now). Unlock {13, 22}. Write. Done.

3. Write “6” using crabbing

Lock {8}. Lock {4}. Unlock {8}. Lock {6}. Unlock {8}. Write. Done.

4. Write “6” using optimistic locking

Visit {8}. Visit {4}. Lock {6}. Write. Done. Only one lock is acquired.

5. We can “validate” our optimistic locking by traversing the tree one more time. Crabbing is preferred if we are concurrently modifying the tree, but optimistic locking is preferred when the data structure is not concurrently modified.

Feedback

As this is a new course, we appreciate any feedback you have for us! If you enjoyed this assignment, hated this assignment, or have other thoughts to share, please do so here!