Homework 4 Databases

Jan C. Bierowiec

1. (10') Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

   View-serializability allows more schedules than conflict-serializability, but it is difficult to enforce efficiently. Since they do not account for the meanings of the operations or the data, neither definition captures all schedules that might be considered serializable. In practice, conflict-serializability is more commonly supported because it can be efficiently enforced.

   Conflict-serializability is stricter, requiring schedules to be transformable into serial schedules by swapping non-conflicting operations, and it imposes additional constraints on the execution order. Conflict-serializability is easier to enforce efficiently because it focuses on the order of conflicting operations, which can be managed using precedence graphs and lock-based protocols.

   View-serializability focuses on ensuring that the final outcome is the same as if the transactions were executed serially, while also ensuring that initial reads come from the same transaction and that intermediate read-write dependencies are preserved. It does not necessarily preserve the order of non-conflicting operations. It is more challenging to enforce efficiently because it involves tracking each transaction's view, including the initial values read, and ensuring that intermediate states align, which is computationally complex.

2. (30') The lost update anomaly is said to occur if a transaction T_j reads a data item, then another transaction T_k writes the data item (possibly based on a previous read), after which T_j writes the data item. The update performed by T_k has been lost, since the update done by T_j ignored the value written by T_k.

   (a) Give an example of a schedule showing the lost update anomaly.

      i. T1 reads A: T1 reads the initial value of A, let's say A = 100.
      ii. T2 reads A: T2 also reads the initial value of A, which is still A = 100.
      iii. T1 writes A: T1 updates A to A = 150.
      iv. T2 writes A: T2 updates A to A = 200.

   In this example, the update made to A by T1 is lost because it is overwritten by T2. The final value of A is set to 200, which does not account for the update to A made by T1. This situation demonstrates how the lost update anomaly can occur when two transactions read the same initial value of a data item and then perform independent updates, without considering each other's changes.

   (b) Give an example schedule to show that the lost update anomaly is possible with the read committed isolation level.
   In the read committed isolation level, a transaction can only read data that has been committed by other transactions. Here's an example schedule demonstrating the lost update anomaly under the read committed isolation level.

   Consider two transactions, T1 and T2, operating on a single data item, A:

i. T1 reads A: T1 reads the initial value of A, let's say A = 100.
ii. T2 reads A: T2 also reads the initial value of A, which is still A = 100.
iii. T1 writes A: T1 updates A to A = 150.
iv. T1 commits
v. T2 reads A: T2 reads the updated value of A, which is now A = 150.
vi. T2 writes A: T2 updates A to A = 200.

In this schedule, T2 reads the updated value of A after T1 has committed its update to A=150. However, when T2 writes to A, it overwrites the value 150 with 200, effectively losing the update made by T1. The final value of A is 200, ignoring the intermediate update made by T1.This demonstrates the lost update anomaly under the read committed isolation level.

(c) Explain why the lost update anomaly is not possible with the repeatable read isolation level.
The lost update anomaly is not possible with the repeatable read isolation level due to the stricter locking mechanisms and guarantees provided by this isolation level. Reads lock are when a transaction reads a data item, it acquires a read lock on that item. This read lock prevents other transactions from writing to the same data item until the reading transaction completes. Write locks are when a transaction writes to a data item, it acquires a write lock on that item. This write lock prevents other transactions from reading or writing to the same data item until the writing transaction completes. Because of these locking mechanisms, the lost update anomaly is prevented.

When Transaction T1 reads a data item A in the repeatable read isolation level, it acquires a read lock on A.

If Transaction T2 attempts to write to A after T1 has acquired its read lock but before T1 commits, T2 will be blocked because it cannot acquire a write lock due to T1's read lock. T2 will have to wait until T1 releases its read lock by committing or rolling back.

Once T1 commits or rolls back, T2 can proceed with its write operation. Therefore, T2 cannot overwrite the update made by T1, preventing the lost update anomaly.

The repeatable read isolation level ensures that transactions acquire appropriate locks to prevent other transactions from modifying data that has been read but not yet committed, thus guaranteeing that the lost update anomaly does not occur. This prevents the anomaly by serializing read and write operations, allowing transactions to either commit or rollback before others can proceed, avoiding lost updates.

3. Consider the following two transactions:

- T_1:
  - read(A);
  - read(B);
  - if A = 0 then B := B + 1;
  - write(B);
- T_2:
  - read(B);
  - read(A);
  - if B = 0 then A := A + 1;
  - write(A);

Add lock and unlock instructions to transactions T1 and T2 so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

T1:	**T2:**

`Lock(A)`
`read(A);`
`Lock(B)`
`read(B);`
`if A = 0 then B := B + 1;`
`Unlock(A)`
`write(B)`
`Unlock(B)`
	`Lock(B)`
	`read(B);`
	`Lock(A)`
	`read(A);`
	`if B = 0 then A := A + 1;`
	`Unlock(B)`
	`write(A)`
	`Unlock(A)`

We have two transactions, T1 and T2, interacting with two shared resources, A and B. The operations performed by these transactions include conditional write operations based on the values they read from these resources.

To comply with the two-phase locking protocol, each transaction must lock the resources it accesses before any operation and release the locks only after completing all accesses. The locks must be acquired before any unlocks begin (growing phase), and once unlocking starts, no new locks should be acquired (shrinking phase).

- T_1:
  - Lock(A) before reading A.
  - Lock(B) before reading B.
  - Process the conditional logic.
  - Unlock(A) after reading and before writing to B.
  - Write(B) and then Unlock(B).
- T_2:
  - Lock(B) before reading B.
  - Lock(A) before reading A.
  - Process the conditional logic.
  - Unlock(B) after reading and before writing to A.
  - Write(A) and then Unlock(A).

The order in which T1 and T2 acquire locks on A and B can lead to a deadlock under certain conditions:

- Suppose T1 acquires a lock on A and proceeds to read A.
- Concurrently, T2 acquires a lock on B and proceeds to read B.
- T1, after reading A, attempts to acquire a lock on B but must wait because T2 currently holds it.
- Simultaneously, T2, after reading B, tries to acquire a lock on A but must wait because T1 holds it.

This creates a classic deadlock scenario where each transaction is waiting for a resource locked by the other, preventing both transactions from proceeding.

To prevent such deadlocks, a consistent locking order for shared resources should be enforced across all transactions. For instance, always acquire locks in a predefined order, such as locking A before B, regardless of transaction logic. This policy eliminates the circular wait condition, one of the necessary conditions for deadlock.

The current locking sequence, as implemented, allows for the possibility of a deadlock. Enforcing a consistent order in which locks on shared resources are acquired can effectively prevent this problem, ensuring the smooth execution of concurrent transactions in the system.

4. (40') Consider the following sequence of actions S,

S: r1(A), r2(A), r3(B), w1(A), r2(C), r2(B), w2(B), w1(C)

and answer the following questions:

(a) Is the schedule S view-serializable? If so, provide a view-equivalent serial schedule.

View-serializability involves ensuring that the execution order of transactions in a concurrent schedule can be transformed into an equivalent serial schedule, where "equivalent" means that the effect and the final values read by each transaction are the same as if the transactions had been run serially, one after another.

A schedule is view-serializable if it can be shown that there exists a serial schedule with the same view (i.e., read-write dependencies) as the given schedule. Specifically, we must ensure the initial read of any data item in both schedules should be from the same transaction, the final write of any data item in both schedules should be by the same transaction and if a transaction T_j reads a data item previously written by transaction T_i in the schedule, this read-write dependency should also exist in the serial schedule.

- Initial Reads:
  - A is initially read by T1 and T2. In a serial schedule, the initial read should come from the same transaction. T2 can be placed before T1 to maintain this order.
  - B is initially read by T3 and then by T2. T3 should come before T2 to maintain this order.
  - C is read only by T2, which doesn't impose a specific order concerning C.
- Final Writes:
  - A is last written by T1.
  - B is last written by T2.
  - C is last written by T1.
- Read-Write Dependencies:
  - T2 reads A before T1 writes to A, so T2 should come before T1.
  - T3 reads B before T2 writes to B, placing T3 before T2.

Given these dependencies, the only view-equivalent serial schedule that respects all these read-write relationships without violating any initial read conditions or final write conditions is T3,T2,T1:

- T3 reads B before any write happens to B by T2.
- T2 reads A and C before any writes happen to these items by T1.
- T1 executes last, aligning with its writes to A and C being the final operations on these data items.

The schedule S is view-serializable because it can be rearranged into the serial order T3,T2,T1 while maintaining all initial read sources and final write targets, as well as preserving the intermediate read-write dependencies as observed in the original concurrent execution. This serial order shows that the effects and visibility of operations are equivalent to those in S, making it view-serializable.

(b) Is the schedule S conflict-serializable? If so, describe all the conflict-equivalent serial schedules.

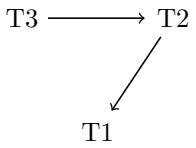
Conflict-serializability involves examining the schedule for conflicts—interactions where one transaction reads or writes a data item followed by another transaction writing to the same item—and ensuring that these can be resolved without cycles in a precedence graph. A schedule is conflict-serializable if its precedence graph, which represents these conflicts, is acyclic.

- Read-Write Conflicts:
  - r2(A) before w1(A): T2 reads A before T1 writes to it, requiring T2→T1.
  - r3(B) before w2(B): T3 reads B before T2 writes to it, requiring T3→T2.
- Write-Read Conflicts:
  - N/A

Precedence Graph:

- Nodes: Transactions: T1, T2, and T3.
- Edges:
  - T2→T1 due to T2's read of A before T1's write.
  - T3→T2 due to T3's read of B before T2's write.

Cycle Check:



There are no cycles in this graph, indicating the schedule is conflict-serializable. Given the acyclic nature of the graph, the conflict-equivalent serial schedule that maintains the order of dependency is T3, T2, T1. This is the only sequence that respects the identified dependencies in the schedule.

The schedule S is conflict-serializable, as confirmed by the acyclic precedence graph. The sequence T3, T2, T1 respects all the read-write conflicts identified, and no other serial orders are possible without violating these dependencies. This order ensures that the operations of the transactions are performed as if they were executed one after the other in this specific sequence, thus maintaining the consistency of the database.

(c) Is the schedule S a 2PL schedule (with exclusive locks)?

- T1:
  - Initially reads A.
  - Later writes to A.
  - During the growing phase, T1 should have an exclusive lock on A when it reads it.
  - However, if T1 releases the lock on A for T2 to read it and then attempts to reacquire it to write to A, this would mean that T1 has entered the shrinking phase prematurely and is trying to lock again, which violates 2PL.
- T2:
  - Initially reads A.
  - Later writes to B.
  - Similar to T1, if T2 releases the lock on A and later attempts to reacquire it, this also violates 2PL.

The schedule S cannot follow the 2PL protocol with exclusive locks as both transactions T1 and T2 exhibit behaviors where they need to reacquire locks after entering the shrinking phase. In 2PL, once a transaction starts releasing locks, it should not attempt to acquire any new locks. This makes the provided schedule non-compliant with the 2PL protocol.

(d) Is the schedule S a 2PL schedule (with shared and exclusive locks)?

SL - Shared Lock
XL - Exclusive Lock
UL - Unlock

SL1(A), r1(A), SL2(A), r2(A), SL3(B), r3(B), SL2(C), XL2(B), UL2(A), XL1(A), w1(A), r2(C), r2(B), w2(B), XL1(C), w1(C)

Yes, when using shared and exclusive locks, the transactions are able to run serial without deadlocks or waiting. We need to anticipate the shared lock on C and the exclusive lock on B in transaction 2. By doing this, transaction 2 can enter the shrinking phase and unlock A. This allows transaction 1 to exclusively lock A and continue without a deadlock.

- Transaction T1:
  - Acquires a shared lock on A for reading.
  - Subsequently, upgrades to an exclusive lock on A for writing.
  - Acquires an exclusive lock on C for writing.
  - Releases all locks appropriately after operations are complete.
- Transaction T2:
  - Acquires shared locks on A, and C for reading.
  - Upgrades to an exclusive lock on B for writing.
  - Unlocks A before entering the shrinking phase.
  - Continues its operations and releases remaining locks.
- Transaction T3:
  - Acquires shared lock on B for reading
  - Unlocks B

By anticipating the need for shared and exclusive locks and acquiring them early in transaction T2, the schedule ensures that it follows a growing phase where all necessary locks are acquired, followed by a shrinking phase where locks are released. Unlocking A allows T1 to acquire the exclusive lock on A and proceed without deadlock.

The revised schedule S adheres to the Two-Phase Locking protocol with shared and exclusive locks, allowing the transactions to run serially without deadlock or unnecessary waiting.

The table is on the last page.

T1	**T2**	**T3**
Begin	**Begin**	**Begin**
SL(A)		
r(A)	SL(A)	
	r(A)	SL(B)
		r(B)
	SL(C)	UL(B)
	XL(B)	
	UL(A)	
XL(A)		
w(A)		
	r(C)	
	r(B)	
	w(B)	
	UL(C)	
	UL(B)	
XL(C)		
w(C)		
UL(A)		
UL(C)		
Commit	**Commit**	**Commit**

Table 1: Transaction Table