ACS Theory Assignment 2

Kai Arne S. Myklebust, Silvan Adrian

Handed in: December 14, 2018

Contents

1 Question 1: Concurrency Control Concepts **2**
- 1.1 Task 1 . . . 2
- 1.2 Task 2 . . . 3
- 1.3 Task 3 . . . 4
- 1.4 Task 4 . . . 4

2 Question 2: Recovery Concepts **4**
- 2.1 Task 1 . . . 4
- 2.2 Task 2 . . . 5
- 2.3 Task 3 . . . 5

3 Question 3: More Concurrency Control **5**
- 3.1 Task 1 . . . 5
- 3.2 Task 2 . . . 6
- 3.3 Task 3 . . . 7
- 3.4 Task 4 . . . 7

4 Question 4: ARIES **8**
- 4.1 Task 1 . . . 8
- 4.2 Task 2 . . . 8
- 4.3 Task 3 . . . 8
- 4.4 Task 4 . . . 8
- 4.5 Task 5 . . . 9
- 4.6 Task 6 . . . 9

5 Question 5: More ARIES **9**
5.1 Task 1 9
5.2 Task 2 10
5.3 Task 3 10
5.4 Task 4 10
5.4.1 a) 11
5.4.2 b) 11

1 Question 1: Concurrency Control Concepts

1.1 Task 1

Following we have 2 schedules which are view equivalent. The first one is therefore view serializable.

T1	T2	T3
R(A)		
	W(A)	
	commit	
W(A)		
Commit		
		W(A)
		Commit

T1	T2	T3
R(A)		
W(A)		
Commit		
	W(A)	
	commit	
		W(A)
		Commit

And the precedence graph of the first schedule shows that there is a cycle, which means it's not conflict serializable.

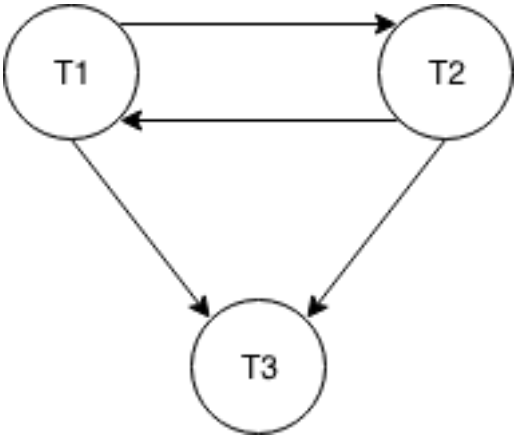

Figure 1: Precedence Graph for shedule

1.2 Task 2

T2 conflicts with T1 and T1 conflict with T3, so there is no cycle in the precedence graph. In T1 the transaction acquires locks for A, then T2 tries to get a shared lock on A which is then not possible since T1 already has an exclusive lock on A.

T1	T2	T3
W(A)		
	R(A)	
	Commit	
		R(B)
		Commit
W(B)		
Commit		

1.3 Task 3

This satisfies 2PL, because it can release exclusive lock on B after it has written B, so T2 can acquire the exclusive lock on B. But it can't satisfy S2PL since T1 will keep the locks until it is finished with all it's operations, so that T2 can't acquire a exclusive lock on B.

T1	T2
W(A)	
W(B)	
	W(B)
W(A)	
Commit	
	Commit

1.4 Task 4

This satisfy S2PL, because the exclusive lock on B in T2 get's released directly after its write so that T1 can obtain the exclusive lock on B. But it can't satisfy CS2PL because T1 acquires all locks at the beginning, which means T2 can't obtain an exclusive lock on B.

T1	T2
W(A)	
	W(B)
	Commit
W(B)	
W(A)	
Commit	

2 Question 2: Recovery Concepts

2.1 Task 1

We do not need a redo scheme, because all changes of committed transactions are guaranteed to have been written to disk at commit time. We also don't need the undo scheme, because the changes of those aborted transactions have not been written to disk.

2.2 Task 2

The difference of stable storage and non-volatile storage is that stable storage is implemented by maintaining multiple copies of information on non-volatile storage. Access time on non-volatile storage is therefore much faster than on stable storage.

Non-volatile storage is guaranteed to survive crashes but can be subject to media failures, while stable storage is guaranteed to survive both.

2.3 Task 3

The log tail needs to be forced to stable storage in following 2 situations:

- When a transaction is committed
- After modifying of pages

For the first situation, when a transaction is committed it has be ensured that the record of changes have been written in the log before it's written to disk. This way we know what changes have been done even after a crash.

For the second situation, when a page gets modified we have to know that there has been a change without committing so that we are able to undo the not committed changes.

For both situations the durability is sufficient because we are able to undo modifications and ensure that all committed transactions survive a crash.

3 Question 3: More Concurrency Control

3.1 Task 1

There are no locks waiting on each other. And after commit all exclusive locks get released and we have in transaction Ta and Tc only a shared lock on Z.

Ta	Tb	Tc
S(Z)		
R(Z)		
	S(Y)	
	R(Y)	
	X(Y)	
	W(Y)	
		S(Z)
		R(Z)
	C	
	Release(Y)	
		X(Y)
		W(Y)
		C
		Release(Z,Y)
X(Y)		
W(Y)		
C		
Release(Z,Y)		

3.2 Task 2

Locks are not waiting on each other and they get released on time before an other transaction wants to get a lock on the same object.

Ta	Tb	Tc
S(Z)		
R(Z)		
	S(Y)	
	R(Y)	
	X(Y)	
	W(Y)	
	Release(Y)	
		S(Z)
		R(Z)
	C	
		X(Y)
		W(Y)
		Release(Y,Z)
		C
X(Y)		
W(Y)		
Release(Y,Z)		
C		

3.3 Task 3

No not possible, since we acquire an exclusive lock on `Y` in `Ta` and afterwords try to get an other exclusive lock on `Y` in `Tb` which is not allowed in `CS2PL`.

Ta	Tb	Tc
S(X)		
X(Y)		
R(X)		
	X(Z)	
	X(Y) - Not possible, abort	

3.4 Task 4

We decided that `Tb` will be the first transaction, `Tc` the second and `Ta` the last.

```
1 Tb: RS(Tb) = {Z,Y}, WS(Tb) = {Z,Y},
2 Tb completes before Tc starts.
3 Tc: RS(Tc) = {Z}, WS(Tc) = {Z},
4 Tc completes before Ta starts with its write phase.
5 Ta: RS(Ta) = {X}, WS(Ta) = {Y}.
```

- `Tb` completes before `Tc` starts, so there weren't any conflicts with other transactions while validation.
- `Tc` completes before `Ta` starts with its write phase and the intersection of the write set from `Tc` and read set from `Ta` is empty.
- So according to the validation phase in `KROCC` the three transactions have no conflicts so this schedule could be generated by `KROCC`.

4 Question 4: ARIES

4.1 Task 1

Following 2 tables are the transaction table and the dirty page table after the analysis phase of ARIES.

Transaction table:

TransID	Status	LastLSN
T1	Active	4
T2	Active	9

Dirty Page table:

PageId	RecLSN
P2	3
P1	4
P5	5
P3	6

4.2 Task 2

The winner set of transactions is: {T3} because it was able to commit and finish. The loser set of transactions is: {T1,T2} because they weren't able to finish and were still active while the crash occured.

4.3 Task 3

The redo phase start with log record with the smallest `recLSN` in the dirty page table, in this case it would be `LSN 3`. The undo phase goes through all loser transactions and ends at the oldest `LSN`, in this case it would be `LSN 3`.

4.4 Task 4

To get the set of log records to be redone we start by taking the smallest `LSN` from the dirty page table. And add the modifications with higher `LSNs` then the smallest `LSN` from the dirty page table to the set as well.

So we get the following set: {3,4,5,6,8,9}

4.5 Task 5

To get the set of records undone, we have to get all the updates from the loser transactions starting from the newest one. We would get the following set: {9,8,5,4,3}

4.6 Task 6

The following table is only the appended log entries after the crash.

LSN	LAST_LSN	TRAN_ID	TYPE	undoNextLSN	PAGE_ID
11	9	T2	ABORT		
12	4	T1	ABORT		
13	11	T2	CLR: Undo LSN 9	8	
14	13	T2	CLR: Undo LSN 8	5	
15	14	T2	CLR: Undo LSN 5	-	
16	15	T2	end		
17	12	T1	CLR: Undo LSN 4	3	
18	17	T1	CLR: Undo LSN 3	-	
19	18	T1	end		

5 Question 5: More ARIES

5.1 Task 1

- A has the value 5, since we are currently undoing 7 and the previous `LSN` in T2 is 5
- B has the value 4, since we are currently undoing 9 and the previous `LSN` in T1 is 4
- C has the value 3, since we are currently undoing 4 and the previous `LSN` in T1 is 3
- D has the value NULL, since we are currently undoing 5 and the previous `LSN` in T2 is NULL
- E has the value NULL, since we are currently undoing 3 and the previous `LSN` in T1 is NULL

5.2 Task 2

Since we don't have a new checkpoint yet, we have to start over from `LSN 3` for the dirty page table. T1 is undone and ended, and T4 is committed and ended, so they are not in the transaction table. So that we ended up on the following 2 tables:

Transaction table:

TransID	Status	LastLSN
T2	Active	17
T3	Active	6

Dirty Page table:

PageId	RecLSN
P3	3
P1	4
P2	7
P4	9

5.3 Task 3

T2 wasn't ended and T3 hadn't committed so they needed to be undone. T2 was already undone before the crash just not ended, that's we only had to add 1 additional log entry for T2. T3 on the other hand had to be aborted since the undone process hasn't started yet.

LSN	LAST_LSN	TRAN_ID	TYPE	undoNextLSN	PAGE_ID
20	6	T3	ABORT		
21	17	T2	end		
22	20	T3	CLR: undo LSN 6	-	
23	22	T3	end		

5.4 Task 4

We understood that the log is still in stable storage and the database is in volatile storage and only gets saved to stable storage on particular log records.

5.4.1 a)

We would say commits definitely need to be written to stable storage. If there occurs a crash all modifications are logged and it would be easy to redo the modifications so that only finished transactions should get written to stable storage.

5.4.2 b)

We think that all information is necessary, so we would save all update/modifications, commits etc. to stable storage for later undo or redo operations. Without any of informations we wouldn't be able to do a proper recovery process, at least we can't think of any information which wouldn't be needed for recovery.