Handin 1 Advanced Concurrency and Recovery

Jens Kristian R. Nielsen & Thomas D. Vinther

31. marts 2019

1 Abstract

In this handin we look at different advanced concurrency controls in concrete examples. First a mulit-version and a single version timestamp scheduler, were we concluded that in these examples perhaps the difference was negligible. Then looking at a validation based scheduler on different sequences, where we noticed that the first transaction that validates, will always do so, and from that point onwards all other transactions validating will have to compare to that first validated Transaction. Then moving on to showing and discussing strict 2PL on a very simple example, showing that for a single transaction, there are a lot of different ways to insert locks. We were unable to make any assertion of the value of using the strict approach in this example. Finally we looked at how we could insert locks, using both regular 2 PL and multiversion 2 PL, depending on how to interpret the question: Is a exclusive lock a fully exclusive lock or not. In either case we found that the transactions would become deadlocked, but that in the multiversion 2 PL a lot more initial transactions were successful, but the certifying locks still failed, causing a deadlock in the multiversion as well.

2 Solution

The topic of this hand-in is advanced concurrency control. The exercises study how different concurrency control strategies work in concrete scenarios.

2.1 A.

What happens for the following three sequences of events if a multi-version timestamp scheduler is used? What happens if the scheduler does not maintain multiple versions (but rather follows the single-version timestamp approach)?
1. st1, st2, r1(A), r2(B), w2(A), w1(B).
2. st1, st2, st3, r1(A), r2(B), w1(C), r3(B), r3(C), w2(B), w3(A).
3. st1, st2, st3, r1(A), r3(B), w1(C), r2(B), r2(C), w3(B), w2(A).
where wi(X) means that transaction i writes item X, and ri(X) means that transaction i reads item X, sti means that transaction i starts.

A.1

Consider the following multiversion timestamp schedule of the first sequence

$T1$	$T2$	$T1'$	A	B	$A1$	$B1$
100	200	300	$rt = wt = 0$	$rt = wt = 0$		
$R1A$			$rt = 100$			
	$R2B$			$rt = 200$		
	$W2A$				$rt = wt = 200$	
$W1B$						
		$R1'A$	$rt = 300$			
		$W1'B$				$rt = wt = 300$

Here the W1B attempt is unable to proceed because $100 < 200$, so T1 is aborted and put back into the queue with a new timestamp sufficient for the write operation to complete. Consider next the single version timestamp procedure

$T1$	$T2$	$T1'$	A	B
100	200	300	$rt = wt = 0$	$rt = wt = 0$
$R1A$			$rt = 100$	
	$R2B$			$rt = 200$
	$W2A$		$wt = 200$	
$W1B$				
		$R1'A$	$rt = 300$	
		$W1'B$		$wt = 300$

Once again W1B cannot proceed so it is aborted and put to the back of the line. In both cases no cascading takes place because W1B is the first write operation that transaction 1 attempts.

A.2

Consider the following multiversion timestamp schedule of the second sequence, where we write $rw = 100$ in place of $rt = wt = 100$ and 0 in place of $rt = wt = 0$ in the interest of maintaining the diagram within the bounds of the page width

$T1$	$T2$	$T3$	$T2'$	A	B	C	$C1$	$A1$	$B1$
100	200	300	400	0	0	0			
$R1A$				$rt = 100$					
	$R2B$				$rt = 200$				
$W1C$							$rw = 100$		
		$R3B$			$rt = 300$				
		$R3C$					$rt = 300$		
	$W2B$								
		$W3A$						$rw = 300$	
			$R2'B$		$rt = 400$				
			$W2'B$						$rw = 400$

The first occurrence of W2B fails because $200 < 300$, i.e. transaction 3 read B before transaction 2 wrote to it, so transaction 2 is rolled back and given a new timestamp. No cascading occurs because this was the first time transaction 2 attempted the write operation. Consider next the single version timestamp schedule.

$T1$	$T2$	$T3$	$T2'$	A	B	C
100	200	300	400	$rt = wt = 0$	$rt = wt = 0$	$rt = wt = 0$
$R1A$				$rt = 100$		
	$R1B$				$rt = 200$	
$R1C$						$wt = 100$
		$R3B$			$rt = 300$	
		$R3C$				$rt = 300$
	$W2B$					
		$W3A$		$wt = 300$		
			$R2'B$		$rt = 400$	
			$W2'B$		$wt = 400$	

We get the same problem with transaction 2 that is pushed to the back of the line.

A.3

Consider the following multiversion timestamp schedule of the third sequence

$T1$	$T2$	$T3$	A	B	C	$C1$	$B1$	$A1$
100	200	300	0	0	0			
$R1A$			$rt = 100$					
		$R3B$		$rt = 300$				
$W1C$					$rt = wt = 100$			
	$R2B$			$rt = 300$				
	$R2C$					$rt = 200$		
		$W3B$					$rt = wt = 300$	
	$W2A$							$rt = wt = 200$

Here everything is smooth. We follow the algorithm granted to us in the first lecture, and for instance the step where rt goes from 300 to 300 it is because transaction 2 reads B, but transaction 3 already read B, so we update the rt timestamp to be the maximumm of 200 and 300, which of course is 300. When a write is performed we create a new version of the table being written to, for instance in the W1C step we find the version of C with the highest wt attribute lower than the timestamp of transaction 1, at this stage only 1 version of C exists, namely C, with wt=0<100 so we create the copy C1 with timestamps 100 because it was transaction 1 that initiated the write procedure.

$T1$	$T2$	$T3$	A	B	C
100	200	300	$rt = wt = 0$	$rt = wt = 0$	$rt = wt = 0$
$R1A$			$rt = 100$		
		$R3B$			$rt = 300$
$W1C$			$wt = 100$		
	$R2B$			$rt = 200$	
	$R2C$				$rt = 200$
		$W3B$		$wt = 300$	
	$W2A$		$wt = 200$		

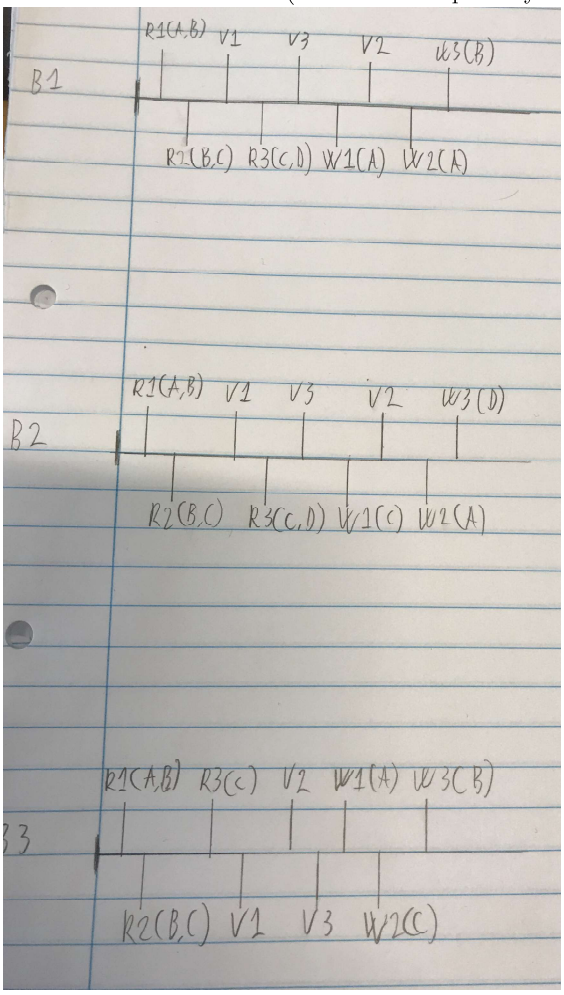
Similarly to the multiversion version this schedule runs through smoothly.

2.2 B.

. What happens for the following sequences of events if a validation-based scheduler is used?
1. R1(A,B), R2(B,C), V1, R3(C,D), V3, W1(A), V2, W2(A), W3(B).
2. R1(A,B), R2(B,C), V1, R3(C,D), V3, W1(C), V2, W2(A), W3(D).
3. R1(A,B), R2(B,C), R3(C), V1, V2, V3, W1(A), W2(C), W3(B).

where Ri(X) means that transaction i starts and its read set is X, Vi means that Ti attempts to validate, and Wi(X) means that Ti finishes and its write set was X.

Observe the timelines (dont mess it up Barry Allen):



We assume, that the transactions are in a reading phase from the first RX(Y) until the first VX, where the transactions goes into a validation phase until the first WX(Y), where it is in a writing phase, until the last write is performed. (In this exercise there is only one write pr. transaction, so the transactions are only in the writing phase at one point) After the last write

the transaction is in a committed phase, presuming that everything was accepted.

We start with the first sequence, using the rules of reading when a validation-based scheduler is used. So we simply read R1(A,B), R2(B,C), and see that the read_sets are easily read. We then arrive at our first V1, a.k.a. the validation phase of transaction 1. We use the rules from the lectures to validate transaction 1, before writing as to ensure serializability. The rules are as follows:
The validation phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:
1. Tj completes its write phase before Ti starts its read phase
2. Ti starts its write phase after Tj completes its write phase, and the read_set of Ti has no items in common with the write_set of Tj
3. Both the read_set and write_set of Ti have no items in common with the write_set of Tj, and Tj completes its read phase before Ti completes is read phase

Because transaction 1, is the only transaction in a validation phase/committed phase, all the conditions hold, and the validation phase for transaction 1 is therefore successful. We then let transaction 3, read (R3(C,D)) updating the read_set accordingly, and we then get to the validate 3. We look at the above rules, and see that only T1 and T3 is in the validation phases or is committed and notice that rule 2 holds, as T3 starts it's write phase after T1 completes it's write phase and that read_set of T3 = {C,D} and the write_set of T1 = {A} has no item in common.
Moving further along the schedule, we simply let T1 write as it has been validated. But letting T2 validate, fails as no of the conditions are met. The read_set of T2 = {B,C} has items in common with the write_set of T3 = {C,D}, and T3 definitely does not complete it's write phase before T2 starts it's read phase. And so T2 is aborted. Lastly T3 writes B.
In total leaving T1 to read A and B and write to A, T3 to read C and D and write to B.

For B2, we more or less repeat the above procedure. As we notice that the two schedules overlap, right up until, T3 tries to validate. This become a problem, as the write_set of T1 ={C} and the read_set of T3 = { C,D} there is an overlap that according to rules 2 and 3, is not allowed. Rule 1's condition is obviously not met, as T3 reads before T1 writes. And therefore T3 is aborted. We then let T1 write on C as it has been validated and continue onward to V2, validating T2. The same argument as before holds, as T2 has items in common with the write_set of T1 ={C} and so T2 is also aborted.
Leaving totally only T1 to read A and B, and write to C.

For B3, we let all 3 transactions read and update the respective read_sets, we then validate T1 without problem as once again there is no other Transaction that is validating or committed, and so T1 is validated. Then we have to validate T2, using the rules, and quickly notice that rule 1 cannot hold, as T1 does not complete (or even start) it's write phase before T2 starts its read phase. But the condition in rule 2 is met, as the write phase of T2, starts after T1 completes its write phase and the read_set of T2 = {B,C} has no item in common with the write_set of T1 = {A}, and so T2 is validated. We then look at V3, validating T3. When comparing to T2 that is committed at this point, rule 1 is dismissed. We notice that the read_set of T3 ={C} and that the write_set of T2 = {C} and so the sets have a common item, and does not live up to the conditions. (as the validating transaction have to live up to every rule for every transaction that is either committed or is in its validation phase, we do not look at T1, although rule 3 is satisfied.)

Again leaving T1 to read A and B, T2 to read B and C, T1 to write to A and lastly T2 to write to C.

2.3 C.

What are all the ways to insert locks (of a single type only) into the sequence of actions r1(A), r1(B), w1(A), w1(B) so that the transaction is
1. Strict two-phase locked
2. Two-phase locked (not strict).
where the notation is as in exercise A.

1

The core idea behind multiversion 2PL is to allow other transactions T' to read an item X while a single transaction T holds a write lock on X (parphrasing from the book) But as we only have a single Transaction T1, there really is no need to use the multiversion 2PL as T' simply does not exsist.
So therefore we will just use the "normal"version of 2PL. In strict two-phase locking, write locks are not released until at commit time, and therefore all our unlocks must be after **w1(B)**. Locking on the other hand is much more free, and we can permute the sequence a lot. All we have to do is make sure that we at least have a read-lock or higher (in other words a write-lock) before we read A or B, and then make sure we have a write-lock before we write on A or B. This means we could upgrade the locks just before we need them, or simply write lock them first, in different sequence. After we have written to B, we can release locks. (Before unlocking we would certify the locks in a multiversion 2PL, and then unlock). This leads to the following cases when only looking at locks of a single type only, i.e. not read AND write locks.

WL1A; WL1B; **r1(A)**; **r1(B)**; **w1(A)**; **w1(B)**; U1A; U1B

WL1A; WL1B; **r1(A)**; **r1(B)**; **w1(A)**; **w1(B)**; U1B; U1A

WL1B; WL1A; **r1(A)**; **r1(B)**; **w1(A)**; **w1(B)**; U1A; U1B

WL1B; WL1A; **r1(A)**; **r1(B)**; **w1(A)**; **w1(B)**; U1B; U1A

WL1A; **r1(A)**; WL1B; **r1(B)**; **w1(A)**; **w1(B)**; U1A; U1B

WL1A; **r1(A)**; WL1B; **r1(B)**; **w1(A)**; **w1(B)**; U1B; U1A

2

As in the previous question, we have the same argument for not using the multiversion 2PL, there is only one transaction. But this time we do not use strict 2PL, this means we can release locks before we have committed everything, this means that we still have disjoint growing and shrinking phases, therefore we can now unlock A, before writing to B, as long as we have already write-locked B, before we unlock A. Again only using one type of lock:

WL1A; WL1B; **r1(A)**; **r1(B)**; **w1(A)**; **w1(B)**; U1A; U1B

WL1A; WL1B; **r1(A)**; **r1(B)**; **w1(A)**; **w1(B)**; U1B; U1A

WL1A; WL1B; **r1(A)**; **r1(B)**; **w1(A)**; U1A; **w1(B)**; U1B

WL1B; WL1A; **r1(A)**; **r1(B)**; **w1(A)**; **w1(B)**; U1A; U1B

WL1B; WL1A;**r1(A)**; **r1(B)**; **w1(A)**; **w1(B)**; U1B; U1A

WL1B; WL1A; **r1(A)**; **r1(B)**; **w1(A)**; U1A; **w1(B)**; U1B

WL1A; **r1(A)**; WL1B; **r1(B)**; **w1(A)**; **w1(B)**; U1A; U1B

WL1A; **r1(A)**; WL1B; **r1(B)**; **w1(A)**; **w1(B)**; U1B; U1A

WL1A; **r1(A)**; WL1B; **r1(B)**; **w1(A)**; U1A; **w1(B)**; U1B

2.4 D.

Assume shared locks are requested immediately before each read, and exclusive locks immediately before each write. Unlocks occur immediately after the final action of the respective transaction. Which actions are denied?
1. r1(A), r2(B), w1(C), r3(D), r4(E), w3(B), w2(C), w4(A), w1(D)
2. r1(A), r2(B), w1(C), w2(D), r3(C), w1(B), w4(D), w2(A)
where the notation is as in A.

1

If we first look at a multiversion two phase locking using certify locks, as if we read the assignment of exclusive lock as in the lectures that, write lock is no longer fully exclusive, but still somewhat exclusive we end up with the following table/scheme of the transactions, where the denied transactions are denoted with a "no"on the right side.

$T1$	$T2$	$T3$	$T4$	A	B	C	D	E
$SL1A$				$SL1$				
$r1A$								
	$SL2B$				$SL2$			
	$r2B$							
$XL1C$						$XL1$		
$w1C$								
		$SL3D$					$SL3$	
		$r3D$						
			$SL4E$					$SL4$
			$r4E$					
		$XL3B$			$XL3$			
		$w3B$						
		$CL3B$			no			
		$CL3D$					$CL3D$	
	$XL2C$					no		
			$XL4A$	$XL4$				
			$w4A$					
			$CL4A$	no				
			$CL4E$					$CL4E$
$XL1D$							$XL1$	
$w1D$								
$CL1D$							no	
$CL1C$						$CL1$		
$CL1A$				no				

The above multi-version two phase locking scheme, ends in a deadlock. When T3 is finished, it cannot certify as T2 has a shared lock on B. But, T2 cannot lock C, as T1 already has an exclusive lock on C, and T1 cannot certify as T3 has a shared lock on D. This is cycle, and so we cannot certify lock any of the transactions completely, and the same goes for transaction 4, as it cannot completely certify lock A before T1 releases the shared lock on A, which cannot happen because of the aforementioned cycle. This could of course be fixed in some form of the implementation, but if the transaction simply waits for the locks to be lifted before certifying, this will not be possible. If however we look at the question as if the exclusive means fully exclusive, we end up with the regular two phase locking, with only the read (shared) lock and the write (exclusive) lock, observe below:

$T1$	$T2$	$T3$	$T4$	A	B	C	D	E
$SL1A$				$SL1$				
$r1A$								
	$SL2B$				$SL2$			
	$r2B$							
$XL1C$						$XL1$		
$w1C$								
		$SL3D$					$SL3$	
		$r3D$						
			$SL4E$					$SL4$
			$r4E$					
		$XL3B$			no			
	$XL2C$					no		
			$XL4A$	no				
$XL1D$							no	

Again we see a cycle between T1, T2 and T3.

2

We apply the 2PL multi version to the problem, like in 2.4.1

$T1$	$T2$	$T3$	$T4$	A	B	C	D
$SL1A$				$SL1$			
$r1A$							
	$SL2B$				$SL2$		
	$r2B$						
$XL1C$						$XL1$	
$w1C$							
	$XL2D$						$XL2$
	$w2D$						
		$SL3C$				$SL3$	
		$r3C$					
$XL1B$					$XL1$		
$w1B$							
$CL1B$					no		
$CL1C$						no	
			$XL4D$				no
	$XL2A$			$XL2$			
	$w2A$						
	$CL2A$			no			
	$CL2D$						$CL2$
	$CL2B$				no		

In the above when a no is in a color, the lock is denied due to the previous lock in the same color in the same column. Note the complexity that T2 tries to get a certified lock on D after the lock attempt on A failed, this may or may not be a good idea in general.
In this solution only transaction 3 is allowed to run, and the other three go in deadlock because T2 wants to lock A, so T3 waits for T1 to be done, but T1 wants to lock B so it is waiting for T1 to release it, deadlock. T4 is waiting for T2 to unlock D, but T2 is stuck in a deadlock so that will never happen.

Next we apply the interpretation that the XL's are fully exclusive, again like 2.4.1

$T1$	$T2$	$T3$	$T4$	A	B	C	D
$SL1A$				$SL1$			
$r1A$							
	$SL2B$				$SL2$		
	$r2B$						
$XL1C$						$XL1$	
$w1C$							
	$XL2D$						$XL2$
	$w2D$						
		$SL3C$				no	
$XL1B$					no		
			$XL4D$				no
	$XL2A$			no			

Here we get a similar deadlock, but T3 is denied access to C and joins T4 in waiting for the deadlock between T1 and T2 to resolve, which of course it does not. As such this interpretation of the problem leads to all transactions denied.

3 Summary

In this handin we worked with advanced concurrency control, in concrete scenarios. We looked at multi-version timestamp scheduling versus single version, that did not maintain multiple versions. Concluding that in these examples, it does not make a significant difference which version we use. It would perhaps be instructive to find an example where it does make a significant difference.
When pondering the validation based scheduler, we found that without explicit phase transitions there is room for interpretation and trouble, we concluded that the read phase runs from the first read to the validation point, that then initiates the validation phase lasting till the transactions first write is scheduled, which in turn initiates the write phase that ends with the last write of the transaction. In the examples covered here the write phase was over quickly since each transaction had only one write operation.
We found that question C was somewhat ambiguous, in asking to: "insert locks (of a single type only)". And we interpreted in the sense that we could only either have a read lock or a write lock, as there was no need for the multiversion because we only had one transaction. The example was peculiar in the sense that locking is interesting when trying to have concurrency of many transactions, here we only had a single transaction which makes the situation very artificial to a dangerous point. The danger intensified in question D as we have been introduced to two ways to interpret an exclusive write lock, we proceeded to apply both, and found a nice example of the multi version procedure producing a better result than the single version procedure.