Problem Set 4: Transactions and Concurrency Control

COMP_SCI 399

Due: Monday, November 20, 2023

Name: Zhuoyuan Li

1 Concurrency Control with Locking

1. (4 points)

   (a) (1 point) Consider a database with objects A, B, and C and assume that there are two transactions T_1 and T_2 with the following I/O requests:

   - $T_1 : R(A),\ R(B),\ W(A),\ W(B),\ Commit$
   - $T_2 : R(A),\ R(B),\ W(B), R(A),\ R(B),\ W(A)\ R(C)\ W(C),\ Commit$

   In other words, Transaction T_1 reads objects A and B, writes A then, and commits. Transaction T_2 reads objects A and B, writes object B. It then reads objects A and B again, writes A. After that, it reads object C, writes it, and commits. Give an example schedule for the transactions T_1 and T_2 to demonstrate each situation below:

   1. A write-read conflict that causes one transaction to attempt a dirty read.
   2. An unrepeatable read. This read-write conflict causes one of transaction to request the same object twice and get different results.
   3. A write-write conflict that causes a lost update.

   In each case, your schedule may contain additional conflicts, but should contain at least one conflict of the type indicated. (In particular you may give a single schedule, which illustrates all three conflicts!) In each case, indicate the conflict of the type you are illustrating and the actions that cause it.

1. $R_1(A), R_1(B), W_1(A)\ W_1(B), R_2(A), R_2(B), W_2(B), R_2(A), R_2(B), W_2(A), R_2(C), W_2(C), C_1, C_2$

Here, T_2 reads A that hasn't commited by T_1, which causes a dirty read

2. $R_1(A), R_1(B), R_2(A), R_2(B), W_1(A), W_1(B), W_2(B), R_2(A), R_2(B), W_2(A), R_2(C), W_2(C), C_1, C_2$

Here, T_2 reads A in the first place, but the T_1 writes A without Commit so the second time T_2 reads A, it will have different results, which causes an unrepeatable read.

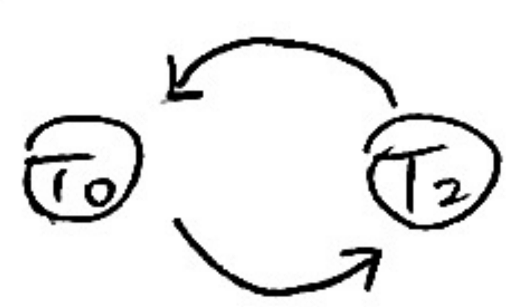
3. $R_1(A), R_1(B), R_2(A), R_2(B), W_1(A), W_2(B), R_2(A), R_2(B), W_2(A), W_1(B), R_2(C), W_2(C), C_2, C_1$

Here, because T_1 writes A first but hasn't commit, then T_2 also writes A, T_2 commits the write first then T_1 commits the write, then A written by T_2 has lost in this case, which causes a lost update.

(b) (1 point) Consider the following three transactions and schedule (time goes from top to bottom). Is this schedule conflict-serializable? Explain why or why not.

Transaction T_0	Transaction T_1	Transaction T_2
$r_0(A)$		
$w_0(A)$		
		$r_2(A)$
		$w_2(A)$
	$r_1(A)$	
$r_0(B)$		
		$r_2(B)$
$w_0(B)$		
		$w_2(B)$
	$r_1(B)$	
	c_1	
c_0		
		c_2

This schedule is not conflict-serializable, because this schedule results in a cyclic graph:



(c) (1 point) Demonstrate how 2PL can ensure a conflict-serializable schedule for the same transactions above. Use the notation $L_i(A)$ to indicate that transaction i acquires the lock on element A and $U_i(A)$ to indicate that transaction i releases its lock on A.

Transaction 0	Transaction 1	Transaction 2
$L_0(A)$		
$r_0(A)$		
$W_0(A)$		
$U_0(A)$		
		$L_2(A)$
		$r_2(A)$
		$W_2(A)$
		$U_2(A)$
	$L_1(A)$	
	$r_1(A)$	
$L_0(B)$		
$r_0(B)$		
		read B has been blocked
$W_0(B)$		
$U_0(B)$		
		$L_2(B)$
		$r_2(B)$
		$W_2(B)$
		$U_2(B)$
	$L_1(B)$	
	$r_1(B)$	
	$U_1(B)$	
	$U_1(A)$	
	Commit1	
Commit 0		
		Commit2

(d) (1 point) If 2PL ensures conflict-serializability, why do we need *rigorous* 2PL?

Different than 2PL, rigorous 2PL holds all locks to the end, and releases locks only after the transaction commits. Rigorous 2PL ensures cascadeless-ness, and that commit order equals serializable order. Therefore, rigorous ensures cascade-free / recoverable schedules,

2 Optimistic Concurrency Control

2. (6 points)

(a) (3 points) Consider the following schedule. Explain what happens when transactions *try* to execute as per this schedule and the DBMS uses timestamp-based concurrency control. We use ST to denote the start of a transaction, C for commit, and A for abort.

$ST_1 \rightarrow ST_2 \rightarrow ST_3 \rightarrow ST_4 \rightarrow R_2(X) \rightarrow R_1(X) \rightarrow W_2(X) \rightarrow W_4(X) \rightarrow W_1(X) \rightarrow C_1 \rightarrow W_3(X) \rightarrow A_4 \rightarrow R_2(Y) \rightarrow W_2(Y) \rightarrow R_3(Y) \rightarrow C_2 \rightarrow W_3(Y) \rightarrow C_3$

<u>Answer</u> (Fill in the table below showing what happens as the transactions execute):

T_1	T_2	T_3	T_4	X	Y
1	2	3	4	RT=0 WT=0 C=true	RT=0 WT=0 C=true
	$R_2(X)$			RT=2	
$R_1(X)$				RT=2	
	$W_2(X)$			WT(X)=2 C(X)=False	
			$W_4(X)$	WT(X)=4 C(X)=False	
$W_1(X)$ Abort					
		$W_3(X)$(delay)			
			Abort	WT(X)=2 C(X)=False	
		$W_3(X)$		WT(X)=3 C(X)=False	
	$R_2(Y)$				RT=2
	$W_2(Y)$				WT(Y)=2 C(Y)=False
		$R_3(Y)$(delay)			
	Commit2				C(Y)=True
		$R_3(Y)$			RT=3
		$W_3(Y)$			WT(Y)=3 C(Y)=False
		Commit3			C(Y)=True

(b) (3 points) Consider the following schedule. Explain what happens when transactions try to execute as per this schedule and the DBMS uses **multiversion** concurrency control:

$ST_1 \rightarrow ST_2 \rightarrow ST_3 \rightarrow ST_4 \rightarrow R_1(X) \rightarrow R_3(X) \rightarrow W_3(X) \rightarrow R_2(X) \rightarrow R_4(X) \rightarrow W_2(X) \rightarrow W_4(X)$

Answer

(Fill in the table below showing what happens as the transactions execute):

T_1	T_2	T_3	T_4	X_0 ...	X_2	X_3	X_4
1	2	3	4				
$R_1(X)$				RT=1			
...							
		$R_3(X)$		RT=3			
		$W_3(X)$				Create	
	$R_2(X)$			RT=3			
			$R_4(X)$			RT=4	
	$W_2(X)$						
	Abort						
			$W_4(X)$				Create

$W_2(X)$ should be aborted because $R_2(X)$ has timestamp 3. Therefore, if we don't want $W_2(X)$ to be aborted, we need to make it appear earlier than $R_2(X)$.