Distributed Systems

CS425/ECE428

March 29 2023

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta and Nikita Borisov

Logistics

- My OH on Monday will be replaced by Jiangran's (over Zoom, but at the same time).
- Regarding HW4
  - We fixed some typos in the Q2 yesterday morning.
  - You should be able to solve all questions upto 2(f) already.
  - You should be able solve 2(g) after today's class, and Q3 after coming Monday's class.

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**
- Distributed Transactions (if time)

Transaction Properties: ACID

- **A**tomic: all-or-nothing
  - Transaction either executes completely or not at all
- **C**onsistent: rules maintained
- **I**solation: multiple transactions do not interfere with each other
  - Equivalent to running transactions in isolation
- **D**urability: values preserved even after crashes

Isolation

How to prevent transactions from affecting each other?

- Execute them serially at the server (one at a time).
  - e.g. through a global lock.
  - But this reduces number of concurrent transactions

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

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

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

Can lead to Deadlocks!

Transaction T1

read_lock(x)

x = getSeats(ABC123);

if(x > 1)

  x = x – 1;

write_lock(x) *Blocked!*

write(x, ABC123);

commit

Transaction T2

read_lock(x)

x = getSeats(ABC123);

if(x > 1)

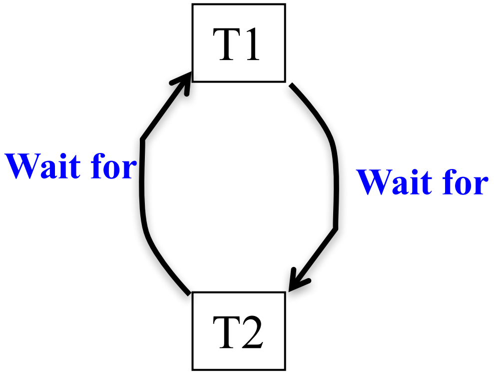
 x = x – 1;

write_lock(x) *Blocked!*

write(x, ABC123);

commit

Deadlock!



When do deadlocks occur?

- 3 <u>necessary</u> 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" = if there's a deadlock, these conditions are all definitely true

- (Conditions not sufficient: if they're present, it doesn't imply a deadlock is present.)

Combating Deadlocks

1. Lock all objects in the beginning in a single atomic step.
   - no circular wait-for graph created (3^{rd} deadlock condition breaks)
   - may not know of all operations a priori.
2. Lock timeout: abort transaction if lock cannot be acquired within timeout
   - (2^{nd} deadlock condition breaks)
   - Expensive; leads to wasted work
   - How to determine the timeout value?
     - Too large: long delays
     - Too small: false positives.
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 (2^{nd} deadlock condition 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
  - 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 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*

T1: time1
T2: time2

Timestamped ordering

- Assign each transaction an id
- Transaction id determines its position in serialization order.
- Ensure that for a transaction T, both are true:
  1. T's write to object O 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 the object
- If rule 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.

A: [3/T1], [5/T_2]

T1 (w A → 3)

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 $T_c \geq$ the 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

&& 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.

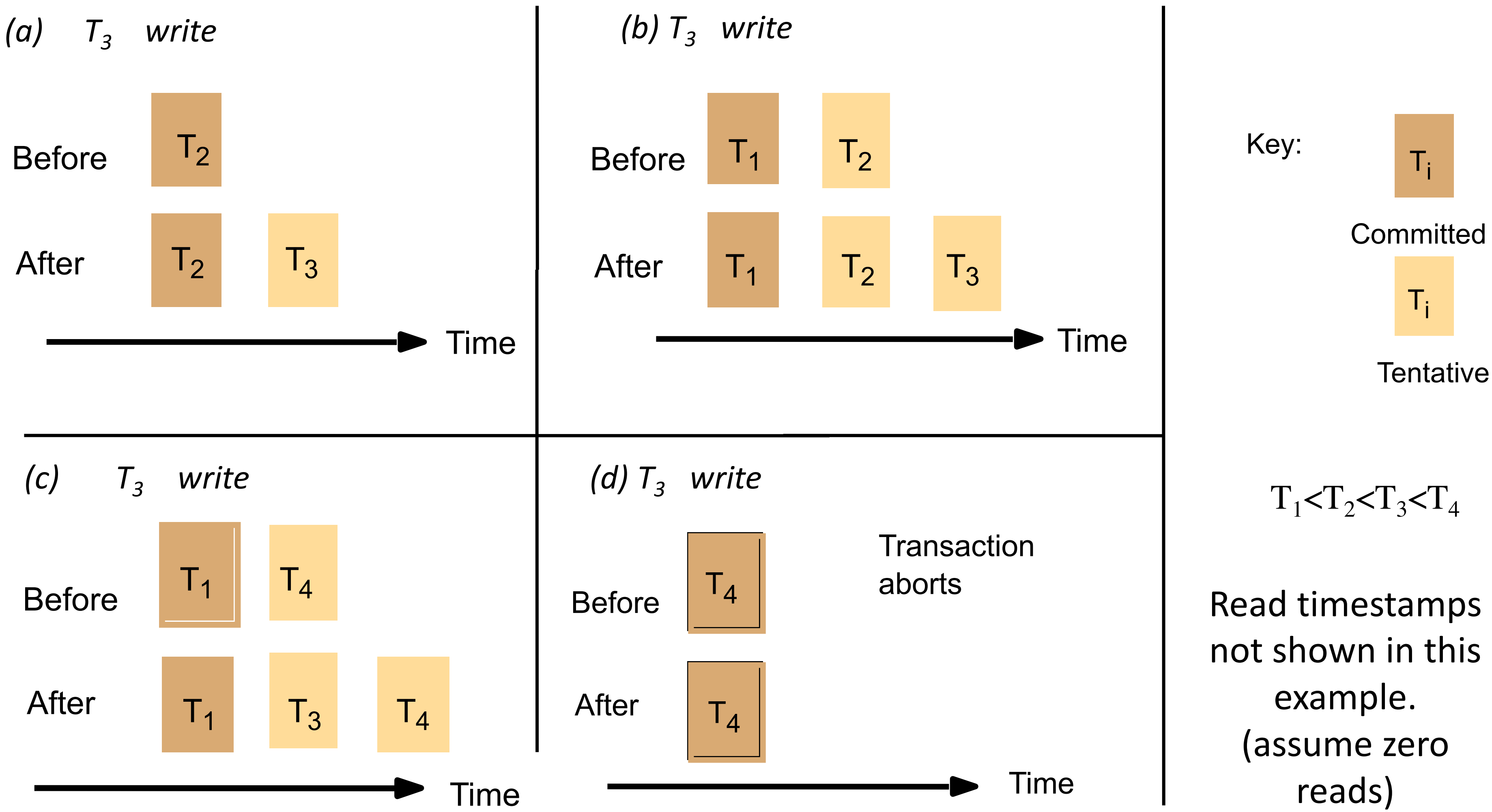
Else, add T_c and its write value to the TW list.

else

abort transaction T_c

//too late; a transaction with later timestamp has already read or written the object.

Timestamped ordering: write rule



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

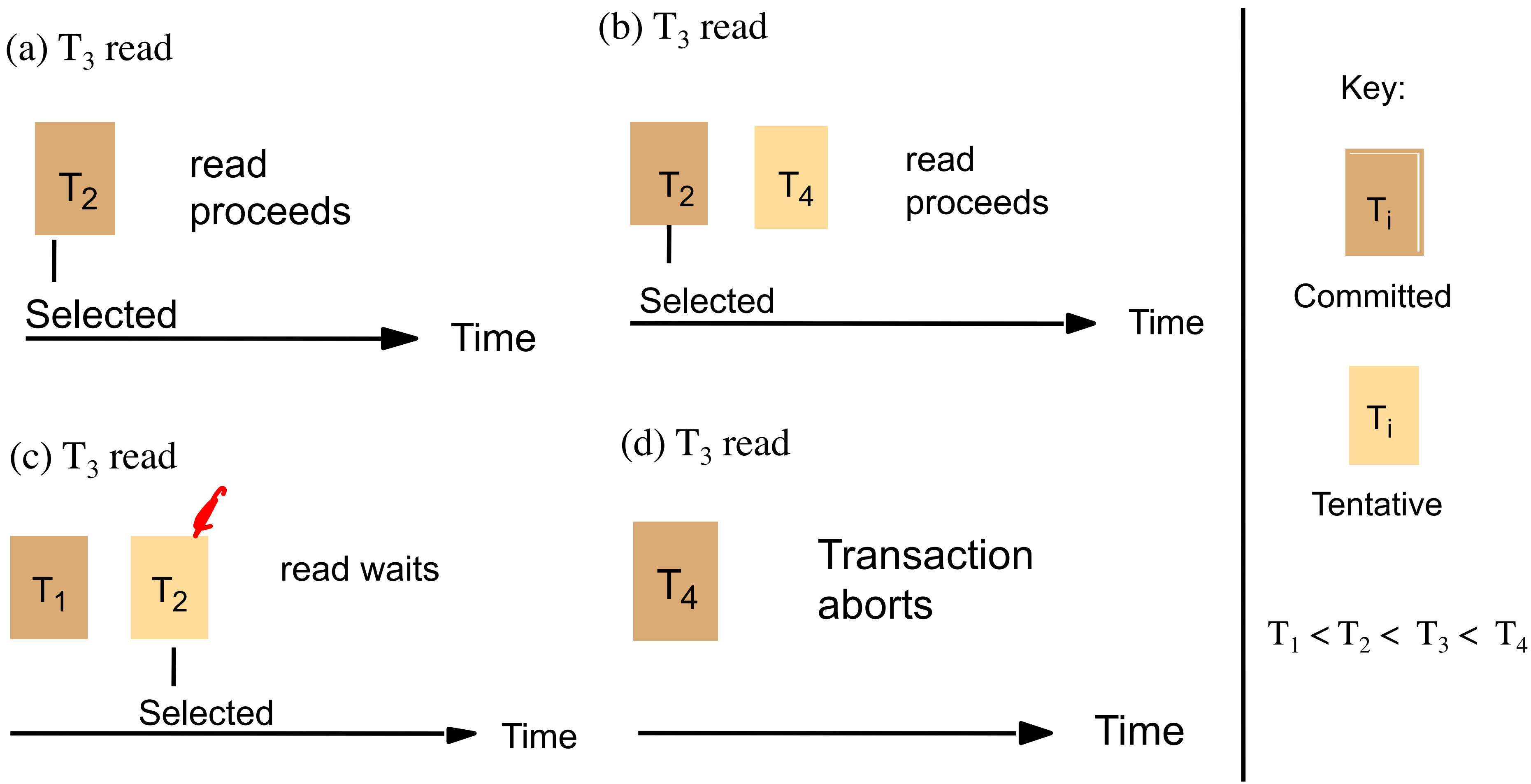
      // if the transaction is aborted, T_c will read the value from an older transaction.

} else

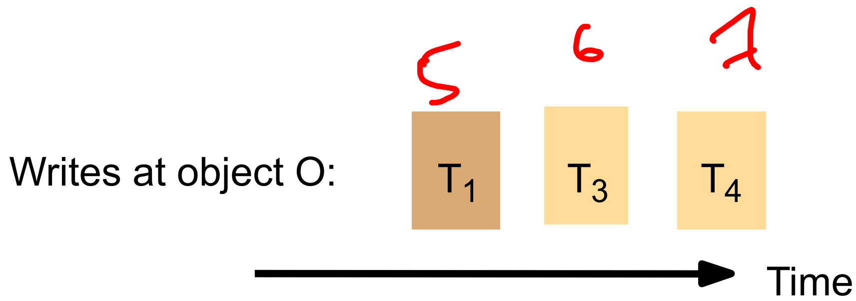
  abort transaction T_c

  //too late; a transaction with later timestamp has already written the object.

Timestamped ordering: read rule

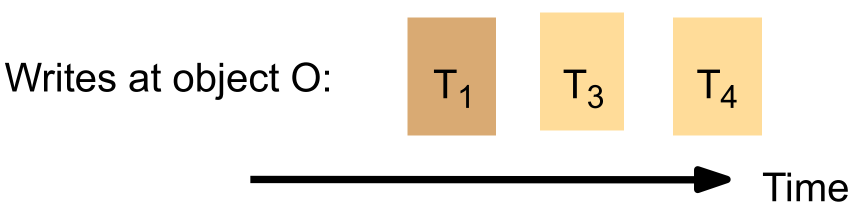


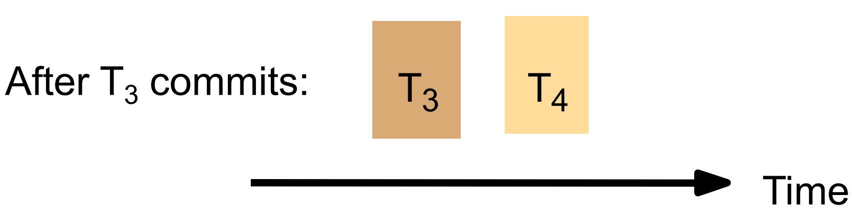
Timestamped ordering: committing



- 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

ABC123: state
committed value = 10
committed timestamp = 0
RTS:
TW:

Transaction T1	**Transaction T2**
x = getSeats(ABC123);	
	x = getSeats(ABC123);
if(x > 1)	if(x > 1)
x = x – 1;	
write(x, ABC123);	
	x = x – 1;
	write(x, ABC123);
commit	
	commit

Lost Update Example with Timestamped Ordering

Transaction T1	**Transaction T2**
x = getSeats(ABC123);	
	x = getSeats(ABC123);
if(x > 1)	if(x > 1)
x = x – 1;	
write(x, ABC123);	
	x = x – 1;
	write(x, ABC123);
commit	
	commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:

Lost Update Example with Timestamped Ordering

Transaction T1	**Transaction T2**
x = getSeats(ABC123);	
	x = getSeats(ABC123);
if(x > 1)	if(x > 1)
x = x – 1;	
write(x, ABC123);	
	x = x – 1;
	write(x, ABC123);
commit	
	commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1, 2
TW:

Lost Update Example with Timestamped Ordering

Transaction T1

x = getSeats(ABC123);

if(x > 1)

x = x – 1;

write(x, ABC123);

commit

Transaction T2

x = getSeats(ABC123);

if(x > 1)

x = x – 1;

write(x, ABC123);

commit

ABC123: state

committed value = 10

committed timestamp = 0

RTS: 1, 2

TW:

Abort!

Next Example with Timestamped Ordering

Transaction T1

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

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS:
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS:
TW:

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	

Next Example with Timestamped Ordering

Transaction T1	**Transaction T2**	
		ABC123: state committed value = 10 committed timestamp = 0 RTS: 1 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	

Next Example with Timestamped Ordering

Transaction T1

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

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS:
TW:

Next Example with Timestamped Ordering

Transaction T1

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

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1

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

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1

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

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123);
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1	**Transaction T2**	
		ABC123: state committed value = 10 committed timestamp = 0 RTS: 1 TW: (5, 1)

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: 1 TW:
commit		
	commit	

Next Example with Timestamped Ordering

Transaction T1

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

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123); wait
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW:

Next Example with Timestamped Ordering

Transaction T1	**Transaction T2**	
		ABC123: state committed value = 10 committed timestamp = 0 RTS: 1 TW: (5, 1)

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

	x = getSeats(ABC123); wait y = getSeats(ABC789);	
write(y+5, ABC789);		
	print("Total:" x+y);	ABC789: state committed value = 5 committed timestamp = 0 RTS: 1 TW:
commit		
	commit	

Next Example with Timestamped Ordering

Transaction T1	Transaction T2	
		ABC123: state committed value = 10 committed timestamp = 0 RTS: 1 TW: (5, 1)

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

	x = getSeats(ABC123); wait y = getSeats(ABC789);	
write(y+5, ABC789);		
	print("Total:" x+y);	ABC789: state committed value = 5 committed timestamp = 0 RTS: 1 TW: (10, 1)
commit		
	commit	

Next Example with Timestamped Ordering

Transaction T1

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

write(y+5, ABC789);

commit

Transaction T2

x = getSeats(ABC123); wait
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW: (5, 1)

ABC789: state
committed value = 5
committed timestamp = 0
RTS: 1
TW: (10, 1)

Next Example with Timestamped Ordering

<u>Transaction T1</u>

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

write(y+5, ABC789);

commit

<u>Transaction T2</u>

x = getSeats(ABC123); wait
y = getSeats(ABC789);

print("Total:" x+y);

commit

ABC123: state
committed value = ~~10~~ 5
committed timestamp = ~~0~~ 1
RTS: 1
TW: ~~(5, 1)~~

ABC789: state
committed value = ~~5~~ 10
committed timestamp = ~~0~~ 1
RTS: 1
TW: ~~(10, 1)~~

Next Example with Timestamped Ordering

Transaction T1	**Transaction T2**	
		ABC123: state committed value = ~~10~~ 5 committed timestamp = ~~0~~ 1 RTS: 1 TW: ~~(5, 1)~~

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

	x = getSeats(ABC123); wait y = getSeats(ABC789);	
write(y+5, ABC789);		
	print("Total:" x+y);	ABC789: state committed value = ~~5~~ 10 committed timestamp = ~~0~~ 1 RTS: 1 TW: ~~(10, 1)~~
commit		
	commit	

T2 then proceeds after T1 commits

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.