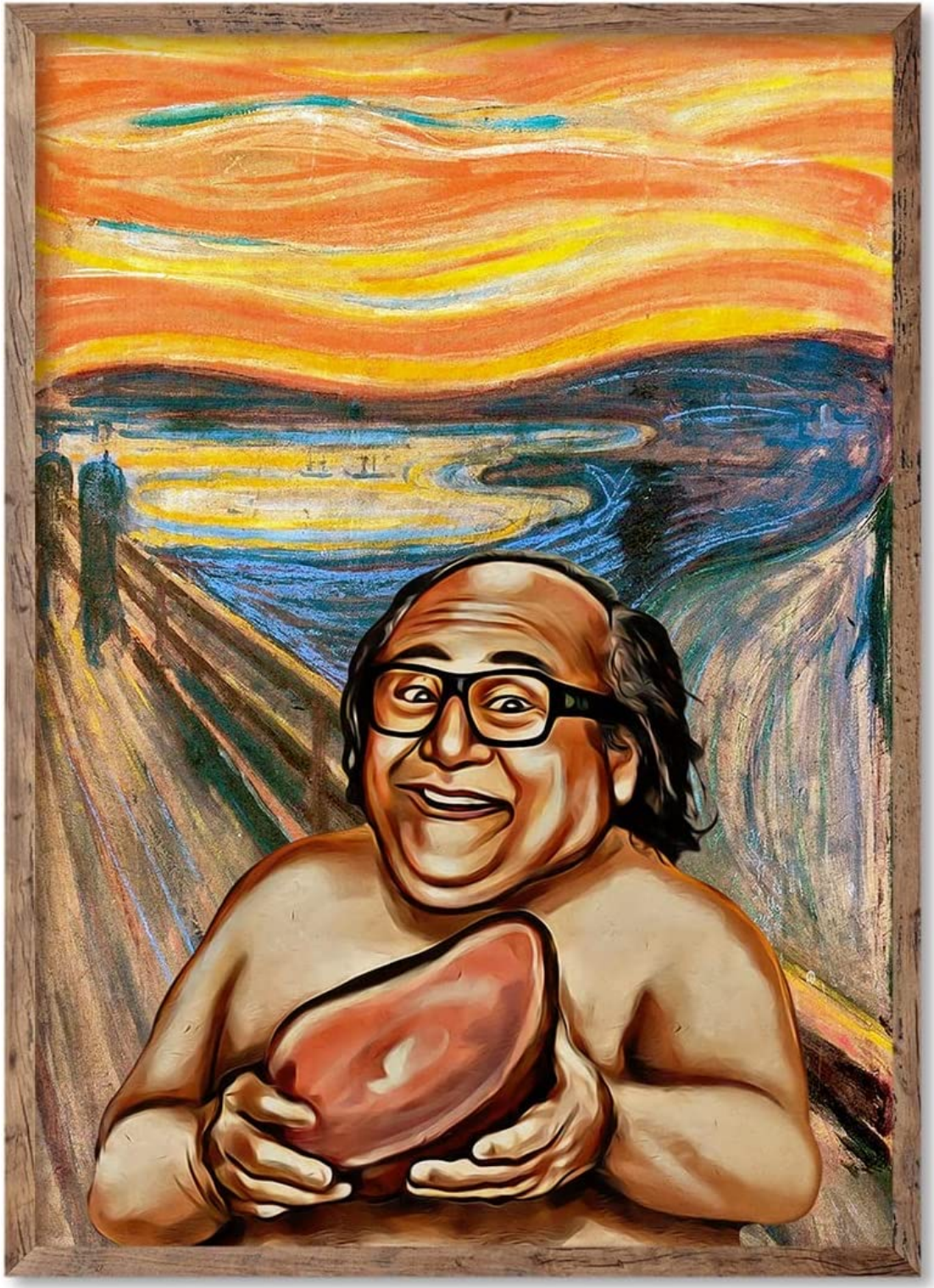
6.5830 Lecture 13
Two-phase Locking Recap
Optimistic Concurrency Control

October 26, 2022

Quiz 1 Back; Mean: 81, Median: 83, Std. Dev: 10

Lab 4 Out



Transactions

- Group related sequence of actions so they are "all or nothing"
  - If the system crashes, partial effects are not seen
  - Other transactions do not see partial effects

- A set of implementation techniques that provides this abstraction with good performance

ACID Properties of Transactions

- **A** tomicity – many actions look like one; “all or nothing”
- **C** onsistency – database preserves invariants
- **I** solation – concurrent actions don’t see each other’s results
- **D** urability – completed actions in effect after crash (“recoverable”)

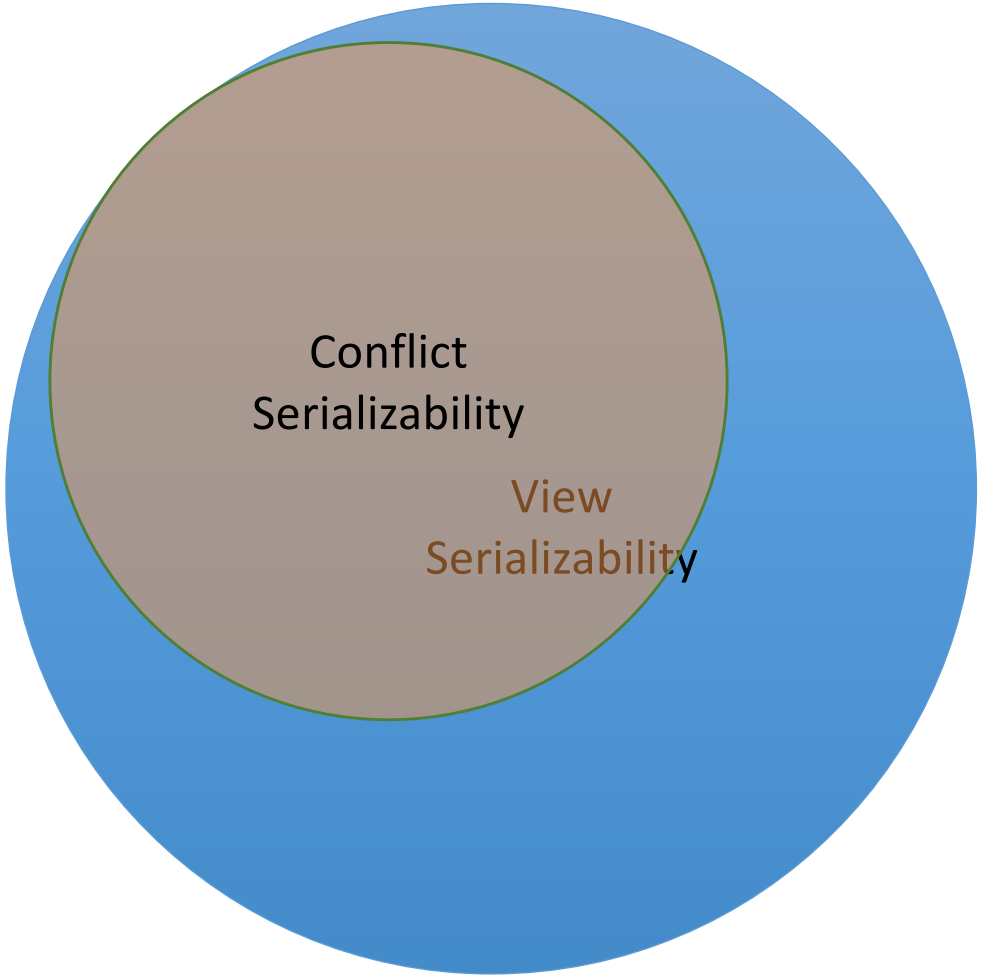
Atomicity/2PL Recap

- Goal: given a set of transactions, consisting of multiple operations, schedule them such that they are equivalent to some serial execution of those transactions.

T1	T2
RA	
WA	
	RA
	WA
RB	
WB	
	RB
	WB

Serially equivalent to T1 then T2

Testing for Serializability



Any schedule that is conflict serializable is view serializable, but not vice-versa

Precedence Graph for Testing Conflict Serializability

Given transactions Ti and Tj,

Create an edge from Ti→Tj if:

- Ti reads/writes some A before Tj writes A

  $RA_{Ti} \prec WA_{Tj}$ or $WA_{Ti} \prec WA_{Tj}$

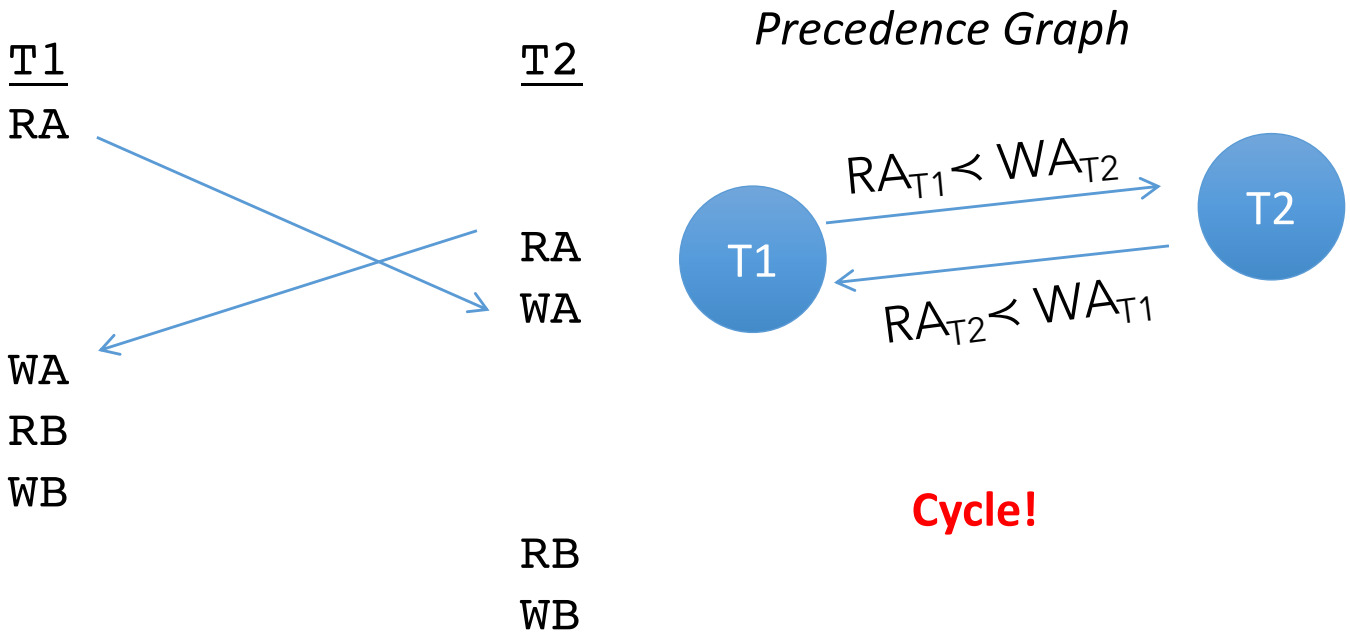
or

- Ti writes some A before Tj reads A

  $WA_{Ti} \prec RA_{Tj}$

If there are cycles in this graph, schedule is not conflict serializable

Non-Serializable Example



Create an edge from Ti→Tj if:

Ti reads/writes some A before Tj writes A, or

$RA_{Ti} \prec WA_{Tj}$ or $WA_{Ti} \prec WA_{Tj}$

Ti writes some A before Tj reads A

$WA_{Ti} \prec RA_{Tj}$

Two Phase Locking (2PL) Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- Release locks only after last lock has been acquired, and ops on that object are finished

Refining 2PL

- Problems:
  - Deadlocks
  - Cascading Aborts

  - How do we know when we are done with all operations on an object?

Rigorous Two-Phase Locking Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- Release locks only after the transaction commits
- Ensures cascadeless-ness, and
- *Commit order = serialization order*

Final Wrinkle: Phantoms

- T1 scans a range; T2 later inserts into that range
- If T1 scans the range again, it will see a new value

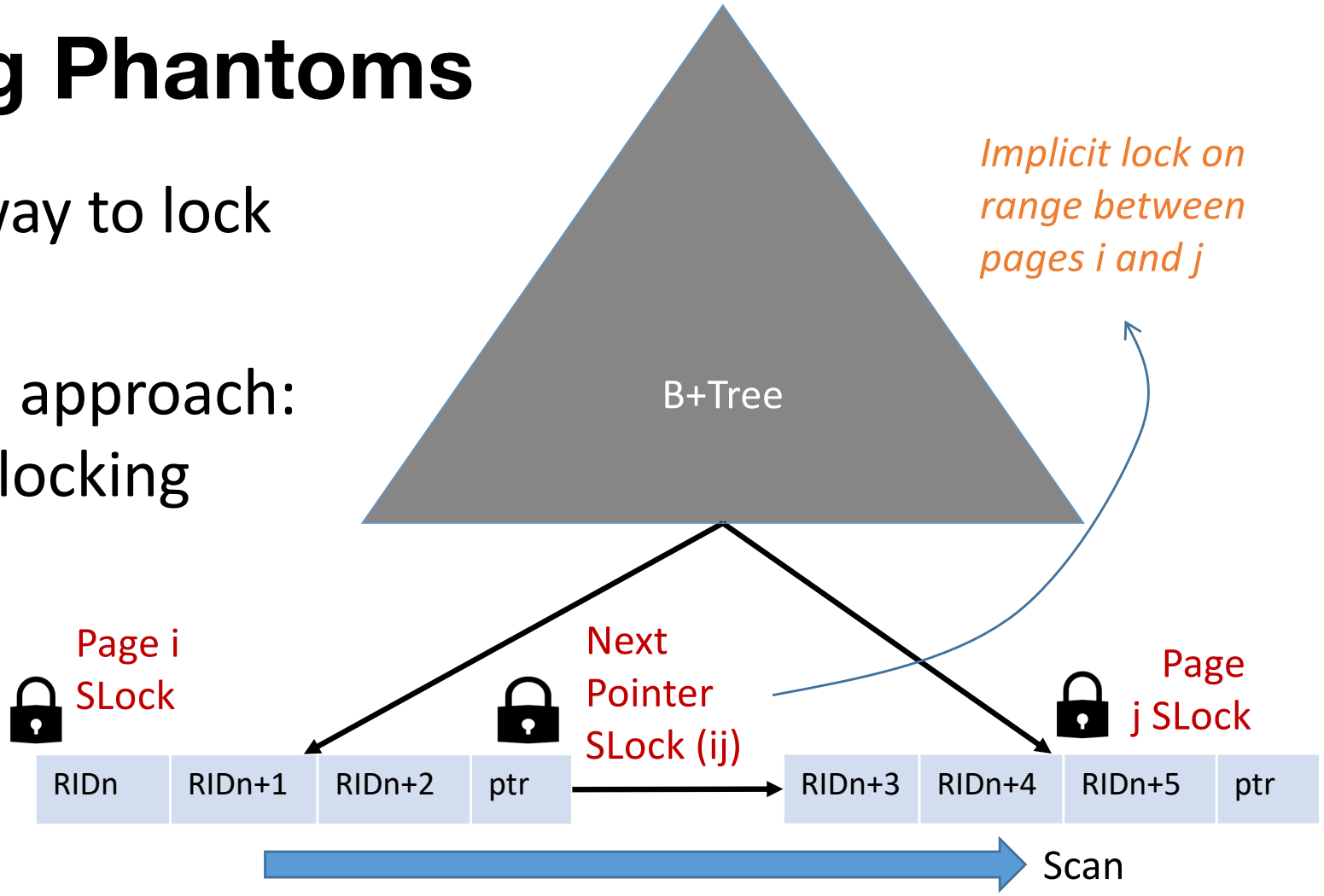
```
T1
BEGIN
      SELECT * FROM emp WHERE SAL > 100
      …
      SELECT * FROM emp WHERE SAL > 200
END
```

```
T2
BEGIN
      INSERT INTO EMP VALUES(…,sal=225)
END
```

If we are just locking, e.g., records, this insertion would be allowed in all 2PL algos we have studied, but is not serializable (since this couldn't happen in a serial execution).

Solving Phantoms

- Need a way to lock ranges
- Common approach: next key locking



On insert(val), Xlock ij next pointer if val> max(page i) and < min(page(j))

Only works for ranges with indexes
For unindexed tables, must read the whole table, so just use a table lock
More details next lecture!

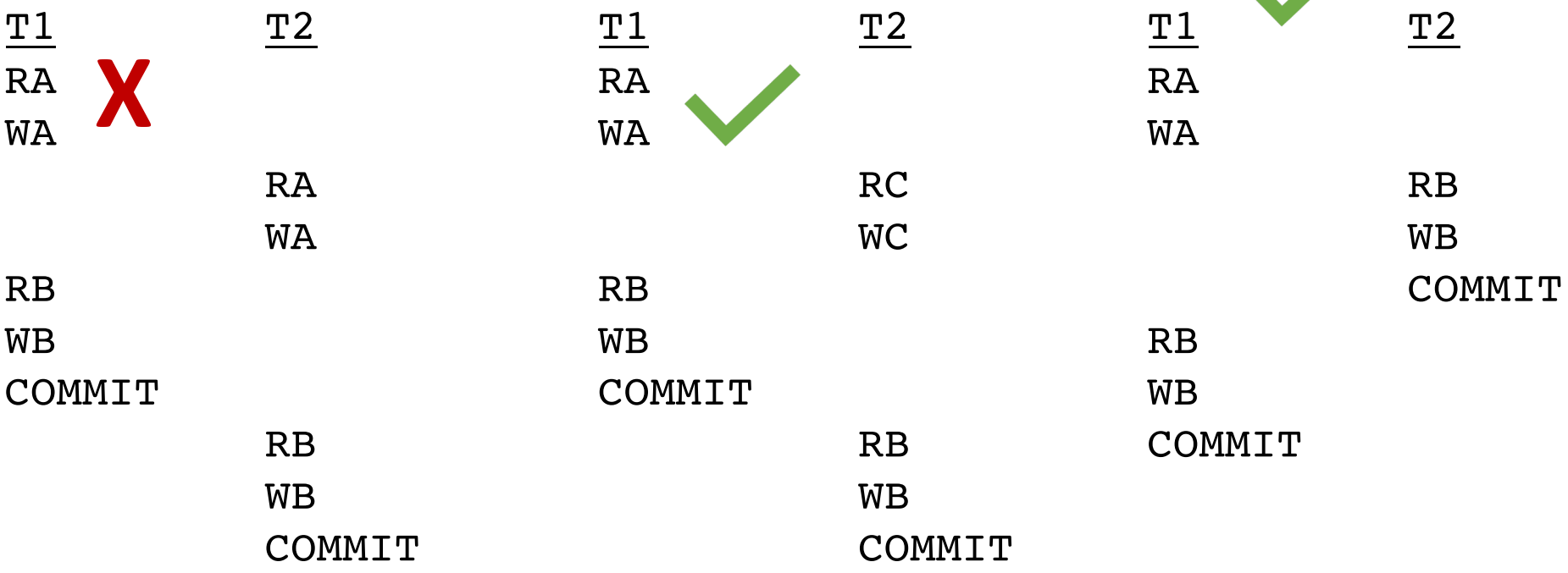
Implementing 2PL

- **SimpleDB**: Lock Table
  - Buffer pool maintains a table of locks per page
  - Transactions acquire locks on reads/writes of pages
  - Release locks at commit

- Access to lock table will need to be synchronized, but not using special “lock” objects to represent the locks.

Study Break

- Which of the following schedules would rigorous 2PL permit?



T1	T2
RA	
WA	
	RA
	WA
RB	
WB	
COMMIT	
	RB
	WB
	COMMIT

T1	T2
RA	
WA	
	RC
	WC
RB	
WB	
COMMIT	
	RB
	WB
	COMMIT

T1	T2
RA	
WA	
	RB
	WB
	COMMIT
RB	
WB	
COMMIT	

T1 does not release lock on A until COMMIT

Optimistic Concurrency Control (OCC)

- Alternative to locking for isolation

- Approach:
  - Store writes in a per-transaction buffer
  - Track read and write sets
  - At commit, check if transaction conflicted with earlier (concurrent) transactions
  - Abort transactions that conflict
  - Install writes at end of transaction
- "Optimistic" in that it does not block, hopes to "get lucky" arrive in serial interleaving

Tradeoff

- In OCC:
  - Never have to wait for locks
  - no deadlocks
- But...
  - Transactions that conflict often have to be restarted
  - Transactions can "starve" -- e.g., be repeatedly restarted, never making progress
- OCC will do better when the restart rate is low
  - (Less contention)
- Recent work on high performance transaction processing has focused on OCC because
  - OCC checks can be done between individual transactions
    - Unlike global shared lock table
  - Modern OCC systems obtain *insane* throughput (> 10M xactions / sec)

E.g., https://people.eecs.berkeley.edu/~wzheng/silo.pdf

OCC Implementation

- Divide transaction execution in 3 phases
  - **Read**: transaction executes on DB, stores local state
  - **Validate**: transaction checks if it can commit
  - **Write**: transaction writes state to DB

Read Phase

- Transactions execute, with updates affecting local copies of the data
- Build a list of data items that were read/written
  - *Read* and *write sets*
- Modify functions to read and write data from DB

OCC Write

```
twrite(object,value):
    if object not in write_set:  // never written, make copy
        m = read(object)
        copies[object] = m
        write_set = write_set U {object}
   write(copies[object], value)
```

By writing to local copies, we ensure dirty results aren't visible to other concurrent transactions

OCC Read

```
tread(object):
        read_set = read_set U {object};
        if object in write_set:
                return read(copies[object]);
        else:
                return read(object);
```

Allows us to read our own writes!

Validation Phase

- How do we know whether a transaction can commit?
  - *Validation Rules*
  - Check concurrent transactions for conflicts
- How do we implement validation efficiently?
  - *Validation Algorithm*
- *But first…* How how do we order transactions?

Transaction Identifier Assignment

- Goal: assign transaction ids T1, … Tn, such that this is the serial equivalent order
- When should we assign transaction identifiers?
- At start of read phase?
  - No! Would be “pessimistic” – don’t want to pre-assign the transaction order before transactions finish running
  - Long running transactions would have to commit before later short transactions
- Assign at end of read phase, just before validation starts

Validation Rules

When Tj completes its read phase, require that for all Ti < Tj, one of the following conditions must be true for validation to succeed (Tj to commit):

1) Ti completes its write phase before Tj starts its read phase

2) W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.

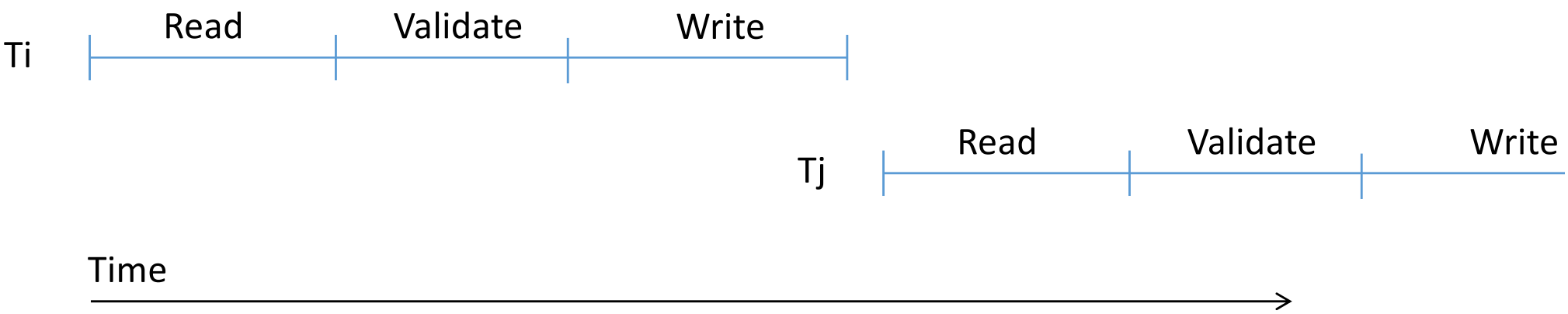
3) W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.

4) W(Ti) does not intersect R(Tj) or W(Tj), and W(Tj) does not intersect R(Ti) [no conflicts]

These rules will ensure serializability, with Tj being ordered after Ti with respect to conflicts

Condition 1

Ti completes its write phase before Tj starts its read phase



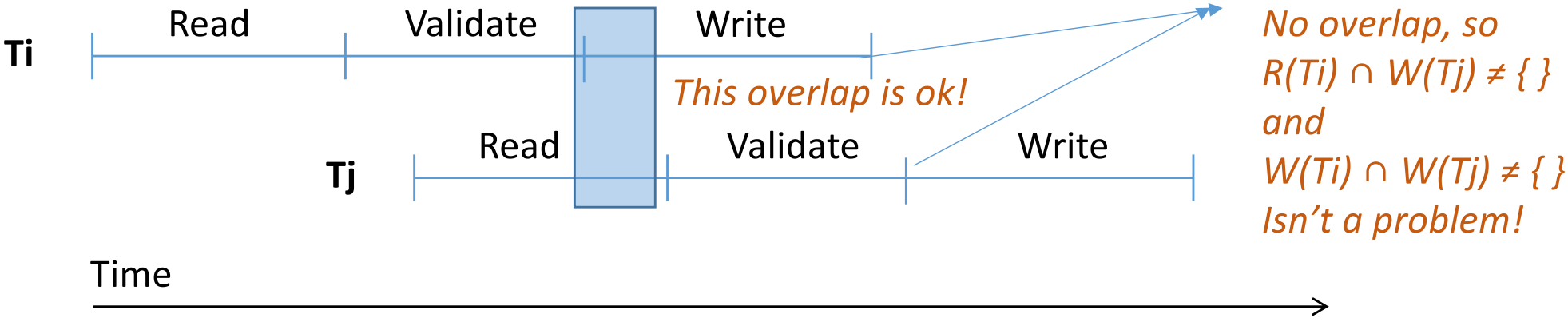
Don't overlap at all.

Condition 2

W(Ti) intersects W(Tj), i.e., Tj wrote something Ti wrote,
or
R(Ti) intersects W(Tj), i.e., Ti read something Tj wrote

W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.

$W(Ti) \cap R(Tj) = \{\}$ $R(Ti) \cap W(Tj) \neq \{\}$ $W(Ti) \cap W(Tj) \neq \{\}$



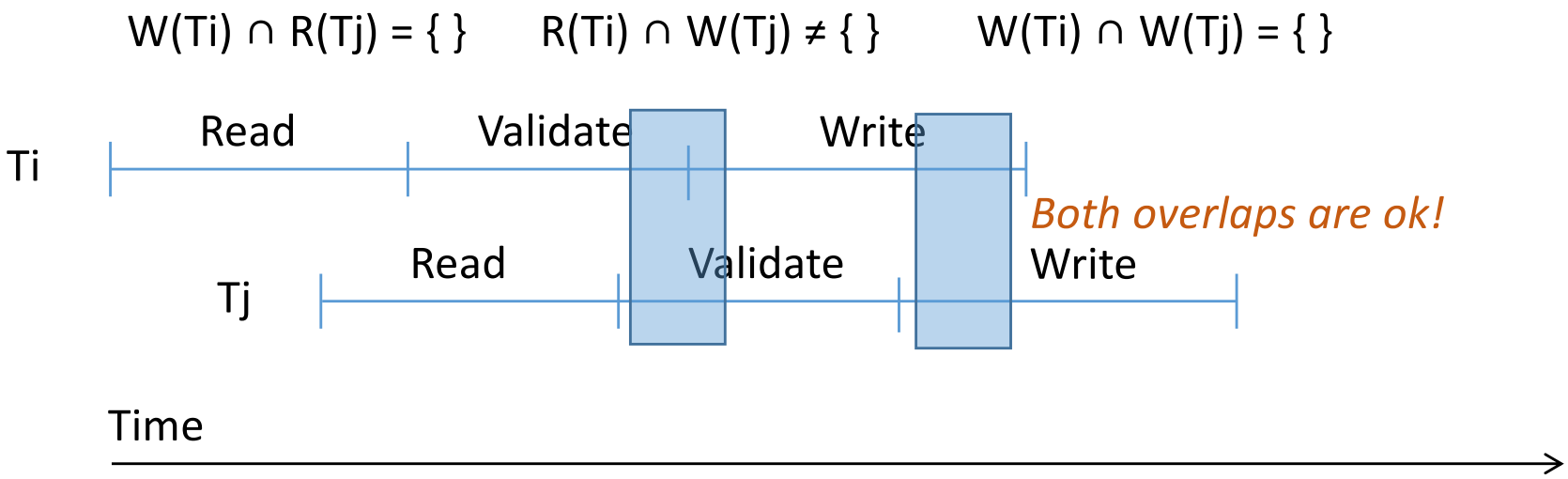
No overlap, so
$R(Ti) \cap W(Tj) \neq \{\}$
and
$W(Ti) \cap W(Tj) \neq \{\}$
Isn't a problem!

Tj doesn't read anything Ti wrote.
Anything Tj wrote that Ti also wrote will be installed afterwards.
Anything Ti read will not reflect Tjs writes

Condition 3

R(Ti) intersects W(Tj), i.e., Ti read something Tj wrote

W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.



Tj doesn't read or write anything Ti wrote (but Ti may read something Tj writes).

Ti definitely won't see any of Tj's writes, because it finishes reading before Tj starts validation, so Ti ordered before Tj.

Ti will always complete its read phase before Tj b/c xaction IDs assigned after read phase

If no conditions apply, abort!

Restating previous rules, aborts required if:

1) $W(Ti) \cap R(Tj) \neq \{\}$, and Ti does not finish writing before Tj starts, Tj must abort, because Tj may have only seen some of what Ti wrote

or

2) $W(Ti) \cap (W(Tj) \cup R(Tj)) \neq \{\}$, and Tj overlaps with Ti validation or write phase, Tj must abort because it needs its writes to all appear after Ti's writes

Validate Implementation

- Several different implementations designed to provide different levels of concurrency during writeback

- Transaction initialization:

```
tnc = 0; // current transaction id
void tbegin {
    read_set = new Set();
    write_set = new Set();
    start_tn = tnc; //the transaction that
                    //finished just before this
                    //one started
}
```

Serial Validation

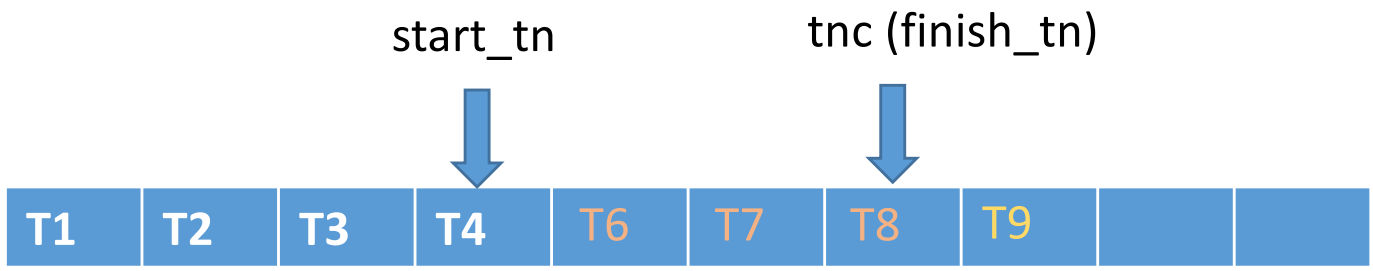
```
validateAndWrite(pastT[], start_tn, my_read_set, my_write_set)
{
    lock();
    int finish_tn = tnc;  //prior transaction
    bool valid = true;
    for(int t = start_tn + 1; t <= finish_tn; t++)
        if(pastT[t].write_set intersects with my_read_set)
            valid = false;
    if (valid) {
        write_phase();
        tnc = tnc+1;
        tn = tnc;
    }
    unlock();
}
```

1. W(Ti) ∩ R(Tj) ≠ { }, and Ti does not finish writing before Tj starts, Tj must abort
2. ~~W(Ti) ∩ (W(Tj) U R(Tj)) ≠ { }, and Tj overlaps with Ti validation or write phase, Tj must abort~~

2nd condition doesn't occur because if Ti completes its read phase before Tj, it will also complete its write phase before Tj.

Example

Last transaction that validated before this transaction started



Transactions that validated and wrote while this transaction was in read phase

Have to compare against T6, T7, and T8

Study Break

- Which of the following transactions would serial validation allow to commit, assuming the transactions are concurrent and Ti completes its write phase before Tj starts its write phase ∀ i < j

Aborts required if:

1) W(Ti) ∩ R(Tj) ≠ { }, and Ti does not finish writing before Tj starts, Tj must abort, because Tj may have only some of what Ti wrote

or

2) W(Ti) ∩ W(Tj) ∩ R(Tj)) ≠ { } , and Tj overlaps with Ti validation or write phase, must abort because it needs writes to all appear after Ti's writes

A.

T1	T2
RA	RC
WA	WA
RB	
WB	

Blind write

C.

T1	T2
RA	RA
WA	RB
	WA

B.

T1	T2	T3
RA	RB	RA
WA	WB	RC
		WC

Problems w/ Serial Validation

- Only one transaction can commit at a time
  - Severely limits transaction throughput
- Idea:
  - Reduce critical sections
  - Allow multiple transactions to write at a time
  - Check that concurrent writers didn't intersect committing transactions write set (condition #2)

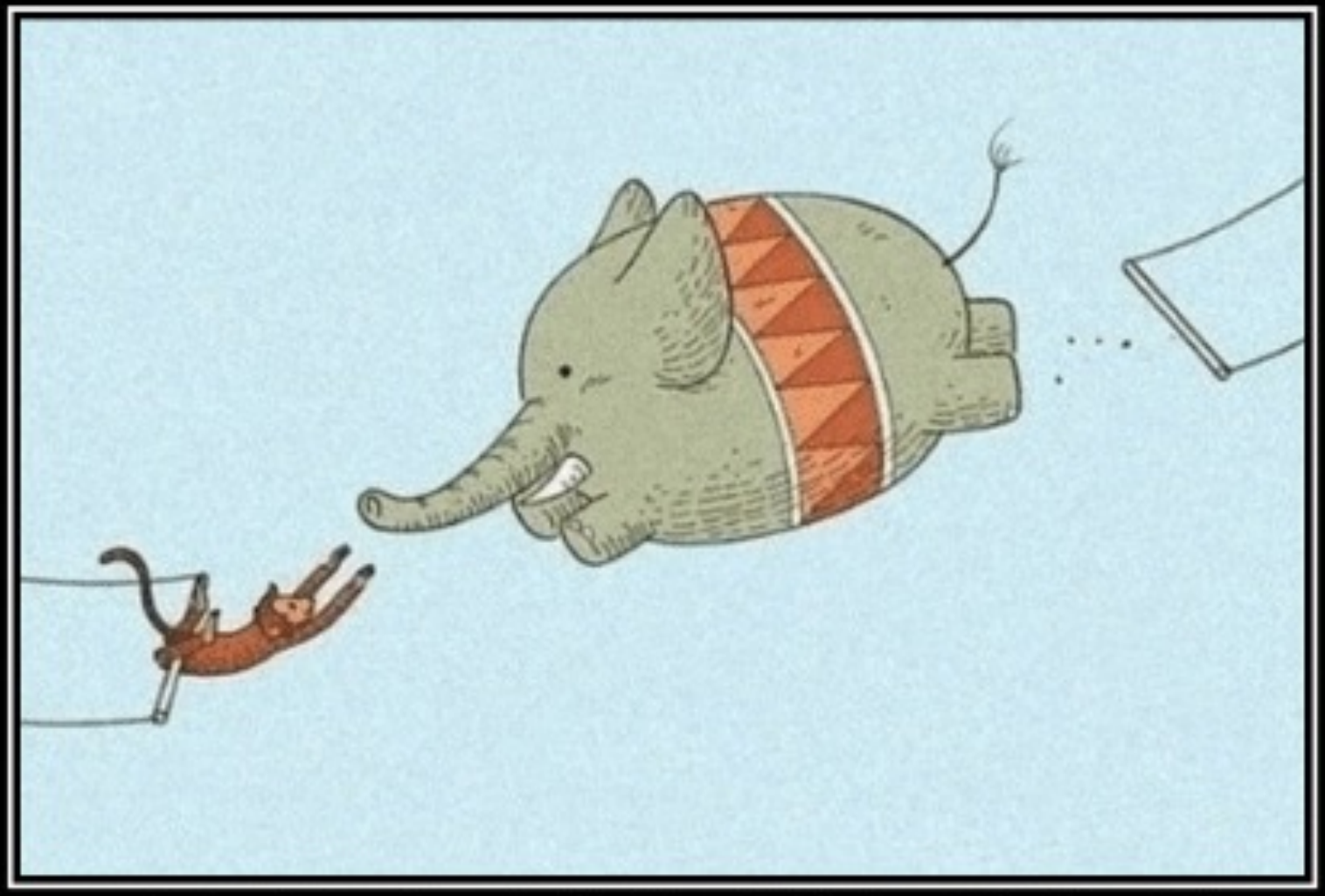
Parallel Validation

```
List<Transaction> active = new List();
validateAndWrite(pastT[], start_tn,
                 my_read_set, my_write_set) {
    lock();
    int finish_tn = tnc;
//transactions writing concurrently
    List<Transaction> finish_active = active.copy();
    active.append(me.id);
    unlock();
    bool valid = true;
    for(int t = start_tn + 1; t <= finish_tn; t++)
        if(pastT[t].write_set intersects with my_read_set)
            valid = false;                          Condition 1
    for (id t : finish_active)
        if(pastT[t].write_set intersects with  Condition 1 &2
                        (my_read_set U my_write_set))
            valid = false;
```

```
if(valid)  {
//note that concurrent writes
//are all to different objects
      write_phase();
       lock();
       tnc = tnc+1;
       tn = tnc;
       active.remove(me.id);
       unlock();
} else {
       lock();
       active.remove(me.id);
       unlock();
}}
```

1. W(Ti) ∩ R(Tj) ≠ { }, and Ti does not finish writing before Tj starts, Tj must abort
2. W(Ti) ∩ (W(Tj) U R(Tj))≠ { }, and Tj overlaps with Ti validation or write phase, Tj must abort

We must check both!



OPTIMISM

Not always the best option.

fakeposters.com

What If Serializability Isn't Needed?

- E.g., application only needs to read committed data
- Databases provide different isolation levels
  - READ UNCOMMITTED
    - Ok to read other transaction's dirty data
  - READ COMMITTED
    - Only read committed values
  - REPEATABLE READS
    - If R1 read A=x, R2 will read A=x ∀ A
- Many database systems default to READ COMMITTED

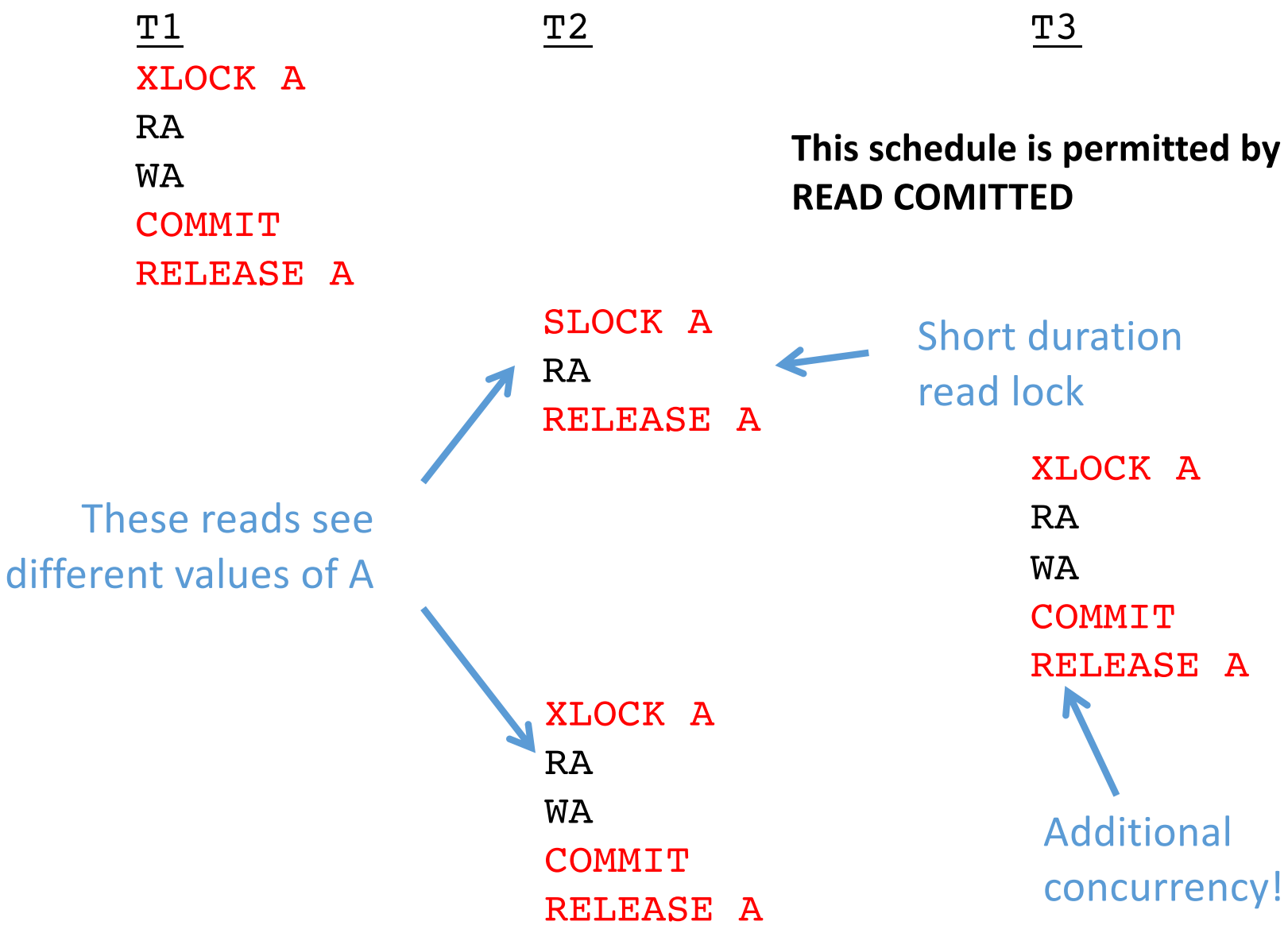
READ UNCOMMITTED w/ Locking

- If OK reading uncommitted data, no need to check if records that are read are locked
- However, to prevent other transactions from seeing dirty data, need to hold write locks for the duration of the transaction
- May be OK if, e.g., just reporting some statistic, like number of users or views

READ COMMITTED w/ Locking

- To ensure that a transaction only reads committed values, need to acquire locks before reading
  - If all other transactions hold write locks (as in READ UNCOMMITTED), it will never read a dirty value
- Since we doesn't care about always reading the same value, OK to release locks after a value is read
- As in READ UNCOMMITTED, write locks still need to be held for the duration of the transaction

READ COMMITTED Example



REPEATABLE READ w/ Locking

- If we want to always read the same value, need to hold read locks for transaction duration

- So how is this different from SERIALIZABLE?

- SERIALIZABLE also needs to prevent phantoms

REPEATABLE READ vs SERIALIZABLE

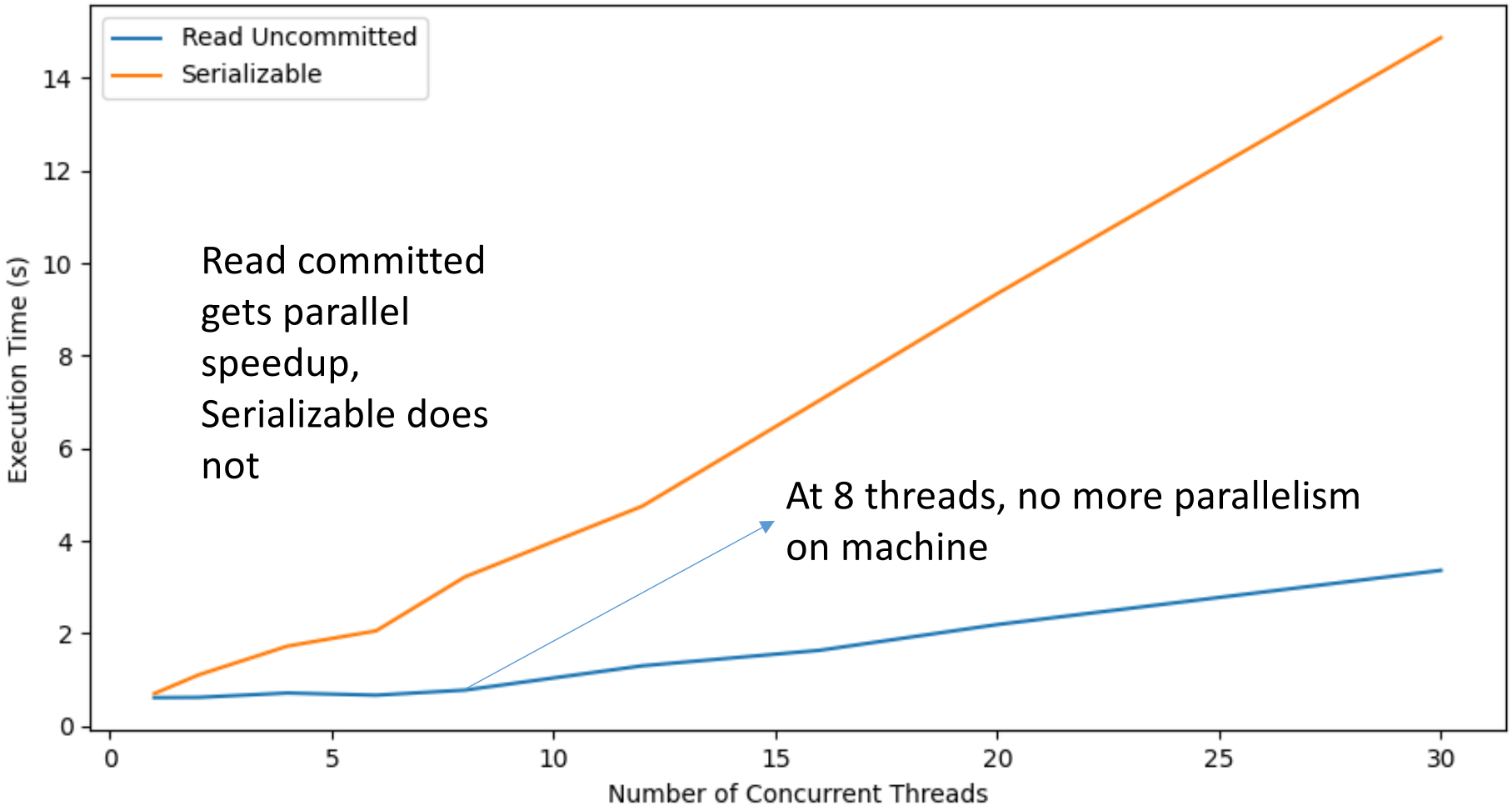
- Some systems, e.g., Postgres implement REPEATABLE READ through a different mechanism based on database snapshots taken at the start of transaction
  - Called “multiversion concurrency control” – yet another way of achieving isolation!

- This has other problems besides phantoms – so called “read skew anomalies”
  - See: https://www.cockroachlabs.com/blog/what-write-skew-looks-like/

Demo

- Table with 1000 rows t (a int, b int, c int)
- Transactions that:
  - Count all rows
  - Update a random row
- Threads that run this transaction 1000 times
- Vary number of threads (level of concurrency) from 1 to 20
  - READ UNCOMMITTED doesn't have conflicts between counts and updates
  - SERIALIZABLE xactions may have to block/abort on update due to concurrent readers
- Measure runtime w/ READ UNCOMMITTED and SERIALIZABLE on Postgres
- Postgres multi-version concurrency can abort transactions on conflict; retry aborts

Results


Read Uncommitted
Serializable
Execution Time (s)
0
2
4
6
8
10
12
14
0
5
10
15
20
25
30
Number of Concurrent Threads
Read committed gets parallel speedup, Serializable does not
At 8 threads, no more parallelism on machine

Summary

- Optimistic concurrency control provides another way to provide serializability
- Good for low contention workloads
- Basis for many modern high throughput transactional systems

- Reduced consistency levels that lock fewer records common in practice
  - Permit greater concurrency at price of serializability