Distributed Systems 1

CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

Instructor: Roxana Geambasu

Spanner

Context

- We learned about several key mechanisms and protocols for achieving scalability and fault tolerance in a strong-semantic, transactional database.
  - Two-phase locking – what's this for?
  - Write-ahead logging – what's this for?
  - Two-phase commit – what's this for?
  - Paxos – what's this for?

Context

- We learned about several key mechanisms and protocols for achieving scalability and fault tolerance in a strong-semantic, transactional database.
  - Two-phase locking – for isolation
  - Write-ahead logging – for atomicity in single-node DB
  - Two-phase commit – for atomicity in sharded DB for scalability
  - Paxos – for consistent replication for fault tolerance
- Today we look at the design and implementation of Spanner, Google's scalable and fault-tolerant ACID database, which **combines all these building blocks** (and more).

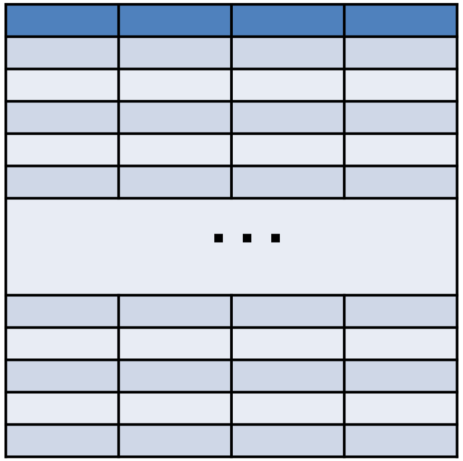
Outline

- Overview and architecture
- TrueTime
- Using TrueTime for efficient linearizable transactions
- Then, YOU will read/view Spanner paper/talk https://www.usenix.org/node/170855.

Overview

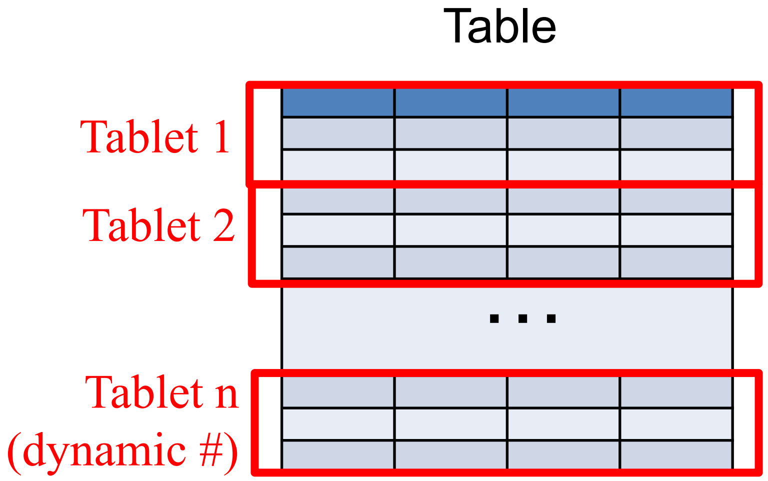
- Relational API
- ACID transactions
- Geographically replicated
- Sharded by rows

Table

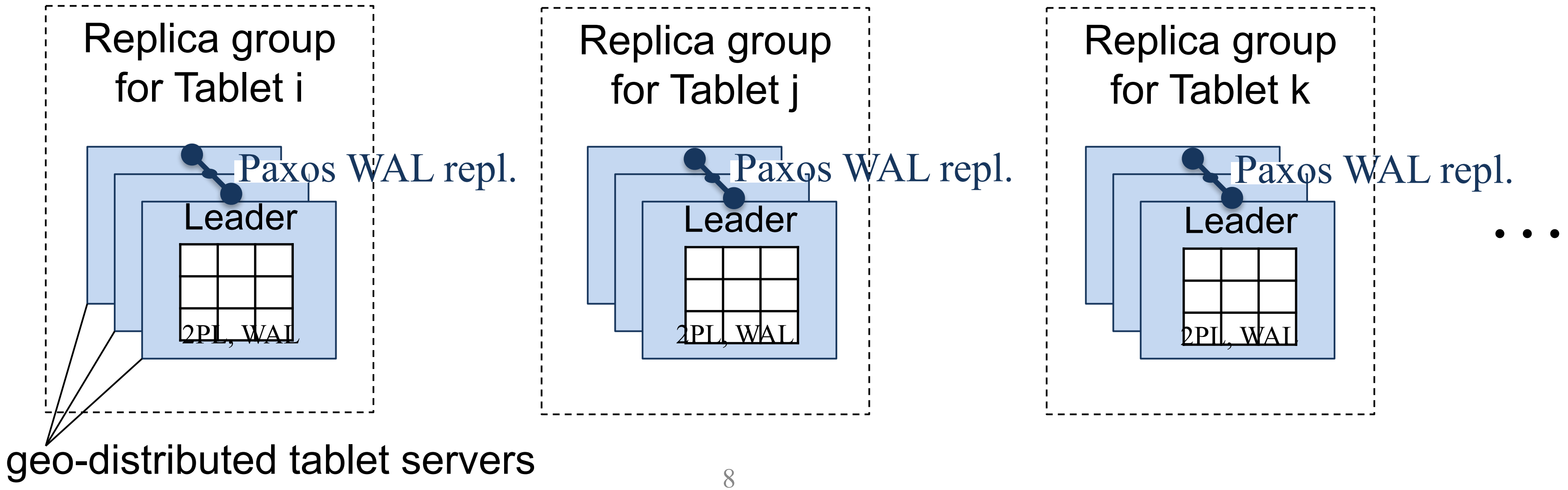


Overview

- Relational API
- ACID transactions
- Geographically replicated
- Sharded by rows

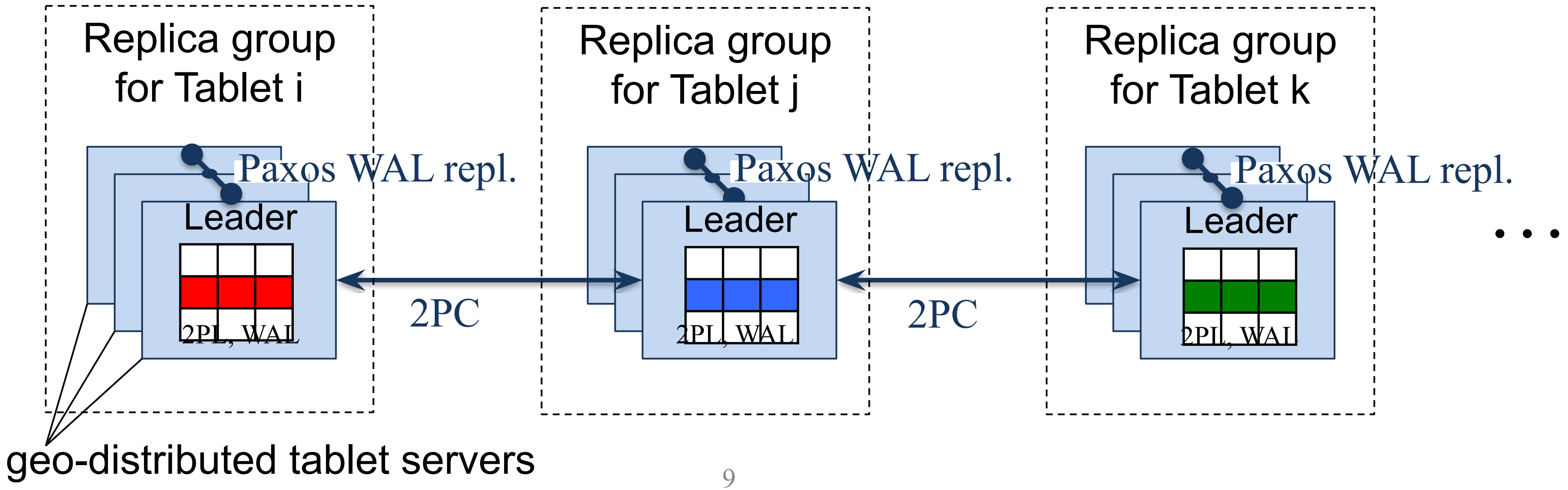


Architecture


Replica group for Tablet i
Paxos WAL repl.
Leader
2PL, WAL
Replica group for Tablet j
Paxos WAL repl.
Leader
2PL, WAL
Replica group for Tablet k
Paxos WAL repl.
Leader
2PL, WAL
. . .
geo-distributed tablet servers

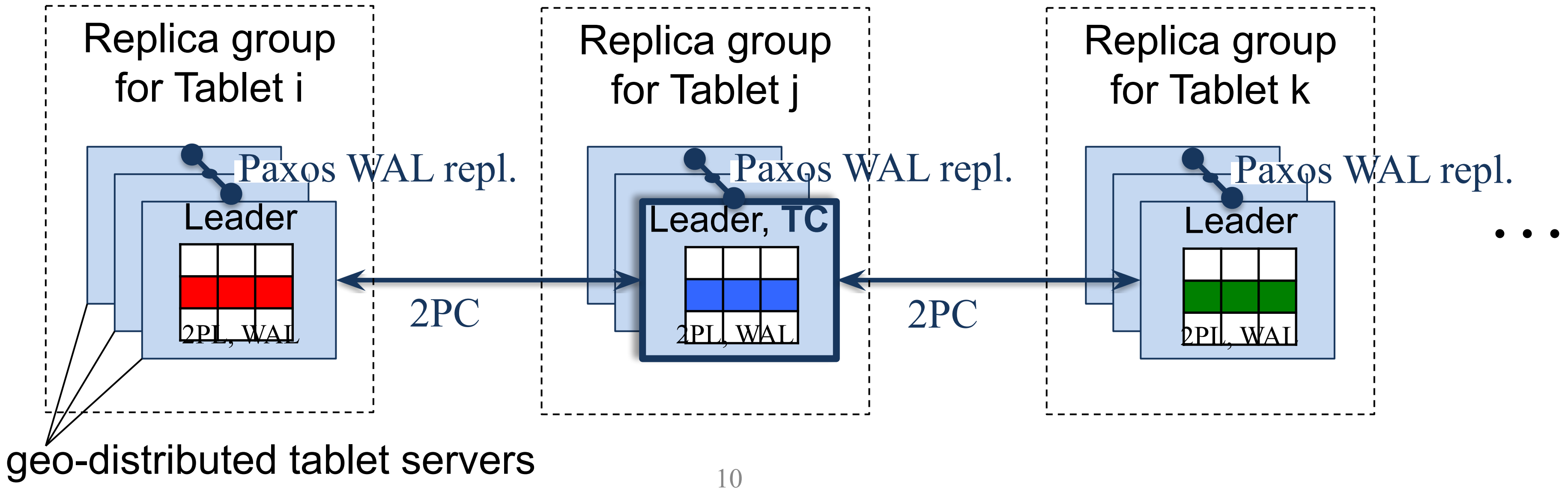
Architecture

Transaction:
Begin(); update row1; update row2; update row3; Commit()


Replica group for Tablet i
Replica group for Tablet j
Replica group for Tablet k
Paxos WAL repl.
Paxos WAL repl.
Paxos WAL repl.
Leader
Leader
Leader
2PL, WAL
2PL, WAL
2PL, WAL
2PC
2PC
. . .
geo-distributed tablet servers

Architecture

Transaction:
Begin(); update row1; update row2; update row3; Commit()


Replica group for Tablet i
Replica group for Tablet j
Replica group for Tablet k
Paxos WAL repl.
Paxos WAL repl.
Paxos WAL repl.
Leader
Leader, TC
Leader
2PL, WAL
2PL, WAL
2PL, WAL
2PC
2PC
. . .
geo-distributed tablet servers

Properties

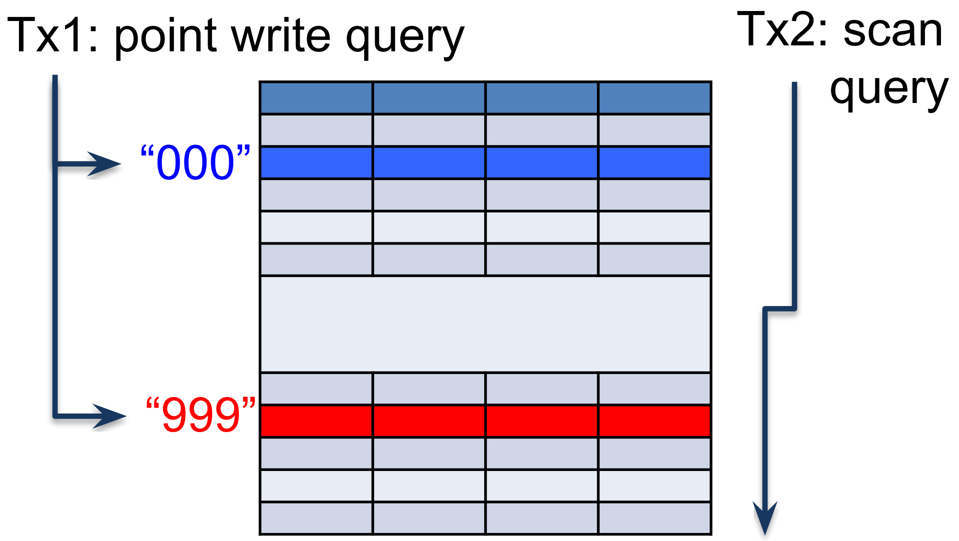
- Semantics?
- Performance?

Semantics/Performance

- Spanner provides **atomic**, **serializable** transactions that can be performed at scale and with geographic fault tolerance.
- Performance: 2PC, especially across geographically replicated groups, is expensive!
- "[S]ome authors have claimed that two-phase commit is too expensive to support, because of the performance or availability problems that it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions." -- Spanner authors

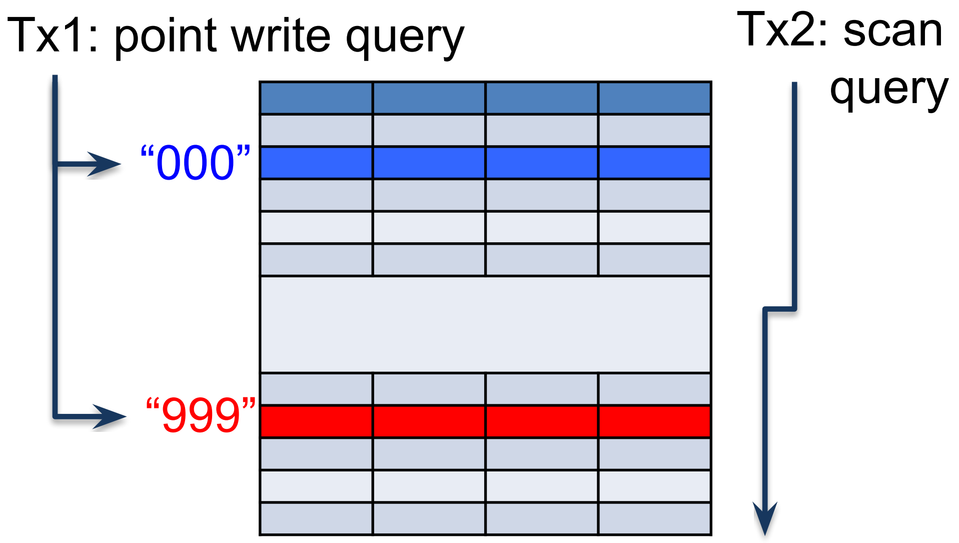
More on Performance

- Consider what happens when you do 2PL + 2PC + Paxos WAL replication across geographies.
  - What locks must be taken to give illusion of serial execution?
  - How long do they need to be retained?
  - How does that impact performance esp. if Tx 2PC on Paxos?



More on Performance

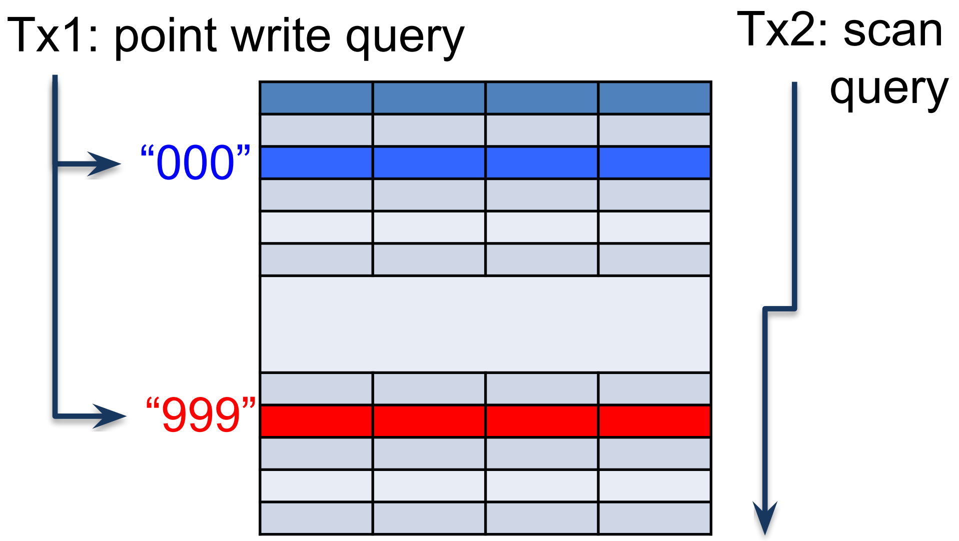
- Need to take r/w locks for all accessed (read/written) rows.
- Locks must be held until the end of the transactions.
- Scan queries, which interact with lots of tablet servers, block other Tx's from executing.



More on Performance

(continued)

- Add to that that 2PC may be involved in some expensive transactions.
- Plus all the cross-geo Paxos to make sure any updates are replicated.
- This interaction **across transactions** is what's expensive!



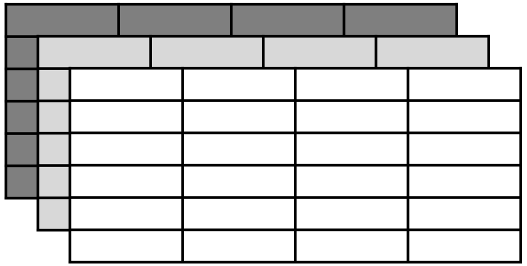
What Can We Do?

- Option 1: Take fewer locks or hold them for less time. This would weaken the semantic (won't be equivalent to serial execution).
  - But "[w]e believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions." – Spanner authors

- Option 2: Distinguish two types of queries -- (1) **point r/w queries** and (2) **r/o scans** -- and treat them differently:
  - Do 2PL for point queries.
  - Do *lockless concurrency control* for r/o scans.
  - Preserves semantics and reduces performance interference.

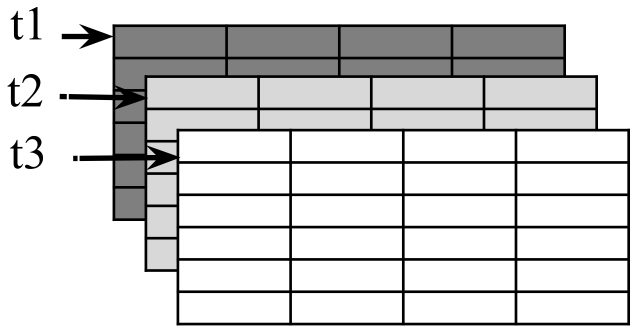
Lockless Concurrency Control

- **Multi-versioned concurrency control (MVCC)** is a popular lockless concurrency control mechanism, which is used in Spanner.
- Each row in the table is versioned. Any r/w transaction creates a new version of the rows it changes.



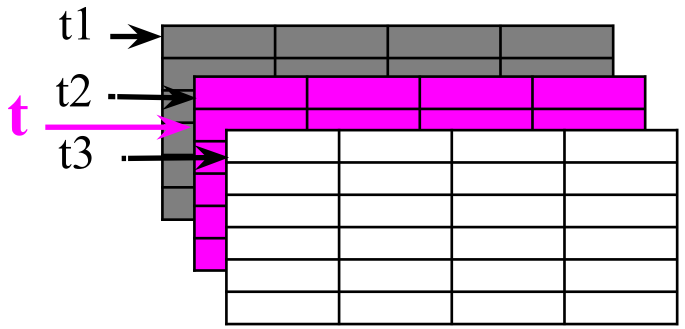
Lockless Concurrency Control

- **Multi-versioned concurrency control (MVCC)** is a popular lockless concurrency control mechanism, which is used in Spanner.
- Each row in the table is versioned. Any r/w transaction creates a new version of the rows it changes.
- **IF** you had a **global notion of time**, then you could timestamp the table version.



Lockless Concurrency Control

- **Multi-versioned concurrency control (MVCC)** is a popular lockless concurrency control mechanism, which is used in Spanner.
- Each row in the table is versioned. Any r/w transaction creates a new version of the rows it changes.
- **IF** you had a **global notion of time**, then you could timestamp the table version.



- Then, when doing a **r/o scan**, you could:
  - Get a timestamp for that transaction: **t**.
  - Read the values of each row you're interested in **at the time t**.
  - **IF** all nodes agreed on the notion of time, you get **serializability**.

What Global Time?

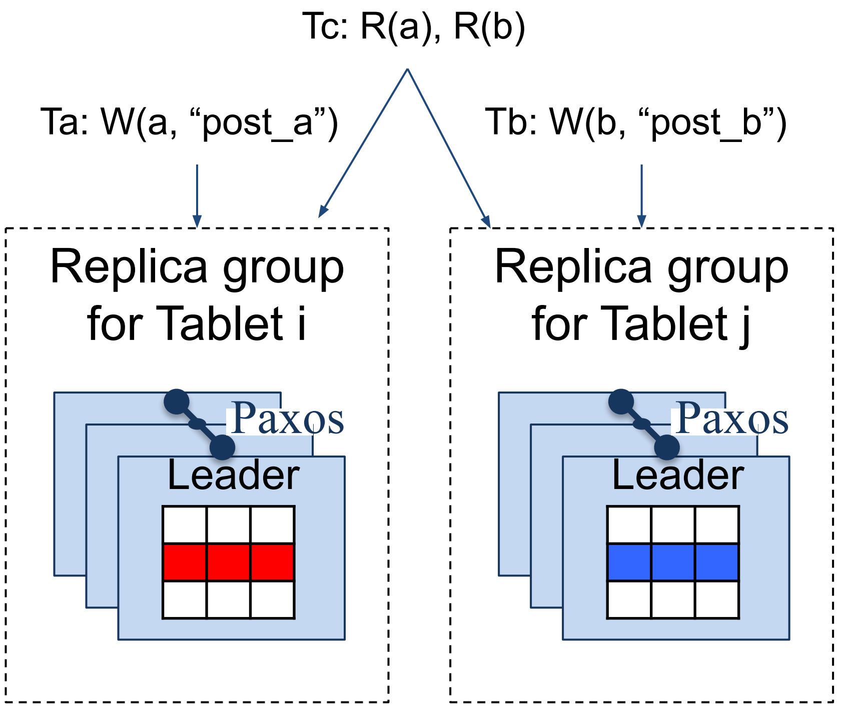
- Option 1: **logical clocks**. There are MVCC protocols based on that and they would give you serializability.
- But logical clocks have some limitations, including that they **only capture internal causality, not external.**
- What problems could arise?

Example

- A writes a post (Ta).
- A calls B to tell them.
- B writes a post with their opinion about A’s post (Tb).
- Friend C does a r/o scan of all posts from their friends (Tc).
- C should not see B’s post without seeing A’s, because Ta committed before Tb started, and hence Tb could have been influenced by Ta (directly or indirectly, internally or externally).

Example

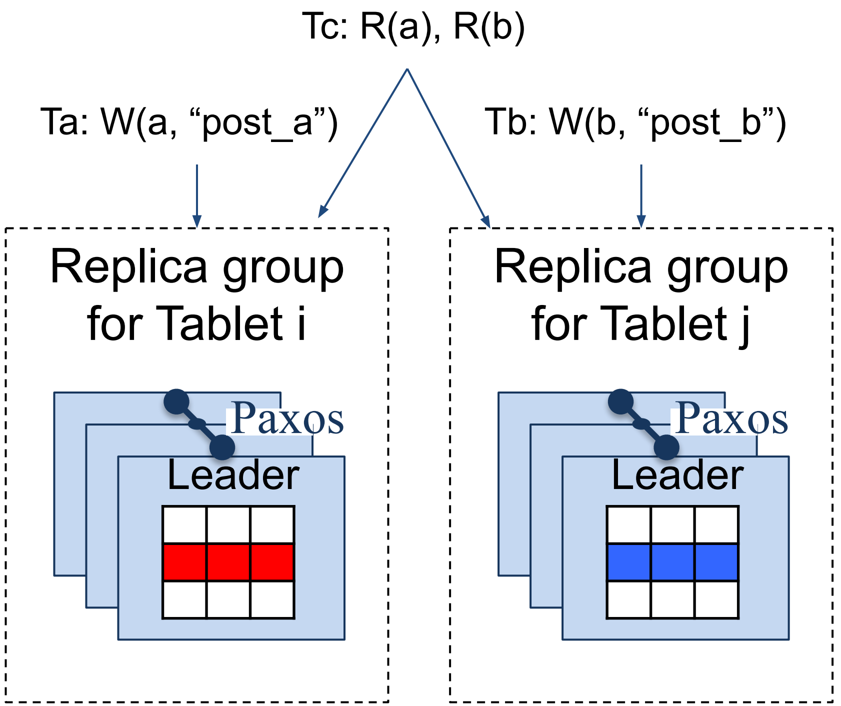
- A writes a post (Ta).
- A calls B to tell them.
- B writes a post with their opinion about A's post (Tb).
- Friend C does a r/o scan of all posts from their friends (Tc).
- C should not see B's post without seeing A's, because Ta committed before Tb started, and hence Tb could have been influenced by Ta (directly or indirectly, internally or externally).



Example

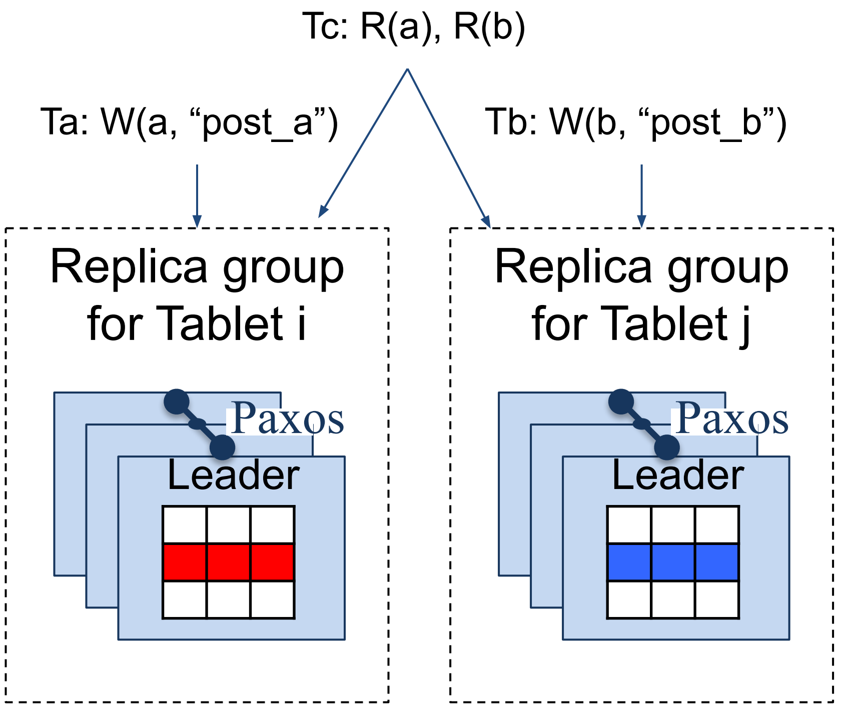
- I.e., the only possibilities for Tc are to read a, b must be:
  - ”post_a”, “post_b”
  - “pre_post_a”, “pre_post_b”
  - “post_a”, “pre_post_b”

- NOT:
  - “pre_post_a”, “post_b”



Example

- I.e., the only possibilities for Tc are to read a, b must be:
  - ”post_a”, “post_b”
  - “pre_post_a”, “pre_post_b”
  - “post_a”, “pre_post_b”
- NOT:
  - “pre_post_a”, “post_b”



That is what Spanner ensures, and the property is called linearizability (stronger than serializability).

How about Physical Clocks?

- Option 2: make physical time work!
  - Google chose this option, with **TrueTime**, their strong-semantic time service.
  - With this notion of time, you can actually get **linearizable consistency and serializable isolation**.

TrueTime

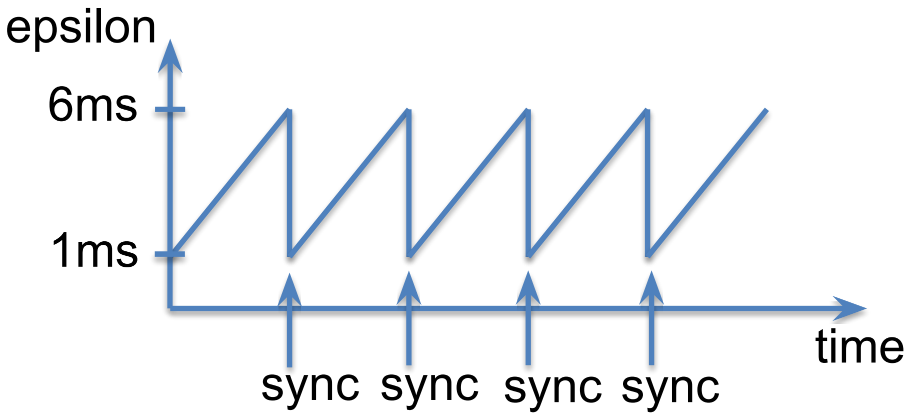
- Highly available, distributed clock that is provided to applications on all Google servers.
- Enables applications to generate monotonically increasing timestamps **across all servers**.
- An application can compute a timestamp T that is guaranteed to be greater than any timestamp T' if T' finished being generated before T started being generated. This guarantee holds even if **T, T' are computed on different machines**.

TrueTime Architecture

- In each data center, they have a set of expensive, but very accurate, atomic clocks. They use these as *reference clocks*.
- On each machine, they have a regular clock (e.g., quartz), which is inexpensive but inexact. The machine runs a *time daemon* that selects multiple reference clocks from multiple data centers and performs a synchronization protocol with them. The protocol bounds the error very precisely -- up to **a few milliseconds** in general (under reasonable assumptions).
- Between synchronizations, the time daemon will accumulate error: about 200usec/second is applied, which is >> than the clock drift on their quartz clocks.

TrueTime Interface

```
typedef struct TT_interval {
    struct timeval earliest;
    struct timeval latest;
} TT_interval;
TT_interval *TT_now(TT_interval *tt);
```

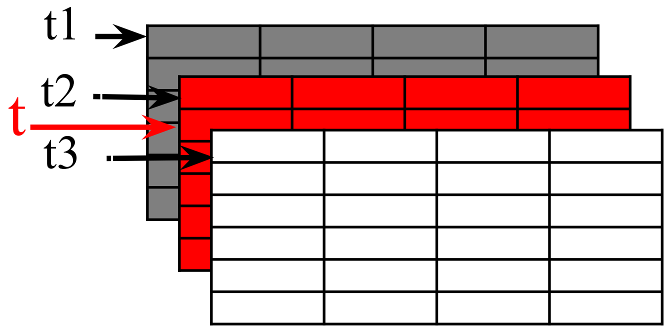


tt.latest – tt.earliest = epsilon

- Epsilon is called the *instantaneous uncertainty bound.*
- In practice, epsilon saw-tooths between 1ms and 6ms.

Serializable Transactions in Spanner

- R/W transactions are executed with 2PL (+2PC+Paxos) and fork a new version of each modified row (copy-on-write style).
- All rows modified by a R/W transaction are tagged with a TrueTime timestamp associated with that transaction.
- R/O scan transactions are executed without locking but with isolation by reading a consistent version of the rows at a TrueTime timestamp associated with the transaction.



- Question: How to associate a timestamp to r/w and r/o transactions to ensure serializability? **(i) What timestamp to select (ii) When to select it?**

Example

- RW T1: ts = 127, updates row1, row2, row3
- RW T2: ts = 129, updates row1, row3, row4
- RO T3: ts = ?

- Row1: 123, 127, 129
- Row2: 123, 127
- Row3: 123, 127, 129
- Row4: 109, 129

Example

- RW T1: ts = 127, updates row1, row2, row3
- RW T2: ts = 129, updates row1, row3, row4
- RO T3: ts = (say) 130

- Row1: 123, 127, 129 |
- Row2: 123, 127 |
- Row3: 123, 127, 129 |
- Row4: 109, 129 |

Example

- RW T1: ts = 127, updates row1, row2, row3
- RW T2: ts = 129, updates row1, row3, row4
- RO T3: ts = (say) 128

- Row1: 123, 127, | 129
- Row2: 123, 127 |
- Row3: 123, 127 |, 129
- Row4: 109 |, 129

Example

- RW T1: ts = 127, updates row1, row2, row3
- RW T2: ts = 129, updates row1, row3, row4
- RO T3: ts = (say) 125

- Row1: 123 | , 127, 129
- Row2: 123 |, 127
- Row3: 123 |, 127, 129
- Row4: 109 |, 129

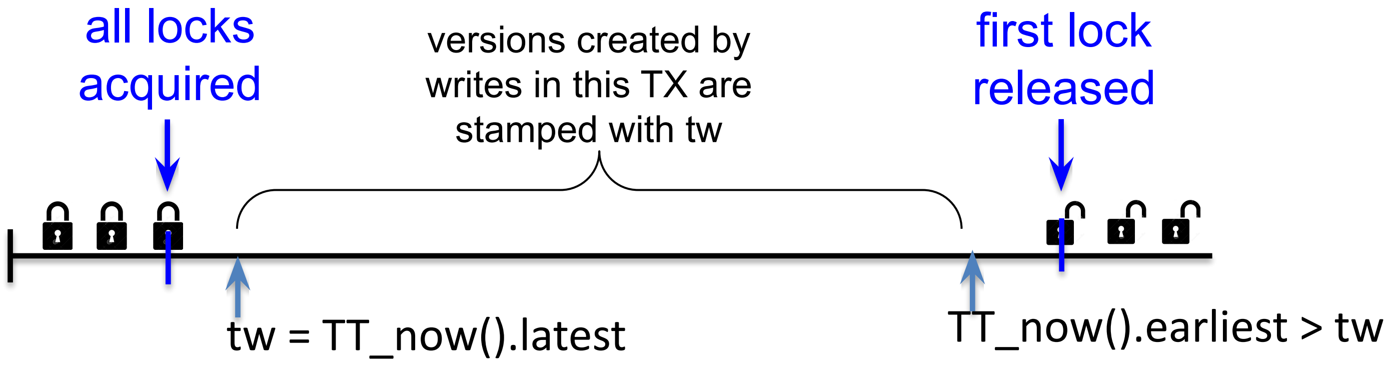
What Timestamp to Select?

- Serializability for r/w transactions is ensured through 2PL. The question is how to ensure it for r/o transactions, which are lockless.
- For a r/o transaction Tr to execute isolated from any ongoing r/w transaction Tw, then Tr's and Tw's timestamps must meet two conditions:
  1. If Tr reads any effect of Tw, then Tr's timestamp > Tw's timestamp.
  2. If Tr doesn't read some effect of Tw, then Tr's timestamp < Tw's timestamp.

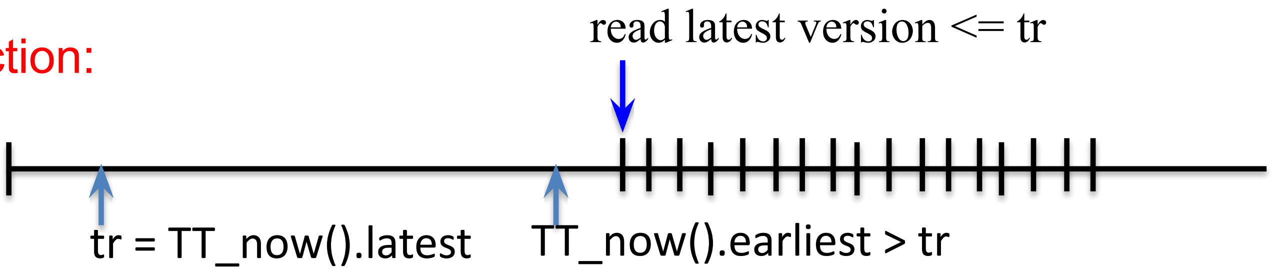
```
Key idea:
    ts = TT_now().latest;
    waitUntil(TT_now().earliest > ts);
    // Then perform transaction using
    // ts as its assigned timestamp.
```

When to Select a Timestamp?

R/W transaction:



R/O transaction:

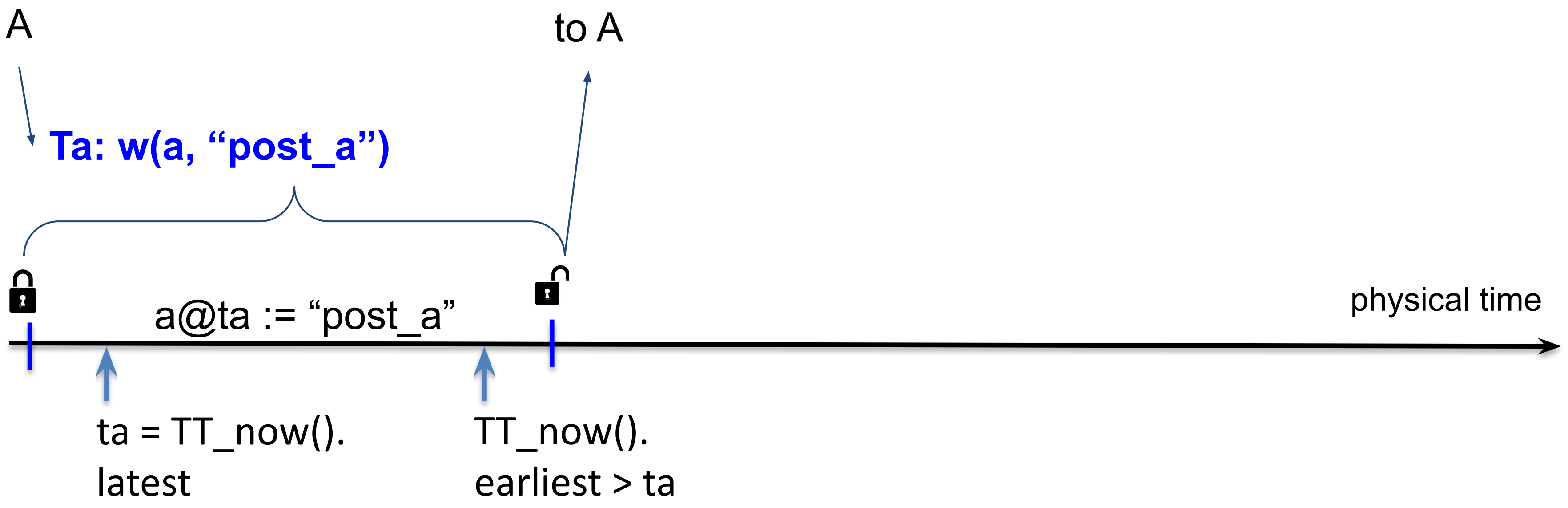


With Multiple Shard Servers

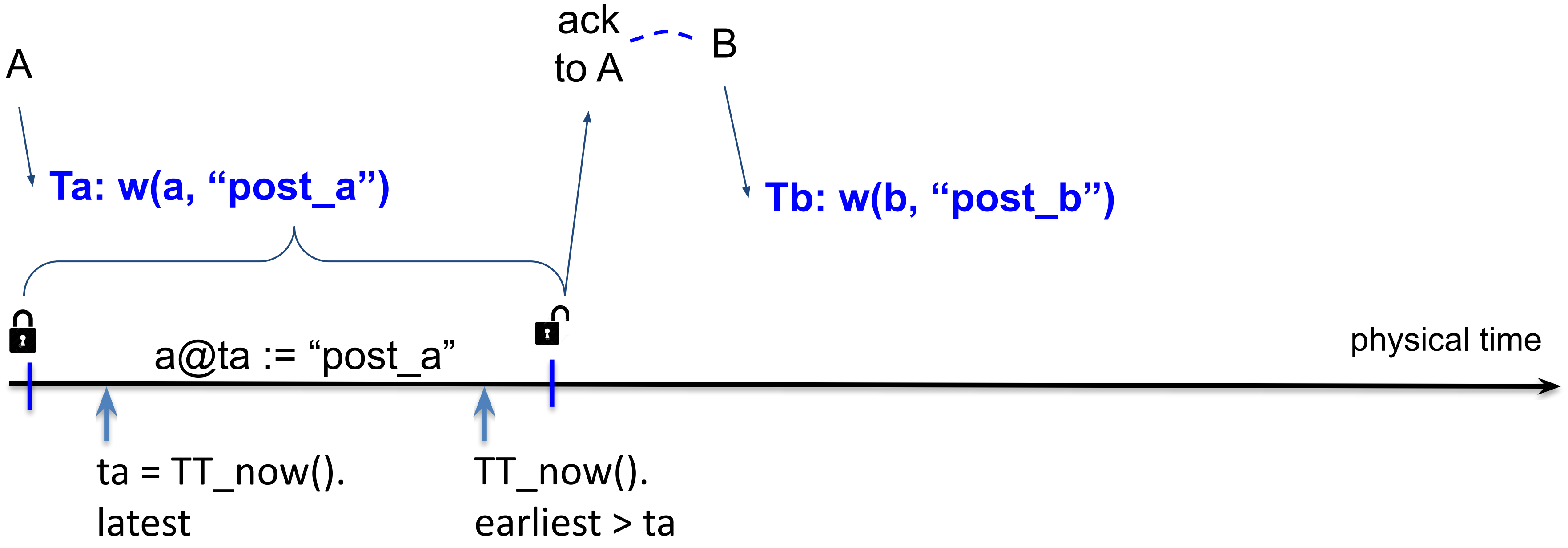
- R/O transactions:
  - Client selects the timestamp ts and shard servers may have to wait until their TT_now().earliest > ts before they give out a value.
- R/W transactions:
  - TC gathers timestamps from Prepare responses from participants (which are Paxos leaders, remember).
  - TC selects ts = max of all the gathered timestamps.
  - Each participant waits until TT_now.earliest > ts before releasing locks.

Execution in Our Post Example

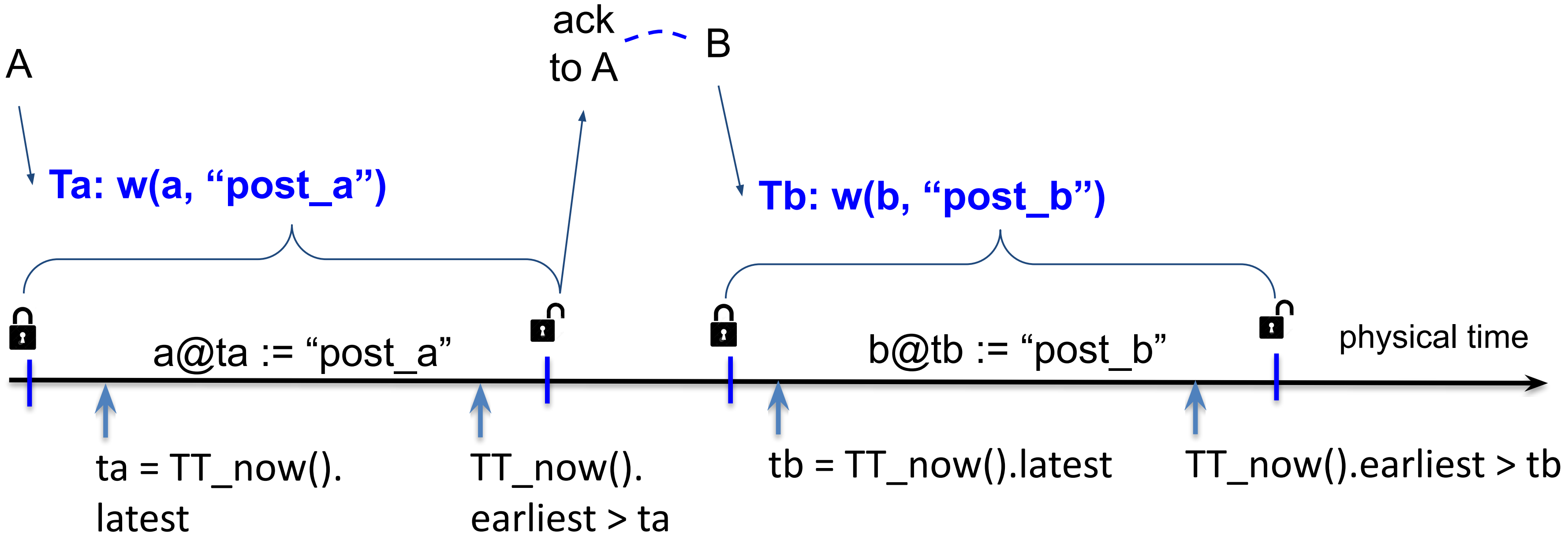
Execution in Our Post Example



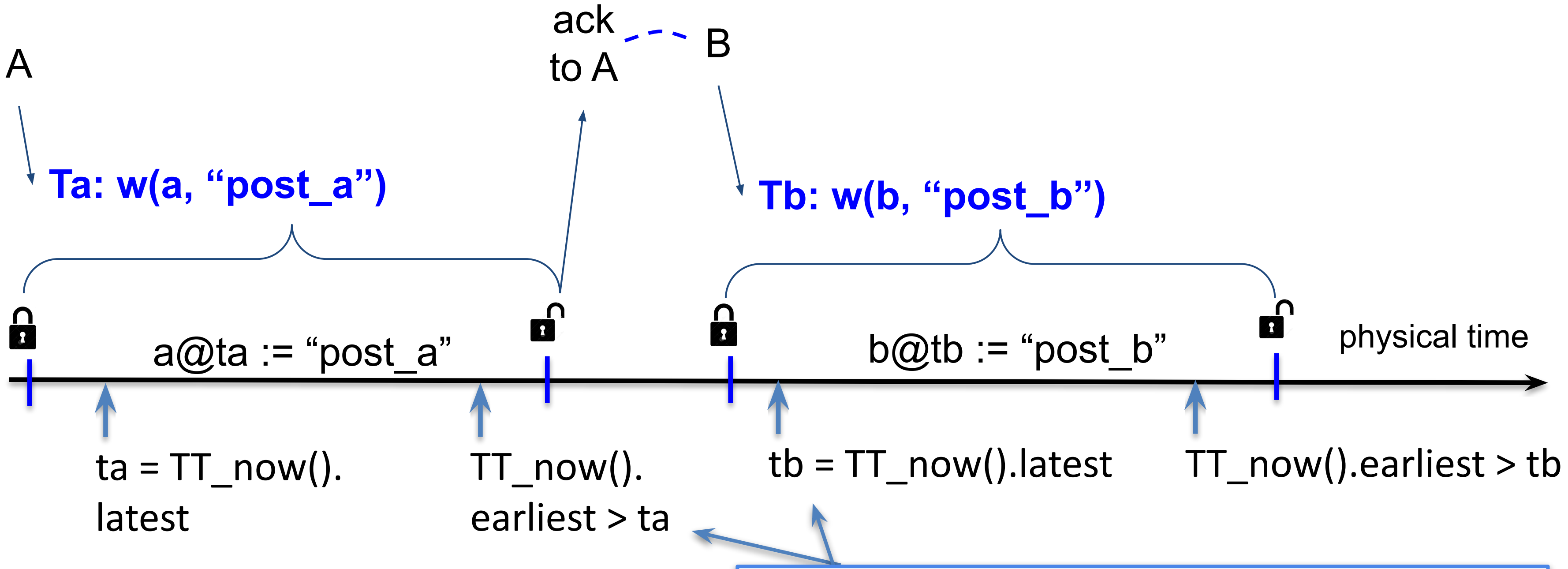
Execution in Our Post Example



Execution in Our Post Example

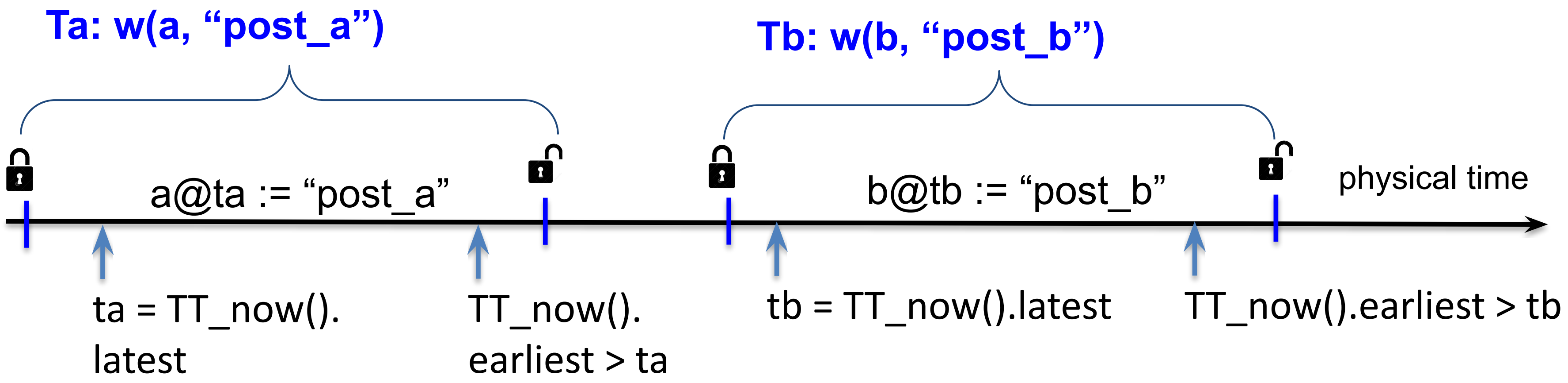


Execution in Our Post Example



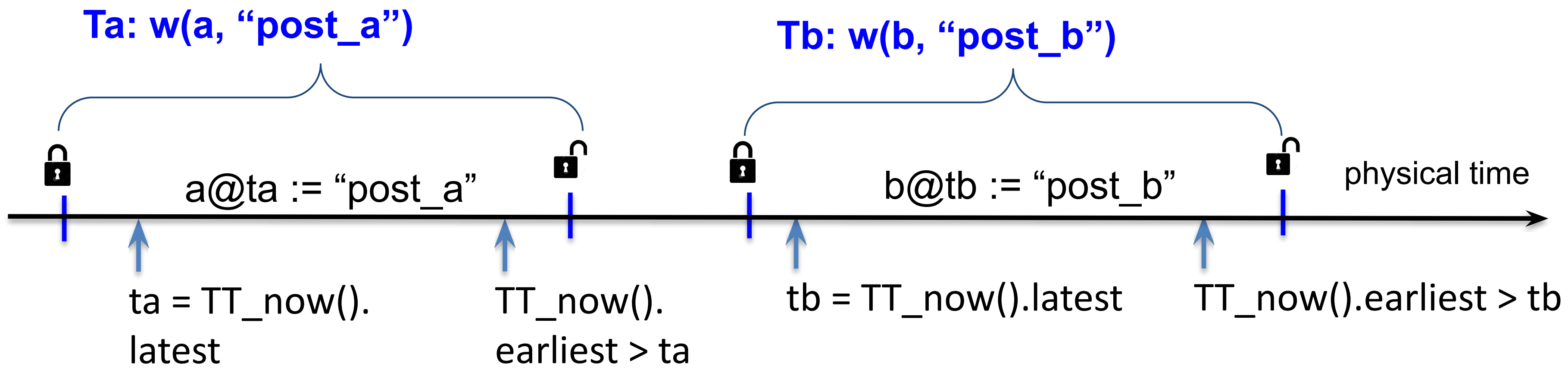
Because TT_now().earliest > ta (on all machines!), it is guaranteed that tb > ta.

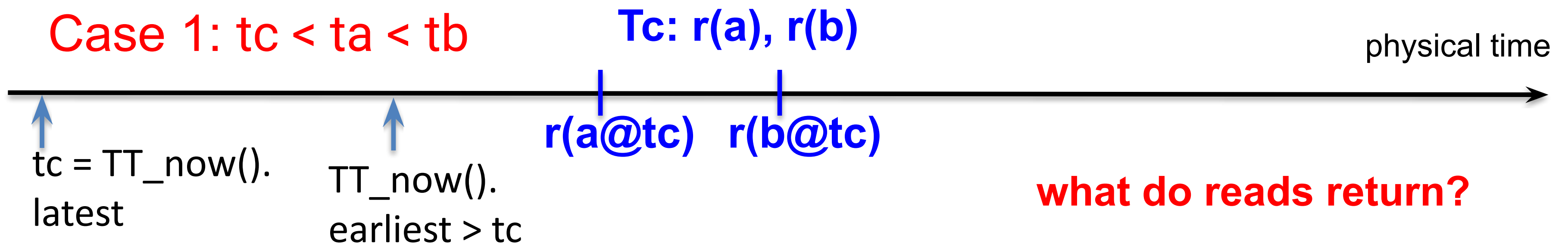
Execution in Our Post Example



Tc: r(a), r(b) ?

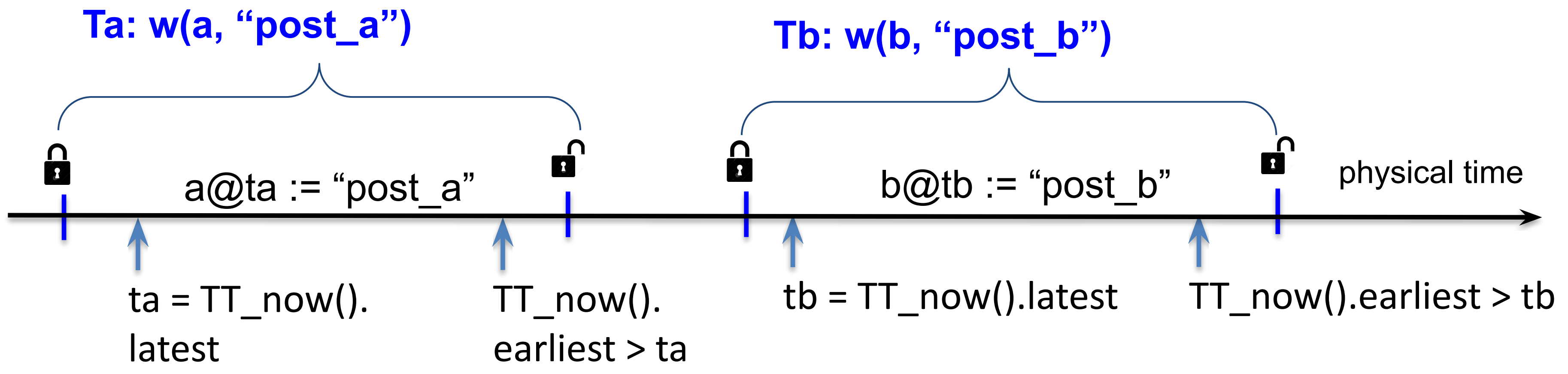
Execution in Our Post Example



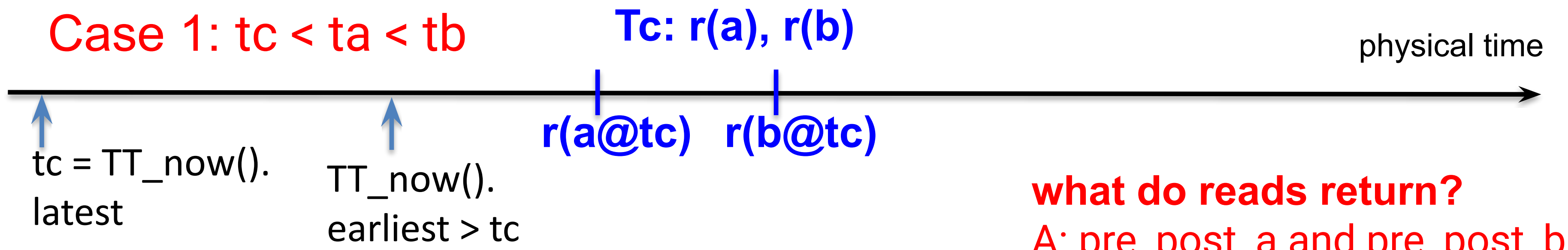


what do reads return?

Execution in Our Post Example



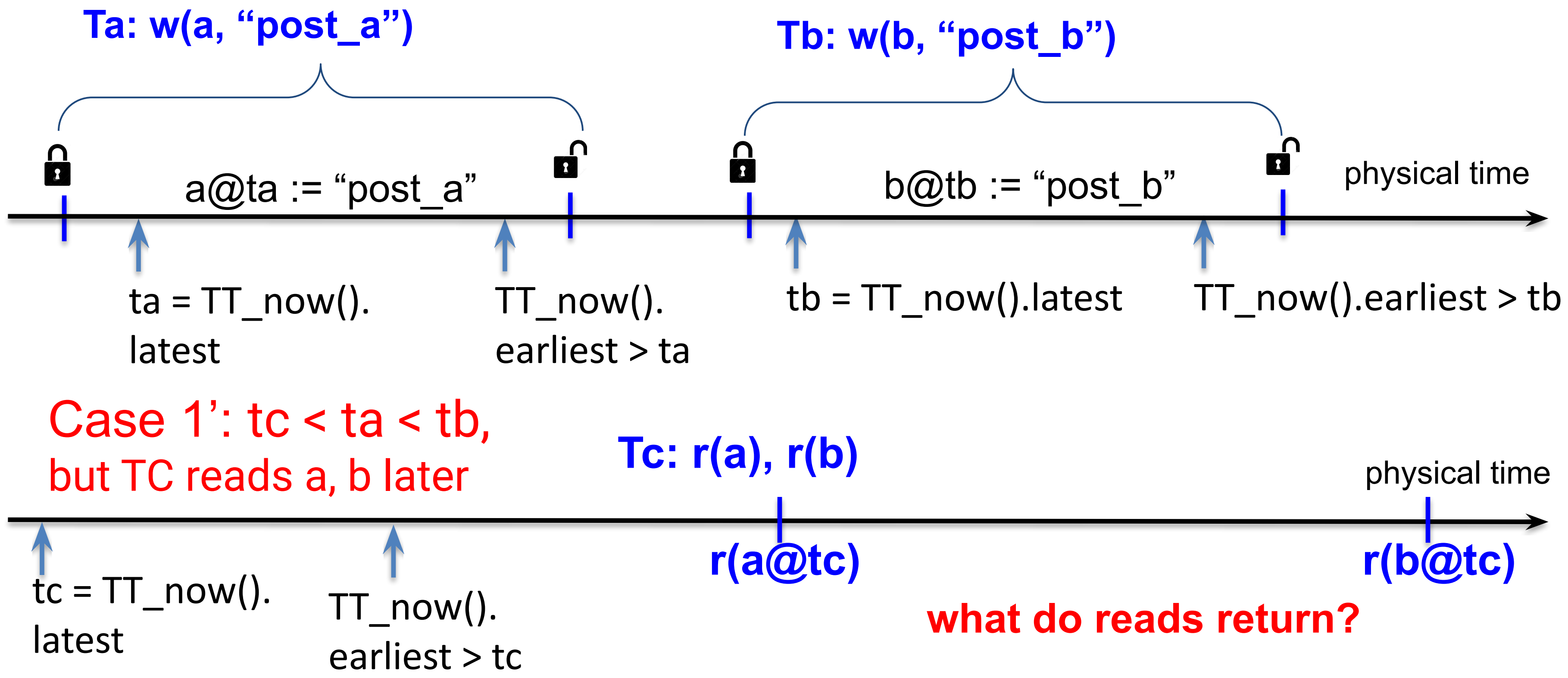
Case 1: tc < ta < tb



what do reads return?

A: pre_post_a and pre_post_b

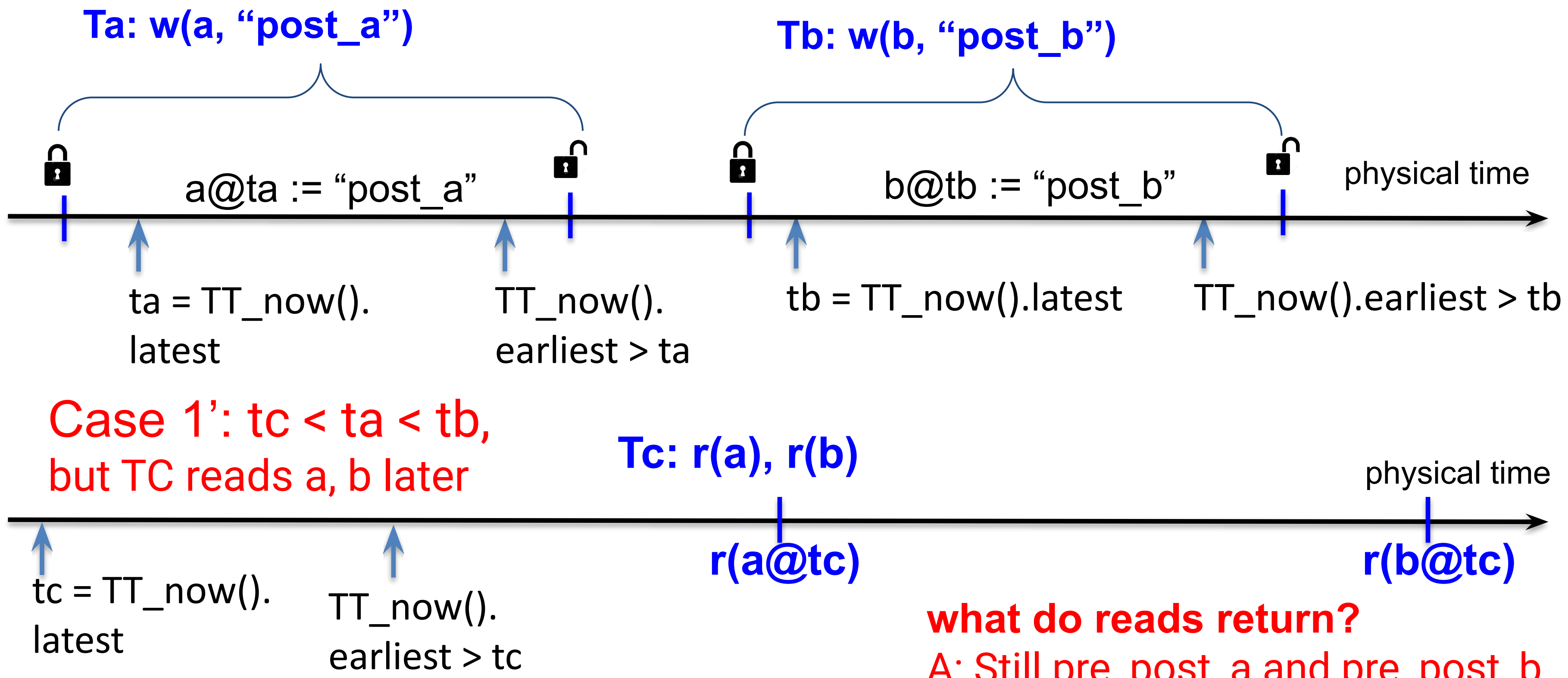
Execution in Our Post Example



Case 1': tc < ta < tb,
but TC reads a, b later

what do reads return?

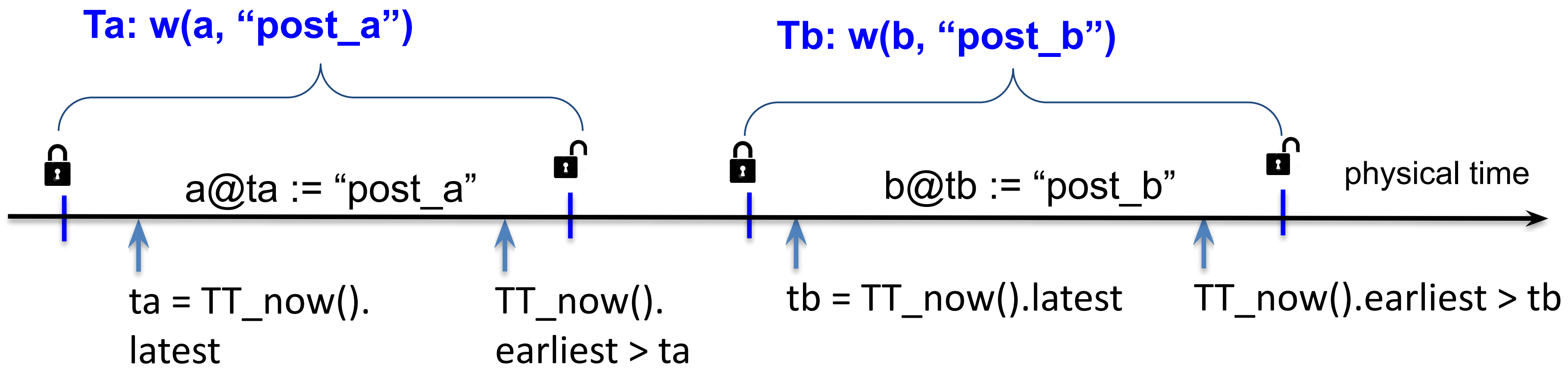
Execution in Our Post Example



Case 1’: tc < ta < tb,
but TC reads a, b later

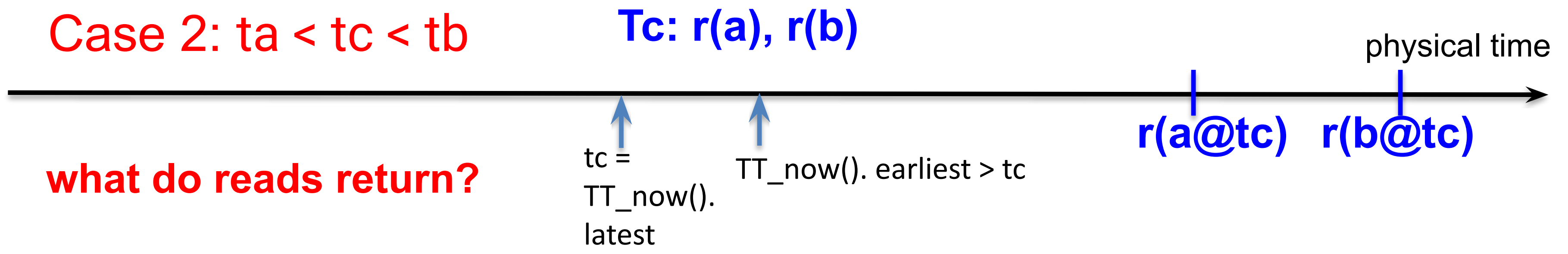
what do reads return?
A: Still pre_post_a and pre_post_b

Execution in Our Post Example

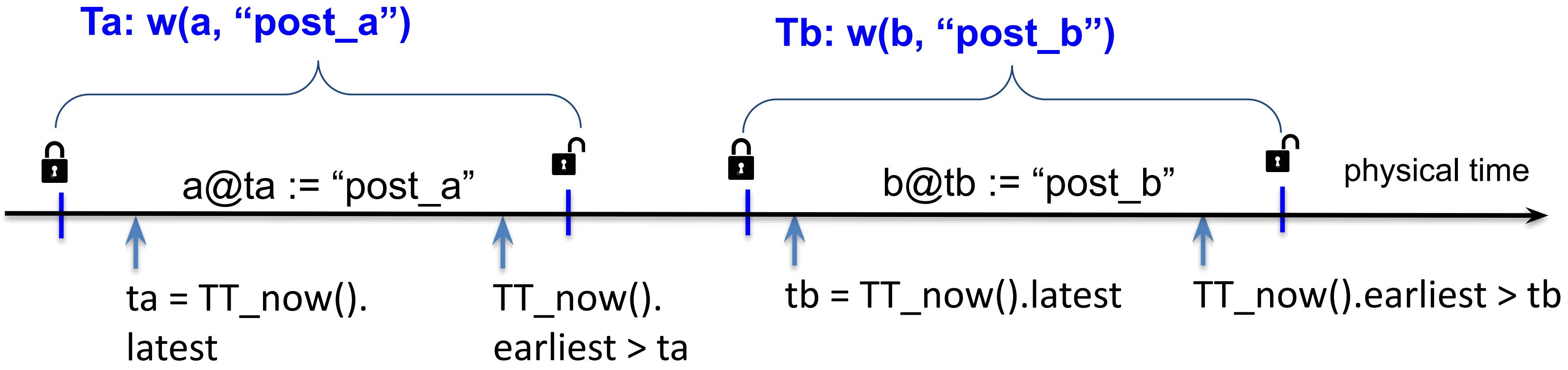


Case 2: ta < tc < tb

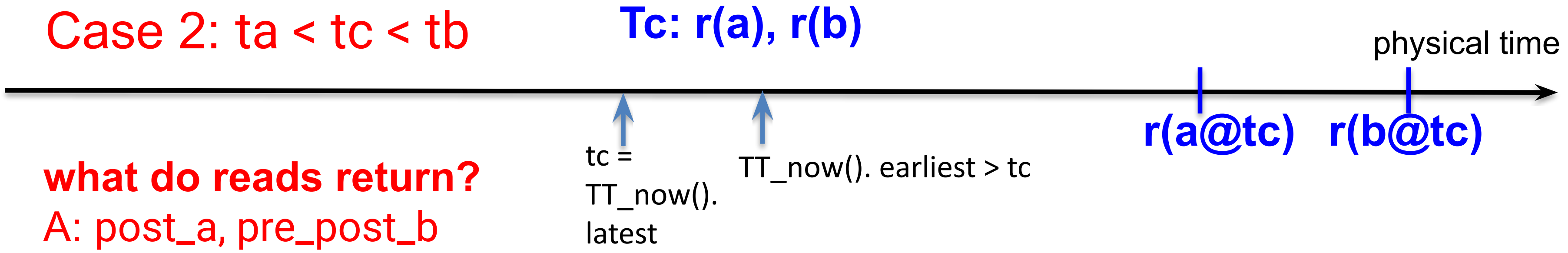
what do reads return?



Execution in Our Post Example



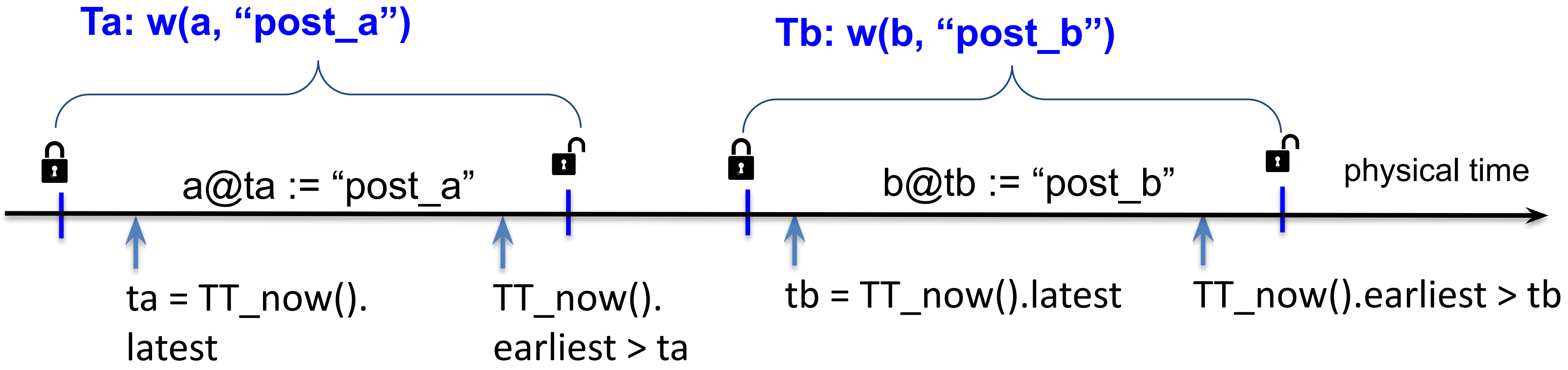
Case 2: ta < tc < tb



what do reads return?

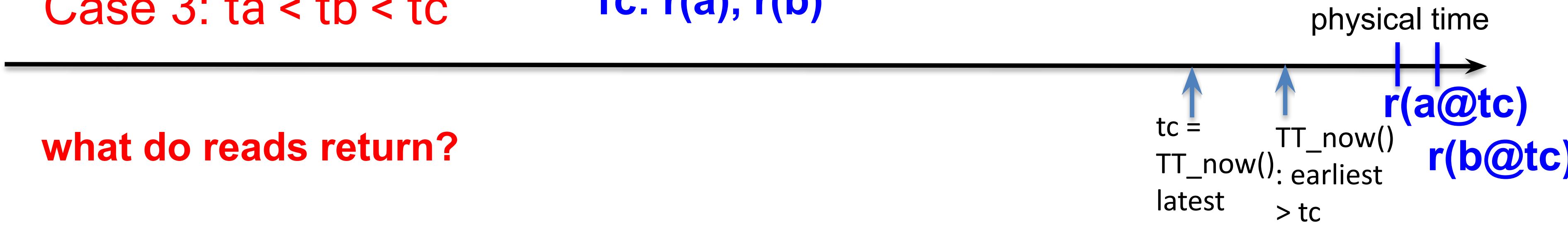
A: post_a, pre_post_b

Execution in Our Post Example



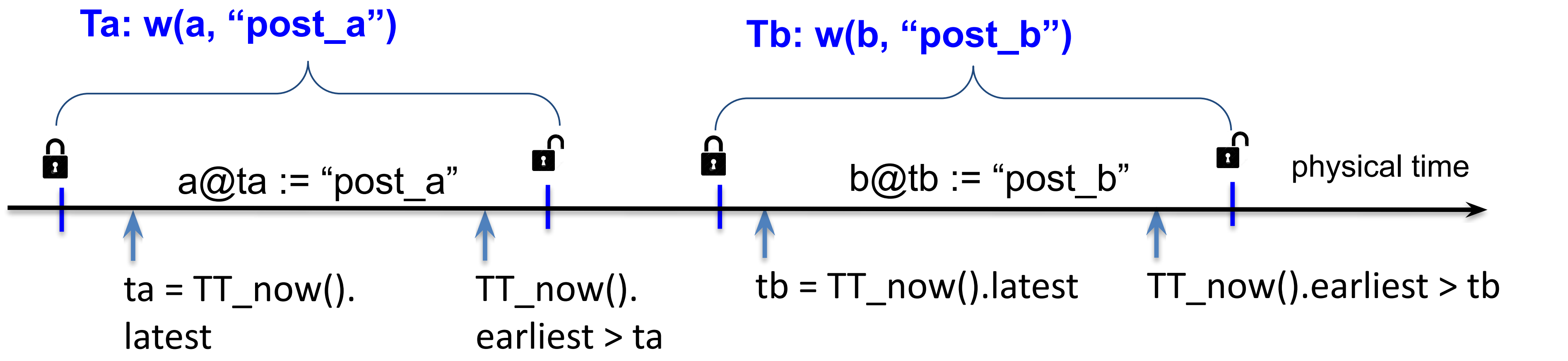
Case 3: ta < tb < tc

Tc: r(a), r(b)

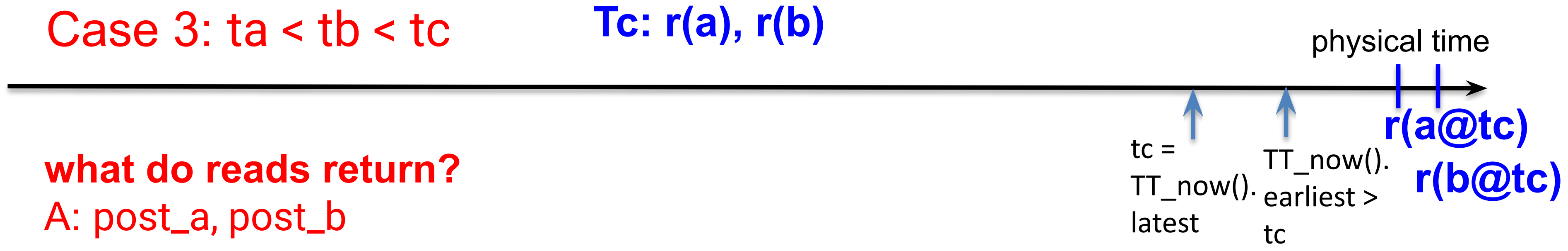


what do reads return?

Execution in Our Post Example



Case 3: ta < tb < tc



what do reads return?

A: post_a, post_b

Implications of Commit Wait

- the larger the uncertainty bound from TrueTime, the longer commit wait period you get
- commit wait will slow down dependent transactions, since locks are held during commit wait
- so, as time gets less certain, Spanner gets slower (!!). View talk or read paper for an evaluation.

Safety are given by the (assumed) correctness of the (earliest, latest) interval.

Performance is given by the size of the interval.

Key Papers

- Spanner paper.