Distributed Systems

ECE428

Lecture 18

Adopted from Spring 2021

Agenda for today

- Transaction Processing and Concurrency Control
  - Chapter 16
    - Transaction semantics: ACID
    - Isolation and serial equivalence
    - Conflicting operations
    - Two-phase locking
    - Deadlocks
    - Timestamped ordering

- First focus on transactions executed on a single server.
- Look into distributed transactions later (Chapter 17)

Transaction Properties: ACID

- Atomic: all-or-nothing
  - Transaction either executes completely or not at all
- Consistent: rules maintained
- Isolation: multiple transactions do not interfere with each other
  - Equivalent to running transactions in isolation
- Durability: values preserved even after crashes

Isolation

How to prevent transactions from affecting each other?

- Option 1: Execute them serially at the server (one at a time).
  - e.g. through a global lock.
  - But this reduces number of concurrent transactions
- Instead of targeting serial execution, target serial equivalence.
  - Conflicting operations executed in same transaction order.
  - How do we ensure this?
- Option 2: at commit point, check if serial equivalence violated. If yes, abort transaction.
  - Too many aborts. Lower transaction throughput.

Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).

Concurrency Control: Two approaches

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking

- Optimistic: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time

Concurrency Control: Two approaches

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking

- Optimistic: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time

Pessimistic: Locking

- Grabbing a global lock is wasteful
  - what if *no* two transactions access the same object?

- Each object has a lock
  - can further improve concurrency.
  - reads on the same object are non-conflicting.

- Per-object read-write locks.
  - Read mode: multiple transactions allowed in
  - Write mode: exclusive lock

When to release locks?

- We can have per-object locks in two modes to increase concurrency.
- Grab the object's lock in the appropriate mode when trying to access an object.
- *When to release locks?*

	write_lock(A)
	write(A)
write_lock(A)	*unlock(A)*
write(A)	*read_lock(A)*
unlock(A)	read(A)
	unlock(A)

Is this a good idea?

When to release locks?

- We can have per-object locks in two modes to increase concurrency.
- Grab the object's lock in the appropriate mode when trying to access an object.
- *When to release locks?*

write_lock(A)
write(A)
unlock(A)

(T2, T1)

write_lock(A)
write(A)
unlock(A)

(T1, T2)

read_lock(A)
read(A)
unlock(A)

Not serially equivalent

Guaranteeing Serial Equivalence with Locks

- Two-phase locking
  - A transaction cannot acquire (or promote) any locks after it has started releasing locks
  - Transaction has two phases
    1. Growing phase: only acquires or promotes locks
    2. Shrinking phase: only releases locks
       - Strict two phase locking: releases locks only at commit point

Two-phase Locking

write_lock(A)
write(A)
unlock(A)

write_lock(A)
write(A)
unlock(A)

read_lock(A)
read(A)
unlock(A)

Not allowed with two-phase locking

Not serially equivalent

Two-phase Locking

	write_lock(A)
	write(A)
	~~*unlock(A)*~~
write_lock(A)	
blocked	
	~~*read_lock(A)*~~
	read(A)
	unlock(A)
write(A)	
unlock(A)	

Serially equivalent!

Two-phase locking - Serial Equivalence?

- Proof by contradiction
- Assume two phase locking system where serial equivalence is violated for some transactions T1 and T2
- Two facts must then be true:
  - (*A*) For some object O1, there were conflicting operations in T1 and T2 such that the time ordering pair is (T1, T2)
  - (*B*) For some object O2, conflict. operation pair is (T2, T1)
  - (*A*) => T1 released O1's lock and T2 acquired it after that

    => T1's shrinking phase is before or overlaps with T2's growing phase
- Similarly, *(B)* => T2's shrinking phase is before or overlaps with T1's growing phase
- But both these cannot be true!

Downside of Locking

- Deadlock!

Lost Update Example with 2P-Lock

Transaction T1

```
read_lock(x)
x = getSeats(ABC123);
if(x > 0)
   x = x – 1;
write_lock(x)
```

Blocked!

```
write(x, ABC123);
unlock(x)
commit
```

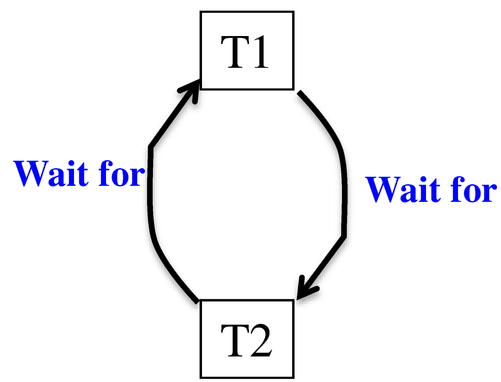
Transaction T2

```
read_lock(x)
x = getSeats(ABC123);
if(x > 0)
 x = x – 1;
write_lock(x)
```

Blocked!

```
write(x, ABC123);
unlock(x)
commit
```

Deadlock!



When do deadlocks occur?

- 3 necessary conditions for a deadlock to occur
  1. Some objects are accessed in exclusive lock modes
  2. Transactions holding locks are not preempted
  3. There is a circular wait (cycle) in the Wait-for graph
- Necessary condition: if there's a deadlock, these conditions are all definitely true
- Not sufficient condition: if they're present, it doesn't imply there is a deadlock.

Combating Deadlocks

1. Lock all objects in the beginning in a single atomic step.
   - no circular wait-for graph created (3rd deadlock condition breaks)
   - may not know of all operations a priori.
2. Lock timeout: abort transaction if lock cannot be acquired within timeout
   - (2nd deadlock condition breaks)
   - Expensive; leads to wasted work
   - How to determine the timeout value?
     - Too large: long delays
     - Too small: false positives, wasted work
3. Deadlock Detection:
   - keep track of Wait-for graph, and find cycles in it (e.g., periodically)
   - If find cycle, there's a deadlock

     => Abort one or more transactions to break cycle (2nd deadlock cond. breaks)

Concurrency Control: Two approaches

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking

- Optimistic: assume the best, allow transactions to write, but check later (for conflicts)
  - E.g., Check at commit time

Optimistic Concurrency Control

- Increases concurrency more than pessimistic concurrency control
- Used in Dropbox, Google apps, Wikipedia, key-value stores like Cassandra, Riak, and Amazon's Dynamo
- Preferable than pessimistic when conflicts are *expected to be* rare
  - But still need to ensure that all conflicts are caught!

First cut approach

- Most basic approach
  - Write and read objects at will
  - Check for serial equivalence at commit time
  - If abort, roll back updates made
  - An abort may result in other transactions that read dirty data, also being aborted
    - Any transactions that read from *those* transactions also now need to be aborted
    - *Cascading aborts*

Timestamped ordering

- Assign each transaction an id
- Transaction id determines its position in serialization order.
- Ensure that for transaction T, both are true:
  1. T's write to object O is allowed only if transactions that have read or written O had lower ids than T.
  2. T's read to object O is allowed only if O was last written by a transaction with a lower id than T.
- Implemented by maintaining read and write timestamps for each object.
- If any of the two rules violated, abort!
- Never results in a deadlock! Older transaction never waits on newer ones.

Timestamped ordering: per-object state

- Committed value.
- Transaction id (timestamp) that wrote the committed value.
- Read timestamps (RTS): List of transaction ids (timestamps) that have read the committed value.
- Tentative writes (TW): List of tentative writes sorted by the corresponding transaction ids (timestamps).
  - Timestamped *versions* of the object.

Timestamped ordering rules

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$ This requires that $T_c \geq$ maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$ This requires that $T_c >$ write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$ This requires that $T_c >$ write timestamp of the committed object.

Timestamped ordering: write rule

Transaction T_c requests a write operation on object D
    if (Tc ≥ max. read timestamp on D and
        Tc > write timestamp on committed version of D)
            Perform a tentative write on D:
                If T_c already has an entry in the TW list for D, update it. Else, add T_c and its write value to the TW list.
    else
        abort transaction T_c
        //too late: there is a transaction with later timestamp that has already read or written the object.

Timestamped ordering: read rule

Transaction T_c requests a read operation on object D

if (T_c > write timestamp on committed version of D) {

D_s = version of D with the maximum write timestamp that is ≤ T_c

//search across the committed timestamp and the TW list for object D.

if (D_s is committed)

read D_s and add T_c to RTS list (if not already added)

else if D_s was written by T_c, simply read D_s

else

wait until the transaction that wrote D_s is committed or aborted, and reapply the read rule.

// if transaction is committed, T_c will read its value after wait.

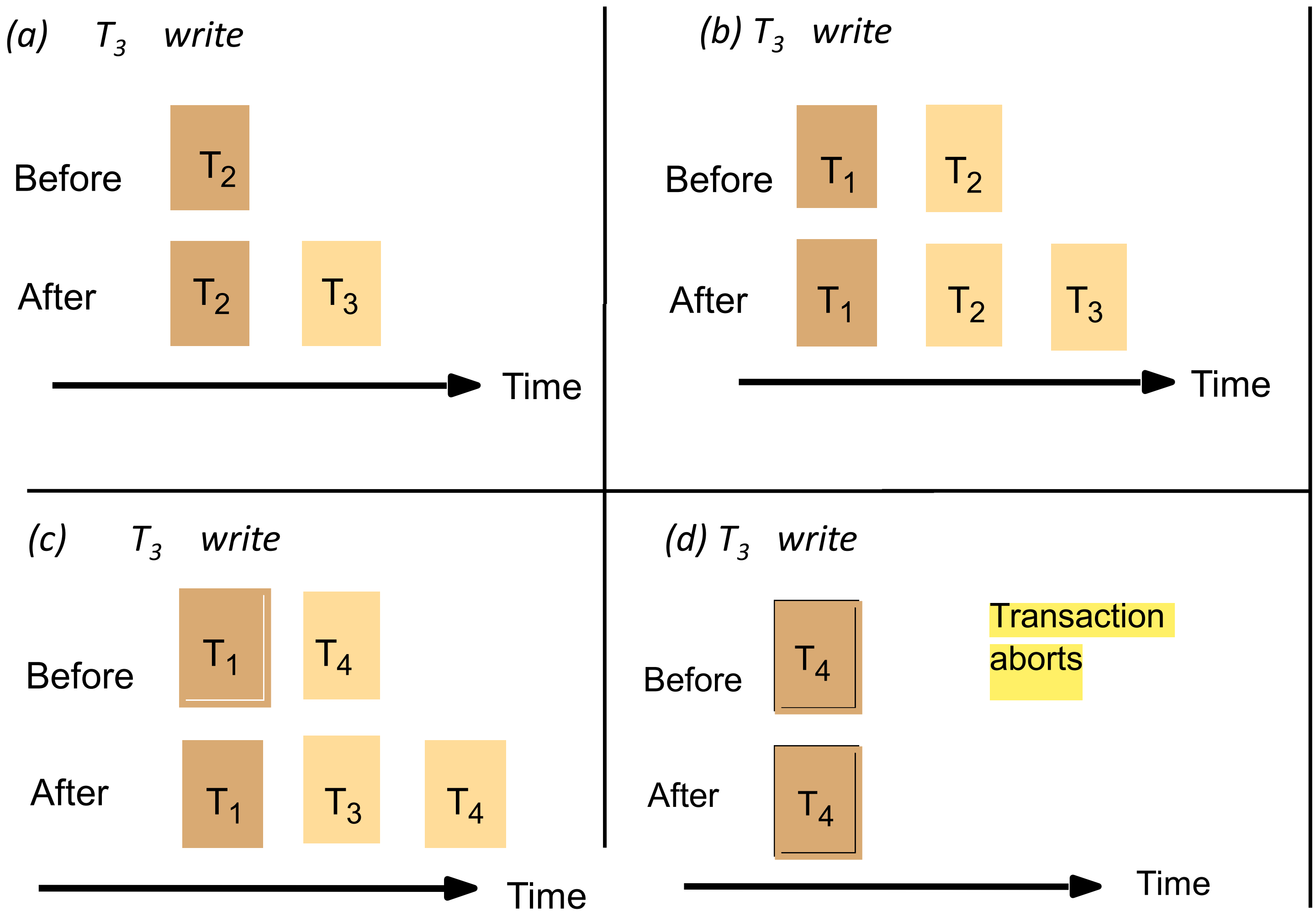
// if the transaction is aborted, T_c will read the value from an older transaction.

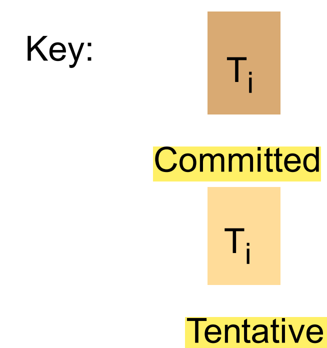
} **else**

abort transaction T_c

//too late: there is a transaction with later timestamp that has already written the object.

Timestamped ordering: write rule

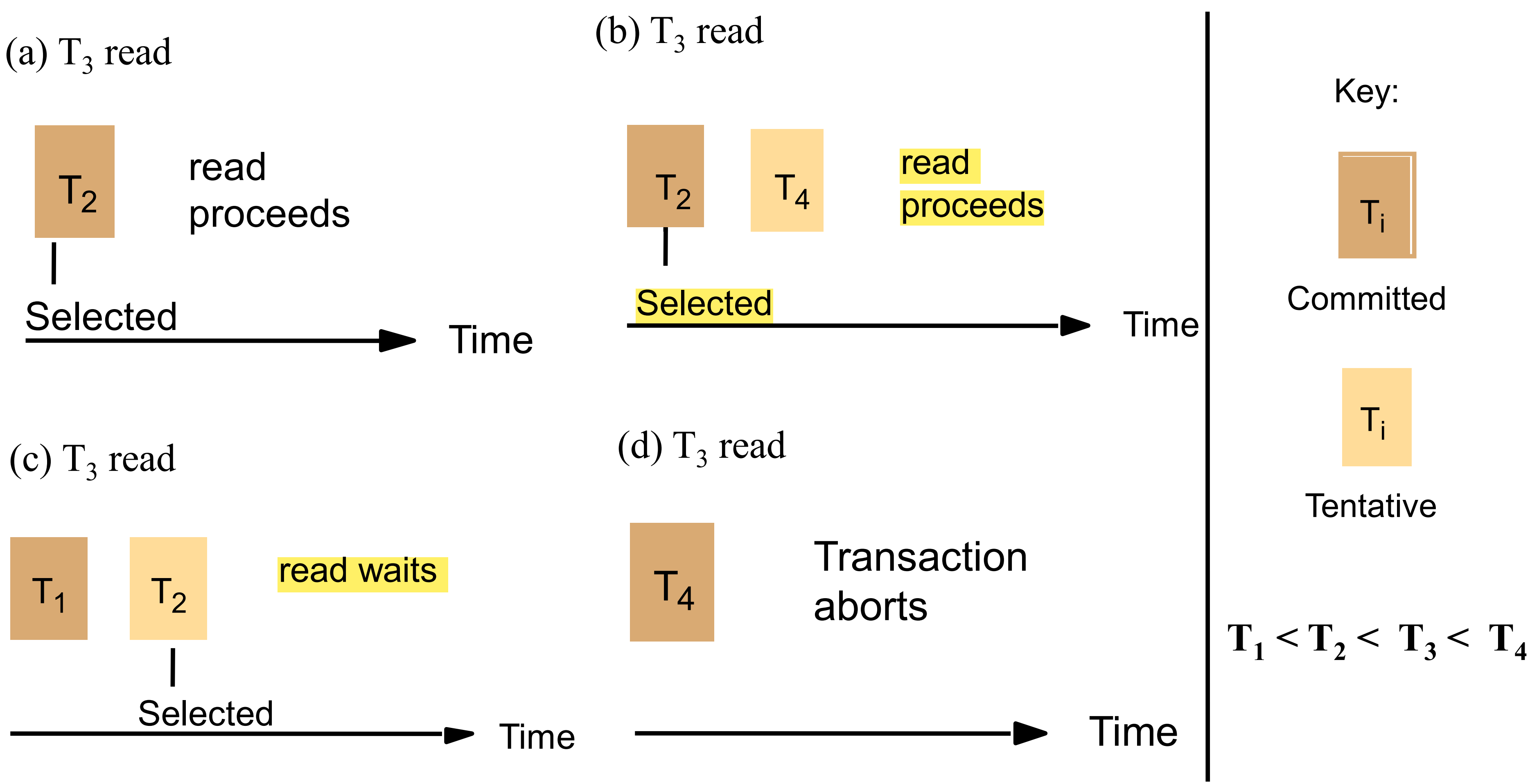




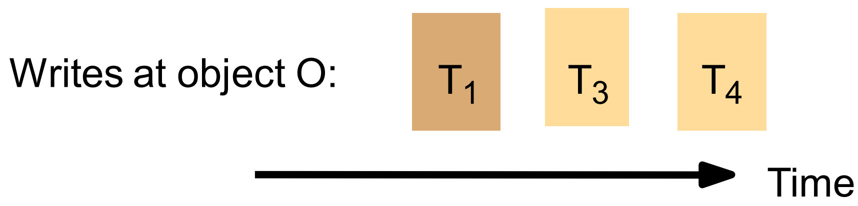
$T_1 < T_2 < T_3 < T_4$

Read timestamps not shown in this example. (assume there are no reads)

Timestamped ordering: read rule

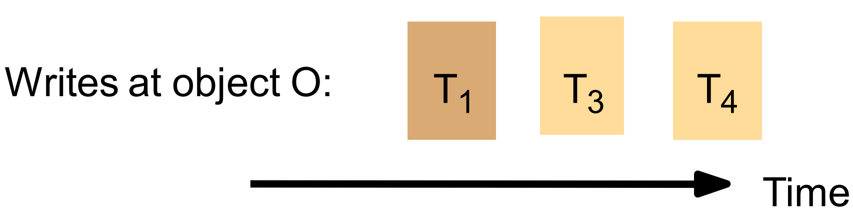


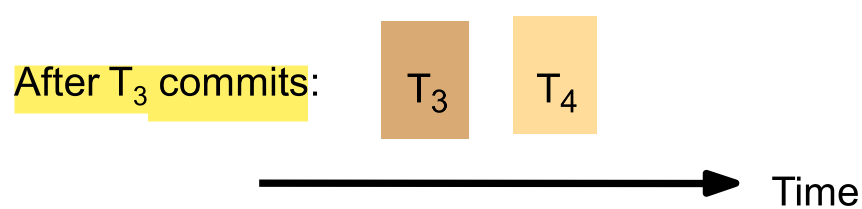
Timestamped ordering: committing



- Suppose T_4 is ready to commit.
- Must wait until T_3 commits or aborts.

- When a transaction is committed, the committed value of the object and associated timestamp are updated, and the corresponding write is removed from TW list.





Lost Update Example with Timestamped Ordering

Transaction T1	Transaction T2	ABC123 state:
x = getSeats(ABC123);		committed value = 10 committed timestamp = 0 RTS: TW:
	x = getSeats(ABC123);	
if(x > 0)	if(x > 0)	
x = x – 1;		
write(x, ABC123);		
	x = x – 1;	
	write(x, ABC123);	
commit		
	commit	

Next Example with Timestamped Ordering

Transaction T1	Transaction T2	
		ABC123 state: committed value = 10 committed timestamp = 0 RTS: TW:

x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);		

	x = getSeats(ABC123); y = getSeats(ABC789);	
write(y+5, ABC789);		
	print(“Total:” x+y);	**ABC789 state:** committed value = 5 committed timestamp = 0 RTS: TW:
commit		
	commit	

Concurrency Control: Summary

- *How to prevent transactions from affecting one another?*
- Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).
- Target *serial equivalence.*
- Two approaches:
  - Pessimistic concurrency control: locking based.
    - read-write locks with two-phase locking and deadlock detection.
  - Optimistic concurrency control: abort if too late.
    - timestamped ordering.

Next Class

- Distributed Transactions.