

# Problem Set 3 Solutions

Assigned 10/13

Due 10/27

## 1 Questions

1. Suppose your database system never STOLE pages (see the paper from Haerder and Reuter) – e.g., that dirty pages were never written to disk. How would that affect the design of the recovery manager of your database system?

*(3 points) With a NO-STEAL policy, dirty pages will never be written to disk. Thus, there is no need for UNDO during recovery. REDO is still needed as some committed transactions may not have flushed all of their data to disk. We will still need the ability to UNDO a transaction that ABORTs – thus we will need an in-memory log.*

2. In this problem, you are given three different workloads, each of which contains of a set of concurrent transactions (consisting of READ and WRITE statements on objects). For each workload, several different interleavings of execution are given. Your job is to indicate whether, for each of the five interleavings:

- The given interleaving has an equivalent serial ordering. If so, indicate what the serial ordering is.
- Whether the given interleaving would be valid using lock-based concurrency control. Assume that locks, when needed, are acquired (with the appropriate lock mode) on an object just before the statement that reads or writes the object and that all locks are released during the COMMIT statement (and no sooner.) If the interleaving is not valid, indicate whether or not it would simply never occur or would result in deadlock, and the time when the deadlock would occur.
- Whether the given ordering would be valid using optimistic concurrency control. If not, indicate which transaction will be aborted. Assume the use of the Parallel Validation scheme described in Section 5 of the Optimistic Concurrency Control paper by Kung and Robinson, and that the validation and the write phases of optimistic concurrency control happen during the COMMIT statement (and no sooner.)

Assume in all cases that written values can depend on previously read values.(The workloads are shown on the next page.)

**Workload 1**

<u>Transaction 1</u>	<u>Transaction 2</u>
READ A	READ A
READ B	WRITE B
WRITE C	WRITE A

Interleaving 1:

```

1 T1:  READ A
2 T2:  READ A
3 T1:  READ B
4 T1:  WRITE C
5 T2:  WRITE B
6 T1:  COMMIT
7 T2:  WRITE A
8 T2:  COMMIT

```

Interleaving 2:

```

1 T1:  READ A
2 T1:  READ B
3 T2:  READ A
4 T2:  WRITE B
5 T2:  WRITE A
6 T2:  COMMIT
7 T1:  WRITE C
7 T1:  COMMIT

```

**Workload 2**

<u>Transaction 1</u>	<u>Transaction 2</u>	<u>Transaction 3</u>
READ A	READ A	READ A
WRITE A	WRITE B	WRITE A

Interleaving 3:

```

1 T1:  READ A
2 T2:  READ A
3 T3:  READ A
4 T2:  WRITE B
5 T2:  COMMIT
6 T3:  WRITE A
7 T3:  COMMIT
8 T1:  WRITE A
9 T1:  COMMIT

```

**Workload 3**

<u>Transaction 1</u>	<u>Transaction 2</u>
WRITE A	WRITE B
READ B	READ C

Interleaving 4:

```

1 T1:  Write A
2 T2:  Write B
3 T1:  Read B
4 T2:  Read C
5 T1:  Commit
6 T2:  Commit

```

Interleaving 5:

```

1 T1:  Write A
2 T2:  Write B
3 T2:  Read C
4 T2:  Commit
5 T1:  Read B
6 T1:  Commit

```

(2 points each)

- **Interleaving 1:**  $T1, T2$  is an equivalent serial order. This would not be valid under locking –  $T2$  would not be allowed to `WRITE B` until after  $T1$  had committed. This is a valid schedule under OCC, since the write set of  $T1$  does not intersect the read set of  $T2$ .
  - **Interleaving 2:**  $T1, T2$  is an equivalent serial order, since  $T1$  reads  $A$  and  $B$  before  $T2$  writes either of them, and  $T2$  does not depend on anything that  $T1$  has written. This schedule would not be allowed under either locking or OCC. In OCC,  $T1$  will be aborted since its read set intersects  $T2$ 's write set.
  - **Interleaving 3:** There is no equivalent serial ordering. OCC would roll back  $T1$ . In locking, deadlock would result at time 6, since  $T3$  cannot get an exclusive lock on  $A$  because  $T1$  holds a shared lock on  $A$ .
  - **Interleaving 4:**  $T1, T2$  is an equivalent serial ordering. This would not be allowed under locking, since  $T1$  would not be allowed to read  $B$  after  $T2$  had written it. OCC allows this schedule, since  $T2$ 's read set does not intersect  $T1$ 's write set.
  - **Interleaving 5:**  $T2, T1$  is an equivalent serial ordering. This schedule is allowed by locking.  $T1$  would abort in OCC, since it reads data that  $T2$  wrote and the read phase of  $T1$  overlapped the read and write phase of  $T2$ .
3. Suppose you are told that the following transactions are run concurrently on a (locking-based, degree 3 consistency) database system that has just been restarted and is fully recovered. Suppose the system crashes while executing the statement marked by an “\*\*\*” in Transaction 1. Suppose that Transaction 2 has committed, and the state of Transactions 3 and 4 are unknown (e.g., they may or may not have committed.) Assume that each object (e.g.,  $X, Y$ , etc.) occupies exactly one page of memory.

Trans 1:	Trans 2:	Trans 3:	Trans 4:
<code>x1 = READ X</code>	<code>WRITE Y, 0</code>	<code>z3 = READ X</code>	<code>a4 = READ A</code>
<code>WRITE X, x1 + 1</code>	<code>WRITE B, 0</code>	<code>a3 = READ A</code>	<code>z4 = READ Z</code>
<code>*** y1 = READ Y</code>	<code>x2 = READ X</code>	<code>WRITE A, a3 + 10</code>	<code>WRITE B, (a4-z4)</code>
<code>WRITE Y, y1 + x</code>	<code>WRITE Z, x2</code>	<code>z3 = READ Z</code>	
	<code>y2 = READ Y</code>	<code>WRITE Z, z3 - 10</code>	
	<code>WRITE A, x2 + x1</code>		

- (a) Show an equivalent serial order that could have resulted from these statements, given what you know about what statement was executing when the system crashed. In addition, show an interleaving of the statements from these transactions that is equivalent to your serial order; make sure this serial order could result from a locking-based concurrency control protocol (again assuming that locks are acquired immediately before an item is accessed and released just before the commit statement.)

(3 points) An equivalent serial order would be  $T2, T4, T1, T3$ . A resulting interleaving might be (there are many possible correct answers):

```

T2  WRITE Y, 0
T2  WRITE B, 0
T2  x2 = READ X
T2  WRITE Z, x2
T2  y2 = READ Y
T2  WRITE A, x2 + x1
T4  a4 = READ A
T1  x1 = READ X
T1  WRITE X, x1 + 1
T1  ***

```

- (b) Show all of the records that should be in the log at the time of the crash (given your serial order), assuming that there have been no checkpoints and that you are using an ARIES-style logging and recovery protocol. Your records should include all of the relevant fields described in Section 4.1 of the ARIES paper. Also show the status of the transaction table (as described in Section 4.3 of the ARIES paper) after the analysis phase of recovery has run.

(5 points) Given the above interleaving, the log contains:

LSN	Type	Tid	PrevLSN	PageID	Data
1	BEGIN	1			
2	BEGIN	2			
3	BEGIN	3			
4	BEGIN	4			
5	UPDATE	2	2	Y	OLD = ?, NEW = 0
6	UPDATE	2	5	B	OLD = ?, NEW = 0
7	UPDATE	2	6	Z	OLD = ?, NEW = x2
8	UPDATE	2	7	A	OLD = ?, NEW = x2 + x1
9	COMMIT	2	8		
10	UPDATE	1	1	X	OLD = ?, NEW = x1 + 1

After recovery, the transaction table contains:

Tid	lastLSN
1	10
3	3
4	4

- (c) Suppose you have 2 pages of memory, and are using a STEAL/NO-FORCE buffer management policy as in ARIES. Given the interleaving you showed above, for each of the 5 pages used in these transactions, show one possible assignment of LSN values for those pages as they are on disk before recovery runs. You should use the value “?” if the LSN is unchanged from the prior state of the page before these transactions began. Finally, indicate which pages will be modified during the UNDO pass, and which will be modified during the REDO pass.

(5 points) To determine which pages are dirty, it's helpful to model the state of the buffer pool. Here we assume that a LRU policy is used for eviction.

Stmt	Buffer Pool	Dirty	Action
W Y	Y	Y	
W B	Y B	Y B	
R X	B X	B	Flush Y (LSN 5)
W Z	X Z	Z	Flush B (LSN 6)
R Y	Z Y	Z	
W A	Y A	A	Flush Z (LSN 7)
R A	Y A	A	
R X	X A	A	
W X	X A	X A	

Thus, the page table for the above interleaving would look like:

Page	Page LSN	Modified by REDO?	Modified by UNDO?
A	?	Y	N
B	6	N	N
X	?	Y	Y
Y	5	N	N
Z	7	N	N

4. Suppose you know that the machine your database will run on has a substantial amount of fast, non-volatile memory – that is, memory that would survive a power failure or restart. How would you exploit this memory to improve the performance of the transaction management and recovery mechanisms in your database system? You should consider issues such as the design of the logging system and the need for various phases of recovery. In what ways would this system be faster or simpler than ARIES-style recovery?

*(5 points) There were a number of creative answers to this question. Many people simply assumed that the entire database would fit into the non-volatile store. In this case, we can eliminate the need for a buffer pool, and simply store each object in memory. In this scheme, we use a global object table that has pointers to the committed version of every object. Transactions then modify a shadow copy and atomically install updates at commit time by updating the object table. In this scheme, no REDO or UNDO is needed at recovery time, since transactions executing at the time of the crash will automatically ABORT on recovery by reverting to the pre-shadow copy. Runtime performance is unaffected (since the usual complaint about shadows — that they increase the amount of non-sequential I/O — doesn't apply to an in-memory database.)*

*If you don't want to assume that the entire database will fit into non-volatile memory, then another good approach is to store just the buffer pool and recovery data structures (e.g., dirty page table and transaction table) in non-volatile. Then, when the system recovers from a crash, the buffer pool and recovery structures will be intact. This will eliminate the need for a REDO phase, without requiring us to FORCE writes to disk. We do need to be a little bit careful because it's possible that the system was in the middle of a write to disk when the crash occurred, causing data on disk to be non-action consistent. To correct this, we can keep an extra bit with each page in the dirty page table that indicates we are in the process of writing the page to disk. We set this bit just before writing the page, and unset it after writing completes. If the bit is set at recovery time, we must write the page out again. This ensures page flushes are atomic.*

*Note that, in this scheme, we will still have to UNDO transactions that were running at the time the system crashed by removing their effects from both the buffer pool and disk.*

*We can further increase the performance of this scheme by keeping the log, which consists only of UNDO records, in non-volatile memory. Since we can delete the log records of a transaction after it commits (since we will never need to REDO), this log only contains records for the running transactions, and thus will be quite compact. Note that the Postgres Storage Manager proposes to use non-volatile memory in very much this way.*