ACS-A3

Rikke Tophøj Heinemann (DTZ190)
Marie Elkjær Rødsgaard (DCK495)
Rasmus Pallisgaard (DWP992)

December 21, 2023

1 Question 1: Concurrency Control Concepts

1.1 Show a schedule that is conflict-serializable, but could not have been generated by a S2PL scheduler

```
T1:         R(X) R(Y) W(X) C
T2: R(X)                    W(Y) C
```

This schedule is conflict-serializable and could not be generated by S2PL. it is conflict-serializable because its precedence graph is acyclic (only edge from T2 to T1). At the start T1 and T2 both acquire a shared lock for X, But when T1 want to get and exclusive lock it cant because T2 is not ready to release its lock yet, which breaks the rules for strict 2PL.

1.2 Show a schedule that could be generated by a S2PL scheduler, but not by a C2PL scheduler

```
T1:         R(X) R(Y) W(Y) C
T2: R(X)                    W(X) C
```

This is S2PL but not C2PL. It is S2PL since T1 and T2 shares the lock for the read operations and T1 has released the lock when T2 needs to write. It is not C2PL because C2PL requires all the locks for a transaction at the beginning making X and exclusive lock and making it so T1 cant obtain a shared lock on X.

1.3 Show a schedule that is serializable, but not conflict-serializable. Is your schedule view-serializable?

```
T1: R(X)        W(X) C
T2:      W(X) C
T3:                  W(X) C
```

This schedule is serializable because it is equivalent to fully completing first T1, T2, and then T3. It is not conflict serializable because the precedence graph has a cycle - specifically one between T1 and T2, as there is a conflict between R(X) in T1 and W(X) in T2, and one between W(X) in T2 and W(X) in T1.

The reason it is not conflict-serializable is that if we draw the precedence graph then we see that it contains cycles. It is view-serializable since all three conditions hold: T1 reads initial value of X in both this and the serial case explained before; No values written are read afterwards; T3 writes final value in both the displayed schedule and the serial one presented earlier.

1.4 Show a schedule of two transactions that is not view-serializable and consists of the minimal possible number of read/write actions to have this property.

```
c
    T1: R(A)
    T2:      W(A)
```

This is the minimum number of actions a schedule with two transactions can have that is not view-serializable. This is not view-serializable because it goes against the definition that If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2.

2 Question 2: More Concurrency Control

2.1 Could Schedule 1 have been generated by a S2PL scheduler?

No, schedule 1 is not a Strict 2PL since transaction C has a shared lock on X as transaction B acquires an exclusive lock on X. In strict 2PL the shared lock that C has is only released upon commit, which happens after B acquires an exclusive lock.

2.2 Could Schedule 1 have been generated by a 2PL scheduler?

No, the schedule cannot have been generated by 2PL since the lock issue explained above still will not have been solved - while 2PL releases locks as they become unnecessary, the shared lock on X is still needed as transaction C will still try to write to X later on.

2.3 Could Schedule 2 have been generated by a C2PL scheduler?

Schedule 2 could not have been generated by a C2PL. C2PL requires all the locks necessary for the transaction at the beginning. We see in schedule 2 that both transaction A and C require exclusive locks on X, and C2PL acquires locks at the beginning of schedule execution.

2.4 Could Schedule 2 have been generated by a 2PL scheduler?

No, this is not 2PL. As we begin, both transaction A and C acquire shared locks on X at some point. Later on transaction C acquires an exclusive lock, while transaction A still has a shared lock, which is incompatible with 2PL.

3 Question 3: Recovery Concepts

3.1 In a system implementing force and steal, is it necessary to implement a scheme for undo? What about a scheme for redo?

If a system implements force it is not necessary to implement a REDO scheme since all update actions will be written to stable storage immediately after being made, making the REDO in case of a crash redundant as data is already in stable storage. If the system implements steal we need to implement an UNDO scheme, as actions that have not yet committed will be able to overwrite data of transactions that have already committed. In cases of abort or crash we need an UNDO to undo the uncommitted changes.

3.2 What is the difference between volatile and stable storage? What types of failures are survived by each type of storage?

The difference between volatile storage and stable storage is that volatile storage is lost in the event of a system crash, usually due to it needing constant power and attendance in order to be active. Stable storage is an almost theoretical level storage type that by definition can survive all types of failures. Volatile storage can survive failures in transactions but fail at both system crashes and media failure (e.g. a stick of ram dies). Stable storage can, as mentioned previously, survive any crash.

3.3 In a system that implements Write-Ahead Logging, which are the two situations in which the log tail must be forced to stable storage? Explain why log forces are necessary in these situations and argue why they are sufficient for durability.

In WAL, just before we write a page to disk we force all logs from this page in the log tail to stable storage. Furthermore if a transaction is committed, then all logs from this transaction that are in the log tail are forced to disk before this commit is concluded.

In the first case this is necessary as we need to force logs to stable storage to have account of what changes are made to the pages on the same storage media. In the second case, given we use non-force we are allowed to data from committed data not be reflected on stable storage, and thus we force the logs to stable storage to be able to reconstruct the given commit in the event of a crash.

Together these ensure that all successful commits can be reconstructed from a crash while ongoing commits can be reconstructed and afterwards rolled back, ensuring durability.

4 Question 4

We are given the following log before a crash

LSN	PREV_LSN	XACT_ID	TYPE	PAGE_ID
1	-	-	begin CKPT	-
2	-	-	end CKPT	-
3	NULL	T1	update	P2
4	3	T1	update	P1
5	NULL	T2	update	P1
6	NULL	T3	update	P2
7	6	T3	commit	-
8	5	T2	commit	-
9	4	T1	update	P2
10	8	T2	end	-
----	XXXXXX	CRASH!	XXXXXX	----

4.1 Show the state of the transaction table and the dirty page table after the recovery phase

We have three transactions: T1, T2, and T3. Transactions have definitively been removed from the transaction table once and `end` record has been logged, which has only happened for T2. Every transaction in the transaction table has

the status Undo except if a commit record has been logged, in which case the status is set to commit. lastLSN is set to the LSN of the last non-end record found through the analysis phase. This gives the following transaction table:

XACT_ID	LAST_LSN	STATUS
T1	9	UNDO
T3	8	COMMIT

The only page chnges in the log are those for pages P1 and P2. T2 has an end record logged, so its changes are reflected on disk, but this isn't necessarily the case for T1 and T3, both of which has made changes to pages P1 and P2, meaning that both are dirty. This gives the following dirty page table:

PAGE_ID	REC_LSN
P1	4
P2	3

4.2 Show the sets of winner and loser transactions

We use the books definition of loser transactions: any transaction that is neither committed nor ended. Let the set T be transactions, T_L be loser transactions and T_W be winner transactions. By the transaction table:

$$T_L = \{T1\},\ \ T_W = T \setminus T_L = \{T2, T3\}$$

4.3 Show the values for the LSNs where the redo phase starts and where the undo phase ends

The redo phase starts at the smallest (earliest) recLSN in the dirty table, i.e. the oldest update that may or may not have been written to disk. Per the dirty table, the redo phase starts at REC_LSN=3.

The undo phase ends at LSN=3, since we start the undo phase at the only active transaction T1 and add prev_LSNs to ToUndo until we reach LSN=3 where prev_LSN is NULL.

4.4 Show the set of log records that may cause pages to be rewritten during the redo phase

In this subsection we assume that for each page, their pageLSN initially starts with a value such that pageLSN¡LSN for the first LSN encountered per page. The redo phase starts at LSN=3 and continues through the log until we reach the final log at LSN=10. Between these we see 4 logs that are either updates or CLRs: LSNs=3, 4, 5, 6, 9. Thus the set of records that can incur potential rewrites to pages are those with LSNs $\{3, 4, 5, 6, 9\}$.

4.5 Show the set of log records undone during the undo phase

As mentioned previously only T1 is active and scheduled for undoing, the undo phase thus begins at LSN=9 and moves through records 9 to 4 to 3. This is because all are updates, so we add their prev_LSN records to ToUndo until we at LSN=3 reach prev_LSN=NULL. Thus the set of LSNs corresponding to logs that are undone is: $\{3, 4, 9\}$

Thus the logs that are undone are those with LSNs 3, 4, and 9.

4.6 Show the contents of the log after the recovery procedure completes

The analysis and redo phases do not add more records to the log. For each update record undone in the undo phase we add a CLR record to signal this, and for each CLR encountered we check if undoNextLSN¬NULL: If it is then undoNetLSN is added to ToUndo, and if not, an end record is written for the transaction. Since we only encounter updates for the three records encountered in the undo phase, we add three CLRs to the log to signify undoing of these. Thus the log will have the following contents:

LSN	PREV_LSN	XACT_ID	TYPE	PAGE_ID
1	-	-	begin CKPT	-
2	-	-	end CKPT	-
3	NULL	T1	update	P2
4	3	T1	update	P1
5	NULL	T2	update	P1
6	NULL	T3	update	P2
7	6	T3	commit	-
8	5	T2	commit	-
9	4	T1	update	P2
10	8	T2	end	-
11	9	T1	CLR	P1
12	11	T1	CLR	P1
13	12	T1	CLR	P1

5 Question 5

5.1 What are the values for the placeholders for undonextLSN in the CLRs?

When a CLR is created, an undoNextLSN is created which points to the record that was undone as the CLR was appended to the log. Since a CLR undoes an update step, we can look at each CLR from the top down and for each,

LSN	PREV_LSN	XACT_ID	TYPE	PAGE_ID	UNDONEXTLSN
1	-	-	begin CKPT	-	-
2	-	-	end CKPT	-	-
3	NULL	T1	update	P3	-
4	3	T1	update	P1	-
5	NULL	T2	update	P3	-
6	NULL	T3	update	P2	-
7	5	T2	update	P2	-
8	4	T1	update	P4	-
9	6	T3	commit	-	-
10	8	T1	abort	-	-
11	7	T2	abort	-	-
12	11	T2	CLR	P2	A
13	9	T3	end	-	-
14	10	T1	CLR	P4	B
15	12	T2	CLR	P3	C
16	14	T1	CLR	P1	D

set `undoLastLSN` to be the `prevLSN` for the update that it undoes, with each successive CLR undoing a write further back in the log for the given transaction. Using this we get that

$$A = 5, B = 4, C = NULL, D = 3$$

5.2 What are the states of the transaction table and of the dirty page table after the analysis phase of recovery?

The analysis starts at the `end_checkpoint` record and looks forward through the log. Through the update steps we add all transaction IDs (T1, T2, T3) to the transaction table. Furthermore, through these all pages (P1, P2, P3, P4) are added to the dirty page table with the `recLSN`s (4, 6, 3, 8) since it is at these LCNs where we first make an update to the pages. T3 is committed and ended, so it is removed from the transaction table. The abort records for T1 and T2 do not change anything. All the CLRs do is change the `LAST_LSN`. Because T1 and T2 haven't seen commits, they both have Undo status. Thus we get the following transaction table and dirty page table.

XACT_ID	LAST_LSN	STATUS
T1	16	UNDO
T2	15	UNDO

PAGE_ID	REC_LSN
P1	4
P2	6
P3	3
P4	8

5.3 Show the additional log contents after the recovery procedure completes.

The additional logs are only produced during the undo phase. We begin with $ToUndo = \{(T1, 16), (T2, 15)\}$, where the first element of a tuple is the transaction and the second element inst `LAST_LSN` field. Upon visiting the first, it is a CLR with an `UNDOLASTLSN`=3, meaning we add $(T1, 3)$ to $ToUndo$. $(T2, 15)$ is a CLR with an `UNDOLASTLSN`=NULL, so we append an end record. $(T1, 3)$ is an update with `PREV_LSN`=NULL, so we add a CLR and an end record afterwards. Thus the additional log contents become the following:

LSN	PREV_LSN	XACT_ID	TYPE	PAGE_ID	UNDONEXTLSN
17	15	T2	end	-	-
18	14	T1	CLR	P3	NULL
19	18	T1	end	-	-

5.4

5.4.1 Which log records could trigger writes to stable storage during normal operation in the scenario above? Why?

The update log, abort, end, and CL records do not under normal operation trigger writes to stable storage. A commit record on the other hand does write to stable storage, because once it is recorded, the log tail up until this commit record is written to stable storage.

5.4.2 Following the ARIES log record structure introduced in the course for update records, is there any information that would not need to be written to stable storage in such a scenario? Why/why not?

While we load all non-volatile storage into volatile storage on start up, we still need to write logs to stable storage so we can recover in the case of a crash. The common fields (`PREV_LSN`, `XACT_ID`, `TYPE`) need to be present for operation of the recovery process - `PREV_LSN` for use in the undo process, `XACT_ID` and `TYPE` for info related to the transaction and type of record. The page id, the length, and the offset are all needed to know where to write data in the pages, and before/after images are needed so we can undo/redo these records.