Middle East Technical University Department of Computer Engineering

CENG 352 - Database Management Systems
Written Assignment 3

Yavuz Selim YESILYURT
2259166

31.05.2020

1 Q1

a) We are asked if conflict serializability is guaranteed or not. To be able to determine this we have a number of options such as creating a precedence graph and checking if there is a cycle in it or not. But in the given schedules for transactions $T1$ and $T2$ we see that a locking/unlocking mechanism has been applied to schedule their working in parallel. So in that case we will look on to the types of schedulers (actually only to the pessimistic scheduler since we only have exclusive locks for the units here) and try to evaluate their conflict serializability by the locking style that is being used. Now we see that, in the schedule of these transactions, all lock requests precede all their corresponding unlock requests, which means this schedule obey $2PL$ rule and which also means conflict serializability is guaranteed here.

b) We can easily say that deadlock it possible here, because when you look to the schedules, you will see that first transaction grabs resource A's lock and requests resource B's lock without unlocking A's lock after 2 actions; second transaction on the contrary grabs resource B's lock and requests resource A's lock without unlocking B's lock after 1 action. This will potentially lead to deadlock situation for these to transactions. If we were to apply $wait - die$ deadlock prevention scheme and assume $T1$ starts first, $T2$ would be rolled back since in the $wait - die$ deadlock prevention scheme the older one transactions holds the lock, the greater its priority is, which will lead to $T1$'s having a greater priority than $T2$.

c) In part a, we deduced that this schedules obey the $2PL$ rule, but this not necessarily means they avoid cascading aborts. For this schedule to avoid cascading aborts it also needs to be *strict*. We see that, all locks held by both transactions are released when the they are completed, namely they release all the held locks at the time of their COMMITs. Therefore we can say that this schedule is not only $2PL$ but also it is $Strict2PL$. So in that case we can say that this schedule avoids cascading rollbacks.

d) Now in the newly modified schedule, we see that lock and unlock actions' locations have been changed. Which may possibly lead to this schedule's no more obeying $2PL$ rule. But we can see that, still, in the schedule of these transactions, all lock requests precede all their corresponding unlock requests, which means this schedule still obey the $2PL$ rule and which also means conflict serializability is guaranteed here.

e) In here we can't say deadlock is possible anymore since in this time $T1$ grabs the lock of resource A and then after some actions again tries to grab the lock of resource B and actually will get the lock, because $T2$ now does not grab the lock of resource B at first, instead it requests lock of resource A which it will be rejected for, since $T1$ would be having that lock already. So everything is going to work smoothly, first $T1$ will finish and then $T2$ will finish.

f) In this newly modified schedule, we need to check that if it still obeys $2PL$ "$Strictly$". For this we check the unlocks times of the the transactions. We see that they do not release their locks at the time of their COMMITs, namely they release some of their locks (resource A's) before they finish their all actions which spoils the $Strict2PL$ rule. So in that case we can't say that this schedule avoids cascading rollbacks. If we assume $T2$ starts first, such a scenario would result in a cascading rollback of the actions since $T1$ reads an uncommitted data at the time of *R1(A)* and if $T2$ aborts $T1$ needs to abort, too:

X2(A) X2(B) R2(B) R2(A) W2(A) U2(A) X1(A) **R1(A)** W1(A) W2(B) U2(B) X1(B) U1(A) R1(B) W1(B) U1(B)

and if we assume $T1$ starts first, such a scenario would result in a cascading rollback of the actions since $T2$ reads an uncommitted data at the time of *R2(B)* and if $T1$ aborts $T2$ needs to abort, too:

X1(A) R1(A) W1(A) X1(B) U1(A) X2(A) R1(B) W1(B) U1(B) X2(B) **R2(B)** R2(A) W2(A) U2(A) W2(B) U2(B)

2 Q2

a)

- We have $TS(T1) = 1$, $TS(T2) = 2$, $TS(T3) = 3$.

Operation	A			B			C		
	RTS	WTS	C	RTS	WTS	C	RTS	WTS	C
r1(A)	1	0	True	0	0	True	0	0	True
r2(B)	1	0	True	2	0	True	0	0	True
r3(A)	3	0	True	2	0	True	0	0	True
w1(A) (Reject and Rollback T1)	3	0	True	2	0	True	0	0	True
r2(C)	3	0	True	2	0	True	2	0	True
w3(B)	3	0	True	2	3	False	2	0	True
w2(C)	3	0	True	2	3	False	2	2	False
c1 (Gets Rejected)	3	0	True	2	3	False	2	2	False
r2(A)	3	0	True	2	3	False	2	2	False
w3(C)	3	0	True	2	3	False	2	3	False
c3	3	0	True	2	3	True	2	3	True
w2(B) (Ignore Write due to Thomas Write Rule)	3	0	True	2	3	True	2	3	True
c2	3	0	True	2	3	True	2	3	True

$w1(A)$ gets rejected since RTS of A has a value of 3 which is greater than $TS(T1)$ which is 1. Therefore this action results in Rollback of $T1$. Then $c1$ operation gets rejected because transaction $T1$ has already been aborted. The other operations are accepted and executed (except $w2(B)$, it results in Ignore Write because it triggers the Thomas Write Rule due to commit bit's being True and WTS's being greater than $TS(T2)$) without any problems.

- We have $TS(T1) = 2$, $TS(T2) = 3$, $TS(T3) = 1$.

Operation	A			B			C		
	RTS	WTS	C	RTS	WTS	C	RTS	WTS	C
r1(A)	2	0	True	0	0	True	0	0	True
r2(B)	2	0	True	3	0	True	0	0	True
r3(A)	2	0	True	3	0	True	0	0	True
w1(A)	2	2	False	3	0	True	0	0	True
r2(C)	2	2	False	3	0	True	3	0	True
w3(B)(Reject and Rollback T3)	2	2	False	3	0	True	3	0	True
w2(C)	2	2	False	3	0	True	3	3	False
c1	2	2	True	3	0	True	3	3	False
r2(A)	3	2	True	3	0	True	3	3	False
w3(C) (Gets Rejected)	3	2	True	3	0	True	3	3	False
c3 (Gets Rejected)	3	2	True	3	0	True	3	3	False
w2(B)	3	2	True	3	3	False	3	3	False
c2	3	2	True	3	3	True	3	3	True

$w3(B)$ gets rejected since RTS of B has a value of 3 which is greater than $TS(T3)$ which is 1. Therefore this action results in Rollback of $T3$. Then $w3(C)$ and $c(3)$ operations get rejected because transaction $T3$ has already been aborted. The other operations are accepted and executed without any problems.

b) *Commit* bit is crucial for timestamp-based scheduler to have (or ensure) **Recoverable** Schedules. We use it to check if the transaction that last wrote to a resource X has committed or not. We do this by adding an additional if check to both writing and reading procedures of transactions in TS-based Scheduler. Transaction checks if the commit bit of resource to write/read is $false$ or not and if it is, transaction gets delayed until commit bit gets *true*. If do not use the commit bit in TS-based scheduler, we would not have schedules that are both recoverable and avoid cascading aborts.

3 Q3

a) So just before the system crash we have the following content in our:

- Write-Ahead-Log (WAL) Table:

LSN	transId	prevLSN	type	pageID	log entry	undoNextLSN
0	T1	-	Update	P1	Write(A)	
1	T2	-	Update	P1	Write(B)	
2	T2	1	Update	P2	Write(C)	
3	T2	2	Abort	-	-	-
4	-	-	-	-	Start Checkpoint & End Checkpoint. Active Transaction and Dirty Page Tables get updated with new values after the checkpoint.	-
5	T2	-	CLR	-	Undo T2 LSN 2	1
6	T2	-	CLR	-	Undo T2 LSN 10	-
7	T2	6	End	-	-	-
8	T3	-	Update	P2	Write(D)	
9	T1	0	Commit	-	-	-
10	T1	9	End	-	-	-
11	T4	-	Update	P1	Write(A)	
12	T3	8	Update	P1	Write(B)	-
13	T4	11	Commit	-	-	-

- Active Transaction ($XACT$) Table:

transId	lastLSN	Status
T3	12	Running
T4	13	Committed

- Dirty Page (DPT) Table:

pageID	recLSN
P1	1
P2	2

In the pages of our memory we have:

- P1:
  - Data item A has the latest value after $T4$'s update action ($LSN = 11$),
  - Data item B has the latest value after $T3$'s update action ($LSN = 12$),
  - $pageLSN = 12$

- P2:
  - Data item C's value hasn't been changed so its value remains as the same,
  - Data item D has the latest value after $T3$'s update action ($LSN = 8$),
  - $pageLSN = 8$

b) Since we know that $T4$ has committed before the system crash, we do not have any problems with having the accumulated log pages in the disk. Therefore our Active Transaction Table ($XACT$) and Dirty Page Table (DPT) remains the same as in part a.

c) To determine at which LSN, Aries Recovery Manager's Redo Phase starts, we check the smallest $recLSN$ in the DPT. We see that it is 1. Therefore starting from the log entry with $LSN = 1$ we will only redo the actions specified by the *log entry* in WAL that is either:

- Has its page listed in DPT,
- Has its LSN smaller than the $recLSN$ (which is determined as 1 in above),
- Has its LSN smaller than or equal to $pageLSN$ that is read from the disk.

Therefore according to those we are going to redo the following changes with LSN 5, 6, 8, 9, 11 and 12. In the pages of our memory at the end of the Redo Phase we have:

- P1:
  - Data item A has the latest value after $T4$'s update action ($LSN = 11$),
  - Data item B has the latest value after $T3$'s update action ($LSN = 12$),
  - $pageLSN = 12$
- P2:
  - Data item C's value hasn't been changed so its value remains as the same,
  - Data item D has the latest value after $T3$'s update action ($LSN = 8$),
  - $pageLSN = 8$