Optimistic Concurrency Control

April 18, 2018

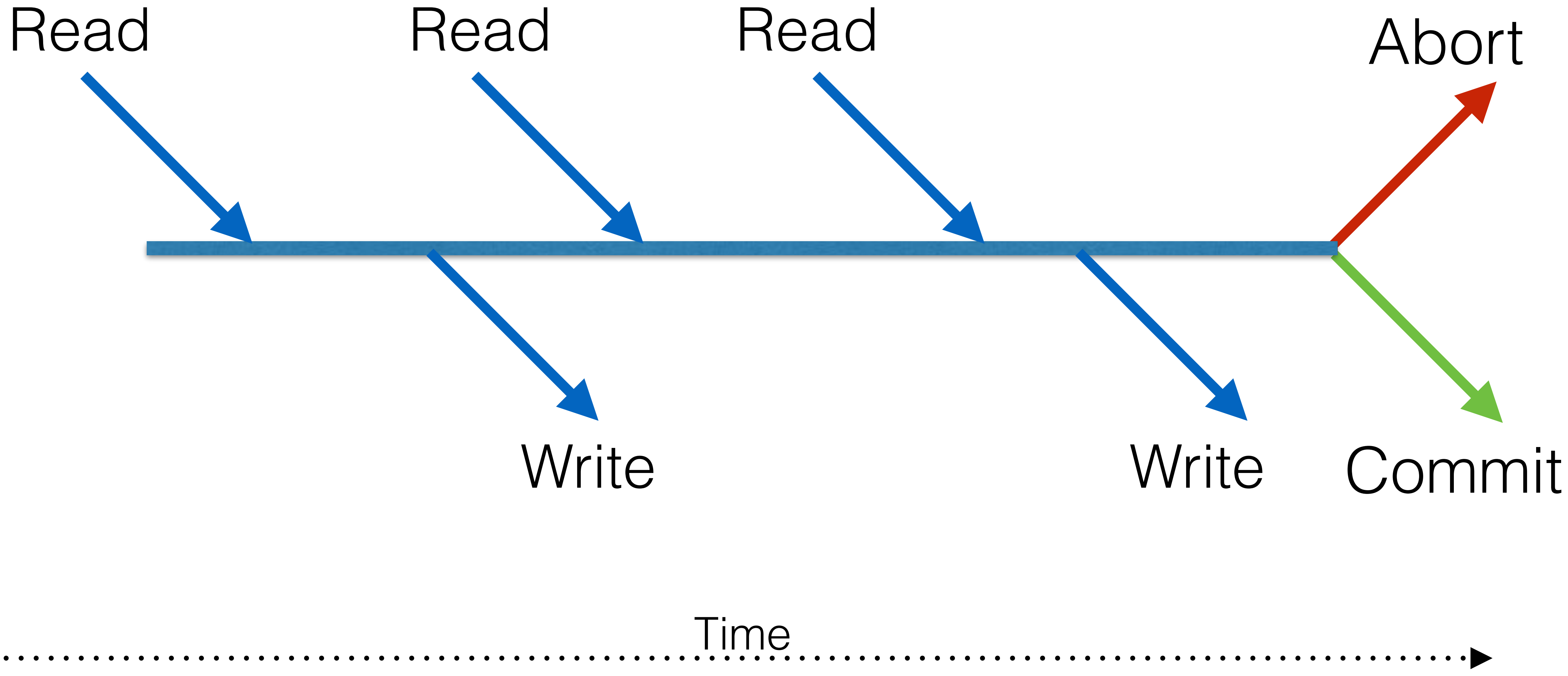
Serializability

Executing transactions serially wastes resources

Interleaving transactions creates correctness errors

Give transactions the *illusion* of isolation

Serializability



Read

Read

Read

Abort

Write

Write

Commit

Time

The Illusion of Isolation

Preserve order of reads, writes across transactions

The Illusion of Isolation

The Illusion of Isolation

Option 1: Avoid situations that break the illusion

Locking

Lock an object before reading or writing it

Unlock it after the transaction ends

This is pessimistic!

Locking

Time	T1	T2
↓	`W(A)`	
	`W(B)`	
		`W(A)`
		`W(B)`
	`COMMIT`	`COMMIT`

Not allowed! T2 has to wait!

Locking

- This is <u>expensive</u>! Locking costs are still incurred even if no conflicts ever actually occur!
- This is <u>restrictive</u>! Don't know in advance what an xact will do, so can't allow all schedules.

We don’t know what a transaction will do until it does.

So let the transaction do it.

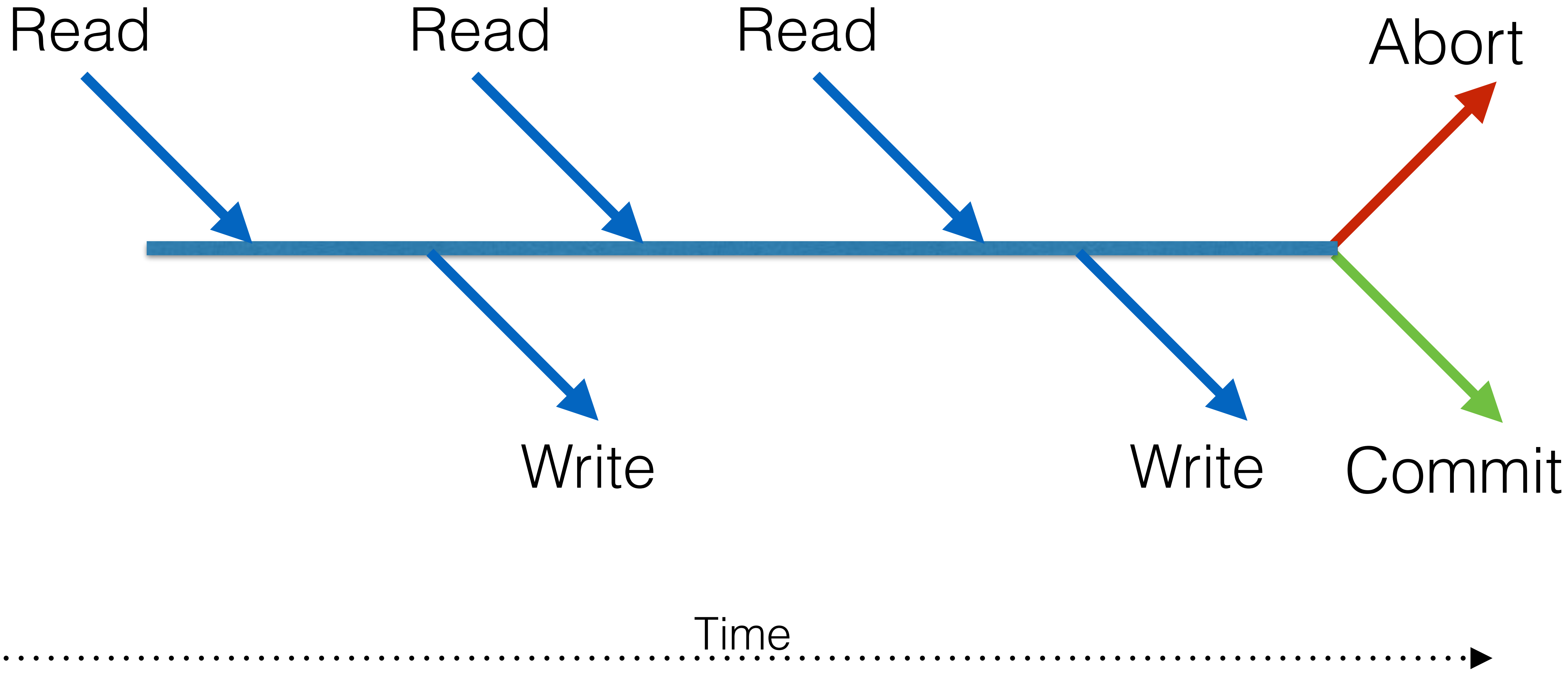
So let the transaction do it.

(Then check if it broke anything later)

Optimistic CC

- **Read Phase**: Transaction executes on a private copy of all accessed objects.
- **Validate Phase**: Check for conflicts.
- **Write Phase**: Make the transaction’s changes to updated objects public.

Read Phase



Read

Read

Read

Abort

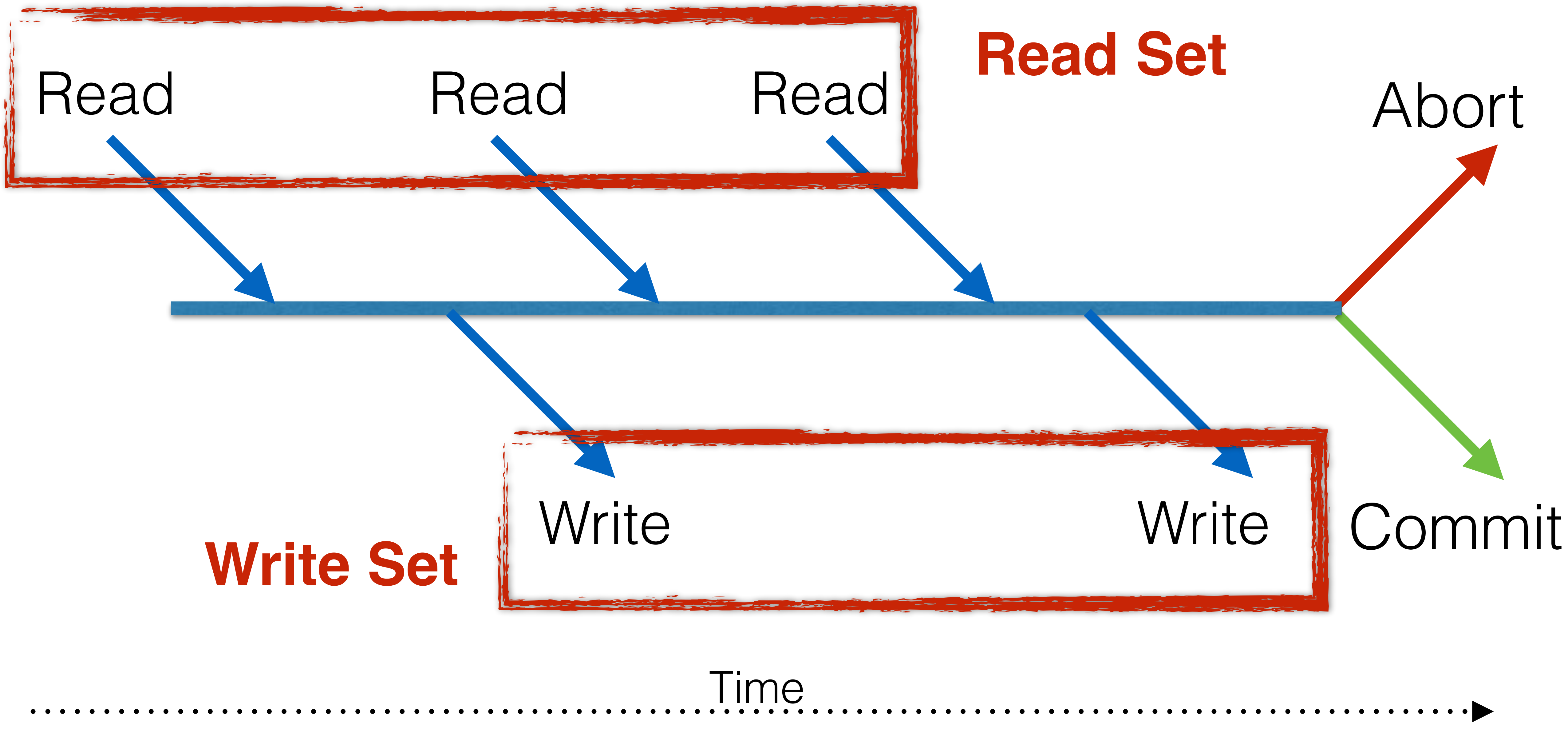
Write

Write

Commit

Time

Read Phase


Read
Read
Read
Read Set
Abort
Write
Write
Commit
Write Set
Time

Read Phase

ReadSet(T_i): Set of objects read by T_i.

WriteSet(T_i): Set of objects written by T_i.

Validation Phase

Pick a serial order for the transactions
(e.g., assign id #s or timestamps)

Validation Phase

Pick a serial order for the transactions
(e.g., assign id #s or timestamps)

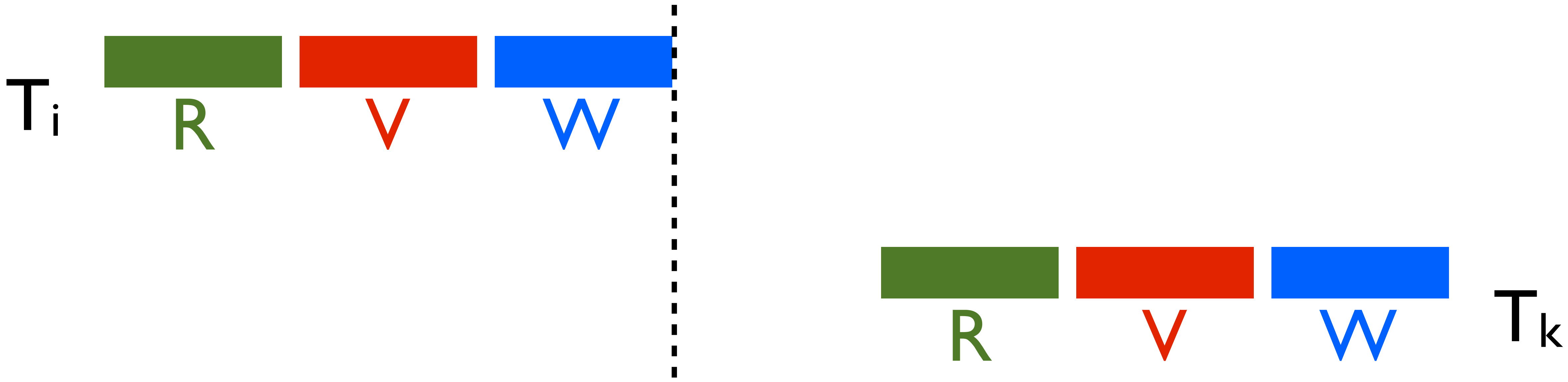
When should we assign Transaction IDs? (Why?)

Validation Phase

What can we test to make sure that transactions applied their effects in the right order?

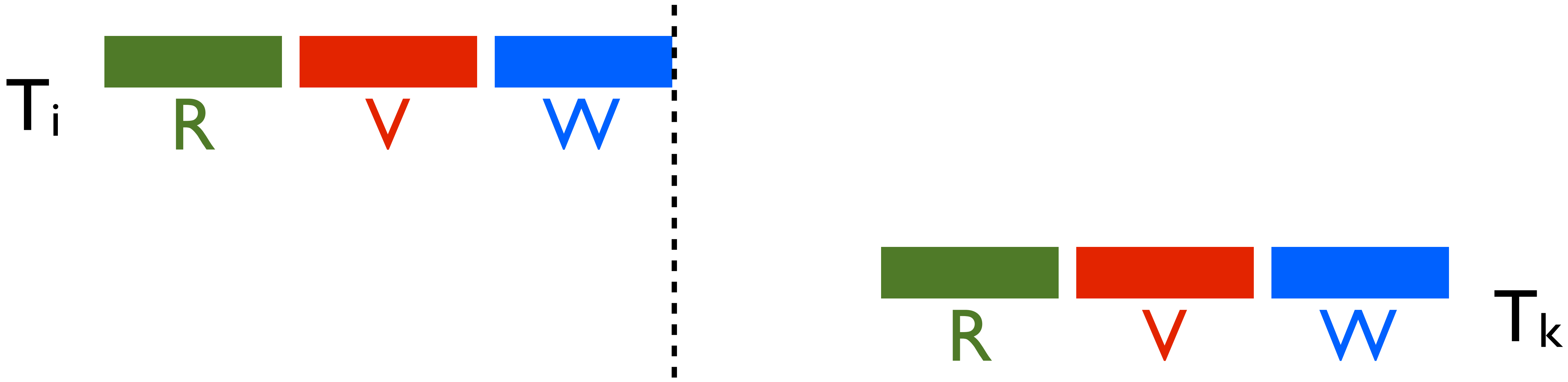
Simple Test

For all i and k for which i < k,
check that Ti completes before Tk begins.



Simple Test

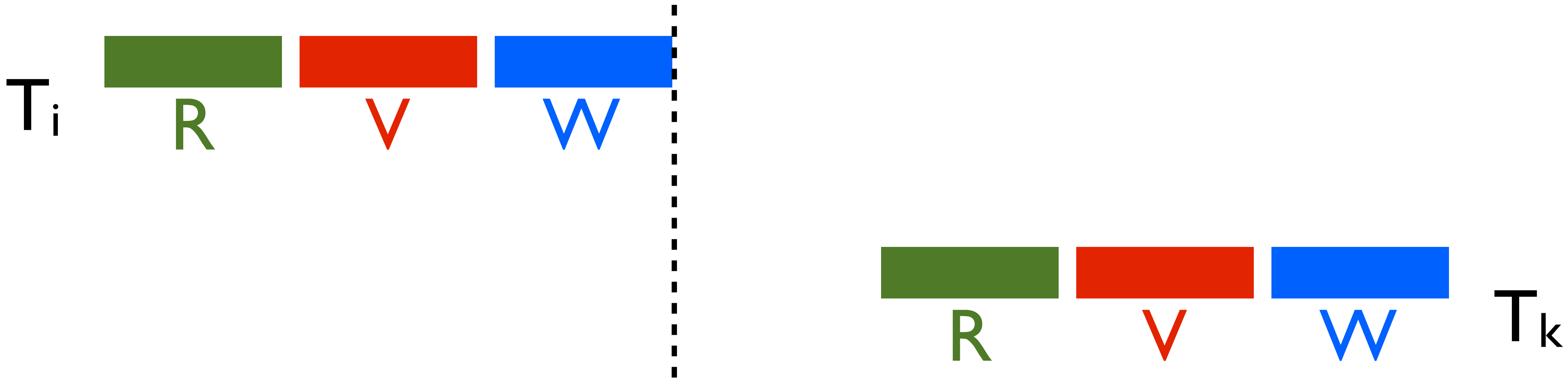
For all i and k for which i < k,
check that Ti completes before Tk begins.



Is this sufficient?

Simple Test

For all i and k for which i < k,
check that Ti completes before Tk begins.

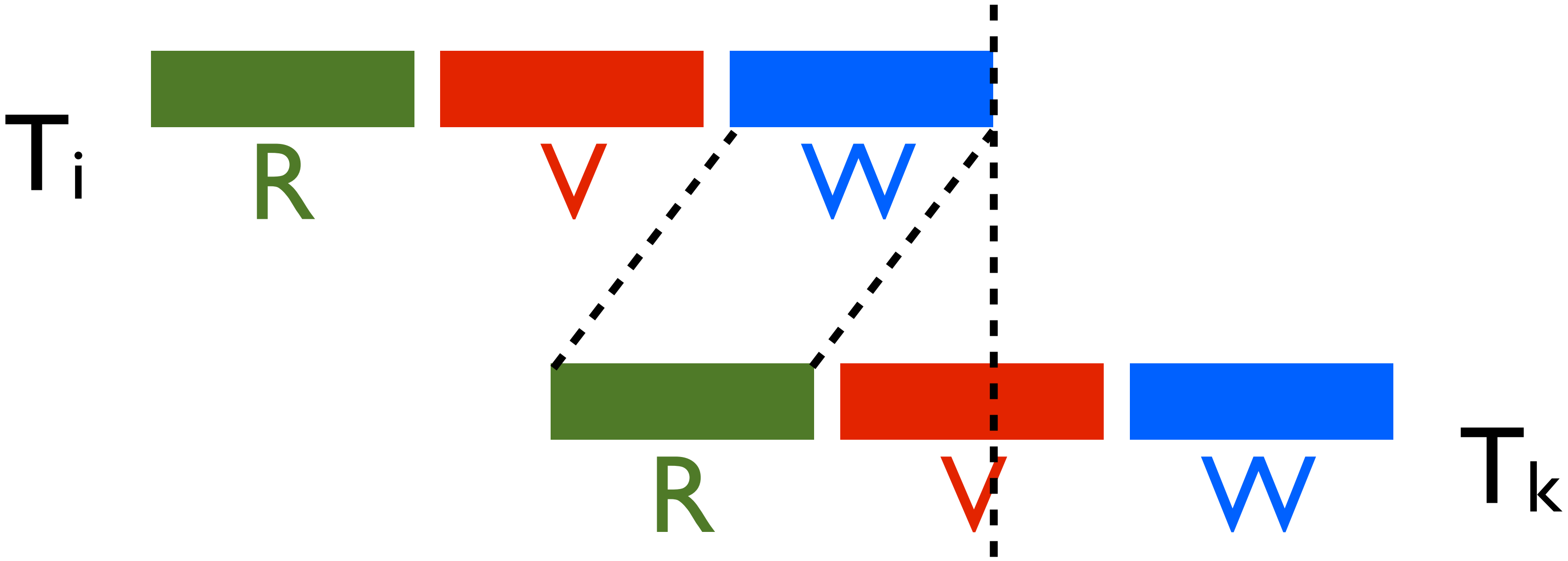


Is this sufficient?

Is this efficient?

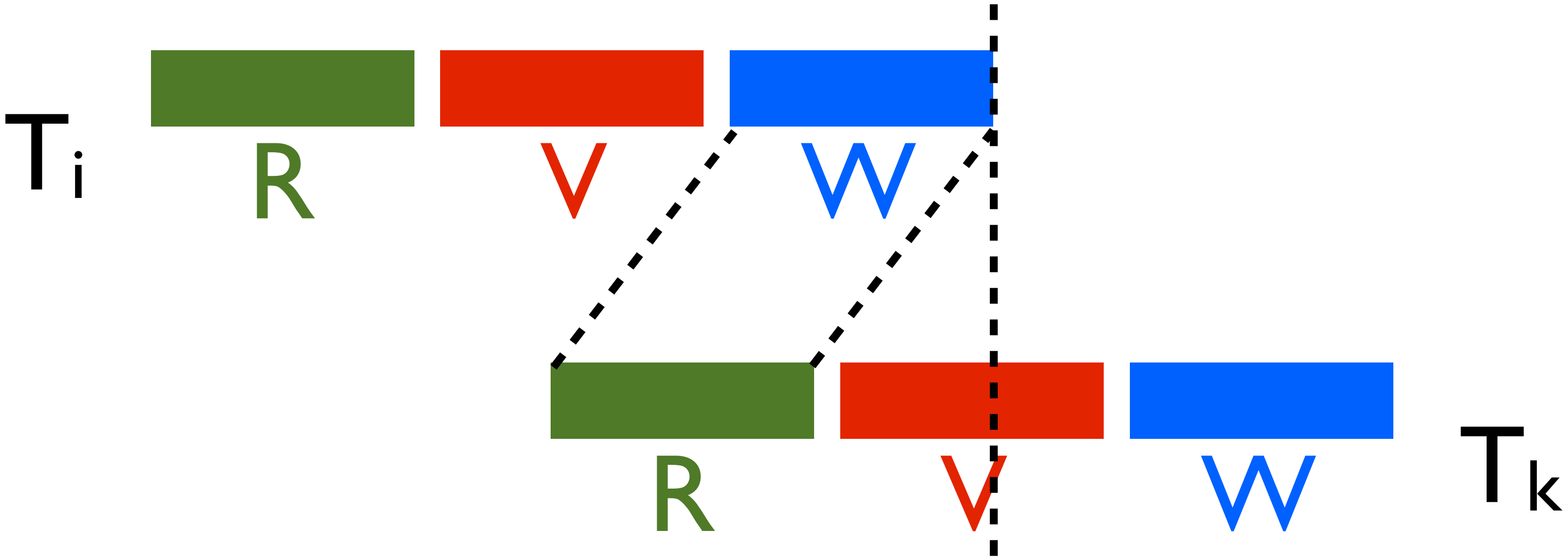
Test 2

For all i and k for which i < k,
check that Ti completes before Tk begins its write phase
AND WriteSet(Ti) ∩ ReadSet(Tk) is empty



Test 2

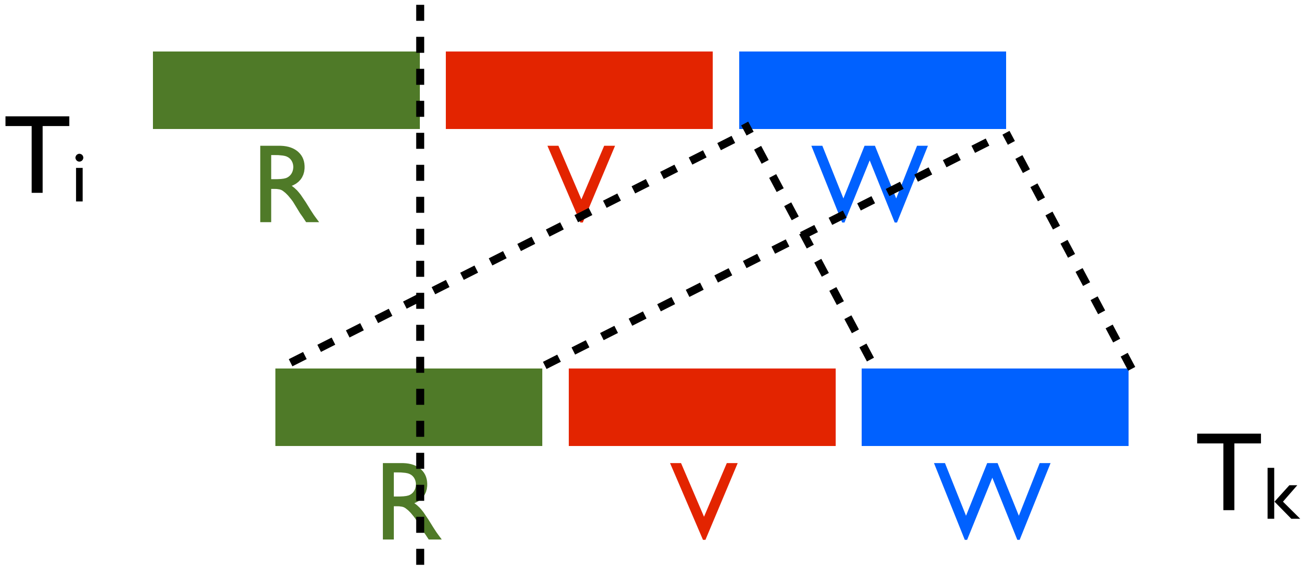
For all i and k for which i < k,
check that Ti completes before Tk begins its write phase
AND WriteSet(Ti) ∩ ReadSet(Tk) is empty



How do these two conditions help?

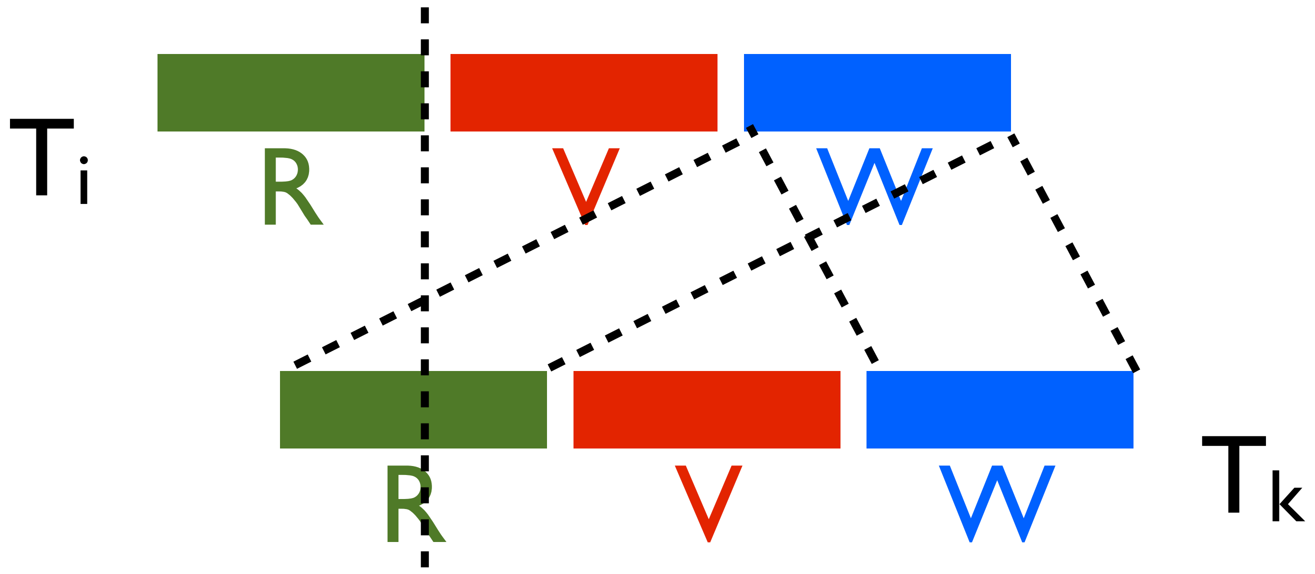
Test 3

For all i and k for which i < k,
check that Ti completes its read phase first
AND WriteSet(Ti) ∩ ReadSet(Tk) is empty
AND WriteSet(Ti) ∩ WriteSet(Tk) is empty



Test 3

For all i and k for which i < k,
check that Ti completes its read phase first
AND WriteSet(Ti) ∩ ReadSet(Tk) is empty
AND WriteSet(Ti) ∩ WriteSet(Tk) is empty



How do these three conditions help?

Which test (or tests) should we use?

Hint: How would you implement each test?

Validation

- Assigning the transaction ID, validation, and the parts of the write phase are a <u>critical section</u>.
  - Nothing else can go on concurrently.
  - The write phase can be long; This is bad.
- **Optimization**: Read-only transactions that don't need a critical section (no write phase).

Optimistic CC Overheads

- Each operation must be recorded in the readset/writeset (sets are expensive to allocate/destroy)
- Must test for conflicts during validation stage
- Must make validated writes “public”.
    - Critical section reduces concurrency.
    - Can lead to reduced object clustering.
- Optimistic CC must **restart** failed transactions.

Timestamp CC

- Give each object a read timestamp (RTS) and a write timestamp (WTS)
- Give each transaction a timestamp (TS) at the start.
- Use RTS/WTS to track previous operations on the object.
    - Compare with TS to ensure ordering is preserved.

Timestamp CC

- When T_i reads from object O:
  - If WTS(O) > TS(T_i), T_i is reading from a ‘later’ version.
    - Abort Ti and restart with a new timestamp.
  - If WTS(O) < TS(T_i), T_i’s read is safe.
    - Set RTS(O) to MAX(RTS(O), TS(T_i))

Timestamp CC

- When T_i writes to object O:
  - If RTS(O) > TS(T_i), T_i would cause a dirty read.
    - Abort T_i and restart it.
  - If WTS(O) > TS(T_i), T_i would overwrite a ‘later’ value.
    - Don’t need to restart, just ignore the write.
  - Otherwise, allow the write and update WTS(O).

Problem: Recoverability

Time	T1	T2
	`W(A)`	
		`R(A)`
		`W(B)`
		`COMMIT`

Problem: Recoverability

Time	T1	T2
	`W(A)`	
		`R(A)`
		`W(B)`
		`COMMIT`

What happens if T1 aborts (or the system crashes)?

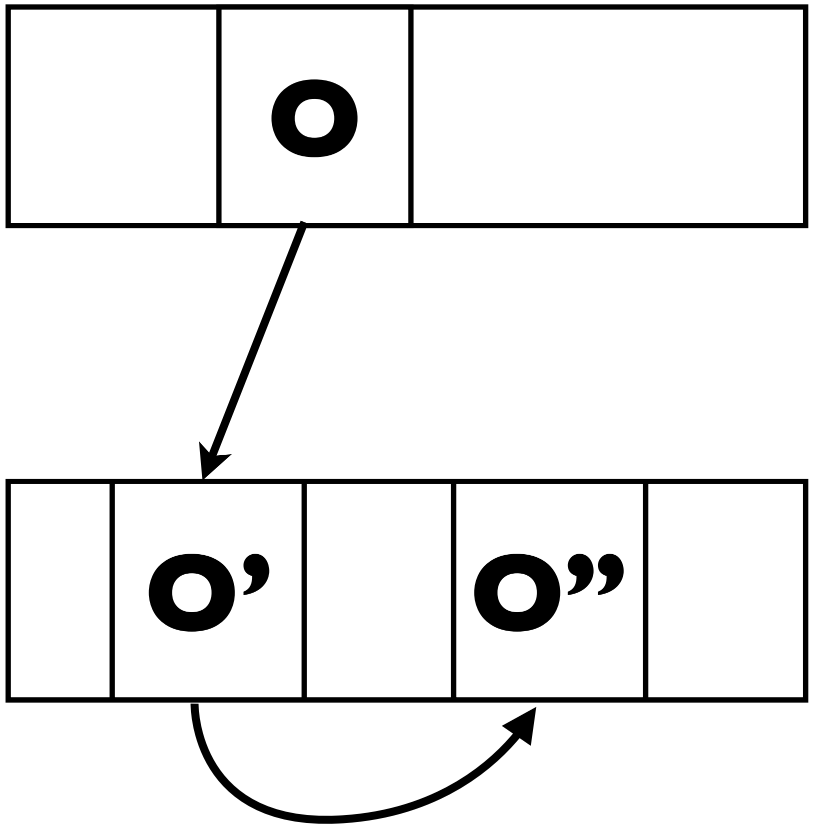
Timestamp CC and Recoverability

- Buffer all writes until a writer commits.
    - But update WTS(O) when the write to O is **allowed**.
- Block readers of O until the last writer of O commits.
- Similar to writers holding X locks until commit, but not quite 2PL.

Can we avoid read after write conflicts?

Multiversion TS CC

- Let writers make a “new” copy, while readers use an appropriate “old” copy.
- Readers are **always** allowed to proceed.
- … but may need to be blocked until a writer commits.



Main Segment

(current version of DB)

Version Pool

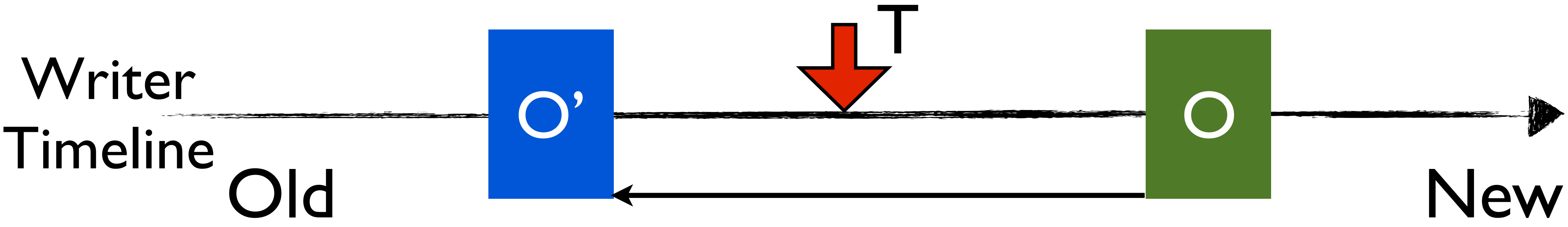
(older versions that can still be useful)

Multiversion TS CC

- Each version of an object has:
  - The writing transaction's TS as its WTS.
  - The highest transaction TS that read it as its RTS.
- Versions are chained backwards in a linked list.
  - We can discard versions that are too old to be "of interest".
- Each transaction classifies itself as a reader or writer for each object that it interacts with.

Reader Transactions

- Find the newest version with WTS < TS(T)
  - Start with the latest, and chain backward.
- Assuming that some version exists for all TS, reader xacts are never restarted!
  - … but may block until the writer commits.



Writer Transactions

- Find the newest version V s.t. WTS < TS(T)
- If RTS(V) < TS(T) make a copy of V with a pointer to V with WTS = RTS = TS(T).
  - The write is buffered until commit, but other transactions can see TS values.
- Otherwise reject the write (and restart)

Logging

- Problem 1: Supporting UNDO
  - How do we recover to an earlier state?
- Problem 2: Mitigating Failures
  - How do we restore un-persisted changes?
- Problem 3: Replication & Distribution
  - How do we synchronize multiple DB instances?