**Theory Assignment – 2**

**Question 3:**

1. In a system implementing force and no-steal it is not necessary to implement a scheme for redo, because all committed transaction have been written to disk at the time of a crash. It is also not necessary for undo, since all dirty writes have not been written to disk at the time of a crash.
2. Non-volatile storage retains data even when power goes off, while the information in stable storage is (theoretically) completely permanent.

By this we mean that events might result in a loss of data on stable storage, but the probability of data loss is negligible. The non-volatile storage is much faster than stable storage in terms of access time.

So non-volatile storage survives system crashes, but it is still subject to media failure. In our model, we assume stable storage to not experience media failures.

1. The log tail needs to be forced to disk in two cases: When a transaction is committed or when pages are written to disk.

For the first case, if a transaction made a change and committed, the no-force approach means that some of the changes may not have been written to disk at the time of subsequent crash. Without a record of these changes, there would be no way to ensure that the changes of a committed transaction survive crash.

For the second case, if the dirty write is written to the disk in yet uncommitted trans- action at the time of subsequent crash, Without a record of these changes, there would be no way to undo the changes.

These rules are sufficient for durability because they support undoing modifications and ensure all actions of committed transactions survive system crashes and media failures.

**Question 4:**

1. Here is the dirty page table computed in the analysis phase:

|  |  |
| --- | --- |
| **PageID** | **RecLSN** |
| P2 | 3 |
| P1 | 4 |
| P5 | 5 |
| P3 | 6 |

And here is the transaction table:

|  |  |  |
| --- | --- | --- |
| **TransID** | **Status** | **LastLSN** |
| T1 | Active | 4 |
| T2 | Active | 9 |

1. The set of winner transactions is {T 3} since it is the only one that finished

The set of loser transactions is {T1,T2} since these did not finish before the crash

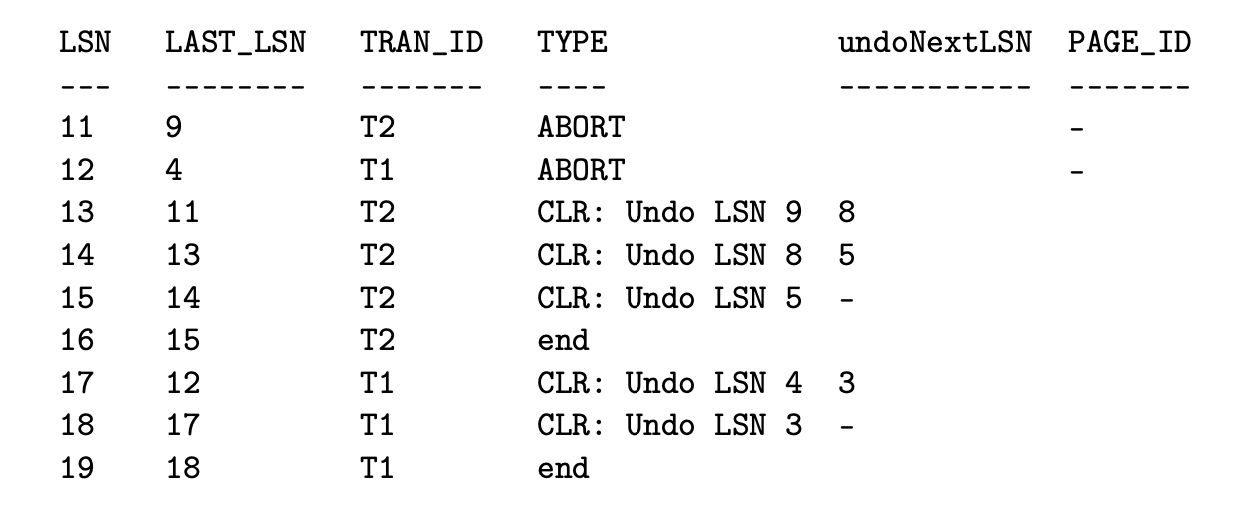
1. The redo phase starts at the minimum recLSN in the dirty page table. This means LSN 3.

The undo phase ends at the oldest LSN of the transactions in the loser set. That would mean LSN 3, since that is the first LSN associated with T1.

1. The set of log records that may ause pages to be rewritten during the redo phase will consist of all update or CLR records after LSN 3, where the redo phase starts.

This means that the set is {3, 4, 5, 6, 8, 9}.

1. The set of log records to undo is the set of updates of the loser transactions. That means LSNs {9, 8, 5, 4, 3}.
2. This is what is appended to the log after the recovery procedure is completed following a crash after LSN 10.



**Question 1:**

**1.**

S = l1(A) r1(A) u1(A) l2(A) w2(A) u2(A) l1(A) w1(A) u1(A)

In this schedule, T1 does not satisfy 2PL; that is it first unlocks A and later locks A again. Also, since w2(A) conflicts with r1(A) and w1(A), H is not conflict-serializable.

**2.** S1: R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)

* Conflicting operations pair (R1(A), W2(A)) because they belong to two different transactions on same data item A and one of them is write operation.
* Similarly, (W1(A), W2(A)) and (W1(A), R2(A)) pairs are also conflicting.
* On the other hand, (R1(A), W2(B)) pair is non-conflicting because they operate on different data item.
* Similarly, ((W1(A), W2(B)) pair is non-conflicting.S1: R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)

**3.**

No schedule exist that is serializable, but not view-serializable.

**4.**

R1[X] → R1[Y ] → R2[X] → W2[X] → R2[Y ] → W1[X] → W1[Y ] → W2[Y ] → C1 → C

A schedule is serializable if it contains the same transactions and operations as a serial schedule and the order of all conflicting operations (read/writes to the same objects by different transactions) is also the same. In the above schedule, T1 reads X before T2 writes X. However, T1 writes X after T2 reads and writes it. The schedule is thus clearly not serializable. Additionally, according to the above schedule, the final content of object X is written by T1 and the final content of object Y is written by T2. Such a result is not possible in any serial execution, where transactions execute one after the other in sequence.