CSE544
Data Management

Lectures 09: Transactions

Announcements

HW4 due on Sunday

HW5 to be released over the weekend

Project feedback: this Friday

Next week:

- Tuesday/Wednesday: 1-1 zoom meetings
- Friday: Project presentations
  - Session 1: 2pm – 6pm
  - Session 2: 6pm – 9pm
  - Times are approximate

Transactions

- We use database transactions everyday
  - Bank $$$ transfers
  - Online shopping
  - Signing up for classes

- Applications that use a DB ***must*** use transactions in order to keep the database consistent.

Motivating Example

Client 1:

```
UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3
```

Client 2:

```
SELECT sum(money)
FROM Budget
```

Would like to treat each group of instructions as a unit

Transaction

Definition: a transaction is a sequence of updates to the database with the property that either all complete, or none completes (all-or-nothing).

```
BEGIN TRANSACTION

[SQL statements]

COMMIT    or    ROLLBACK (=ABORT)
```

If autocommit = off, then first query starts TXN
If autocommit = on, then each query is one TXN

Motivating Example

```
BEGIN TRANSACTION
    UPDATE Budget
    SET money=money-100
    WHERE pid = 1

    UPDATE Budget
    SET money=money+60
    WHERE pid = 2

    UPDATE Budget
    SET money=money+40
    WHERE pid = 3
COMMIT  (or ROLLBACK)
```

```
SELECT sum(money)
FROM Budget
```

If autocommit=ON,
no need to begin txn

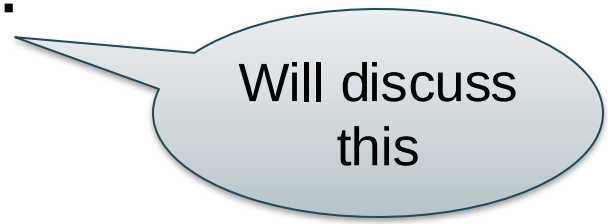
ACID Properties

- **A**tomicity: Either all changes performed by transaction occur or none occurs
- **C**onsistency: A transaction as a whole does not violate integrity constraints
- **I**solation: Transactions appear to execute one after the other in sequence
- **D**urability: If a transaction commits, its changes will survive failures

Transaction Manager

Concurrency control manager:

- Ensures Isolation



Recovery manager:

- Ensures Atomicity and Durability

Consistency is an implied property

Concurrency Control

Overview

- The CC manager, or scheduler, is responsible for ensuring isolation
- Rigorous definitions and correctness proofs are important

Schedules

Important Convention

- Database = a set of *elements*
  - DB = X_1, X_2, X_3, …
  - E.g. Elements are records, or disk pages
  - For concurrency control: records

- Txn = sequence of READ/WRITE ops:
  - READ(A); READ(B); …; WRITE(A);…

Schedules

A *schedule* is a sequence of interleaved actions from all transactions

Example

A and B are elements in the database
t and s are variables in tx source code

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A Serial Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

A Serial Schedule

A = 2
B = 2

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

A Serial Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

A = 2
B = 2

A = 102
B = 102

A Serial Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

A = 2
B = 2

A = 102
B = 102

A = 204
B = 204

A Serial Schedule

T1	T2
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	

A = 2
B = 2

A Serial Schedule

T1	T2
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	

A = 2
B = 2

A = 4
B = 4

A Serial Schedule

T1	T2
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	

A = 2
B = 2

A = 4
B = 4

A = 104
B = 104

Serializable Schedule

A schedule is *serializable* if it is equivalent to a serial schedule

A Serializable Schedule

T1	T2	
		A = 2 B = 2
READ(A, t)		
t := t+100		
WRITE(A, t)		A = 102 B = 2
	READ(A,s)	
	s := s*2	
	WRITE(A,s)	A = 204 B = 2
READ(B, t)		
t := t+100		
WRITE(B,t)		A = 204 B = 102
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	A = 204 B = 204

This is a serializable schedule.
This is NOT a serial schedule

A Non-Serializable Schedule

T1	T2	
		A = 2 B = 2
READ(A, t)		
t := t+100		A = 102 B = 2
WRITE(A, t)		
	READ(A,s)	
	s := s*2	A = 204 B = 2
	WRITE(A,s)	
	READ(B,s)	
	s := s*2	A = 204 B = 4
	WRITE(B,s)	
READ(B, t)		
t := t+100		A = 204 B = 104
WRITE(B,t)		

Serializable Schedules

- The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

Serializable Schedules

- The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

A: Because of very poor throughput due to disk latency.

Lesson: main memory databases *may* schedule TXNs serially

Still Serializable, but…

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s + 200
	WRITE(A,s)
	READ(B,s)
	s := s + 200
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Still Serializable, but…

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s + 200
	WRITE(A,s)
	READ(B,s)
	s := s + 200
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Serializable because
(A+100)+200 = (A+200)+100

Still Serializable, but…

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s + 200
	WRITE(A,s)
	READ(B,s)
	s := s + 200
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Serializable because
(A+100)+200 = (A+200)+100

…but don't expect the scheduler to know this!

A Simplified Model

- TXN is a sequence of Reads and Write
- Ignore what the TXN does to the elements

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

A Simplified Model

- TXN is a sequence of Reads and Write
- Ignore what the TXN does to the elements

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

- Given schedule, check if we can swap operations into a serial schedule
- Ops that cannot be swapped are in conflict

Conflicts

Two actions by same transaction T_i: $r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element $w_i(X); w_j(X)$

Read/write by T_i, T_j to same element $w_i(X); r_j(X)$

$r_i(X); w_j(X)$

Conflict Serializability

Definition A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

- Every conflict-serializable schedule is serializable
- The converse is not true in general

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

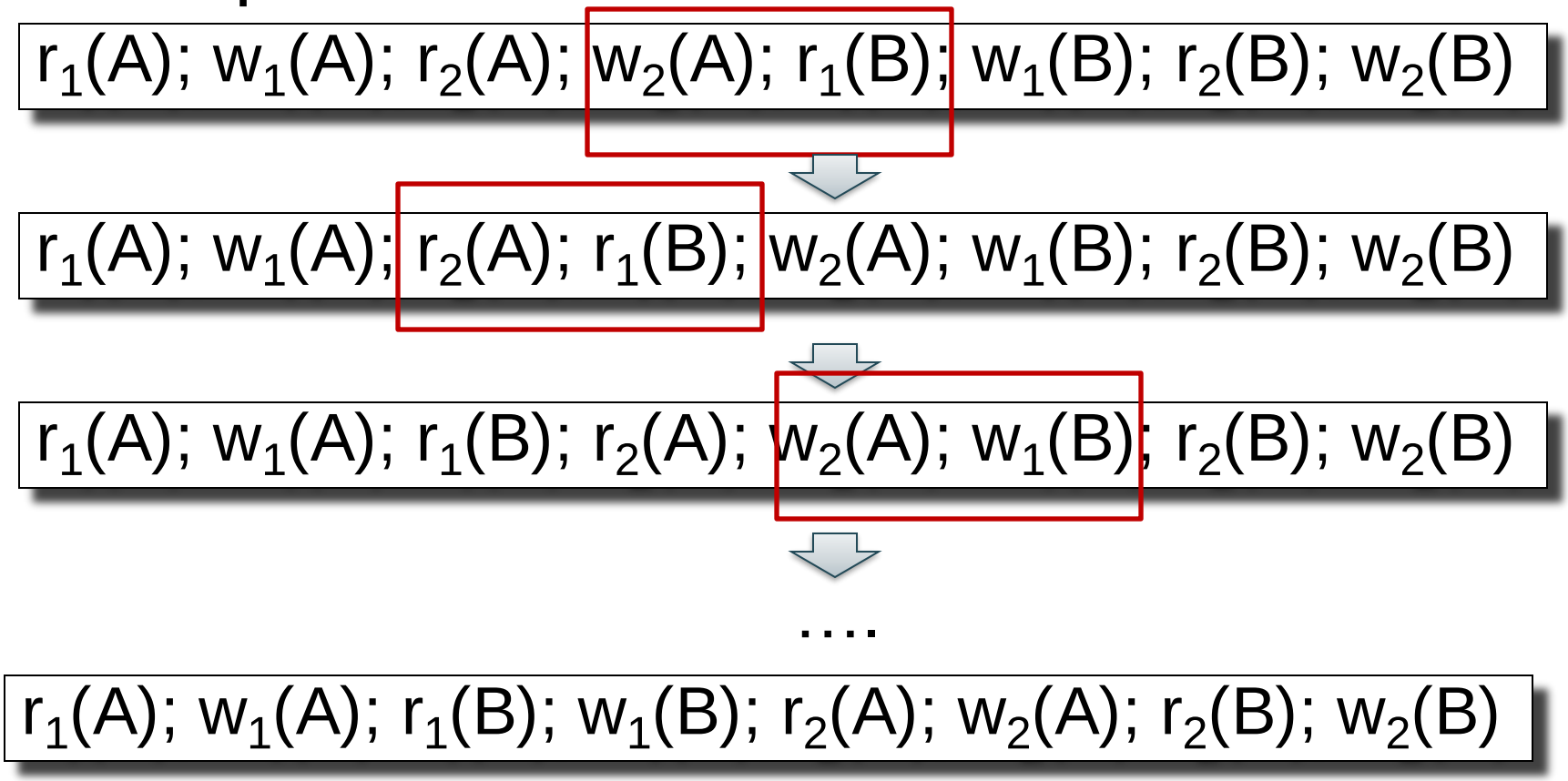
$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:



Serializable, Not Conflict-Serializable

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s + 200
	WRITE(A,s)
	READ(B,s)
	s := s + 200
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Serializable, Not Conflict-Serializable

T1	T2
$r_1(A)$	
~~t := t+100~~	
$w_1(A)$	
	$r_2(A)$
	~~s := s + 200~~
	$w_2(A)$
	$r_2(B)$
	~~s := s + 200~~
	$w_2(B)$
$r_1(B)$	
~~t := t+100~~	
$w_1(B)$	

Serializable, Not Conflict-Serializable

T1	T2
$r_1(A)$	
$w_1(A)$	
	$r_2(A)$
	$w_2(A)$
	$r_2(B)$
	$w_2(B)$
$r_1(B)$	
$w_1(B)$	

Testing for Conflict-Serializability

- Given a schedule, do a post-mortem test to see if it was conflict-serializable
- In practice we never to that
- HOWEVER understanding this test is important to design the scheduler

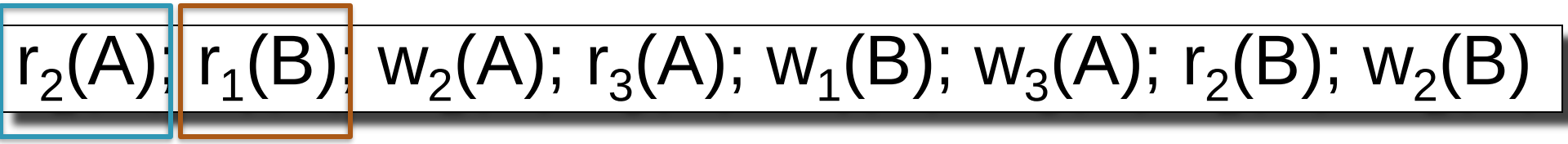
Testing for Conflict-Serializability

Precedence graph:

- One node for each transaction T_i,
- One edge $T_i \rightarrow T_j$ when an action in T_i conflicts with a later an action in T_j

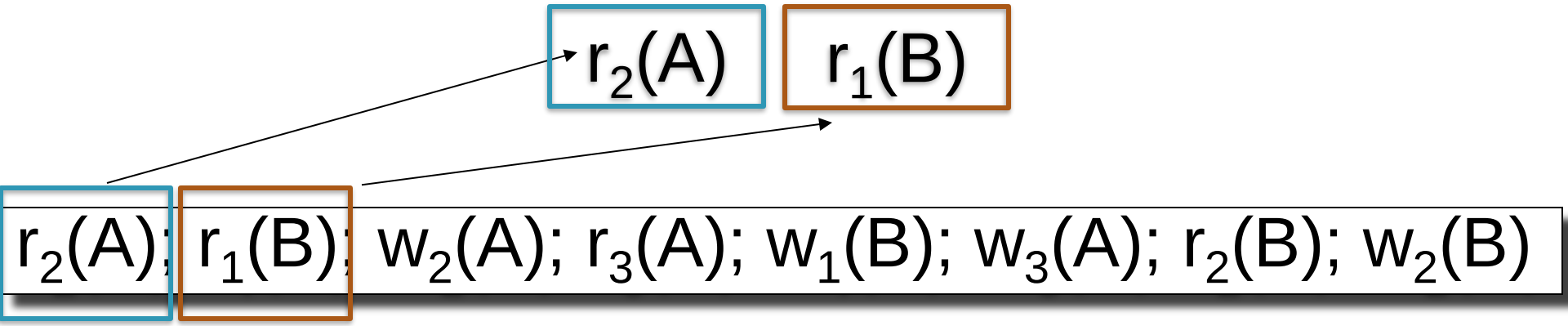
The schedule is conflict serializable iff the precedence graph is acyclic

Example 1



1 2 3

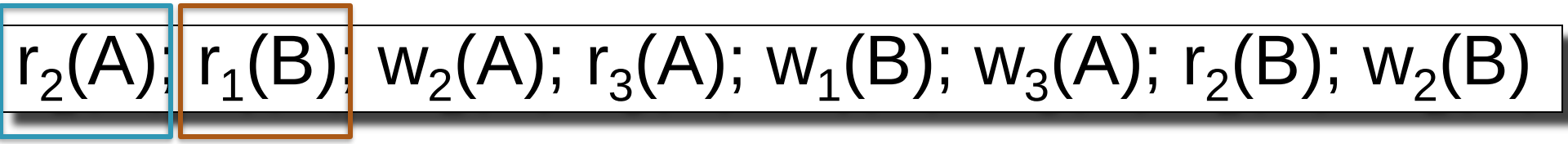
Example 1



1 2 3

Example 1

$r_2(A)$ $r_1(B)$



1 2 3

Example 1

$r_2(A)$ $r_1(B)$

No edge because no conflict (A != B)

$r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$

Example 1

$r_2(A)$ $w_2(A)$

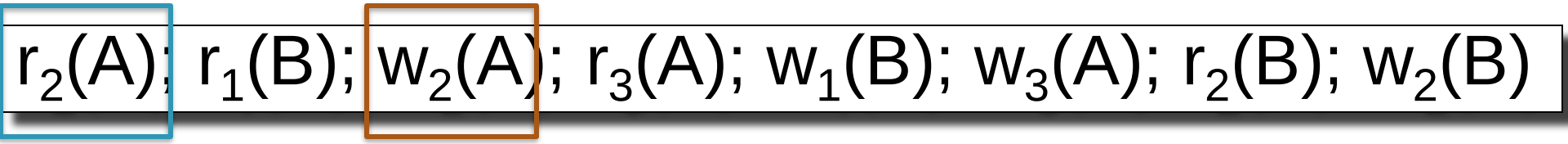
$r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$

1 2 3

Example 1

$r_2(A)$ $w_2(A)$

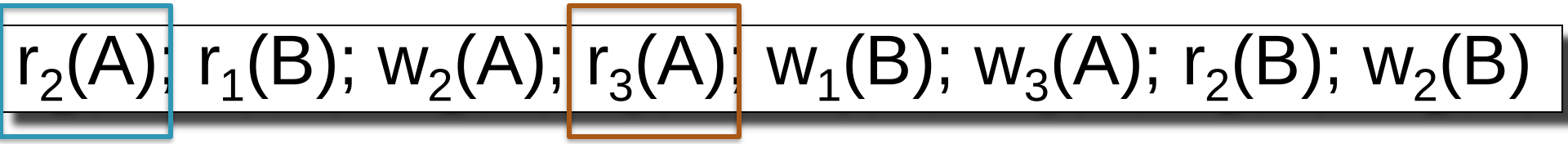
No edge because same txn (2)



1 2 3

Example 1

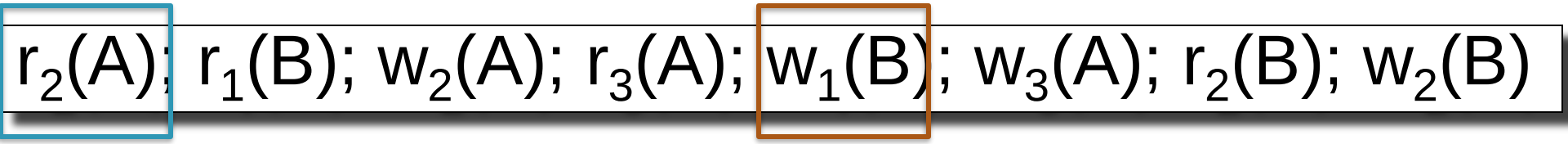
$r_2(A)$ $r_3(A)$?



1 2 3

Example 1

$r_2(A)$ $w_1(B)$?



1 2 3

Example 1

$r_2(A)$ $w_3(A)$?

$r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$

1 2 3

Example 1

$r_2(A)$ $w_3(A)$

Edge! Conflict from T_2 to T_3

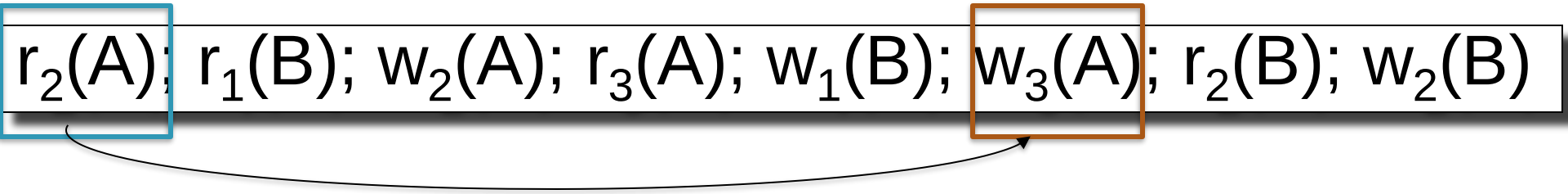
$r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$

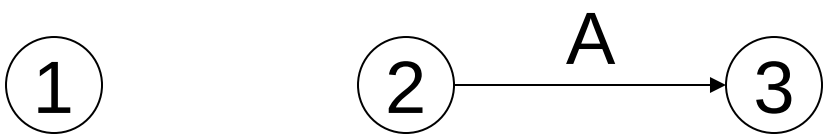
1 2 3

Example 1

$r_2(A)$ $w_3(A)$

Edge! Conflict from T_2 to T_3



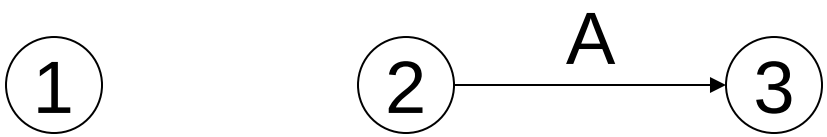


Example 1

$r_2(A)$ $r_2(B)$?

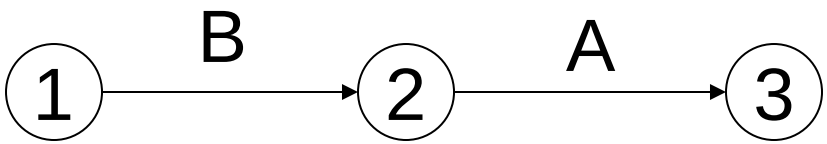
$r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$

And so on until compared every pair of actions…



Example 1

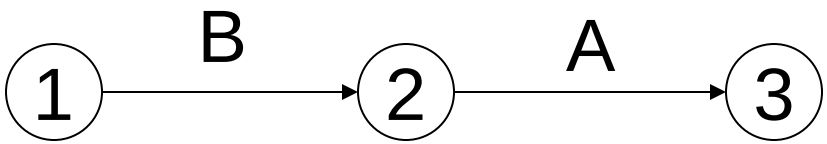
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



More edges, but repeats of same edge not necessary

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

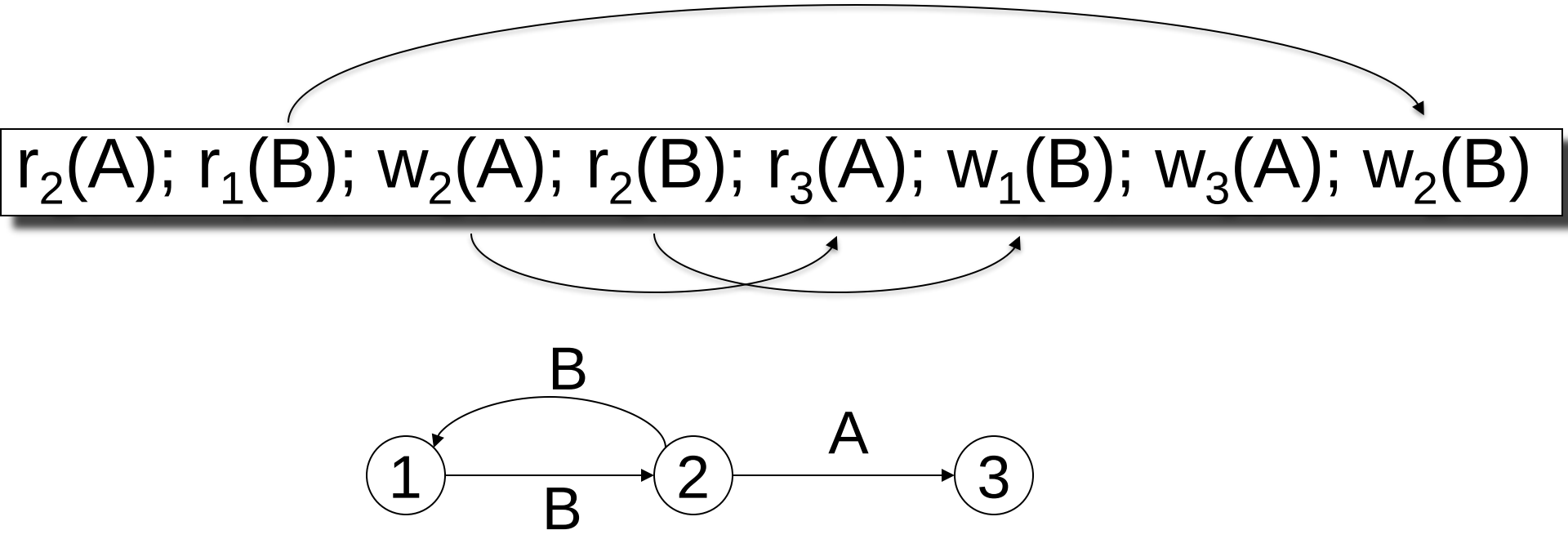


This schedule is **conflict-serializable**

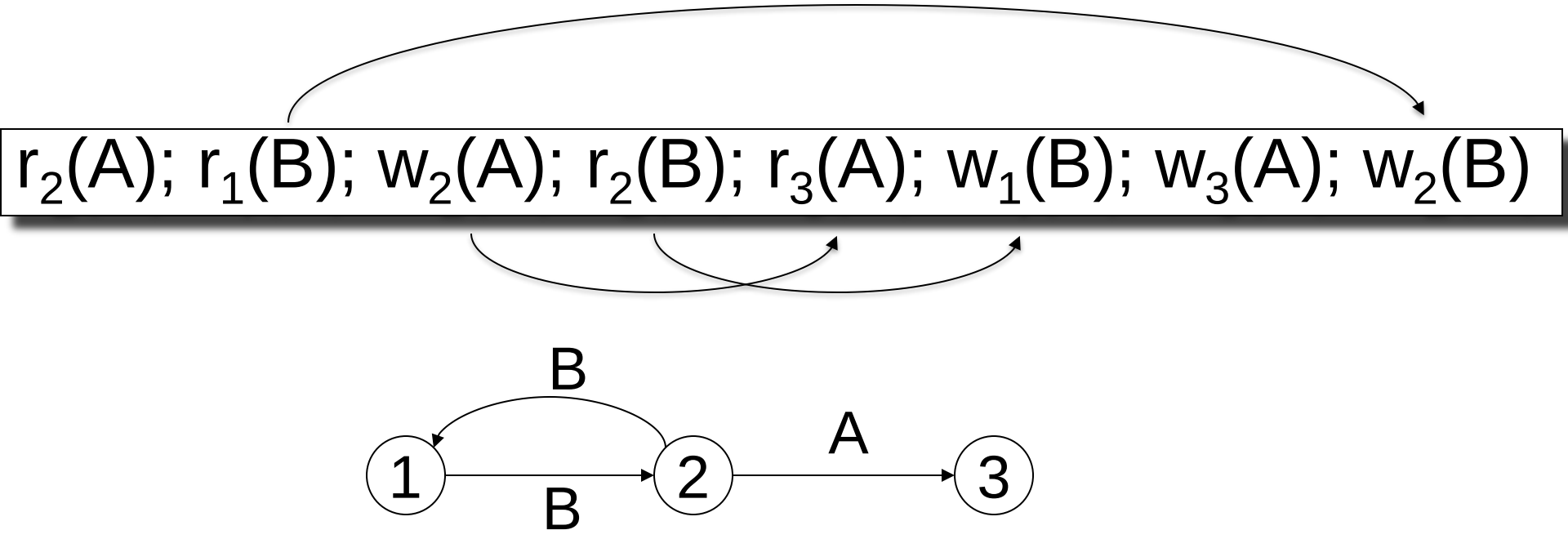
Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Example 2



Example 2



This schedule **is NOT conflict-serializable**

View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Is this schedule conflict-serializable ?

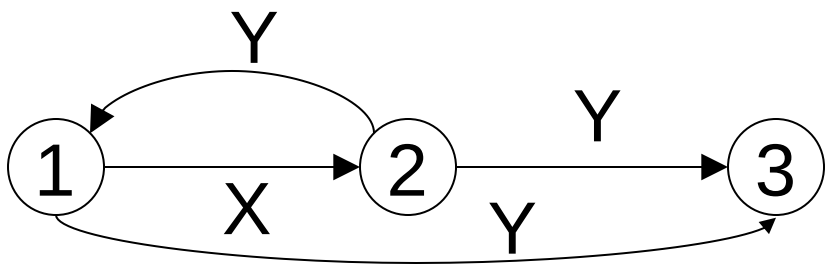
View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

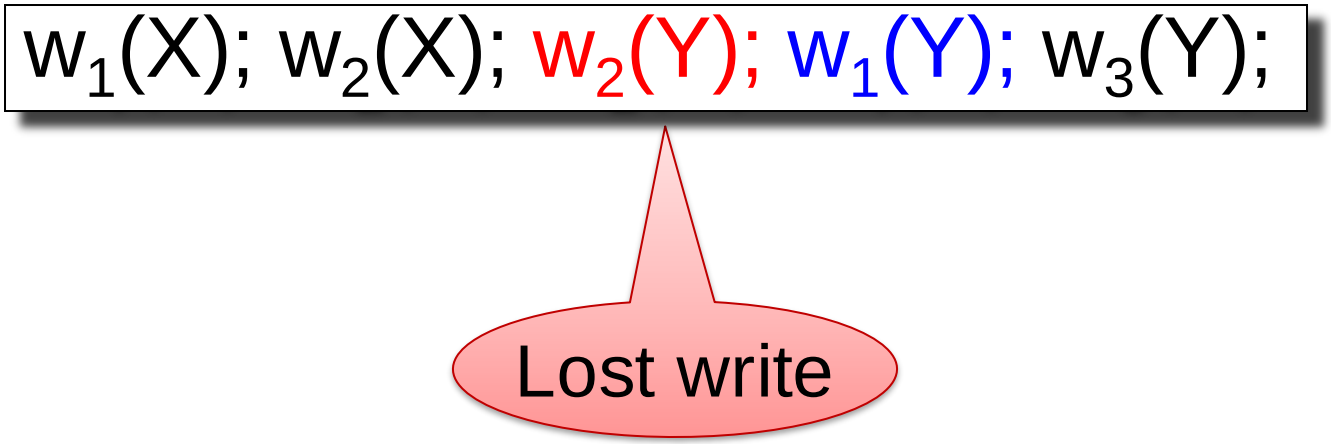
Is this schedule conflict-serializable ?

No…



View Equivalence

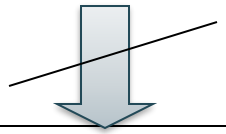
- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption



View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

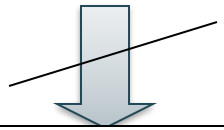


$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

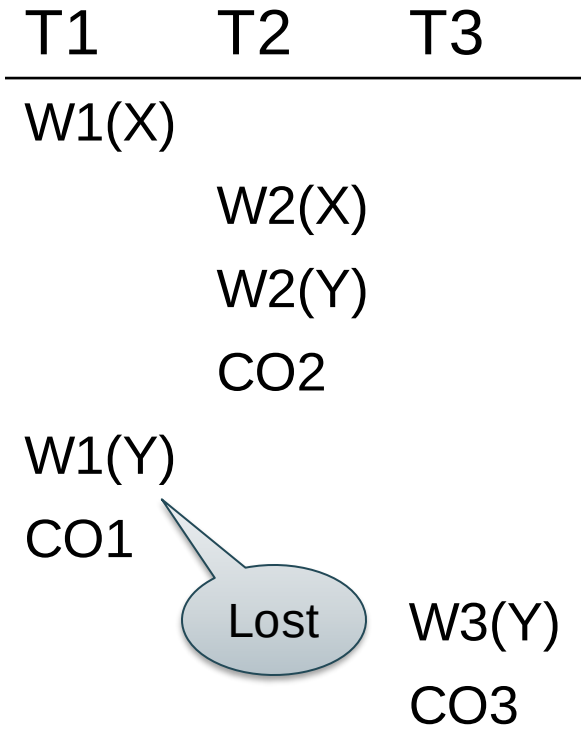
$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

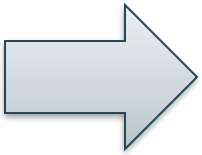


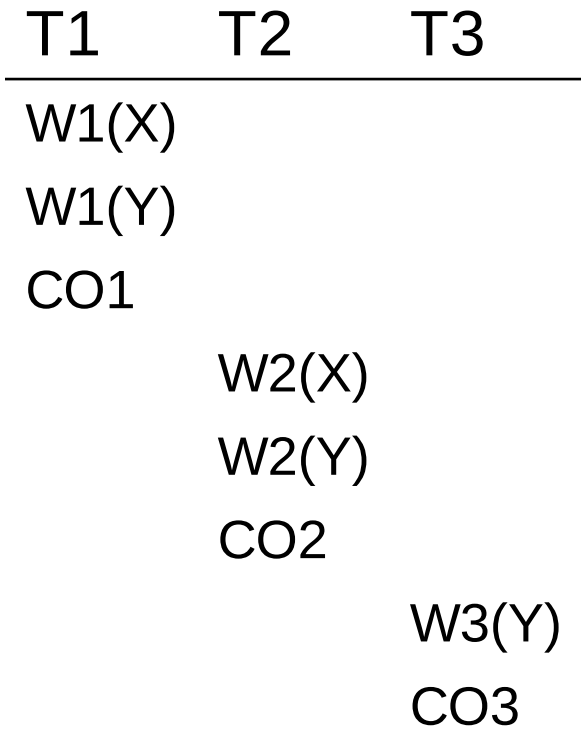
$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

Equivalent, but not conflict-equivalent

View Equivalence







Serializable, but not conflict serializable

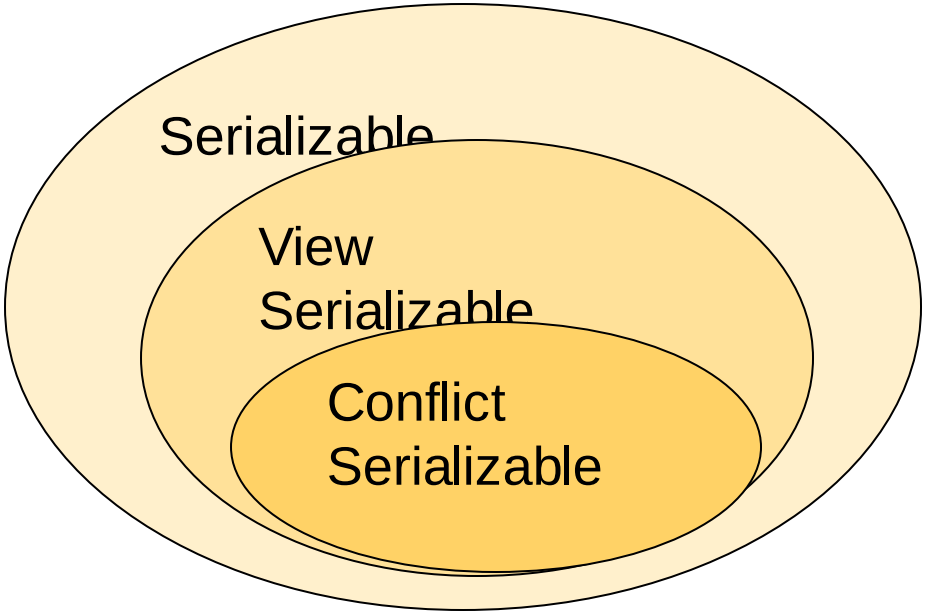
View Equivalence

Two schedules S, S' are *view equivalent* if:

- If T reads an initial value of A in S, then T reads the initial value of A in S'
- If T reads a value of A written by T' in S, then T reads a value of A written by T' in S'
- If T writes the final value of A in S, then T writes the final value of A in S'

View-Serializability

A schedule is *view serializable* if it is view equivalent to a serial schedule



Schedules with Aborted Transactions

- When a transaction aborts, the recovery manager undoes its updates
- But some of its updates may have affected other transactions !

Schedules with Aborted Transactions

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

Schedules with Aborted Transactions

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

What's wrong?

Cannot abort T1 because cannot undo T2

Recoverable Schedules

A schedule is *recoverable* if:

- It is conflict-serializable, and
- Whenever a transaction T commits, all transactions that have written elements read by T have already committed

Recoverable Schedules

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
?	

Nonrecoverable

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
Commit	
	Commit

Recoverable

Recoverable Schedules

T1	T2	T3	T4
R(A)			
W(A)			
	R(A)		
	W(A)		
	R(B)		
	W(B)		
		R(B)	
		W(B)	
		R(C)	
		W(C)	
			R(C)
			W(C)
			R(D)
			W(D)
Abort			

How do we recover ?

Cascading Aborts

A schedule *avoids cascading aborts* if whenever a transaction reads an element, the transaction that has last written it has already committed.

Avoiding Cascading Aborts

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
. . .	
	. . .

With cascading aborts

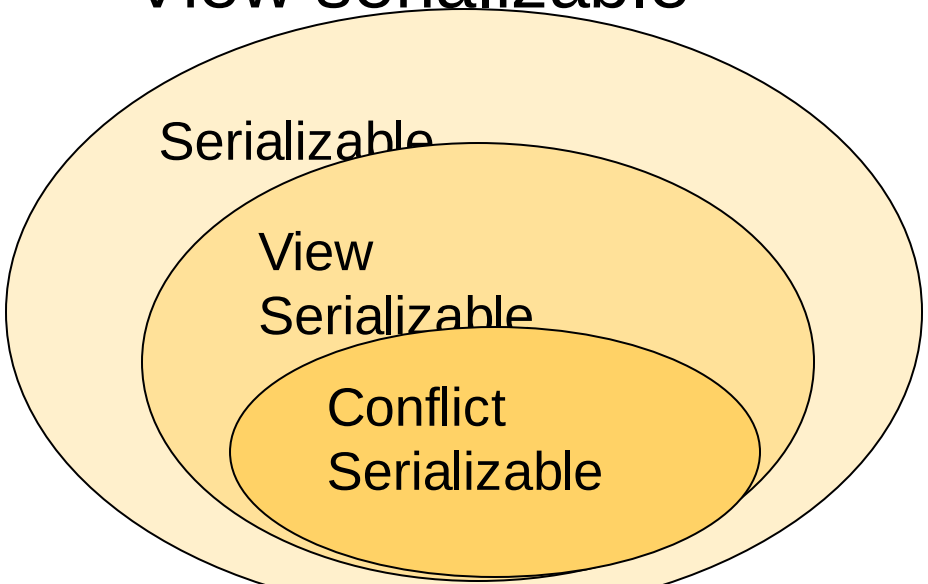
T1	T2
R(A)	
W(A)	
Commit	
	R(A)
	W(A)
	R(B)
	W(B)
	. . .

Without cascading aborts

Recap

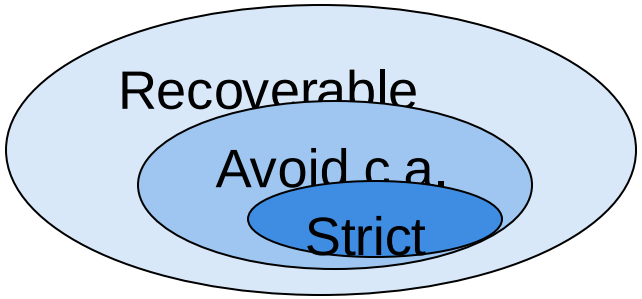
Serializability

- Serial
- Serializable
- Conflict serializable
- View serializable



Recoverability

- Recoverable
- Avoids cascading aborts



Implementing Transactions

Scheduler

A.k.a. Concurrency Control Manager

- The module that schedules the transaction
- TXN T requests: READ(X) or WRITE(X),
- Scheduler answers one of:
  - Proceed
  - Put in a wait queue, schedule another TXN T'
  - Abort (!!)

Implementing a Scheduler

- Locking Scheduler
  - Aka “pessimistic concurrency control”
  - SQLite, SQL Server, DB2
- Multiversion Concurrency Control (MVCC)
  - Aka “optimistic concurrency control”
  - Postgres, Oracle: Snapshot Isolation (SI)

Locking Scheduler

Locking Scheduler

- Each element has a unique lock
- TXN must acquire lock before read/write
- If the lock is taken: wait
- When done: release the lock

Actions on Locks

$L_i(A)$ = T_i acquires lock for element A

$U_i(A)$ = T_i releases lock for element A

A Non-Serializable Schedule

T1	T2
READ(A)	
A := A+100	
WRITE(A)	
	READ(A)
	A := A*2
	WRITE(A)
	READ(B)
	B := B*2
	WRITE(B)
READ(B)	
B := B+100	
WRITE(B)	

Example

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Example

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Schedule is conflict-serializable

But…

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$;	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;
$L_1(B)$; READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	

But…

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$;	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;
$L_1(B)$; READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	

Locks did not enforce conflict-serializability!

Two Phase Locking (2PL)

The 2PL rule:

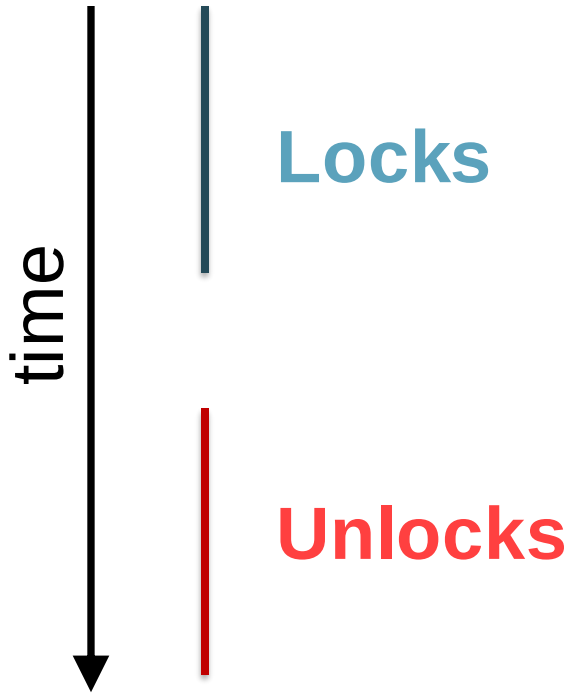
In every transaction, all lock requests must precede all unlock requests

Example: 2PL transactions

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;

Conflict-serializable

Two-Phase Locking



Two-Phase Locking

T1
L1(A)
READ(A, t)
t := t+100
WRITE(A, t)
U1(A)
L1(B)
READ(B, t)
t := t+100
WRITE(B,t)
U1(B)

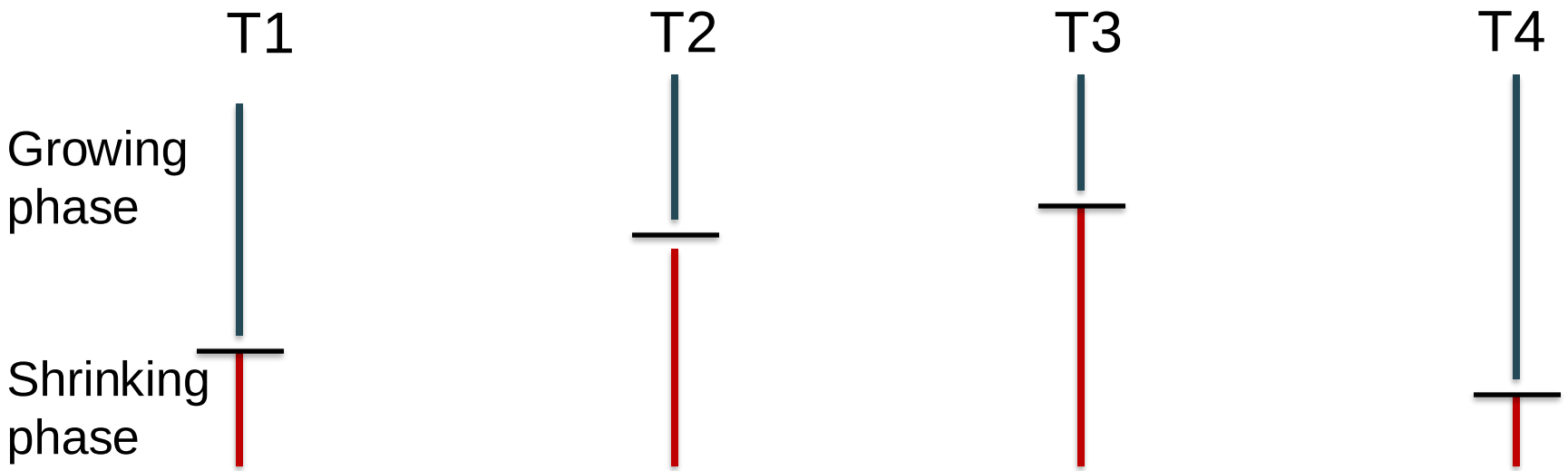
Not 2PL

T1
L1(A)
READ(A, t)
t := t+100
WRITE(A, t)
L1(B)
U1(A)
READ(B, t)
t := t+100
WRITE(B,t)
U1(B)

T1
L1(A)
L1(B)
READ(A, t)
t := t+100
WRITE(A, t)
U1(A)
READ(B, t)
t := t+100
WRITE(B,t)
U1(B)

2PL

Example with Multiple Transactions



Equivalent to each TXN executing entirely the **moment** it enters shrinking phase

Two-Phase Locking

Theorem: 2PL ensures conflict serializability

Two-Phase Locking

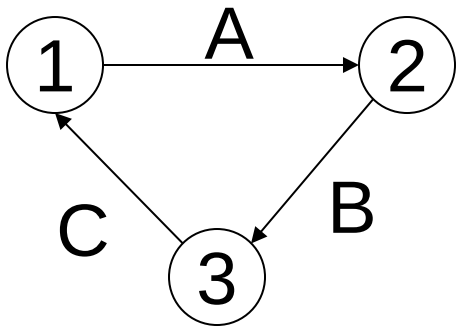
Theorem: 2PL ensures conflict serializability

Proof. Suppose precedence graph has a cycle

Two-Phase Locking

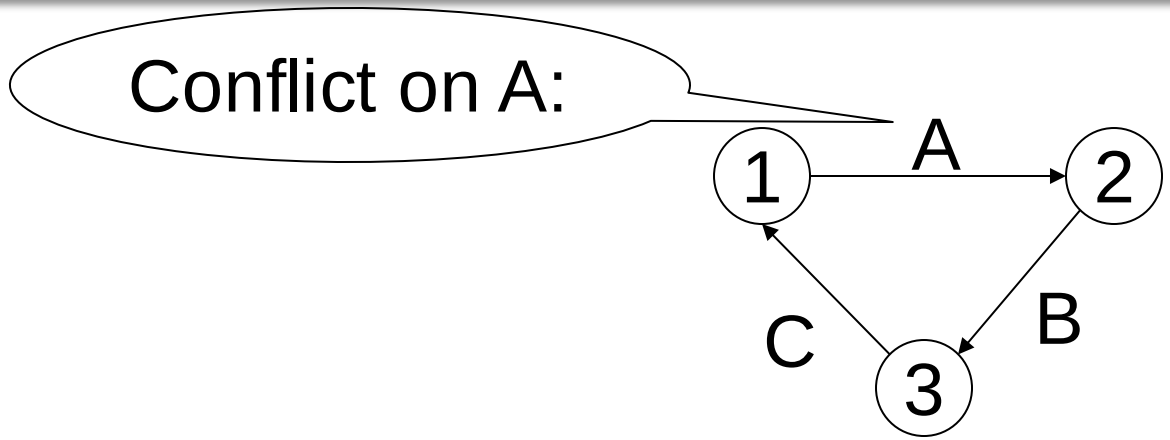
Theorem: 2PL ensures conflict serializability

Proof. Suppose precedence graph has a cycle



Two-Phase Locking

Theorem: 2PL ensures conflict serializability

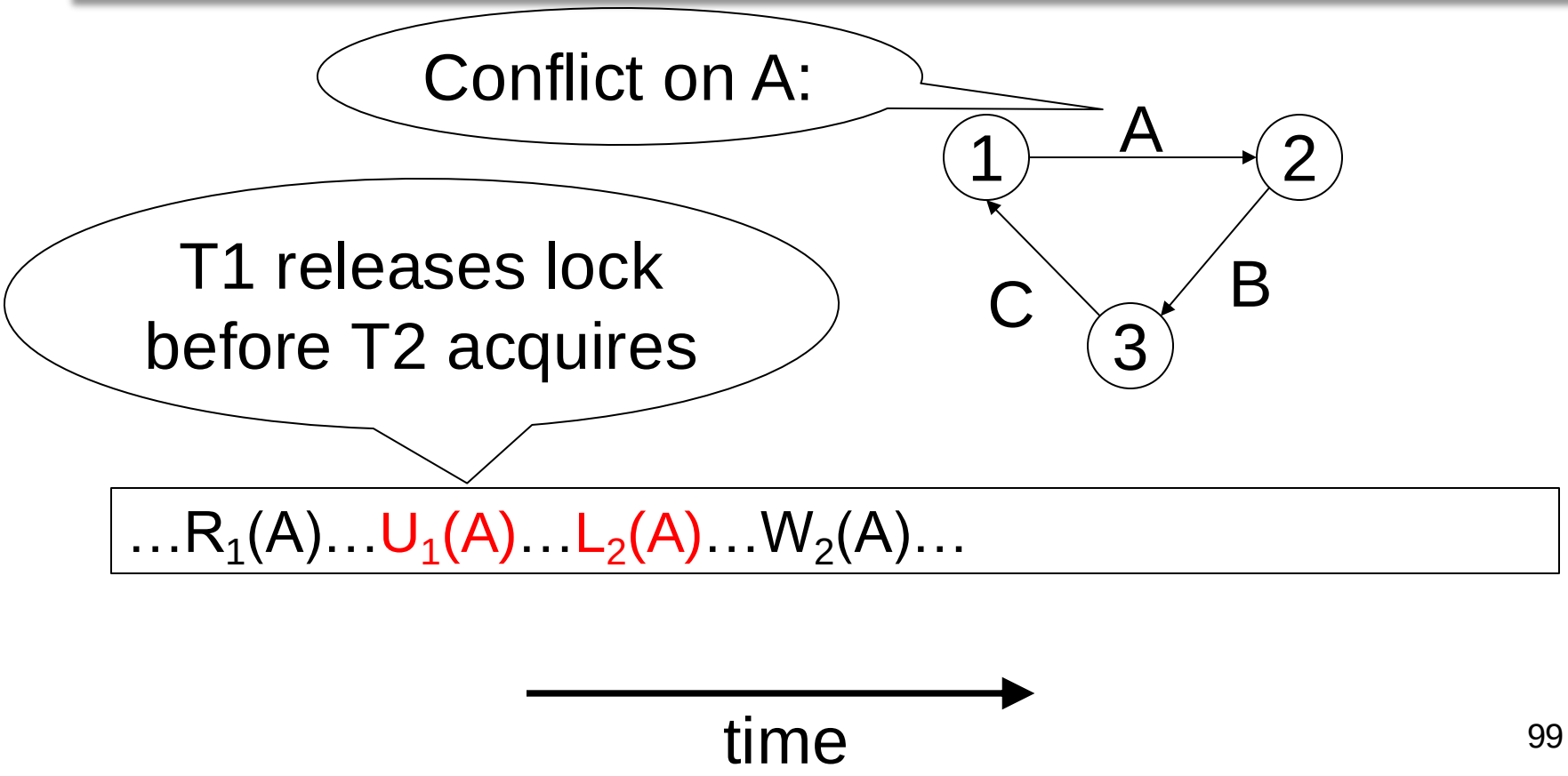


…$R_1(A)$… …$W_2(A)$…

time

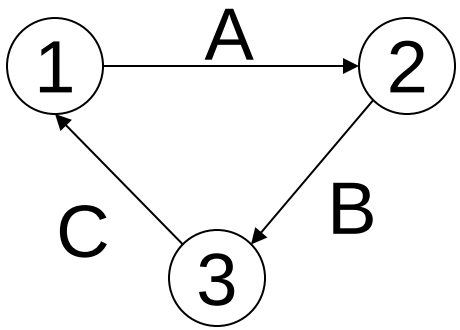
Two-Phase Locking

Theorem: 2PL ensures conflict serializability



Two-Phase Locking

Theorem: 2PL ensures conflict serializability

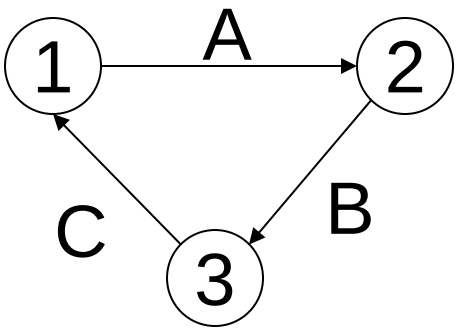


…$R_1(A)$…$U_1(A)$…$L_2(A)$…$W_2(A)$…

time

Two-Phase Locking

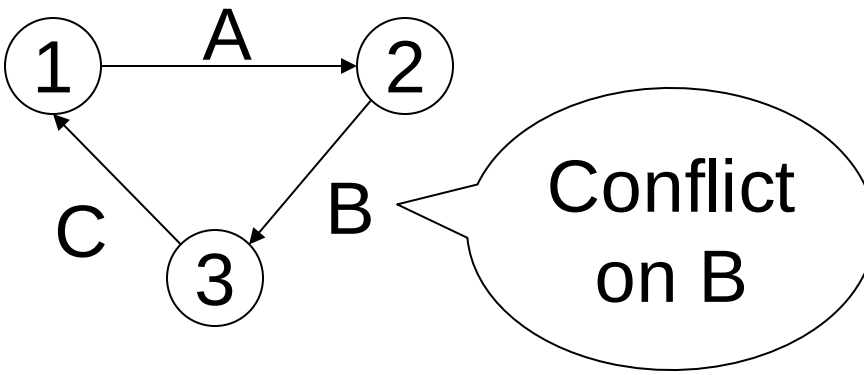
Theorem: 2PL ensures conflict serializability



$\ldots U_1(A) \ldots L_2(A) \ldots$

Two-Phase Locking

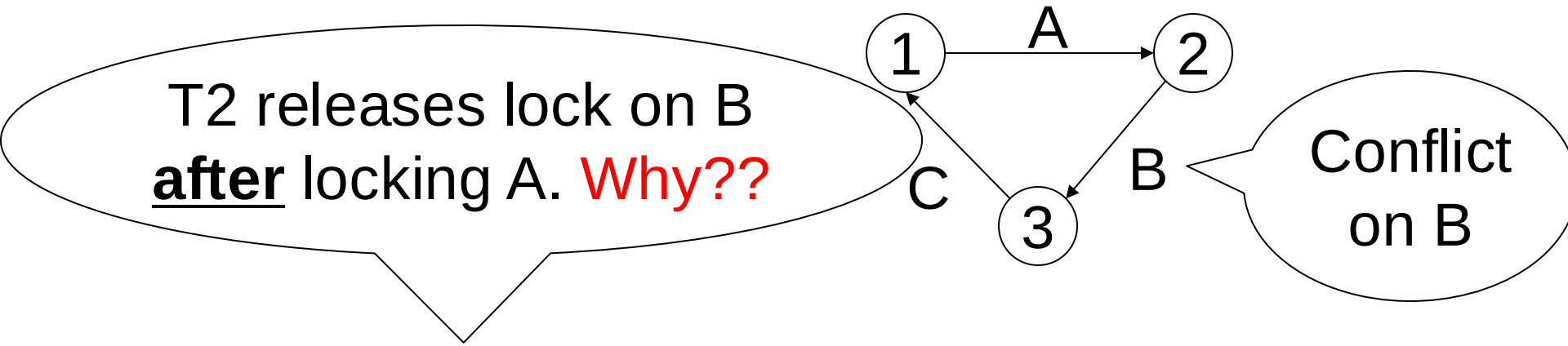
Theorem: 2PL ensures conflict serializability



…$U_1(A)…L_2(A)$…

Two-Phase Locking

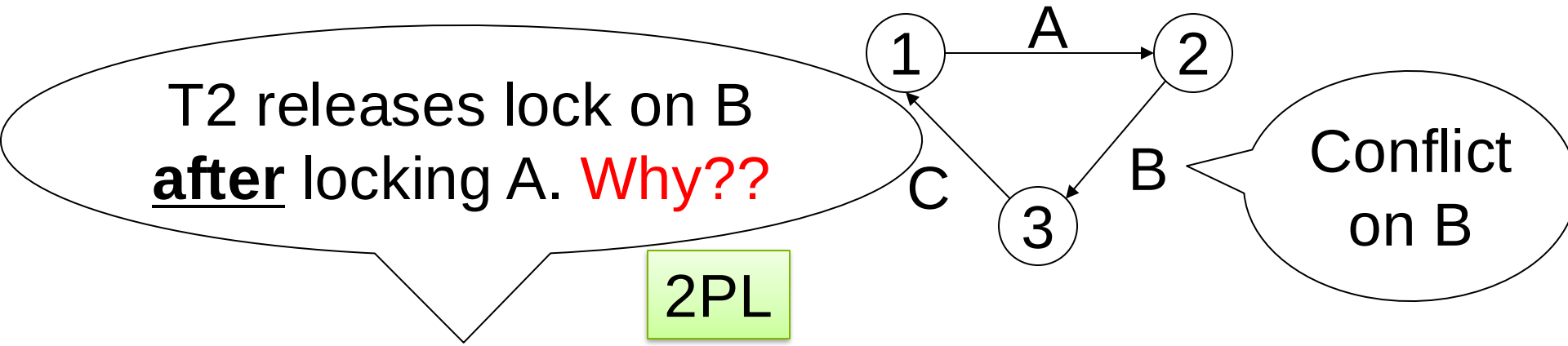
Theorem: 2PL ensures conflict serializability



$\ldots U_1(A) \ldots L_2(A) \ldots U_2(B) \ldots$

Two-Phase Locking

Theorem: 2PL ensures conflict serializability

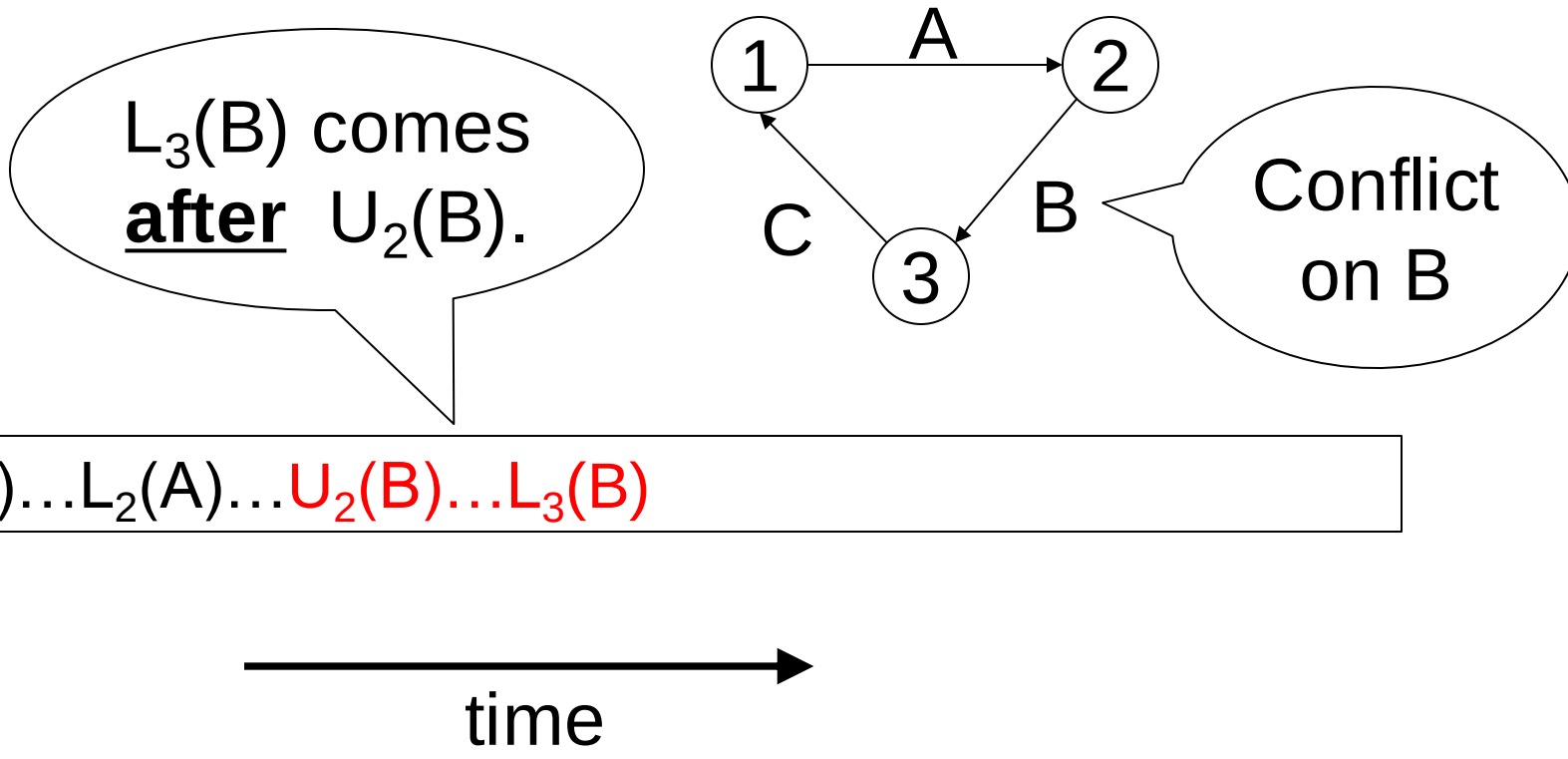


$\ldots U_1(A) \ldots L_2(A) \ldots U_2(B) \ldots$

time

Two-Phase Locking

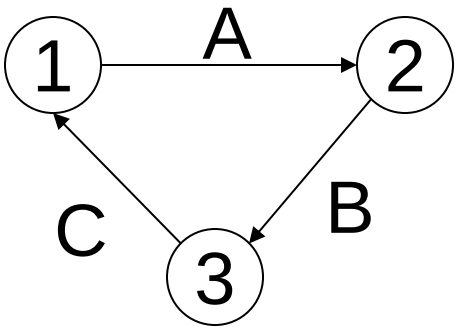
Theorem: 2PL ensures conflict serializability



Two-Phase Locking

Theorem: 2PL ensures conflict serializability

Continue the same reasoning…

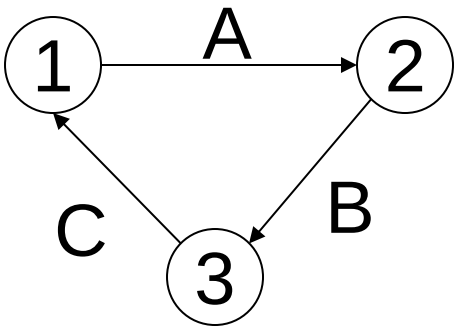


…$U_1(A)…L_2(A)…U_2(B)…L_3(B)…U_3(C)…$

Two-Phase Locking

Theorem: 2PL ensures conflict serializability

Continue the same reasoning…

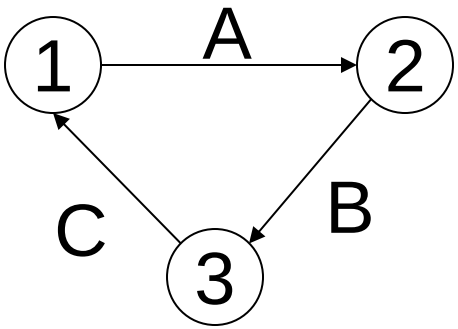


…$U_1(A)…L_2(A)…U_2(B)…L_3(B)…U_3(C)…L_1(C)…$

Two-Phase Locking

Theorem: 2PL ensures conflict serializability

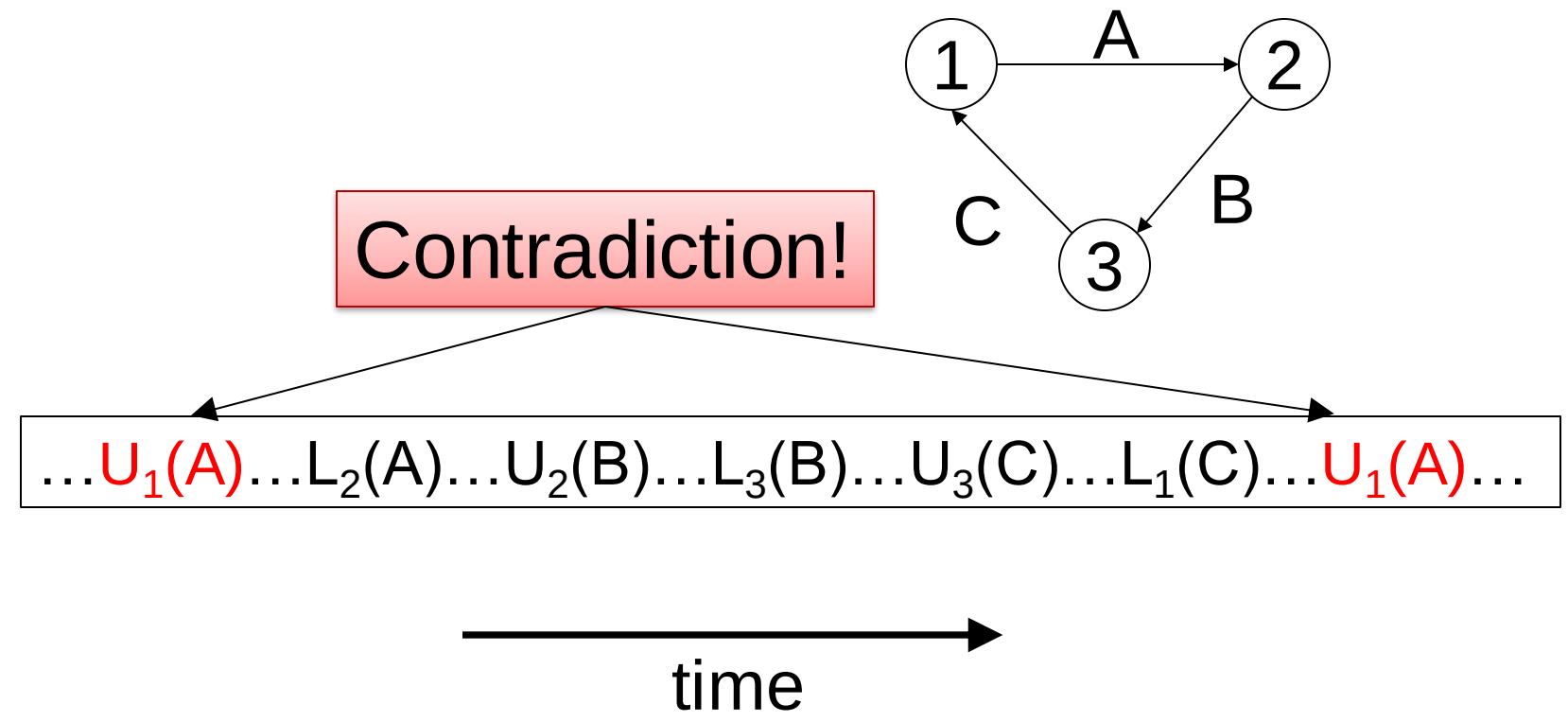
Continue the same reasoning…



…$U_1(A)…L_2(A)…U_2(B)…L_3(B)…U_3(C)…L_1(C)…U_1(A)…$

Two-Phase Locking

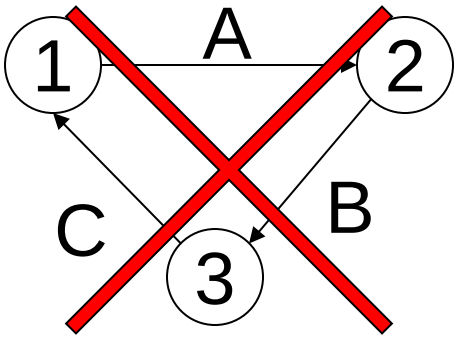
Theorem: 2PL ensures conflict serializability



Two-Phase Locking

Theorem: 2PL ensures conflict serializability

Assumption led to
a contradiction.
Hence, precedence
graph has no cycle.



Schedule is conflict serializable

A New Problem: Non-recoverable Schedule

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
	Commit
Rollback	

A New Problem: Non-recoverable Schedule

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
	Commit
Rollback	

Non-recoverable schedule

Strict 2PL

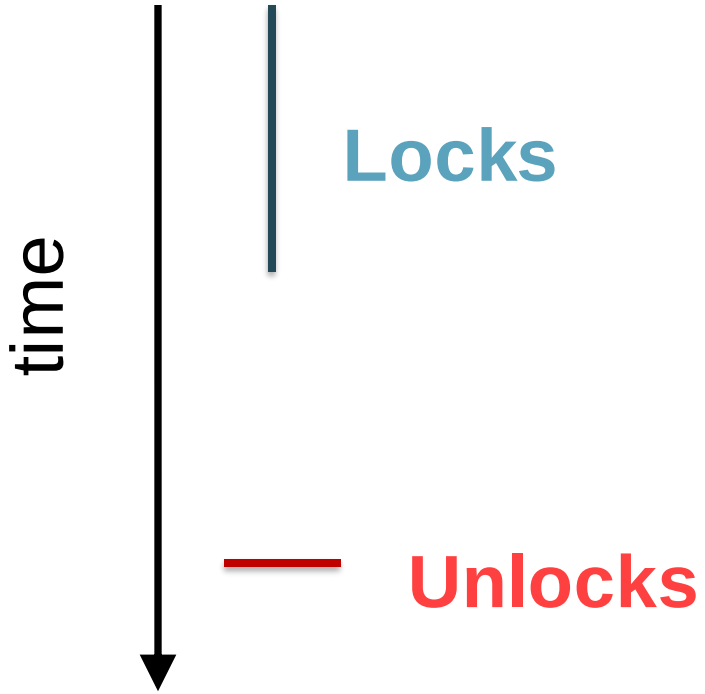
The Strict 2PL rule:

All unlocks are done *with* commit/abort.

Strict 2PL

T1	T2
$L_1(A)$; READ(A)	
A :=A+100	
WRITE(A);	
	$L_2(A)$; BLOCKED…
$L_1(B)$; READ(B)	
B :=B+100	
WRITE(B);	
Rollback & $U_1(A)$;$U_1(B)$;	
	…GRANTED; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B);
	Commit & $U_2(A)$; $U_2(B)$;

Strict 2PL



Theorem Strict 2PM ensures both conflict-serializable and recoverable schedule

Strict 2PL

- Lock-based systems always use strict 2PL
- Easy to implement:
  - READ(X) or WRITE(X) request: insert L(X)
  - COMMIT/ROLLBACK: insert U(X), U(Y), ...
- Conflict-serializable and avoids-cascading aborts

Deadlocks

T_1: L(A), R(A), L(B),…. W(B)

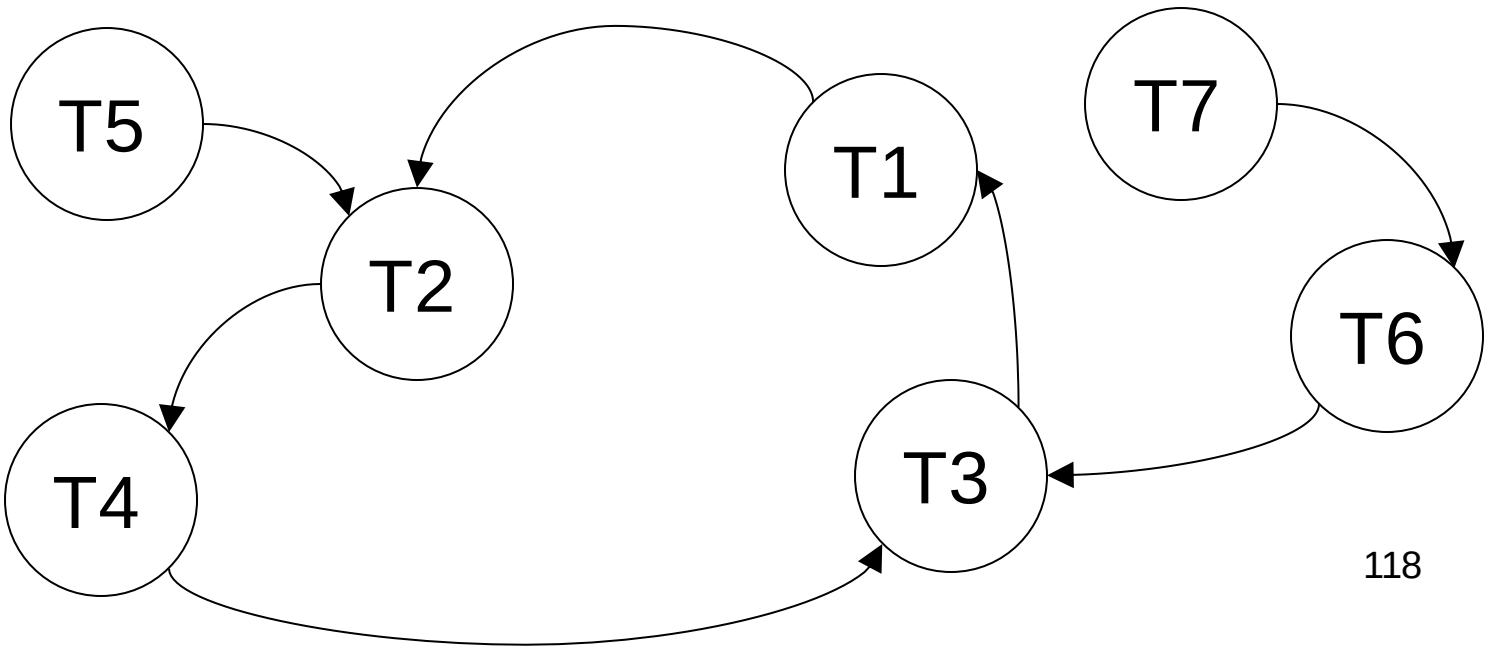
T_2: L(B), R(B), L(A),…. W(A)

This is a deadlock!

Deadlocks

- Deadlock = when *waits-for* graph has a cycle

- Check the graph periodically; if deadlock is detected then pick a txn T and abort it; recheck more often.



Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)

Lock compatibility matrix:

	None	S	X
None			
S			
X			

Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)

Lock compatibility matrix:

	None	S	X
None	✓	✓	✓
S	✓	✓	✗
X	✓	✗	✗

Discussion

- Shared locks allow reads to proceed
- Issues:
  - Deadlocks more likely:
    T1, T2 both acquire S(A), then both want to escalate their lock to X(A): deadlock
  - Need to avoid starvation:
    T1, T2, T3, … read element A and commit
    T0 waits forever for X(A)

Optimistic concurrency control

Optimistic CC

- Proceeds more aggressively, but more likely to require abort
- Three main abstractions:
  - Timestamps
  - Multiversions
  - Validation
- Will illustrate them separately

Timestamps

Timestamps

- Each transaction receives a unique timestamp TS(T)

Could be:

- The system's clock
- A unique counter, incremented by the scheduler

Timestamps

Main invariant:

> The timestamp order defines
> the serialization order of the transaction

Will generate a schedule that is view-equivalent to a serial schedule, and strict

Timestamps

With each element X, associate

- RT(X) = the highest timestamp of any transaction U that read X
- WT(X) = the highest timestamp of any transaction U that wrote X
- C(X) = the commit bit: true when transaction with highest timestamp that wrote X committed

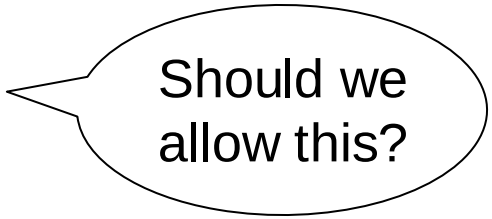
Confusing Notation

- $r_T(X)$ = txn T reads element X
- RT(X) = the “read timestamp” of X
- TS(T) = the ”timestamp” of txn T

Simplified TS

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$w_U(X)$ . . . $r_T(X)$
$r_U(X)$ . . . $w_T(X)$
$w_U(X)$ . . . $w_T(X)$

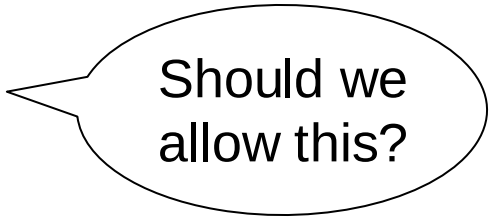


- Similarly for the other cases

Simplified TS

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$w_U(X)$ . . . $r_T(X)$



Simplified TS

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$w_U(X)$ . . . $r_T(X)$

Should we allow this?

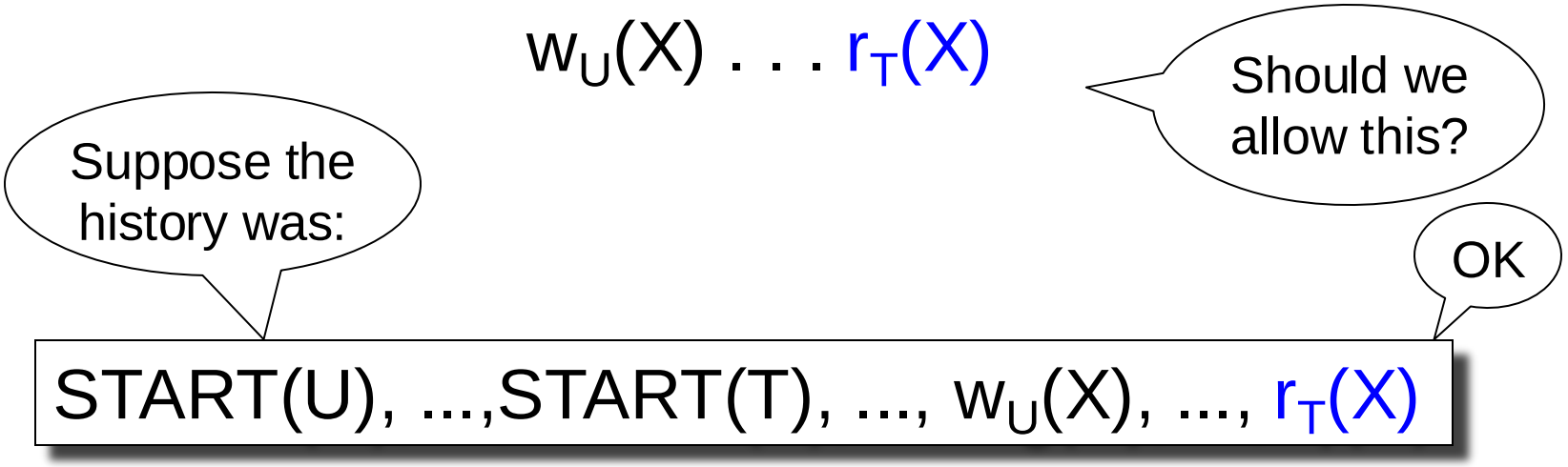
Suppose the history was:

START(U), ...,START(T), ..., $w_U(X)$, ..., $r_T(X)$

Simplified TS

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

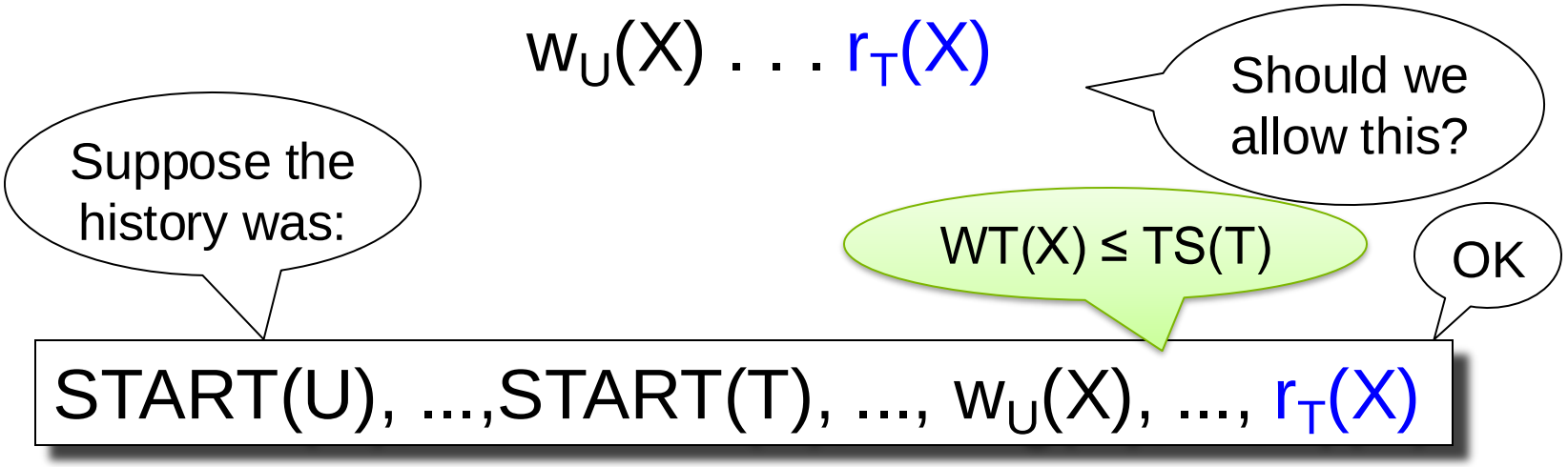
$w_U(X)$ . . . $r_T(X)$



Simplified TS

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

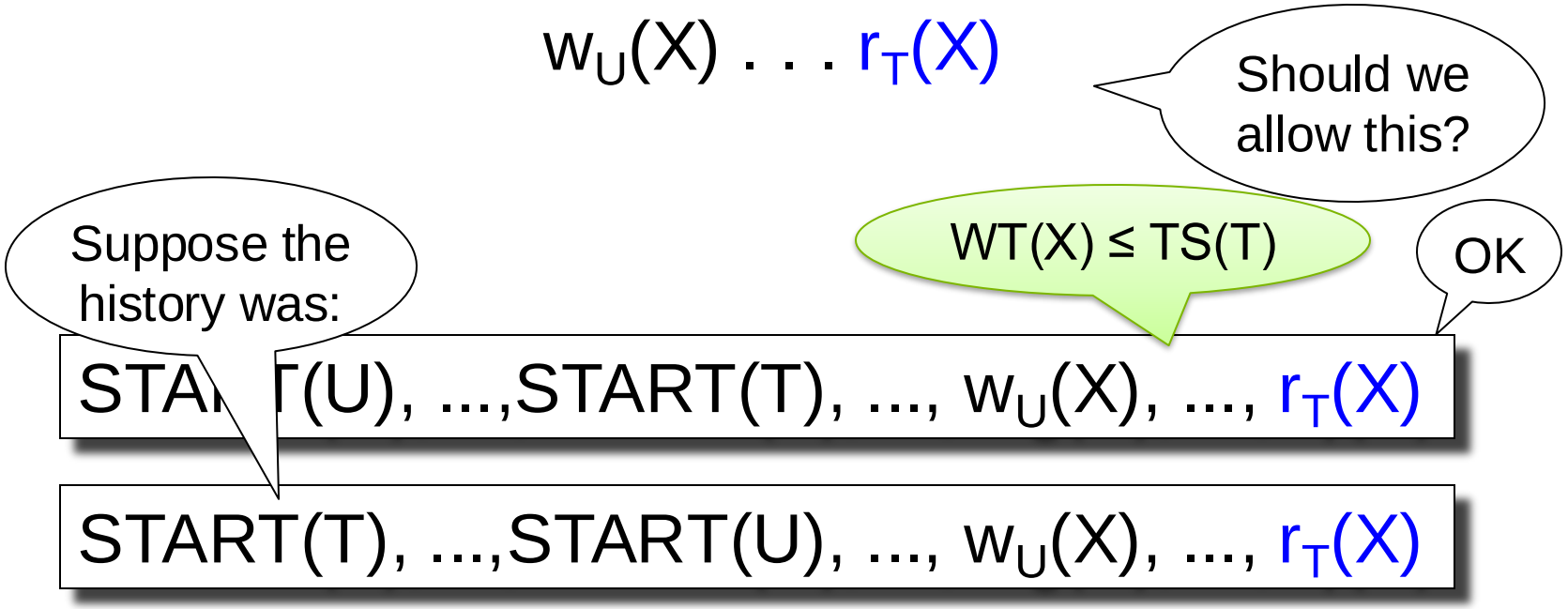
$w_U(X)$ . . . $r_T(X)$



Simplified TS

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

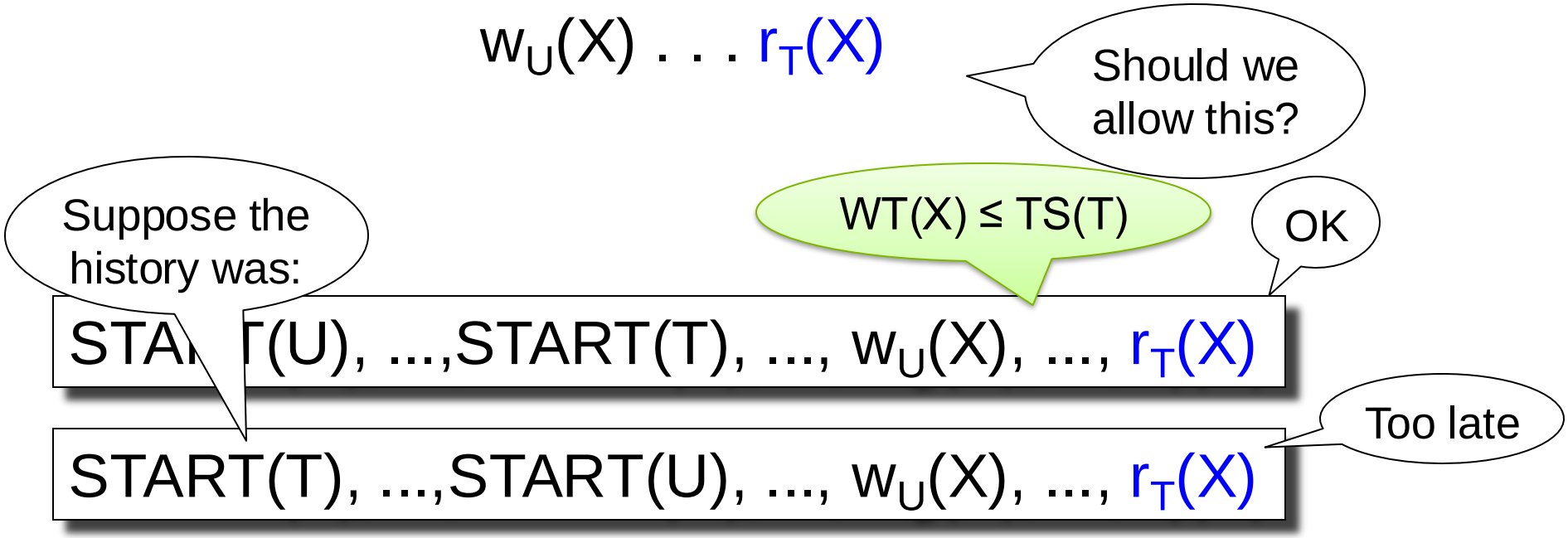
$w_U(X)$ . . . $r_T(X)$



Simplified TS

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

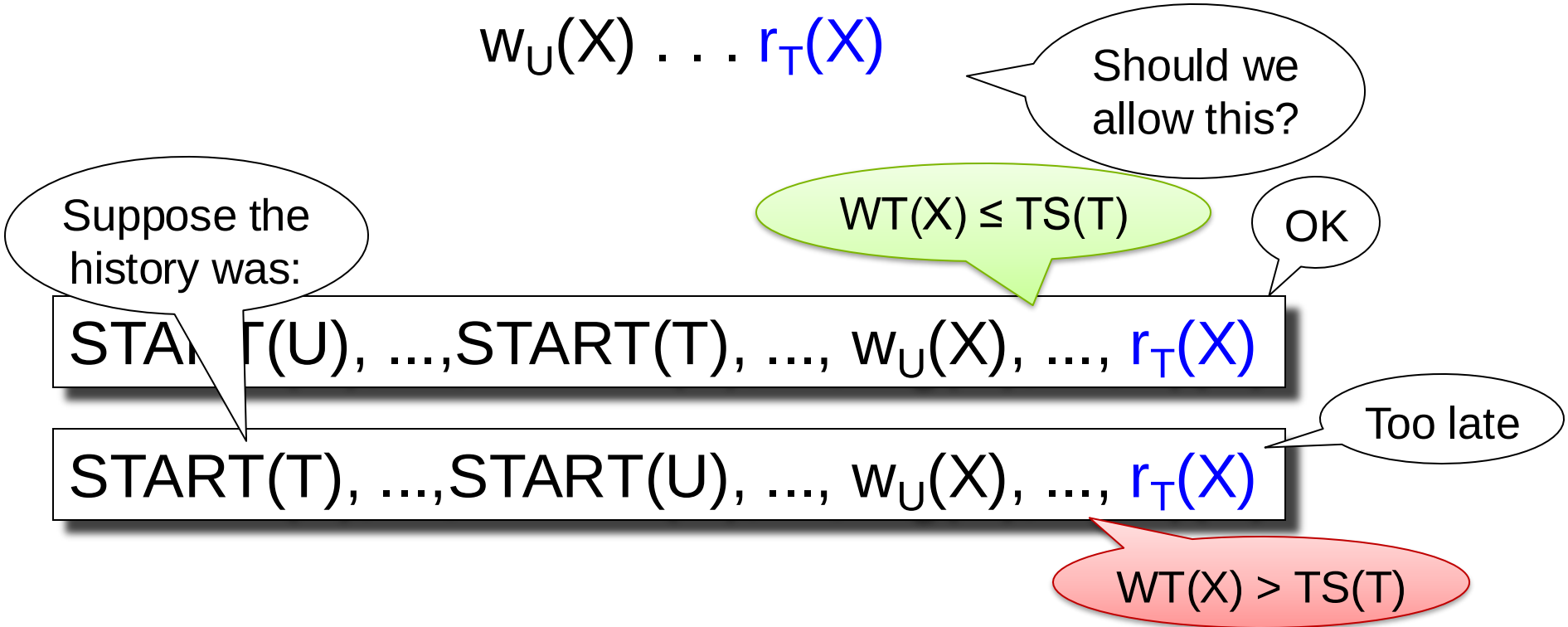
$w_U(X)$ . . . $r_T(X)$



Simplified TS

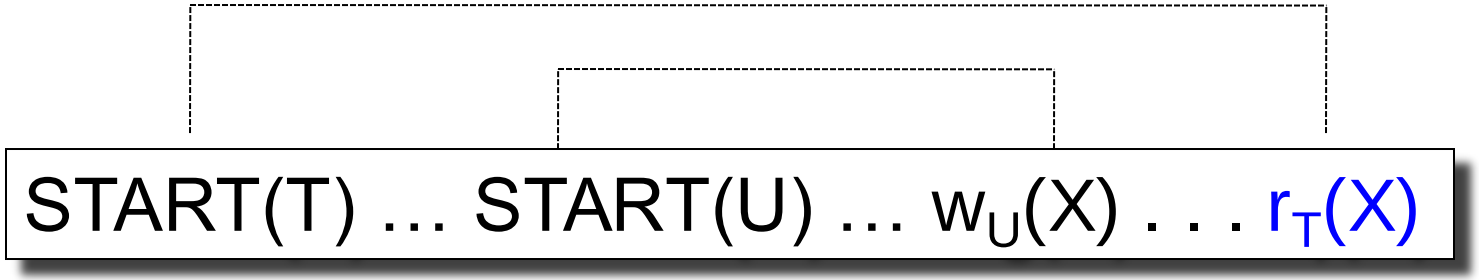
- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$w_U(X)$ . . . $r_T(X)$



Summary for $r_T(X)$

- If WT(X) > TS(T): "read too late"

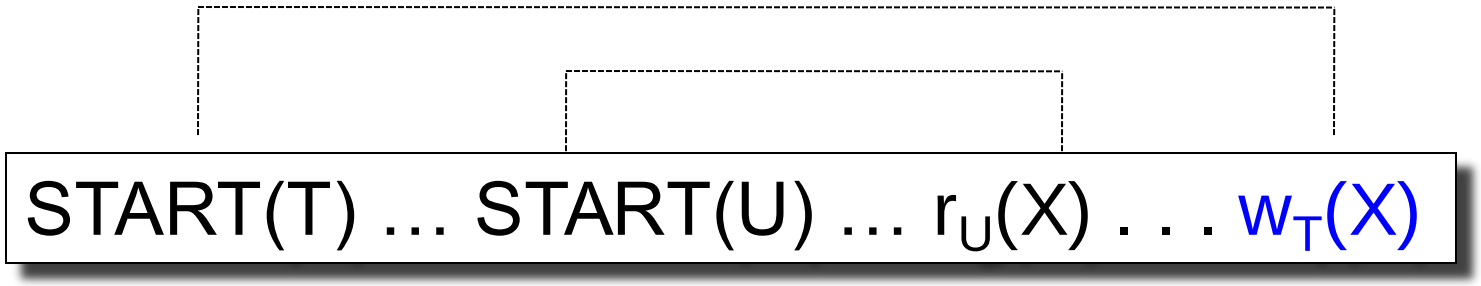


Rollback T

- Otherwise proceed

Summary for $w_T(X)$

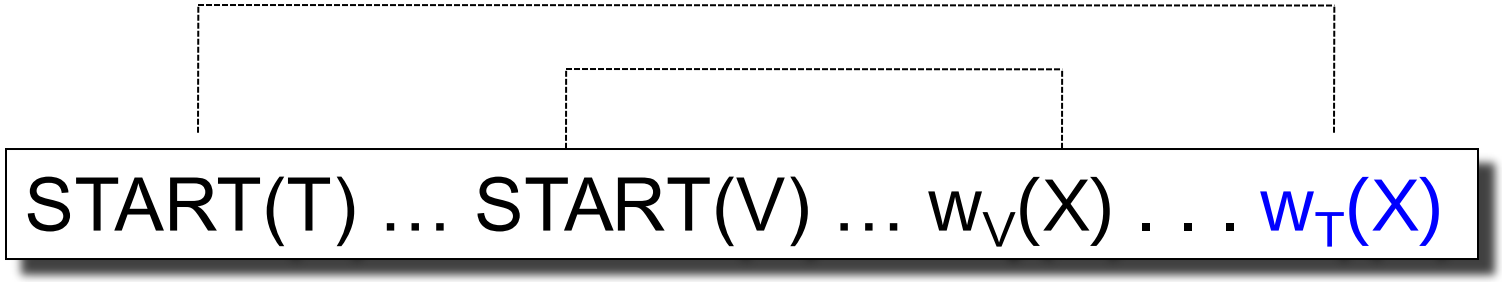
- If RT(X) > TS(T)



- Otherwise… (next slide!)

Summary for $w_T(X)$

- If RT(X) ≤ TS(T) and WT(X) > TS(T)



Don’t write X at all !
Thomas’ rule

- Otherwise, proceed

Simplified TS

Only for transactions that do not abort

Request is $r_T(X)$

If WT(X) > TS(T) then ROLLBACK

Else READ and update RT(X) to larger of TS(T) or RT(X)

Request is $w_T(X)$

If RT(X) > TS(T) then ROLLBACK

Else if WT(X) > TS(T) ignore write & continue (Thomas Write Rule)

Otherwise, WRITE and update WT(X) =TS(T)

Simplified TS

- **Fact**: the simplified timestamp-based scheduling with Thomas' rule ensures that the schedule is view-serializable

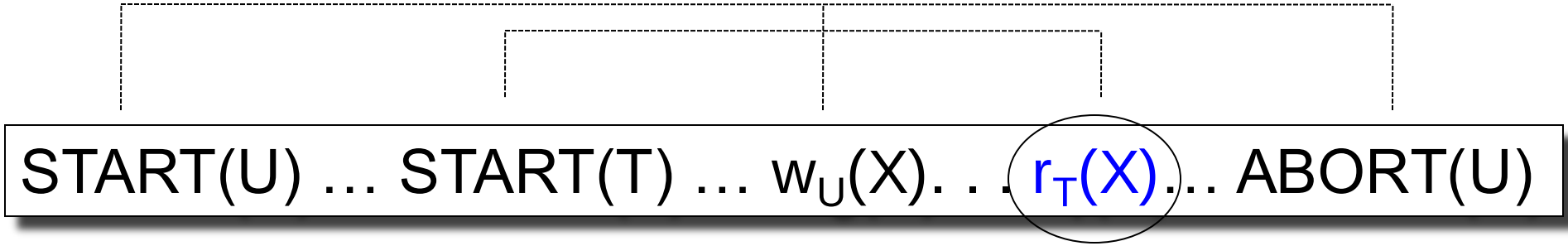
Full TS

- Use the commit bit C(X) to keep track if the transaction that last wrote X has committed

Full TS

Read dirty data:

- T wants to read X, and WT(X) < TS(T)
- Seems OK, but…

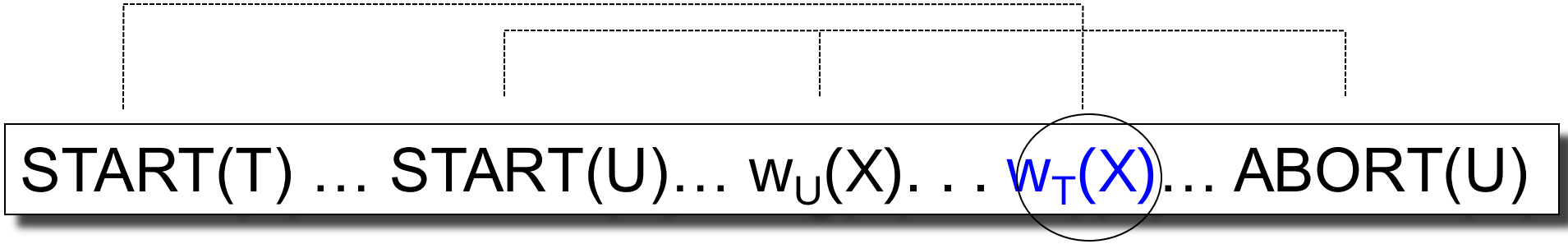


If C(X)=false, T needs to wait for it to become true

Full TS

Thomas' rule needs to be revised:

- T wants to write X, and WT(X) > TS(T)
- Seems OK not to write at all, but …



If C(X)=false, T needs to wait for it to become true

Full TS

Request is $r_T(X)$
If WT(X) > TS(T) then ROLLBACK
Else If C(X) = false, then WAIT
Else READ and update RT(X) to larger of TS(T) or RT(X)

Request is $w_T(X)$
If RT(X) > TS(T) then ROLLBACK
Else if WT(X) > TS(T)
Then If C(X) = false then WAIT
else IGNORE write (Thomas Write Rule)
Otherwise, WRITE, and update WT(X)=TS(T), C(X)=false

Full TS

- Fact: full timestamp-based scheduling is view-serializable and avoids cascasing aborts

Timestamps

Main takeaway:

- TS defines the serialization order
- Scheduler decides whether to allow operation to proceed, or wait, or abort

Multiversion

Multiversion Timestamp

- Problem: when T requests $r_T(X)$ but WT(X) > TS(T), then T must abort

- Solutions: keep multiple versions of X:

$$X_t, X_{t-1}, X_{t-2}, \ldots$$

$$TS(X_t) > TS(X_{t-1}) > TS(X_{t-2}) > \ldots$$

- Let T read an older version, with appropriate timestamp

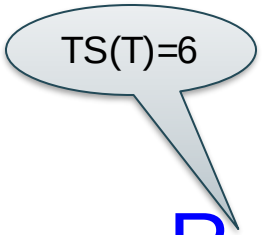
Details

- When $w_T(X)$ occurs,
  create a new version, denoted X_t where t = TS(T)

- When $r_T(X)$ occurs,
  find most recent version X_t such that t <= TS(T)
  Notes:
  - $WT(X_t)$ = t and it never changes
  - $RT(X_t)$ must still be maintained to check legality of writes

- Can delete X_t if we have a later version X_{t1} and all active transactions T have TS(T) > t1

Example (in class)



X_3 X_9 X_{12} X_{18}

$R_6(X)$ -- what happens?

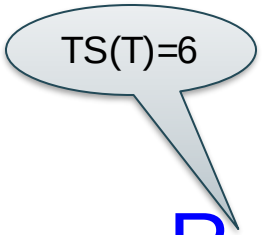
$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)



X_3 X_9 X_{12} X_{18}

$R_6(X)$ -- what happens? Return X_3

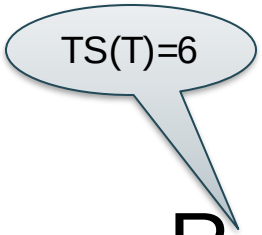
$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)



X_3 X_9 X_{12} X_{18}

$R_6(X)$ -- what happens? Return X_3

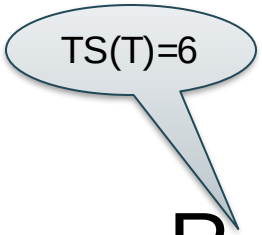
$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)



X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

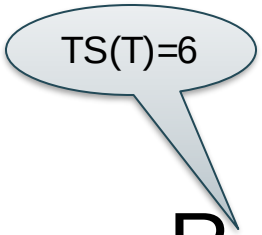
$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)



X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

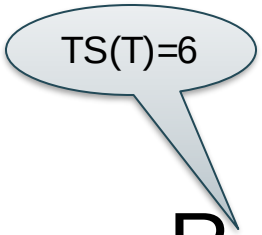
$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)



X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

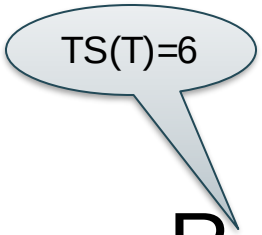
$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens? Return X_{14}

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)



X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

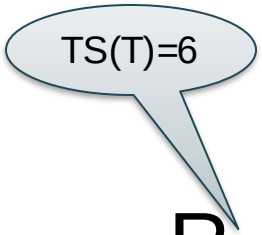
$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens? Return X_{14}

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)



X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

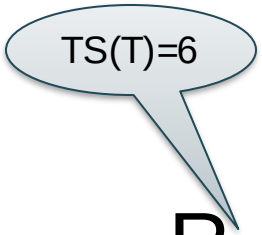
$R_{15}(X)$ – what happens? Return X_{14}

$W_5(X)$ – what happens? ABORT

Because of $R_6(X)$

When can we delete X_3?

Example (in class)

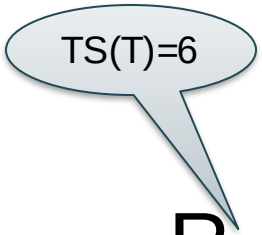


X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens? Return X_{14}
$W_5(X)$ – what happens? ABORT

When can we delete X_3?

Example (in class)



X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens? Return X_{14}

$W_5(X)$ – what happens? ABORT

When can we delete X_3? When min TS(T)≥ 9

Multiversion

Takeaways:

- Reduces the number of aborts due to late reads
- Simplifies rollback

Validation

Concurrency Control by Validation

- TXN reads elements, performs all updates on local copies

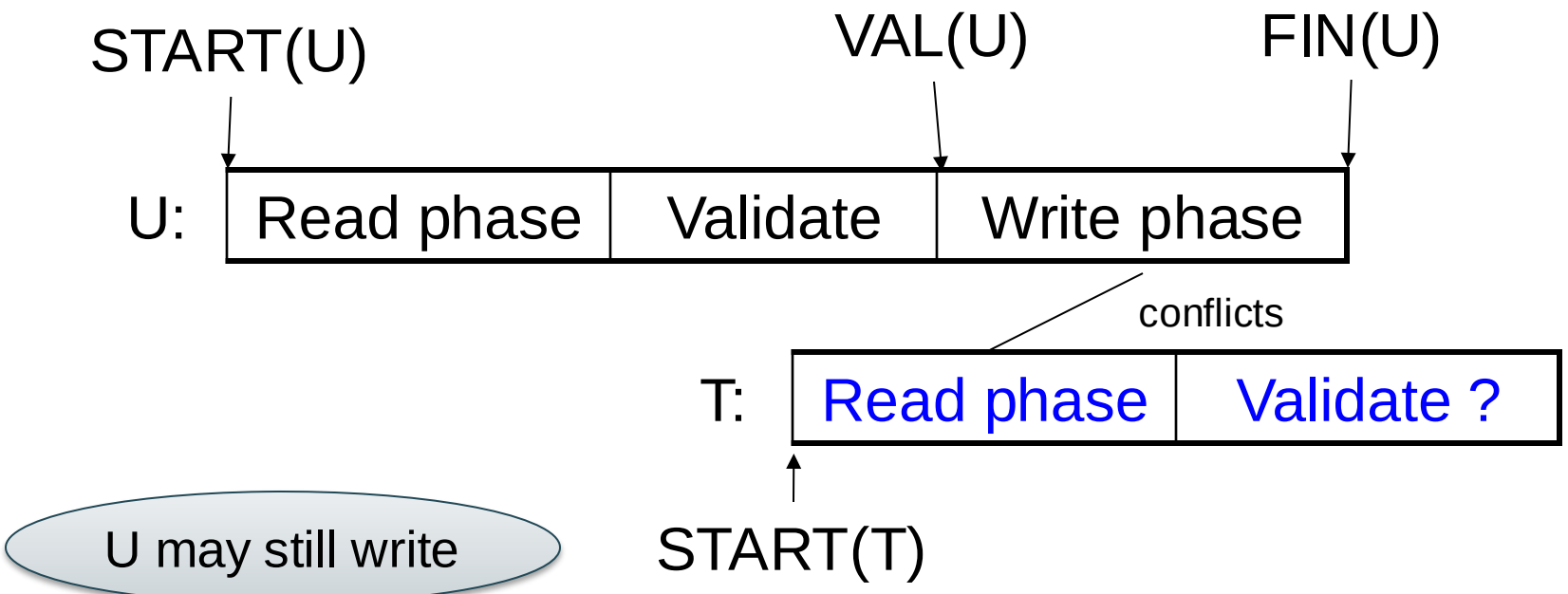
- At commit time:
  - CC manager performs *validation*
  - If OK, then it writes the local copies to disk
  - If not OK then aborts

Concurrency Control by Validation

- Each transaction T defines:
  - a *read set* RS(T) and
  - a *write set* WS(T)
- Each TXN has three phases:
  - Read elements RS(T): Time = START(T)
  - Validate: Time = VAL(T)
  - Writes elements WS(T). Time = FIN(T)

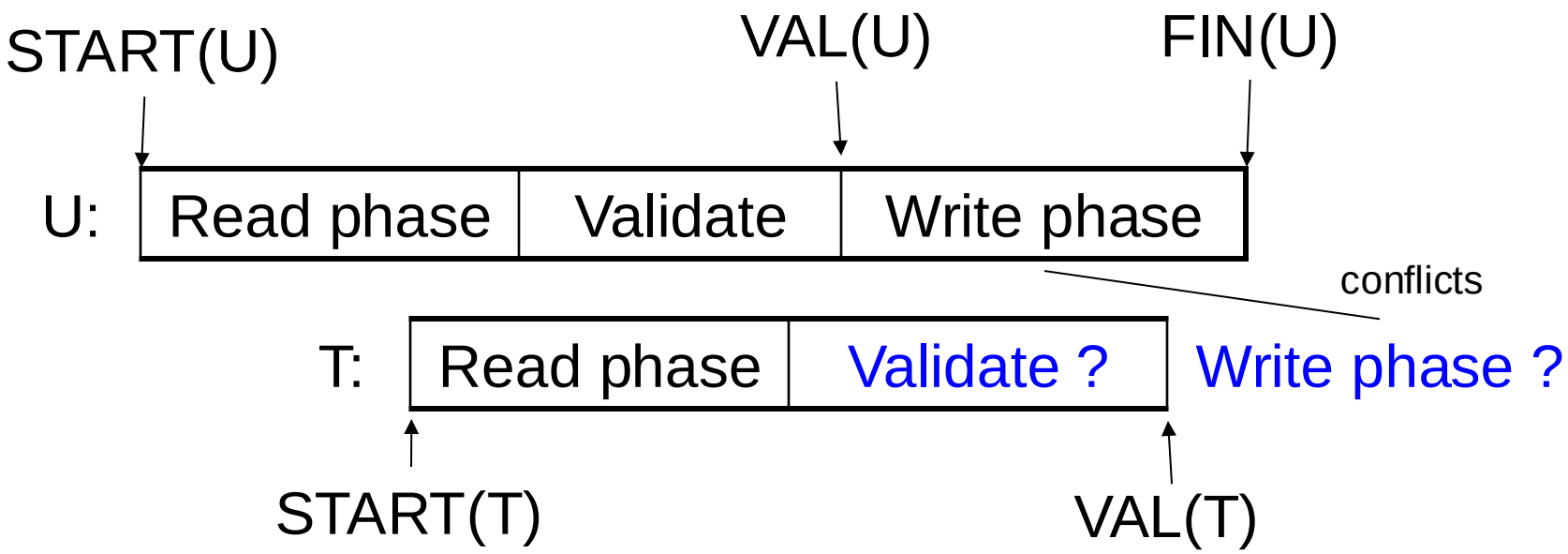
Main invariant: the serialization order is VAL(T)

Avoid $r_T(X)$ - $w_U(X)$ Conflicts



If FIN(U) > START(T) and RS(T) ∩ WS(U)
Then ROLLBACK(T)

Avoid $w_T(X)$ - $w_U(X)$ Conflicts



If FIN(U) > VAL(T) and WS(T) ∩ WS(U)
Then ROLLBACK(T)

Validation

Takeaways:

- READs/WRITEs proceed without delay
- Only delay happens at validation time
- May abort aggressively

Snapshot Isolation (SI)

Snapshot Isolation (SI)

A variant of multiversion/validation

- Very efficient, and very popular
- Warning: classical SI is not serializable

Snapshot Isolation Rules

- TXN T receives timestamp TS(T)
- Sees snapshot of DB at time TS(T)
- T writes in its local memory only
- When T commits, updates are written to disk
- Write/write conflicts: “first committer wins” rule
- Read/write conflicts are ignored

Snapshot Isolation (Details)

- Multiversion concurrency control:
  - Versions of X: X_{t1}, X_{t2}, X_{t3}, . . .

- When T reads X, return X_t,
  where t is max s.t. t ≤ TS(T)

- When T writes X:
  if other transaction updated X, abort
  Otherwise: new version $X_{TS(T)}$

What Works and What Not

- Reads are never delayed
- No dirty reads, no inconsistent reads
- No lost updates (“first committer wins”)
- But: read-write conflicts! Write skew

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
         then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
         then X = -50; WRITE(X)
   COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
         then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
         then X = -50; WRITE(X)
   COMMIT
```

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
         then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
         then X = -50; WRITE(X)
   COMMIT
```

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
         then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
         then X = -50; WRITE(X)
   COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
        then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
        then X = -50; WRITE(X)
   COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

X_0 Y_0

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
        then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
        then X = -50; WRITE(X)
   COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

$X_0 \quad Y_0$

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
        then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
        then X = -50; WRITE(X)
   COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

$X_0 \quad Y_0 \quad Y_1$

Should have aborted T1, but SI doesn't keep RT(Y)

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
        then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
        then X = -50; WRITE(X)
   COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

$X_0 \quad Y_0 \quad Y_1 \quad X_2$

Should have aborted T1, but SI doesn't keep RT(Y)

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
        then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
        then X = -50; WRITE(X)
   COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

Should have aborted T1, but SI doesn't keep RT(Y)

$X_0 \quad Y_0 \quad Y_1 \quad X_2$

Write Skew

X=my income
Y=my spending

Invariant: X + Y >= 0

```
T1:  // spend more
   READ(X);
   if X >= 50
        then Y = -50; WRITE(Y)
   COMMIT
```

```
T2:  // earn less
   READ(Y);
   if Y >= 50
        then X = -50; WRITE(X)
   COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

Should have aborted T1, but SI doesn't keep RT(Y)

$X_0 \quad Y_0 \quad Y_1 \quad X_2$

Starting with X=50,Y=50, we end with X=-50, Y=-50.
Non-serializable !!!

Discussions

- Snapshot isolation (SI) is like repeatable reads but also avoids some (not all) phantoms
- If DBMS runs SI and the app needs serializable:
  - use dummy writes for all reads to create write-write conflicts… but that is confusing for developers
- Extension of SI to make it serializable is implemented in postgres

Phantoms

Phantom Problem

- So far we have assumed the database to be a *static* collection of elements (=tuples)

- If tuples are inserted/deleted then the *phantom problem* appears

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

Is this schedule serializable ?

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

Is this schedule serializable ?

No: T1 sees a "phantom" product A3

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

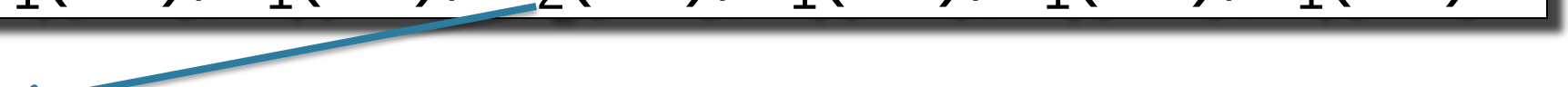
$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$



$W_2(A3);R_1(A1);R_1(A2);R_1(A1);R_1(A2);R_1(A3)$

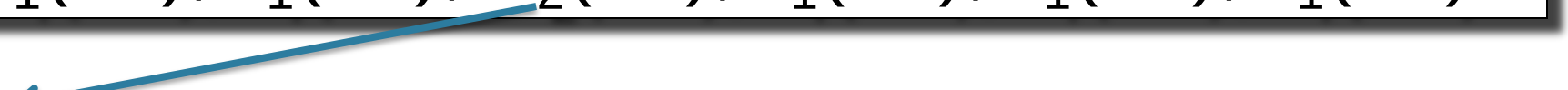
Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

But this is conflict-serializable!

$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$



$W_2(A3);R_1(A1);R_1(A2);R_1(A1);R_1(A2);R_1(A3)$

Phantom Problem

- A “phantom” is a tuple that is
  invisible during part of a transaction execution
  but not invisible during the entire execution

- In our example:
  - T1: reads list of products
  - T2: inserts a new product
  - T1: re-reads: a new product appears !

Phantom Problem

- In a ***static*** database:
  - Conflict serializability implies serializability
- In a ***dynamic*** database, this may fail due to phantoms
- Strict 2PL guarantees conflict serializability, but not serializability

Dealing With Phantoms

- Lock-based CCs:
  - Place a lock on entire table
  - Expensive
- Multiversion-based CC:
  - Resolved by keeping track of the version

Summary of Serializability

- **Static database**:
  - Conflict serializability implies serializability
- **Dynamic database**:
  - Conflict serializability plus phantom management implies serializability

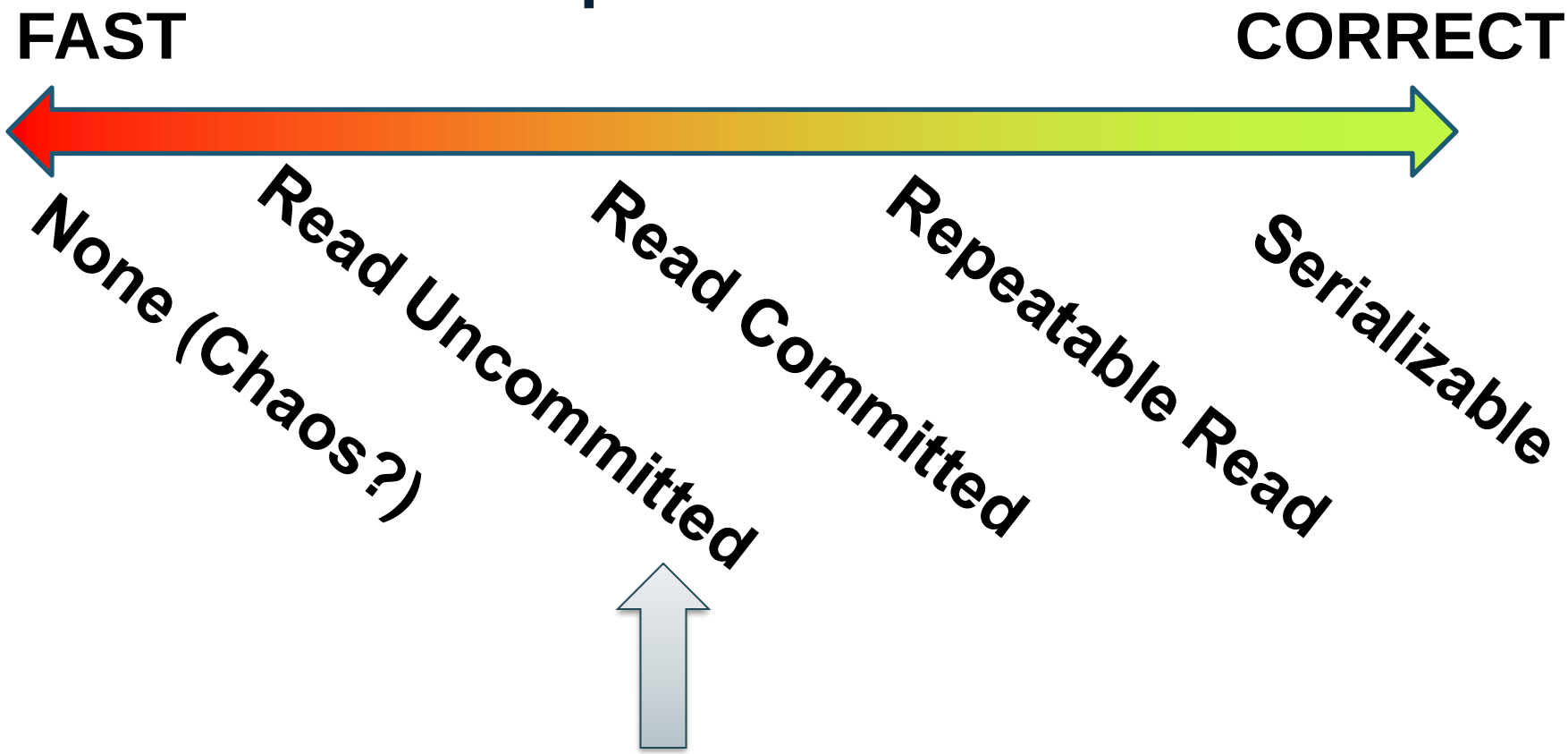
Weaker Isolation Levels

Isolation Level Design Spectrum

Isolation Levels

- SET TRANSACTION ISOLATION LEVEL ...
  - READ UNCOMMITED
  - READ COMMITED
  - REPEATABLE READ
  - SERIALIZABLE
  - SNAPSHOT ISOLATION (MVCC)

Isolation Level Design Spectrum



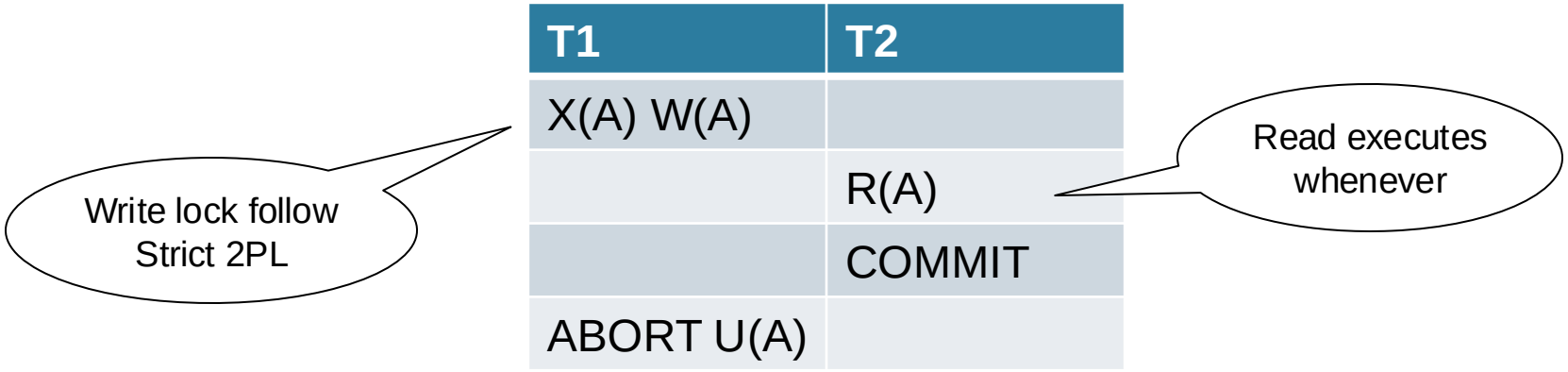
READ UNCOMMITTED

- Writes → Strict 2PL write locks
- Reads → No locks needed

T1	T2
X(A) W(A)	
	R(A)
	COMMIT
ABORT U(A)	

READ UNCOMMITTED

- Writes → Strict 2PL write locks
- Reads → No locks needed

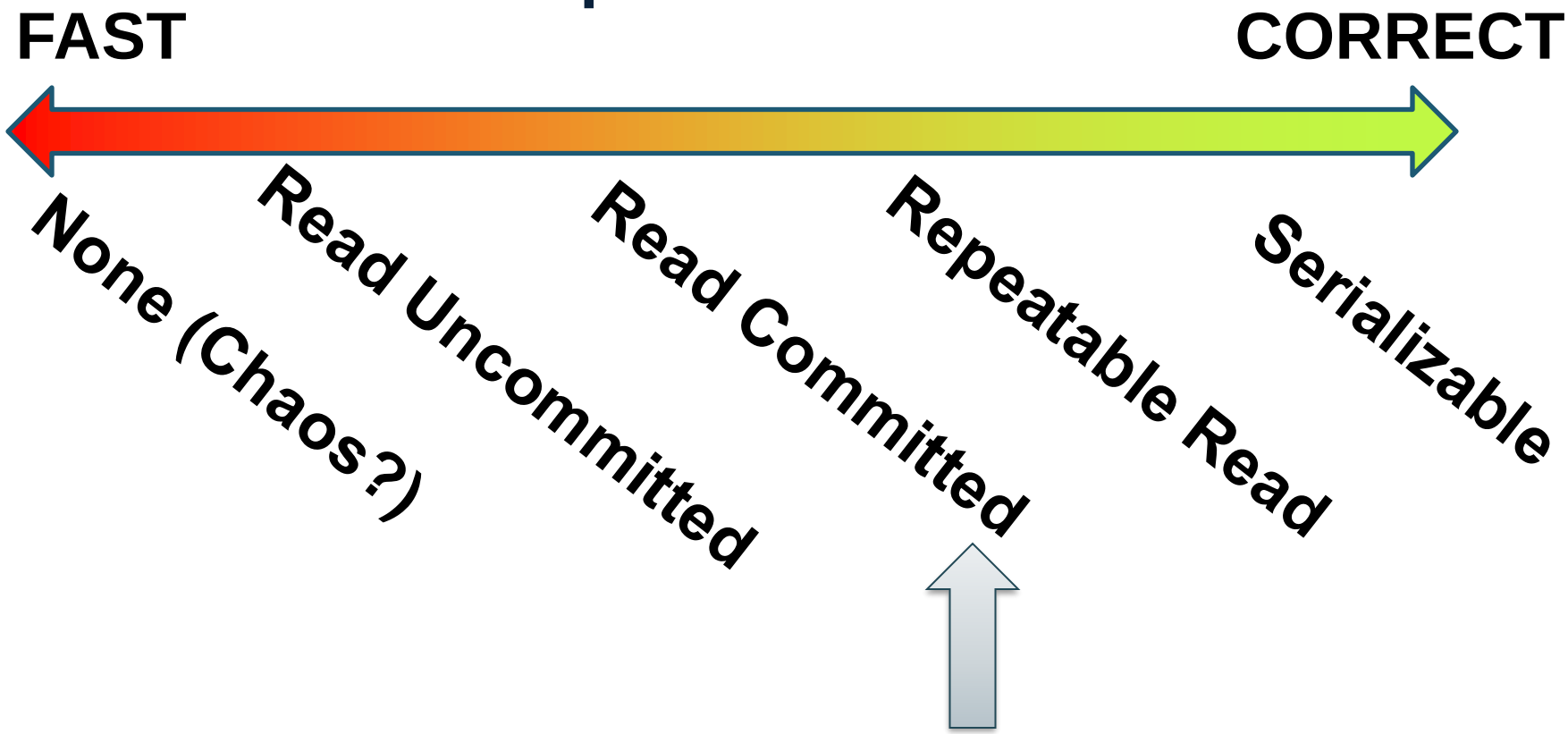


READ UNCOMMITTED

Reads never wait! Use cases:

- Static data (few or no writes after data initialization)
- Read coverage/accuracy is not mission critical

Isolation Level Design Spectrum



READ COMMITTED

- Writes → Strict 2PL write locks
- Reads → Short-duration read locks

READ COMMITTED

- Writes → Strict 2PL write locks
- Reads → Short-duration read locks

T1	T2
X(A) W(A)	
	R(A)
	COMMIT
ABORT U(A)	

READ COMMITTED

- Writes → Strict 2PL write locks
- Reads → Short-duration read locks

T1	T2
X(A) W(A)	
	R(A)
	COMMIT
ABORT U(A)	

T1	T2
X(A) W(A)	
	S(A) blocked…
ABORT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

READ COMMITTED

- But non-repeatable reads possible.

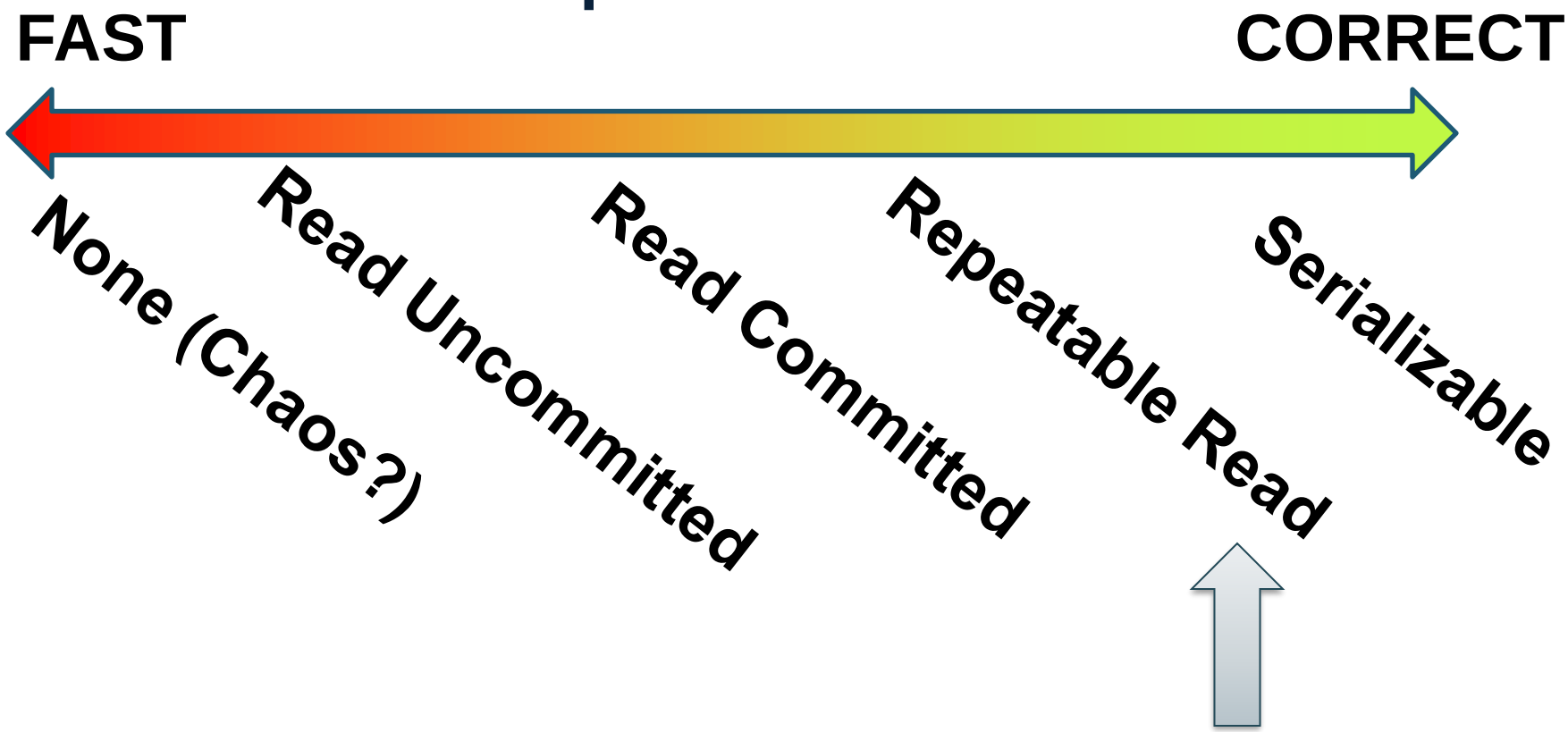
T1	T2
	R(A)
W(A)	
COMMIT U(A)	
	R(A)
	COMMIT U(A)

READ COMMITTED

- But non-repeatable reads possible.

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	X(A)
	W(A)
	COMMIT U(A)

Isolation Level Design Spectrum



REPEATABLE READ

- Read/Writes → Strict 2PL locks

REPEATABLE READ

- Read/Writes → Strict 2PL locks

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

REPEATABLE READ

- Read/Writes → Strict 2PL locks

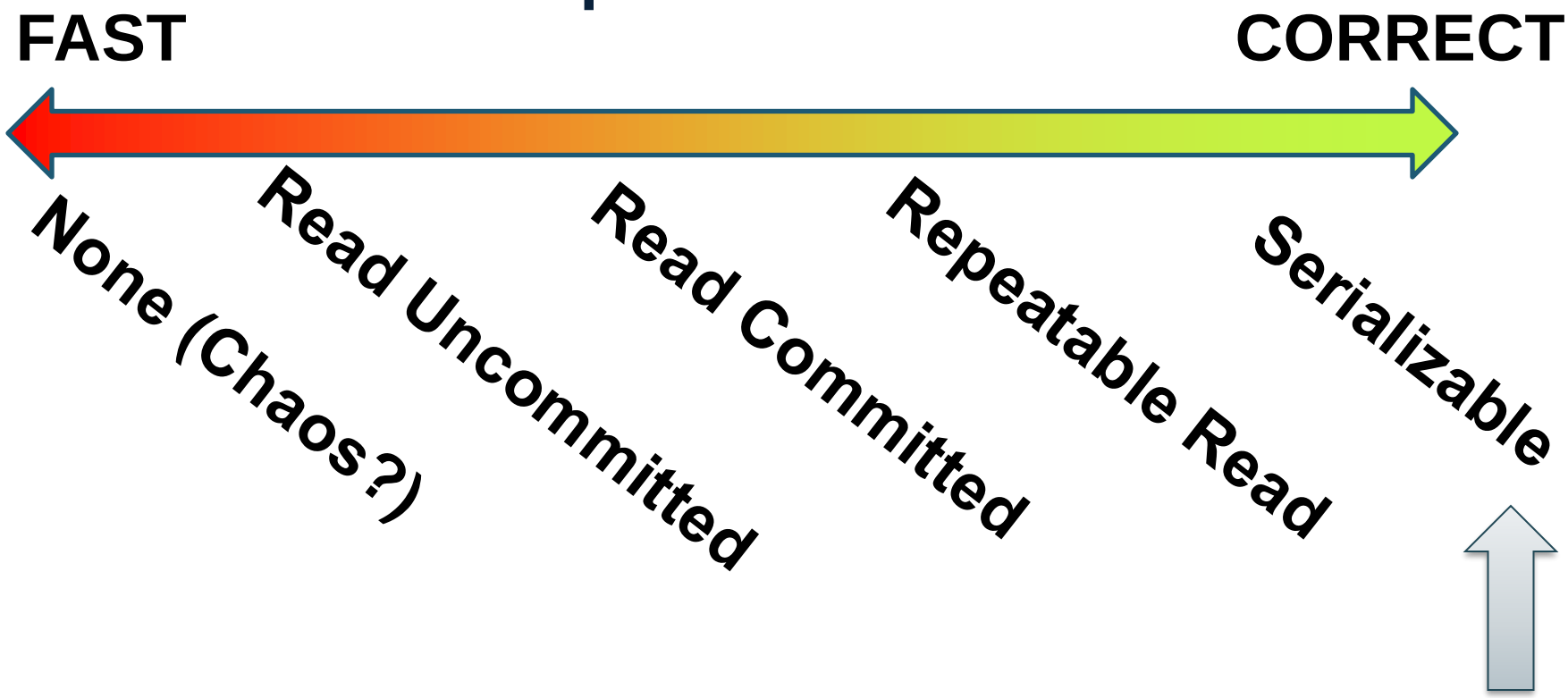
T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…granted X(A)	U(A)
	S(A) blocked…
W(A)	…
COMMIT U(A)	…granted S(A)
	R(A)
	COMMIT U(A)

T1	T2
	S(A)
X(A) blocked…	
…	**R(A)**
…	**R(A)**
…granted X(A)	COMMIT U(A)
W(A)	
COMMIT U(A)	

REPEATABLE READ

- Conflict serializability
- Use cases: few insert/deletes

Isolation Level Design Spectrum



SERIALIZABLE Level

- Write Lock → Strict 2PL
- Read Lock → Strict 2PL
- Locks on tables to handle phantom problem

Final Thoughts on Transactions

Enforcing ACID properties slows down RDBMs

- Concurrency (I): need to wait, need to abort
- Recovery (A): need to double write to the log

Typical throughput: 1000 – 10000 TPS

Lock Performance