ECEN 757: Transactions and Concurrency

Chapter 16

Concurrent Transactions

- To prevent transactions from affecting each other
  - Could execute them one at a time at server
  - But reduces number of concurrent transactions
  - *Transactions per second* directly related to revenue of companies
    - This metric needs to be maximized
- Goal: increase concurrency while maintaining correctness (ACID)

Serial Equivalence

- An interleaving (say O) of transaction operations is serially equivalent iff (if and only if):
  - There is some ordering (O’) of those transactions, one at a time, which
  - Gives the same end-result (for all objects and transactions) as the original interleaving O
  - Where the operations of each transaction occur consecutively (in a batch)
- Says: Cannot distinguish end-result of real operation O from (fake) serial transaction order O’

Checking for Serial Equivalence

- An operation has an effect on
  - The server object if it is a write
  - The client (returned value) if it is a read
- Two operations are said to be conflicting operations, if their *combined effect* depends on the order they are executed
  - read(x) and write(x)
  - write(x) and read(x)
  - write(x) and write(x)
  - NOT read(x) and read(x): swapping them doesn't change their effects
  - NOT read/write(x) and read/write(y): swapping them ok

Checking for Serial Equivalence (2)

- *Two transactions are serially equivalent if and only if all pairs of conflicting operations (pair containing one operation from each transaction) are executed in the same order (transaction order) for all objects (data) they both access.*
  - Take all pairs of conflict operations, one from T1 and one from T2
  - If the T1 operation was reflected first on the server, mark the pair as "(T1, T2)", otherwise mark it as "(T2, T1)"
  - All pairs should be marked as either "(T1, T2)" or all pairs should be marked as "(T2, T1)".

1. Lost Update Problem – Caught!

At Server: seats = 10

Transaction T1

x = getSeats(ABC123);

// x = 10

if(x > 1)

x = x – 1;

write(x, ABC123);

commit

(T2, T1)

(T1, T2)

(T1, T2)

Transaction T2

x = getSeats(ABC123);

if(x > 1) **// x = 10**

x = x – 1;

write(x, ABC123);

commit

T1's or T2's update was lost!

seats = 9

seats = 9

2. Inconsistent Retrieval Problem – Caught!

Transaction T1

x = getSeats(ABC123);

y = getSeats(ABC789);

write(x-5, ABC123);

write(y+5, ABC789);

commit

(T1, T2)

(T2, T1)

Transaction T2

x = getSeats(ABC123);

y = getSeats(ABC789);

// x = 5, y = 15

print("Total:" x+y);

// Prints "Total: 20"

commit

At Server:
ABC123 = 10
ABC789 = 15

T2's sum is the wrong value!
Should have been "Total: 25"

What’s Our Response?

- At commit point of a transaction T, check for serial equivalence with all other transactions
  - Can limit to transactions that overlapped in time with T
- If not serially equivalent
  - Abort T
  - Roll back (undo) any writes that T did to server objects

Dirty Read

Transaction *T*:		**Transaction *U*:**	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	\$100		
a.setBalance(balance + 10)	\$110		
		balance = a.getBalance()	\$110
		a.setBalance(balance + 20)	\$130
		commit transaction	
abort transaction			

Premature Write

Transaction *T*:		**Transaction *U*:**	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

Can We do better?

- Aborting => wasted work
- Can you prevent violations from occurring?

Two Approaches

- Preventing isolation from being violated can be done in two ways
    1. Pessimistic concurrency control
    2. Optimistic concurrency control

Pessimistic vs. Optimistic

- 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, multi-version approaches

Pessimistic: Exclusive Locking

- Each object has a lock
- At most one transaction can be inside lock
- Before reading or writing object O, transaction T must call lock(O)
  - Blocks if another transaction already inside lock
- After entering lock T can read and write O multiple times
- When done (or at commit point), T calls unlock(O)
  - If other transactions waiting at lock(O), allows one of them in
- Sound familiar? (This is Mutual Exclusion!)

Can we improve concurrency?

- More concurrency => more transactions per second => more revenue ($$$)
- Real-life workloads have a lot of read-only or read-mostly transactions
  - Exclusive locking reduces concurrency
  - Hint: Ok to allow two transactions to concurrently read an object, since read-read is not a conflicting pair

Another Approach: Read-Write Locks

- Each object has a lock that can be held in one of two modes
  - Read mode: multiple transactions allowed in
  - Write mode: exclusive lock
- Before first reading O, transaction T calls read_lock(O)
  - T allowed in only if *all* transactions inside lock for O all entered via read mode
  - Not allowed if *any* transaction inside lock for O entered via write mode

Read-Write Locks (2)

- Before first writing O, call write_lock(O)
  - Allowed in only if no other transaction inside lock
- If T already holds read_lock(O), and wants to write, call write_lock(O) to *promote* lock from read to write mode
  - Succeeds only if no other transactions in write mode or read mode
  - Otherwise, T blocks
- Unlock(O) called by transaction T releases any lock on O by T

Guaranteeing Serial Equivalence With Locks

- Two-phase locking
  - A transaction cannot acquire (or promote) any locks after it has started releasing locks (why?)
  - 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

2. Inconsistent Retrieval Problem – Caught!

Transaction T1	**Transaction T2**
Obtain lock: ABC123	
write(x-5, ABC123);	
Release lock: ABC123	
	x = getSeats(ABC123);
	y = getSeats(ABC789);
Obtain lock:ABC789	
write(y+5, ABC789);	
Release lock: ABC789	
	print("Total:" x+y);
commit	
	commit

At Server:
ABC123 = 10
ABC789 = 15

T2's sum is the wrong value!
Should have been "Total: 25"

Why Two-phase Locking => Serial Equivalence?

- Proof by contradiction
- Assume two phase locking system where serial equivalence is violated for some two transactions T1, 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, the conflicting 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

- Deadlocks!

Downside of Locking – Deadlocks!

<u>Transaction T1</u>

Lock(ABC123);

x = write(10, ABC123);

Lock(ABC789);

// Blocks waiting for T2

…

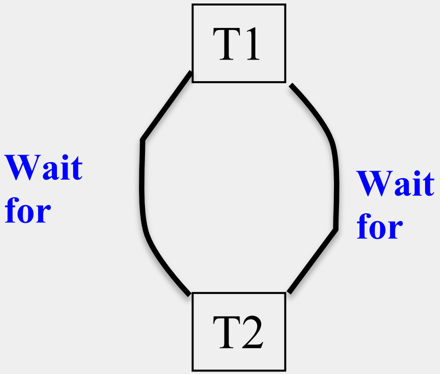
<u>Transaction T2</u>

Lock(ABC789);

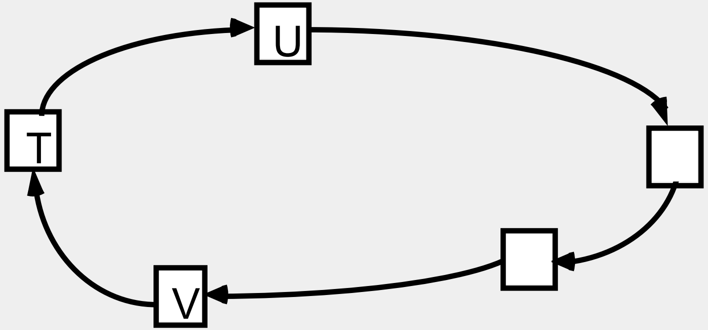
y = write(15, ABC789);

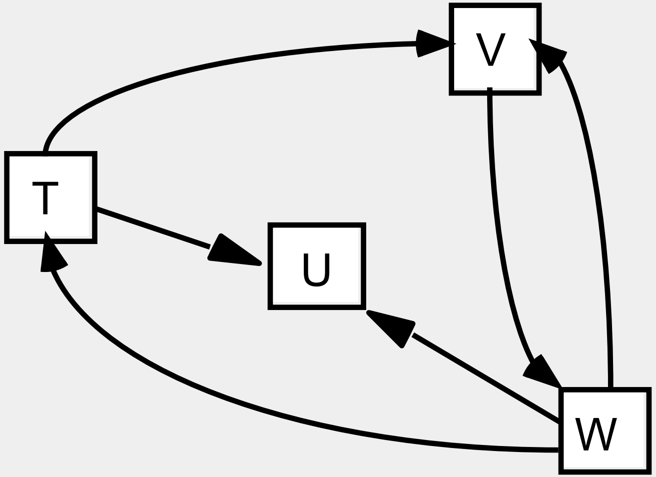
Lock(ABC123);

… **// Blocks waiting for T1**



More Examples of Deadlocks





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 cannot be 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 timeout: abort transaction if lock cannot be acquired within timeout

   ☹️ Expensive; leads to wasted work

2. Deadlock Detection:

   –keep track of Wait-for graph (e.g., via Global Snapshot algorithm), and

   –find cycles in it (e.g., periodically)

   –If find cycle, there’s a deadlock => Abort one or more transactions to break cycle

   ☹️ Still allows deadlocks to occur

Combating Deadlocks (2)

3. Deadlock Prevention

- Set up the system so one of the *necessary conditions* is violated
  1. *Some objects are accessed in exclusive lock modes*
     - Fix: Allow read-only access to objects
  2. *Transactions holding locks cannot be preempted*
     - Fix: Allow preemption of some transactions
  3. *There is a circular wait (cycle) in the Wait-for graph*
     - Fix: Lock all objects in the beginning; if fail any, abort transaction => No cycles in Wait-for graph
     - Fix#2: Locks can only be obtained in ascending order. That is, one cannot obtain lock 2 first, and then obtain lock 1.
     - Either fix reduces concurrency

Next

- Can we allow more concurrency?
- Optimistic Concurrency Control

Optimistic Concurrency Control

- Increases concurrency more than pessimistic concurrency control
- Increases transactions per second
- For non-transaction systems, increases operations per second and lowers latency
- 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

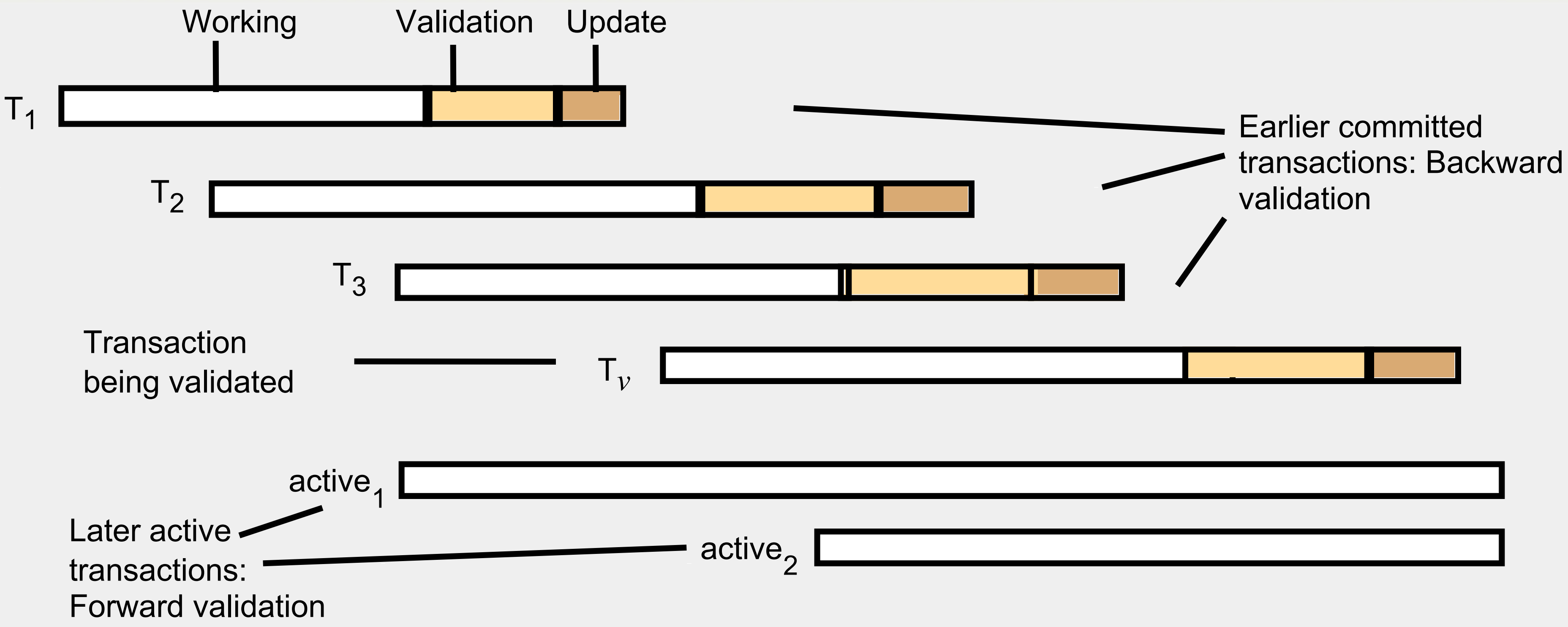
- Most basic approach
  - Working phase: Write and read objects at will
    - Values of write operations are stored locally, and invisible to all other transactions
    - Actual writes happen at the time of commit
  - Validation phase: Check for serial equivalence at commit time
  - Update phase: If 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*

Validation Phase

- Need to check the following three cases
- Here "write" means the write in the update phase
- Only allow one process in the Validation/Update phase
  - These two phases are usually short. This eliminates the third possibility

T_v	T_i	Rule
write	*read*	1. T_i must not read objects written by T_v
read	*write*	2. T_v must not read objects written by T_i
write	*write*	3. T_i must not write objects written by T_v and T_v must not write objects written by T_i

Two Types of Validation



Backward Validation

- Rule 1 is satisfied automatically
- Only need to check rule 2

T_v	T_i	Rule	
write	*read*	1.	T_i must not read objects written by T_v
read	*write*	2.	T_v must not read objects written by T_i
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i

Forward Validation

- Rule 2 is satisfied automatically
- Only need to check rule 1

T_v	T_i	Rule
write	*read*	1. T_i must not read objects written by T_v
read	*write*	2. T_v must not read objects written by T_i
write	*write*	3. T_i must not write objects written by T_v and T_v must not write objects written by T_i

Backward validation of transaction T_v

```
boolean valid = true;
for (int Ti = startTn+1; Ti <= finishTn; Ti++){
        if (read set of Tv intersects write set of Ti) valid = false;
}
```

Forward validation of transaction T_v

```
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
        if (write set of Tv intersects read set of Tid) valid = false;
}
```

Second approach: Timestamp 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

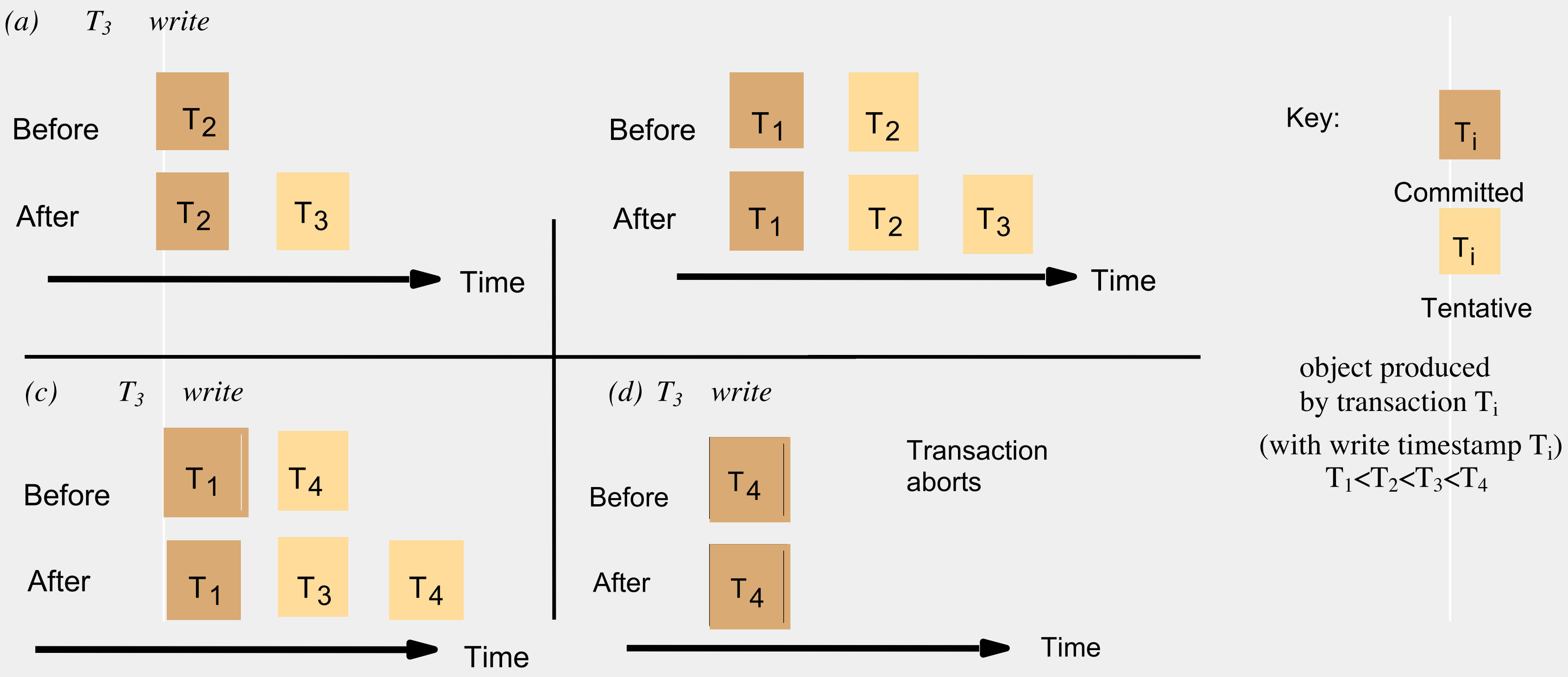
Timestamp Ordering

- Every object has a (committed) write timestamp, a set of read timestamps, and a set of tentative versions
  - Tentative versions are invisible to other processes
  - Each tentative version also has a write timestamp
- If a “write” operation is accepted => Create a tentative version
- If a “read” operation is accepted:
  - Read the version with the maximum write timestamp less than the transaction timestamp
  - Add the transaction timestamp to the set of timestamp
- When a transaction commits:
  - Tentative version becomes the value of the object
  - Timestamp of tentative version becomes timestamp of the object

Operation Conflicts

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.

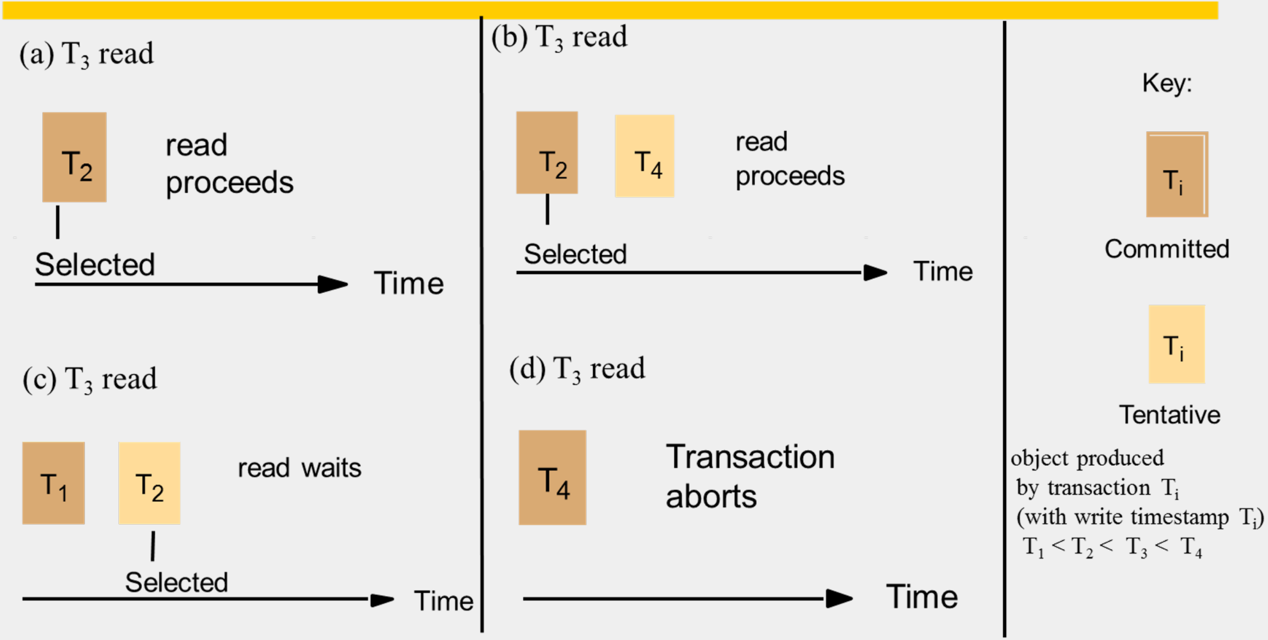
Write Operation (If timestamp > max read timestamp)



Write Operation

if ($T_c \geq$ maximum read timestamp on D &&
$T_c >$ write timestamp on committed version of D)
perform write operation on tentative version of D
with write timestamp T_c
else /* write is too late */
Abort transaction T_c

Read Operation



Read Operation

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

let D_{selected} be the version of D with the maximum write timestamp $\leq T_c$

if ($\mathrm{D}_{\text{selected}}$ is committed)

perform *read* operation on the version D_{selected}

else

Wait until the transaction that made version D_{selected} commits or aborts then reapply the *read* rule

} else

Abort transaction T_c

Summary

- RPCs and RMIs
- Transactions
- Serial Equivalence
  - Detecting it via conflicting operations
- Pessimistic Concurrency Control: locking
- Optimistic Concurrency Control