Concurrency control

11/30/18

Problems caused by concurrency?

Lost update: the result of a txn is overwritten by another txn

Dirty read: uncommitted results are read by a txn

Non-repeatable read: two reads in the same txn return different results

Phantom read: later reads in the same txn return extra rows

Serial schedule — no problems

T1: R(A), W(A), R(B), W(B), Abort

T2: R(A), W(A), Commit

time

Quiz: Which concurrency problem is this?

T1: R(A), W(A)   R(B), W(B), Abort

T2:   R(A), W(A), Commit

→ time

Dirty read

Quiz: Which concurrency problem is this?

T1: R(A) R(A), W(A), Commit

T2: R(A), W(A), Commit

time

Non-repeatable read

Quiz: Which concurrency problem is this?

T1: R(A), W(A) W(B), Commit

T2: R(A) W(A), W(B), Commit

→ time

Lost update

Quiz: Which concurrency problem is this?

T1: R(A), W(A) W(A), Commit

T2: R(A), R(B), W(B) Commit

time

Dirty read

How to ensure *correctness* when running concurrent txns?

What does correctness mean?

Transactions should have property of *isolation*, i.e., where all operations in a transaction appear to happen together

Fixing concurrency problems

Strawman: Just run txns serially — prohibitively bad performance

Observation: Problems only arise when

1. Two txns touch the same data
2. At least one of these txns involves a *write* to the data

Key idea: Permit schedules whose effects are *equivalent* to serial schedules

Serializability of schedules

Two **operations conflict** if

1. They belong to different txns
2. They operate on the same data
3. One of them is a write

Two **schedules are equivalent** if

1. They involve the same transactions and operations
2. All *conflicting* operations are ordered the same way

A schedule is serializable if it is equivalent to a serial schedule

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit

time

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit

time

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit

time

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B) W(B) Commit

time

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit

time

Serializable

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

time

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

time

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A) W(B), Commit

T2: R(B), W(B), R(A) Commit

time

Testing for serializability

Intuition: Swap *non-conflicting* operations until you reach a serial schedule

T1: R(A), W(A), W(B), Commit

T2: R(B), W(B), R(A) Commit

time

NOT serializable

Testing for serializability

Another way to test serializability:

Draw arrows between conflicting operations

Arrow points in the direction of time

If no cycles between txns, the schedule is serializable

Testing for serializability

Another way to test serializability:

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles between txns, the schedule is conflict serializable

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit

time

Testing for serializability

Another way to test serializability:

Draw arrows between conflicting operations

Arrow points in the direction of time

If no cycles between txns, the schedule is serializable

T1: R(A), W(A), Commit

T2: R(A), R(B), W(B) Commit

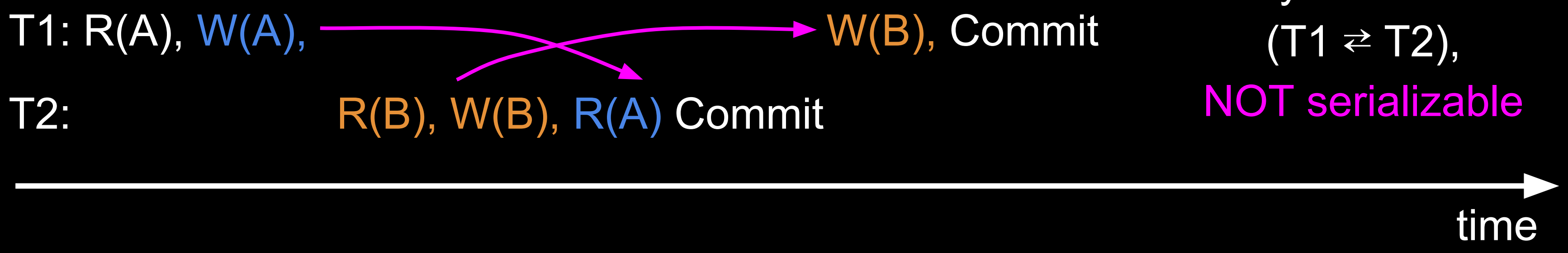
time

No cycles, serializable

Testing for serializability

Another way to test serializability:

- Draw arrows between conflicting operations
- Arrow points in the direction of time
- If no cycles **between txns**, the schedule is serializable



Cycle exists (T1 ⇄ T2), NOT serializable

Implementing serializability: 2PL

Two-phase locking (2PL): acquire all locks before releasing any locks

Each txn acquires shared locks (S) for reads and exclusive locks (X) for writes

- Growing phase: transaction acquires all necessary locks
- Shrinking phase: transaction releases all locks

Cannot acquire more locks after *any* locks are released

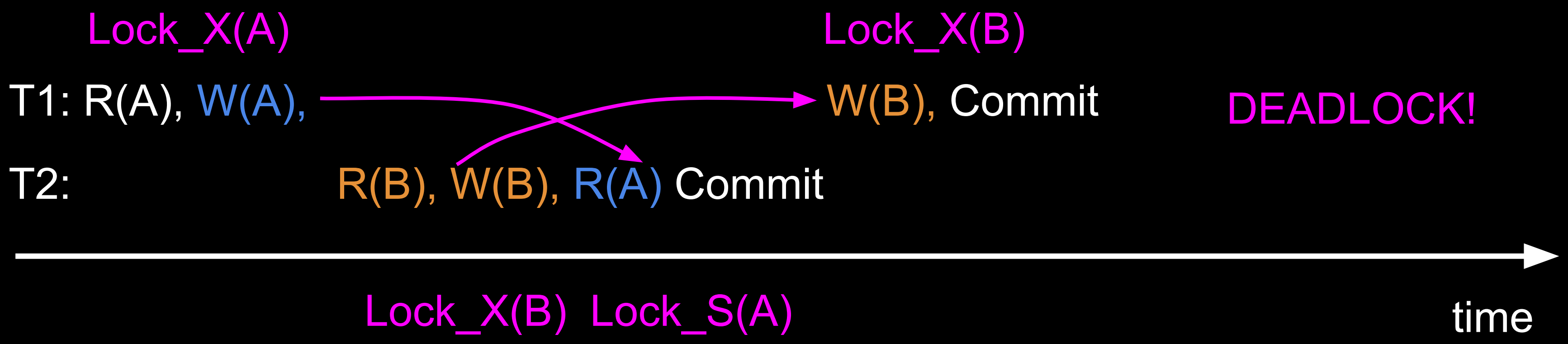
2PL

2PL guarantees serializability by disallowing cycles between txn operation execution

But there could be dependencies among transactions waiting for locks

- Edge from Ti to Tj means Ti acquired lock first and Tj has to wait
- Edge from Tj to Ti means Tj acquired lock first and Ti has to wait
- Cycles mean DEADLOCK!

2PL



Deal with deadlocks by aborting one of the two txns (e.g. detect with timeout)

2PL: Releasing locks too soon?

What if we release the lock as soon as we can?

Lock_X(A) Unlock_X(A)

T1: R(A), W(A), Abort

T2: R(B), W(B), R(A) Abort

time

Lock_X(B) Lock_S(A)

Rollback of T1 requires rollback of T2, since T2 read a value written by T1

Cascading aborts: the rollback of one txn causes the rollback of another

Strict 2PL

Release locks at the *end* of the txn

Variant of 2PL implemented by most databases in practice

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Unlock(A)	<granted>
	Read(A)
	Unlock(A)
	Lock_S(B) <granted>
Lock_X(B)	
	Read(B)
<granted>	Unlock(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

Is this a 2PL schedule?

No

Is this a serializable schedule?

No

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Unlock(A)	<granted>
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	<granted>
	Unlock(A)
	Read(B)
	Unlock(B)

Is this a 2PL schedule?

Yes, and it is serializable

Is this a Strict 2PL schedule?

No, cascading aborts possible

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	<granted>
	Read(A)
	Lock_S(B) <granted>
	Read(B)
	Unlock(A)
	Unlock(B)

Is this a 2PL schedule?

Yes, and it is serializable

Is this a Strict 2PL schedule?

Yes, cascading aborts not possible

Two ways of implementing serializability: 2PL, OCC

2PL (pessimistic):

1. Assume conflict, always lock
2. High overhead for non-conflicting txn (but low for low-conflict workloads)
3. Must check for deadlock

Optimistic concurrency control (OCC):

1. Assume no conflict
2. Low overhead for low-conflict workloads (but high for high-conflict workloads)
3. Ensure correctness by aborting txns if conflict occurs

Optimistic concurrency control

Execute optimistically: Read committed values, write changes locally

Validate: Check if data has changed since original read

Commit (Write): Commit if no change, else abort

These should happen together!

Atomic commit for OCC

Use two-phase commit (2PC) to achieve atomic commit (validate + commit writes)

Recall 2PC protocol:

1. Send *prepare* messages to all nodes, other nodes vote *yes* or *no*
   a. If all nodes accept, proceed
   b. If **any** node declines, abort

2. Coordinator sends *commit* or *abort* messages to all nodes, and all nodes act accordingly

Optimistic concurrency control

Execute optimistically: Read committed values, write changes locally

Validate: Check if data has changed since original read — Phase 1

Commit (Write): Commit if no change, else abort — Phase 2

- Phase 1: send *prepare* to each shard: include buffered write + original reads for that shard
  - Shards validate reads and acquire locks (exclusive for write locations, shared for read locations)
  - If this succeeds, respond with *yes*; else respond with *no*
- Phase 2: collect votes, send result (*abort* or *commit*) to all shards
  - If *commit*, shards apply buffered writes
  - All shards release locks