CS 6400 A

Database Systems Concepts and Design

Lecture 19

11/04/26

Characterizing Schedules based on Serializability (1)

Serial schedule

- A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
  - Basically, actions from different transactions are NOT interleaved
  - Otherwise, the schedule is called nonserial schedule.

Serializable schedule

- A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

Characterizing Schedules based on Serializability (2)

Conflict equivalent

- Two conflict equivalent schedules have the same effect on a database
- All pairs of conflicting actions are in same order
- one schedule can be obtained from the other by swapping “non-conflicting” actions
  - either on two different objects
  - or both are read on the same object

Conflict serializable

- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.

Testing for conflict serializability

Through a precedence graph:

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a precedence graph (serialization graph) - a graph with directed edges
- An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is conflict-serializable if and only if the precedence graph has no cycles.

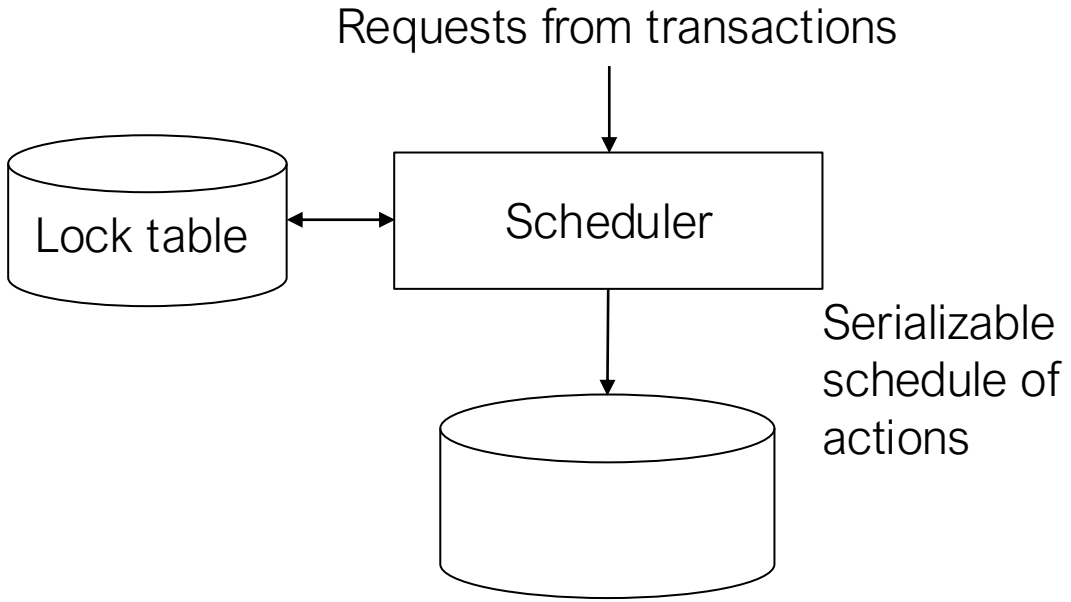
2. Lock-based Concurrency Control

Enforce serializability with locks

$l_i(X)$: Ti requests lock on X
$u_i(X)$: Ti releases lock on X

Consistency of transactions
- Can only read/write element if granted a lock
- A locked element must later be unlocked

Legality of schedules
- No two transactions may lock element at the same time



Enforce serializability with locks

- Legal, but not serializable schedule

T1	*T2*	*A*	*B*
		25	25
$l_1(A)$; $r_1(A)$;			
$A := A+100$			
$w_1(A)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$		
	$A := A*2$		
	$w_2(A)$; $u_2(A)$	250	
	$l_2(B)$; $r_2(B)$		
	$B := B*2$		
	$w_2(B)$; $u_2(B)$		50
$l_1(B)$; $r_1(B)$			
$B := B+100$			
$w_1(B)$; $u_1(B)$;			150

Locking itself is NOT sufficient for enforcing serializability

Two-phase locking (2PL)

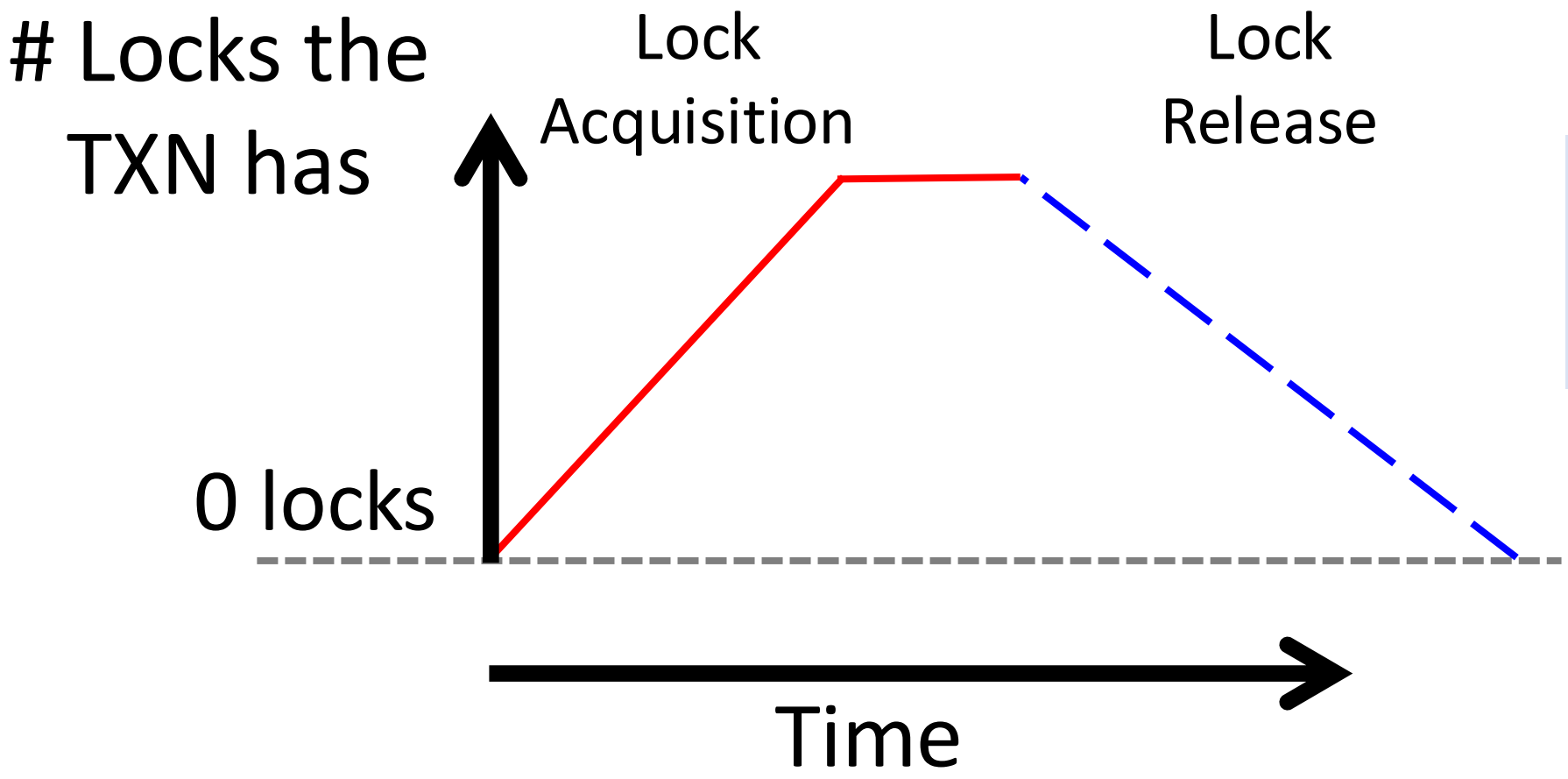
In every transaction, all lock actions precede all unlock actions

- Growing phase: Acquire locks as needed, cannot release any locks
- Shrinking phase: Release locks as needed, cannot acquire any new locks

Guarantees a legal schedule of consistent transactions is conflict serializable

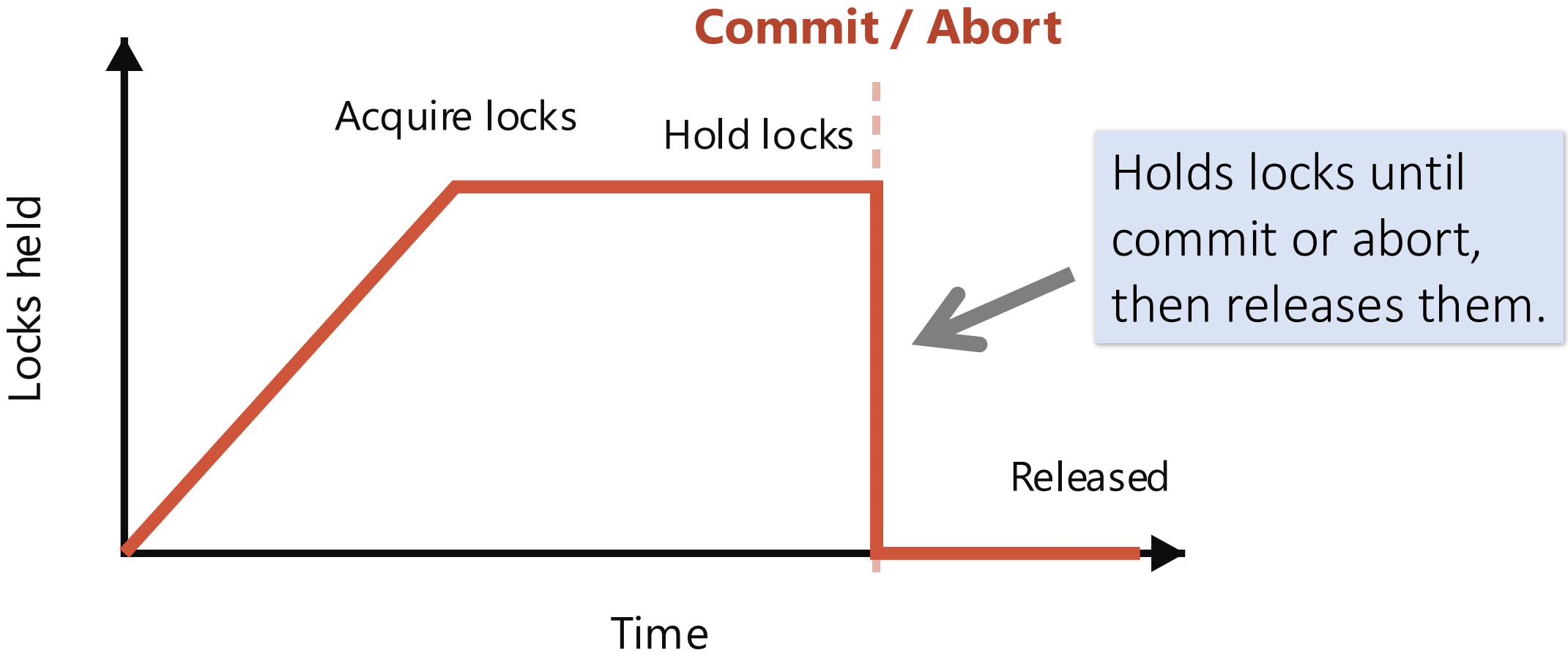
2PL is sufficient, but not necessary, for conflict serializability: some conflict-serializable schedules cannot be generated by 2PL.

Picture of 2PL (Basic 2PL)



Locks may be released before commit

Strict 2PL

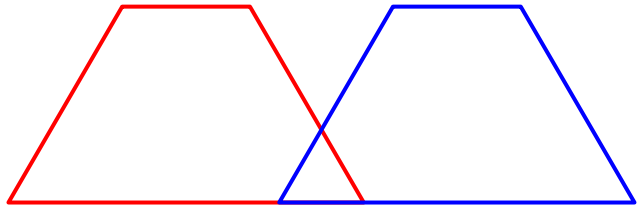


Strict 2PL prevents other transactions from reading or overwriting uncommitted writes, simplifying recovery.

Two-phase locking (2PL)

- This is now conflict serializable

T1	*T2*	*A*	*B*
		25	25
$l_1(A)$; $r_1(A)$;			
$A := A+100$			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$		
	$A := A*2$		
	$w_2(A)$;	250	
	$l_2(B)$ Denied		
$r_1(B)$; $B := B+100$			
$w_1(B)$; $u_1(B)$;			125
	$l_2(B)$; $u_2(A)$; $r_2(B)$		
	$B := B*2$		
	$w_2(B)$; $u_2(B)$		250



Locking with several modes

Using one type of lock is not efficient when reading and writing

Instead, use shared locks for reading and exclusive locks for writing

$sl_i(X)$: Ti requests shared lock on X
$xl_i(X)$: Ti requests exclusive lock on X

Requirements: analogous notions of consistent transactions, legal schedules, and 2PL

Locking with several modes

- Compatibility matrix

		Lock **requested**	
		S	X
Lock **held** in mode	S	Yes	No
	X	No	No

Locking with several modes

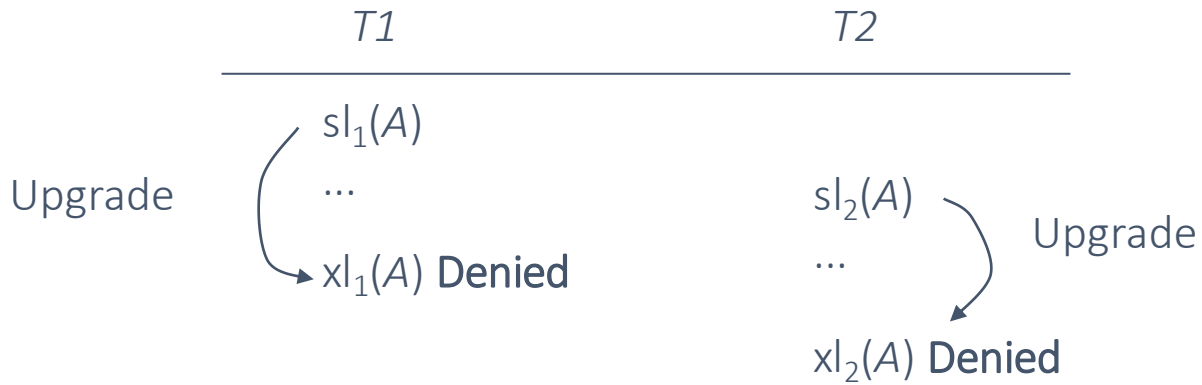
- More efficient than previous schedule

T1	*T2*
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$xl_1(B)$ Denied	
	$u_2(A)$; $u_2(B)$;
$xl_1(B)$; $r_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_1(B)$;	

- T1 and T2 can read A at the same time
- T1 and T2 use 2PL, so the schedule is conflict serializable

Lock Modes Beyond S/X: Update locks

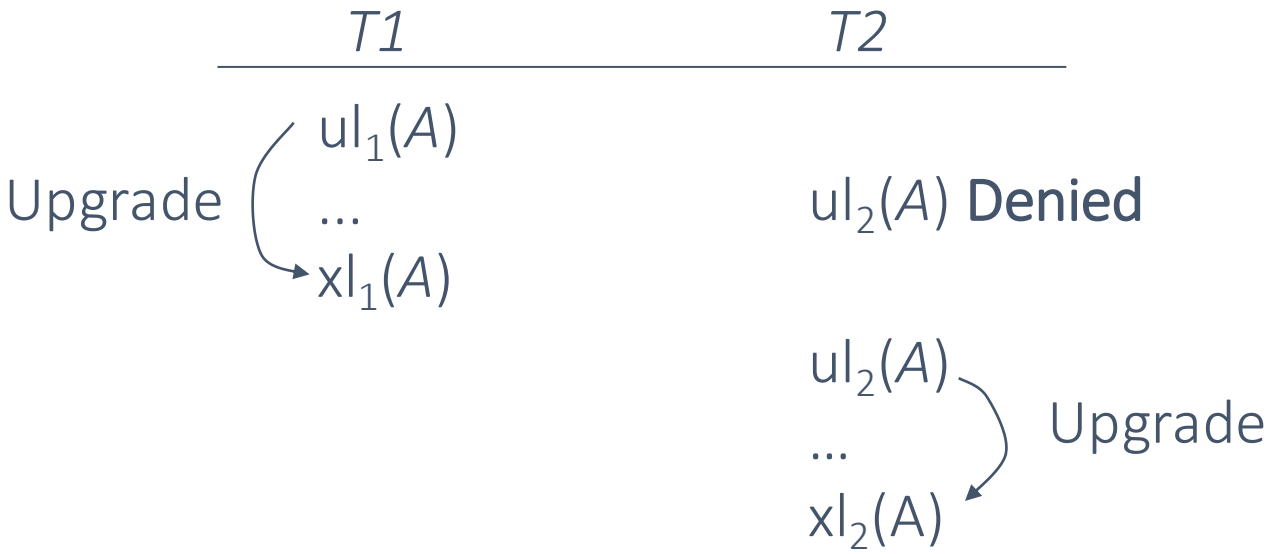
- If T reads and writes the same X, enable lock to upgrade from shared to exclusive
  - Obviously allows more parallelism
- However, a simple upgrading approach may lead to deadlocks



Lock Modes Beyond S/X: Update locks

$ul_i(X)$: Ti requests an update lock on X

- Solution: introduce new type called update locks
- Only an update lock can be updated to an exclusive lock later



Compatibility matrix

	S	X	U
S	Yes	No	Yes
X	No	No	No
U	No	No	No

Deadlocks

Deadlock: Cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

1. Deadlock detection
2. Deadlock prevention (see Database Systems Book Ch19.2)

Deadlock Detection: Example

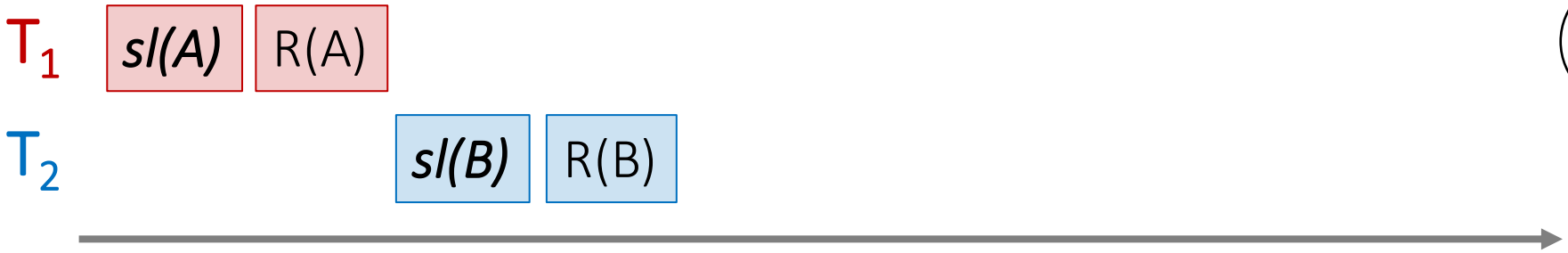


Waits-for graph:



First, T_1 requests a shared lock on A to read from it

Deadlock Detection: Example

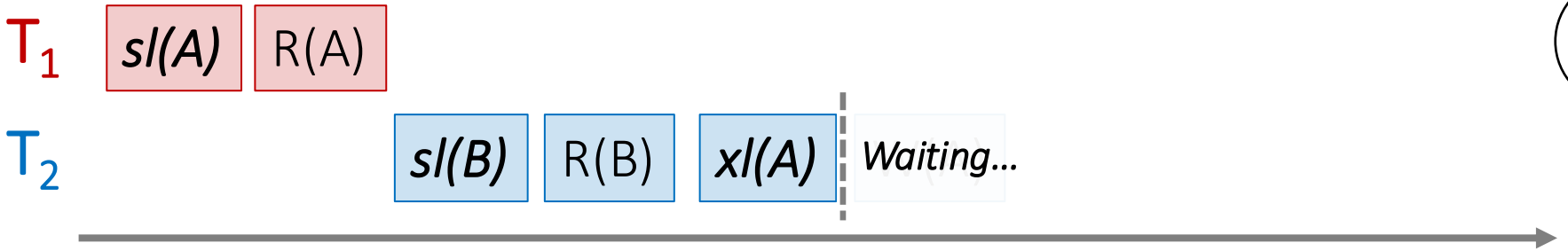


Waits-for graph:



Next, T_2 requests a shared lock on B to read from it

Deadlock Detection: Example

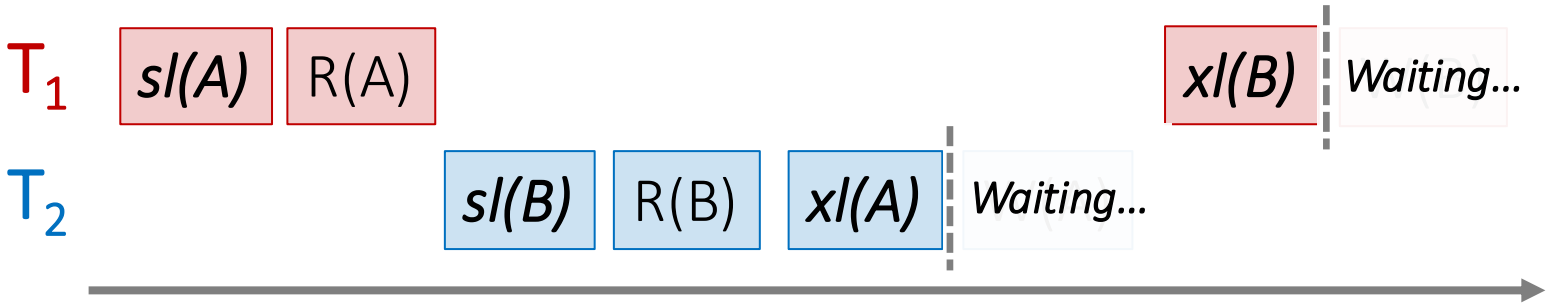


Waits-for graph:

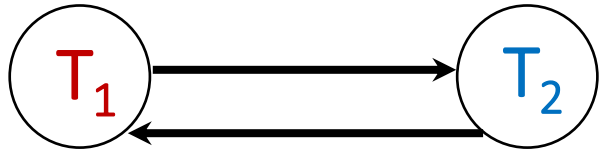


T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1…**

Deadlock Detection: Example



Waits-for graph:



Cycle = DEADLOCK

Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2… DEADLOCK!**

Deadlock Detection

Create the **waits-for graph**:

- Nodes are transactions
- There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*

Periodically check for (***and break***) cycles in the waits-for graph

- e.g., roll back transaction that introduces a cycle

In class Activity: Transaction Simulator

- T1: W(A) → R(B) → W(C) → Commit
- T2: R(C) → R(A) → Commit
- T3: R(B) → W(C) → Commit

Assume transactions go in round-robin fashion

RULES:

- Request locks before operations
- Cannot request locks after releasing ANY lock
- Release ALL locks when done

Locks With Multiple Granularity

So far, we haven't explicitly defined which "database elements" the transaction should acquire locks on.

A few options:

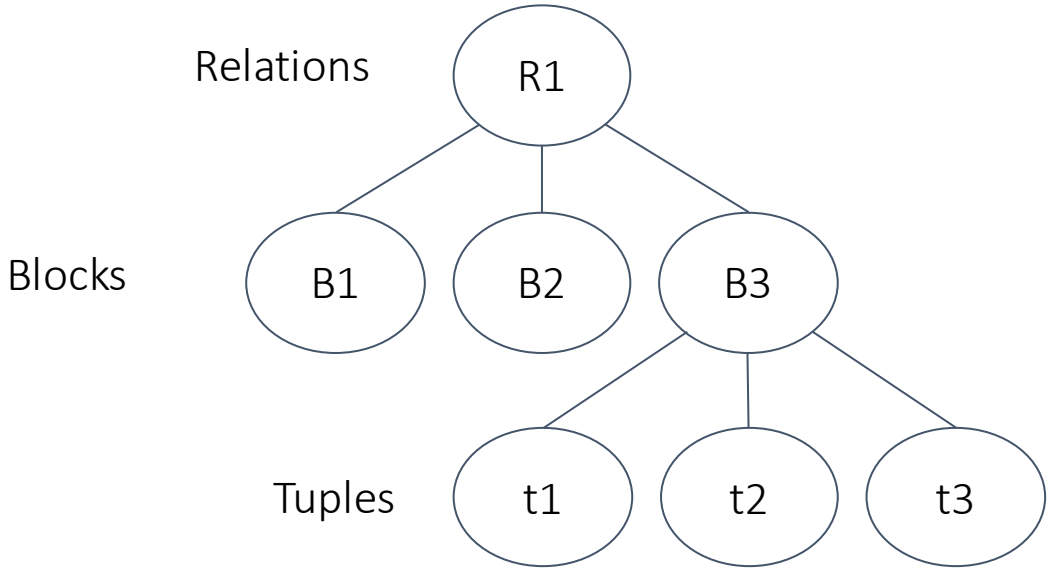
- Relations → Least concurrency
- Pages or data blocks
- Tuples → Most concurrency, but also expensive

Having locks with multiple granularity could lead to unserializable behavior

- e.g., a shared lock on the relation + an exclusive lock on tuples

Intention locks

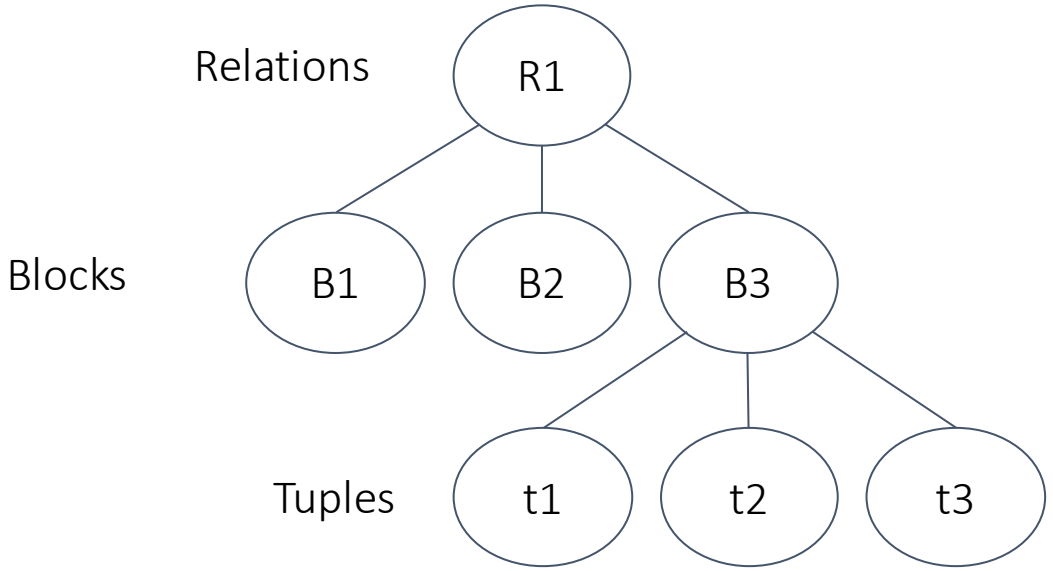
- Ordinary locks: S and X
- Intention locks: I (shows intention to lock)



Intention locks

- Ordinary locks: S and X
- Intention locks: I (shows intention to lock)

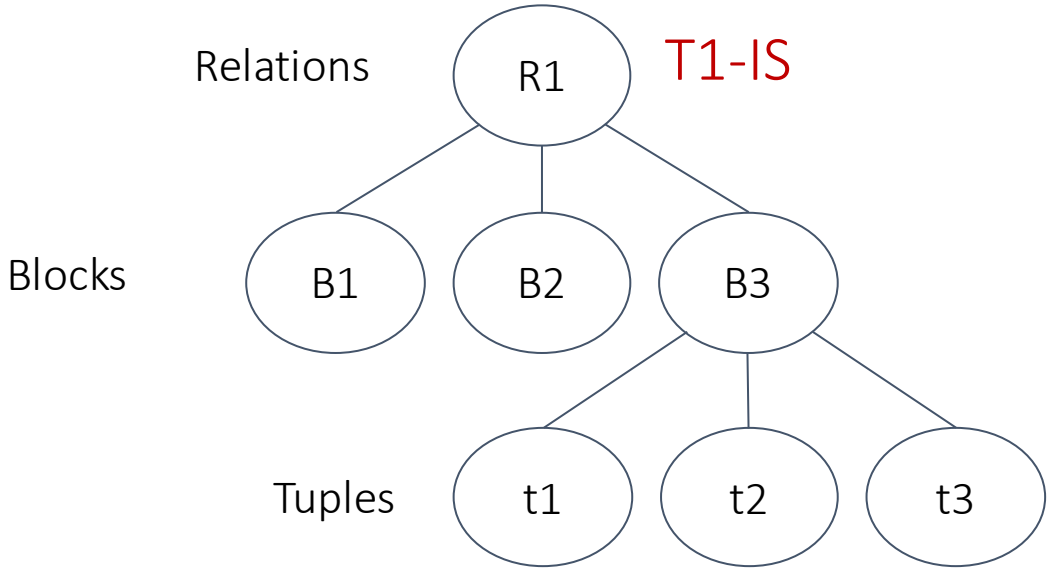
T1 wants to read t3



Intention locks

- Ordinary locks: S and X
- Intention locks: I (shows intention to lock)

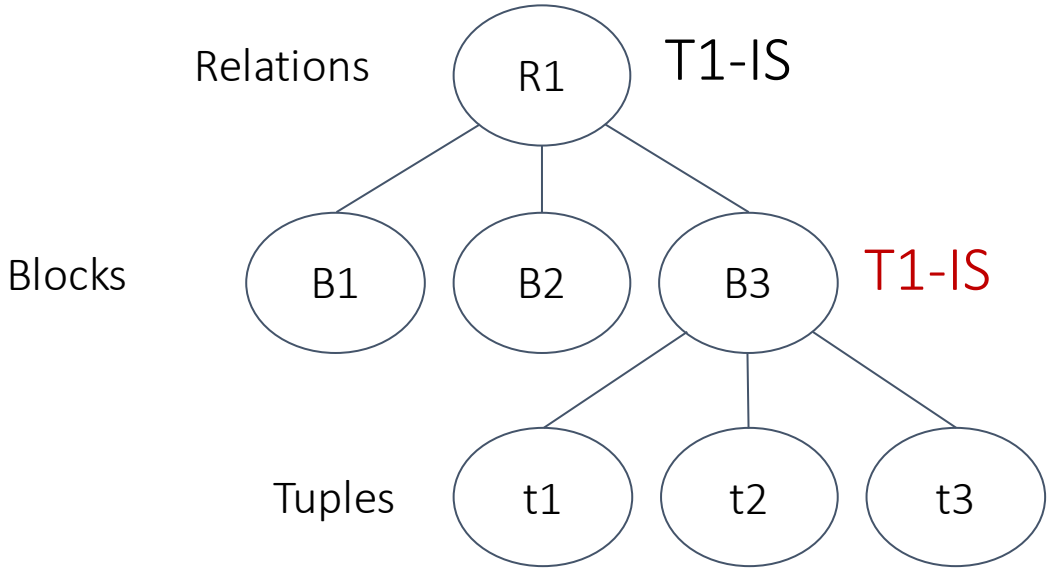
T1 wants to read t3



Intention locks

- Ordinary locks: S and X
- Intention locks: I (shows intention to lock)

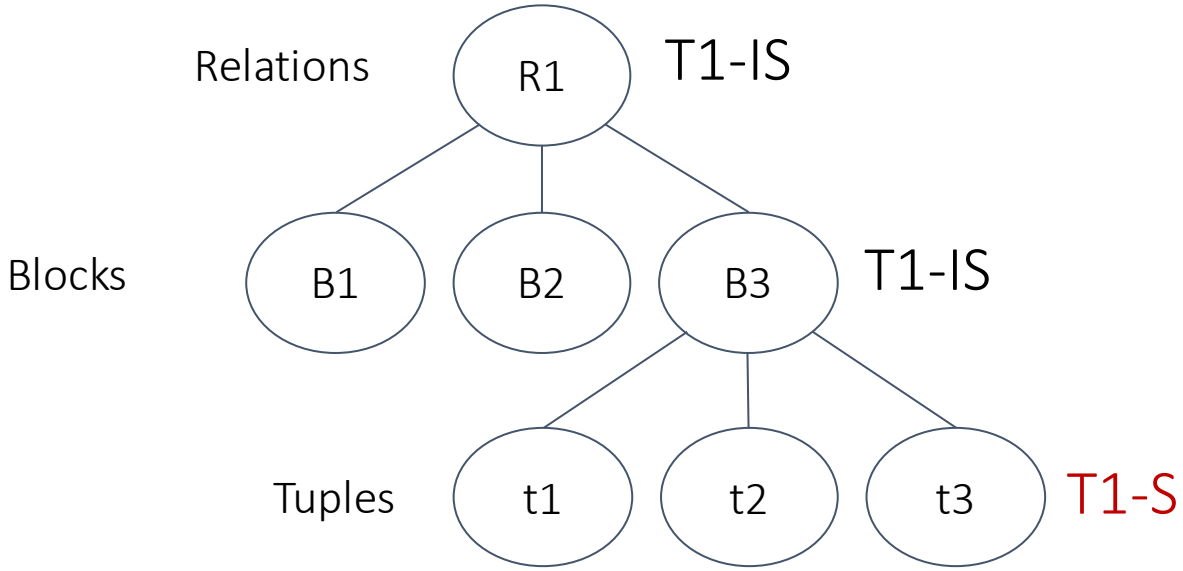
T1 wants to read t3



Intention locks

- Ordinary locks: S and X
- Intention locks: I (shows intention to lock)

T1 wants to read t3

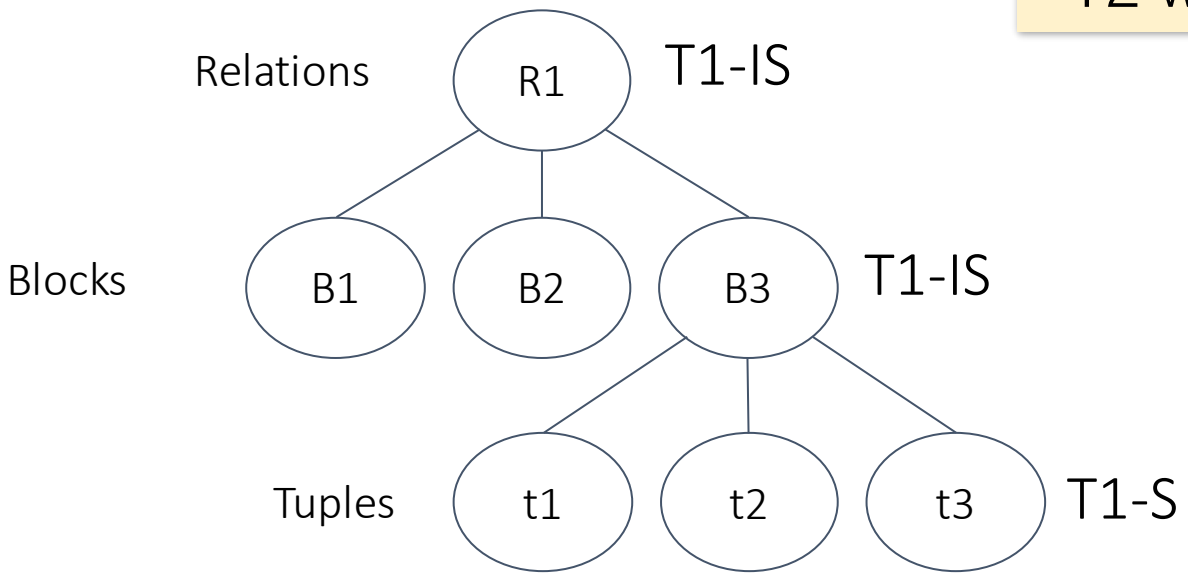


Intention locks

- Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).

T1 wants to read t3

T2 wants to write B2

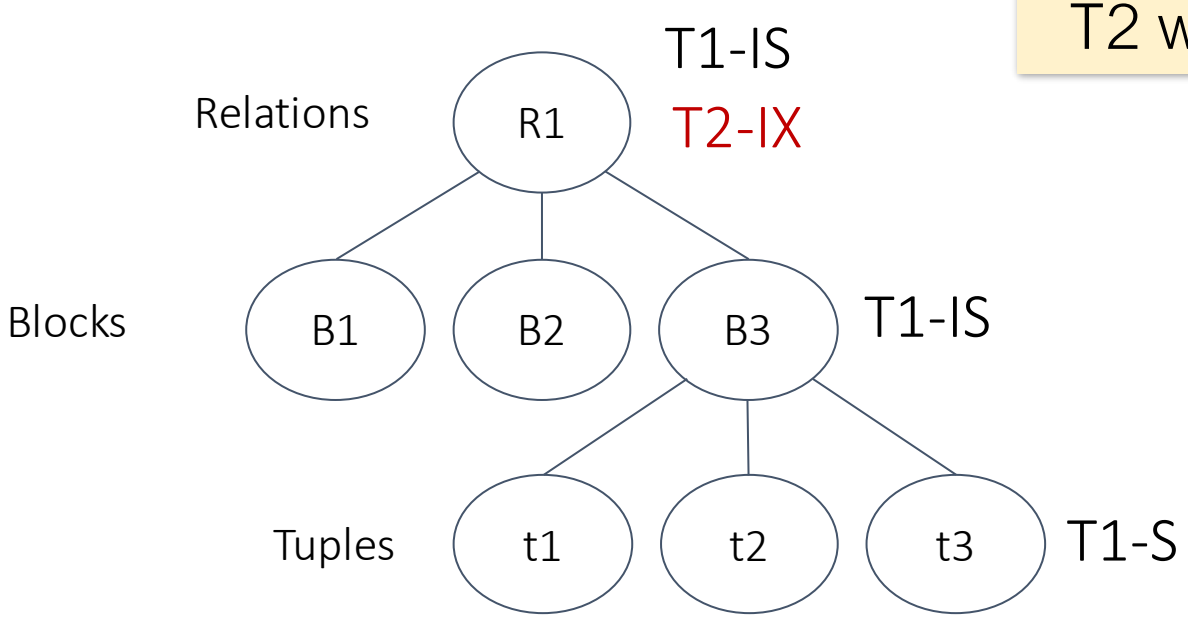


Intention locks

- Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).

T1 wants to read t3

T2 wants to write B2

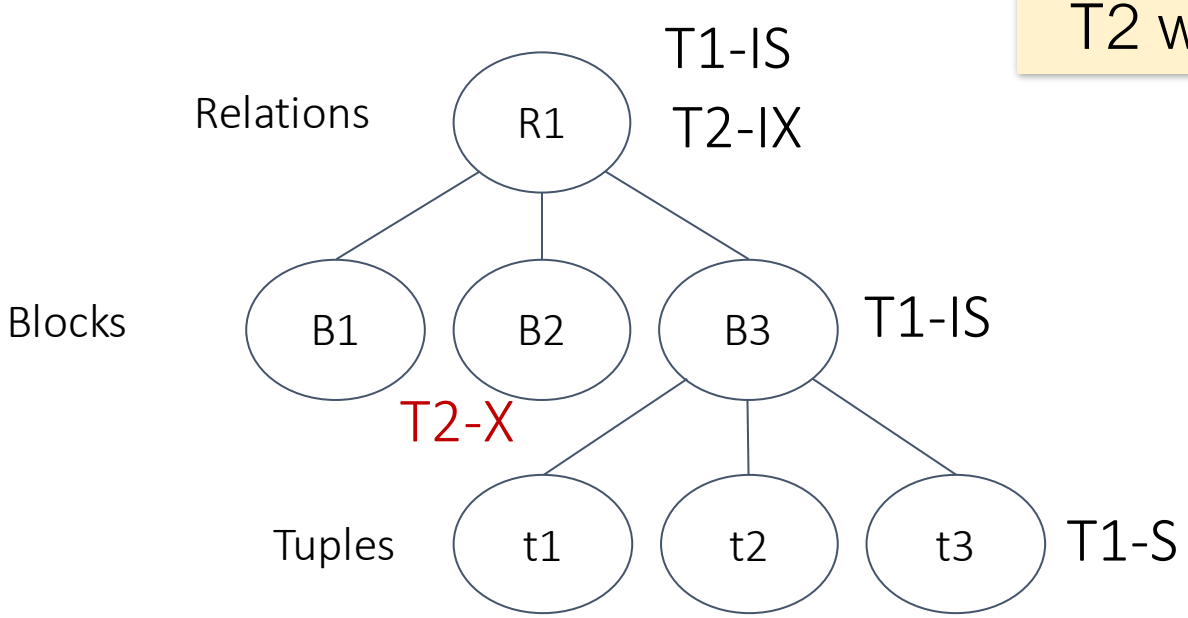


Intention locks

- Ordinary locks: S and X
- Intention locks: I (shows intention to lock)

T1 wants to read t3

T2 wants to write B2



Compatibility matrix

- For shared, exclusive, and intention locks

Requestor

Holder	IS	IX	S	X
IS	Yes	Yes	Yes	No
IX	Yes	Yes	No	No
S	Yes	No	Yes	No
X	No	No	No	No

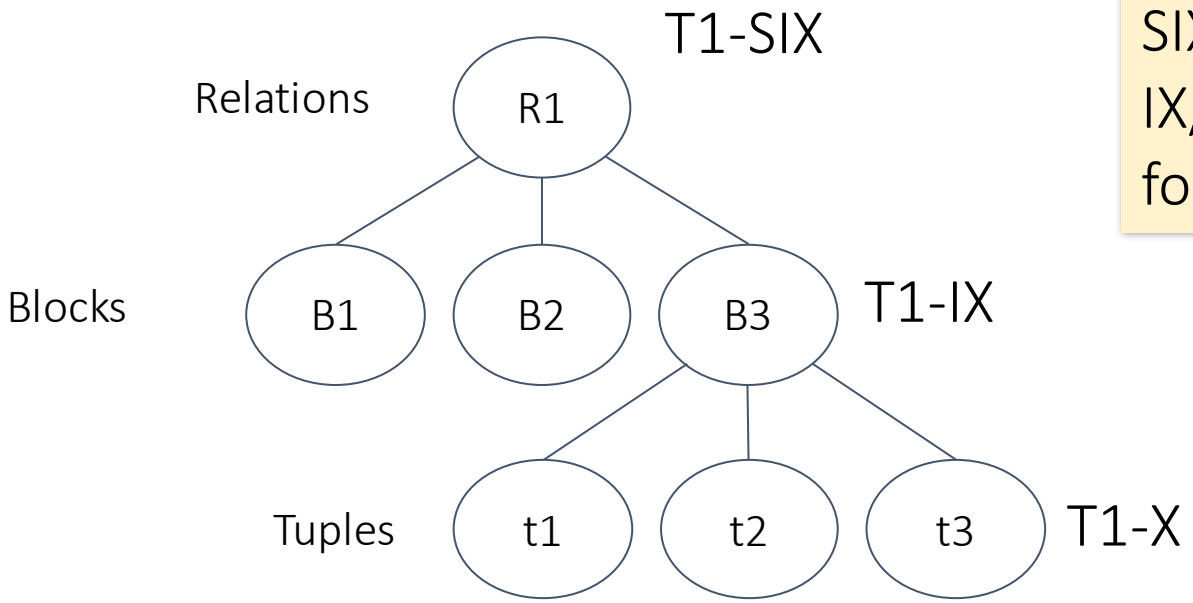
Combining Lock Modes — SIX

What if a transaction wants to read an entire relation but update only a few tuples?

- **S on the relation**: Allows reading the entire relation.
- **IX on the relation**: Indicates an intention to acquire X locks on some descendants.
- **Shared-intention-exclusive (SIX)** indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

Combining Lock Modes — SIX

- Example: A transaction reads all employee records to calculate salary adjustments, then updates a few employees.



SIX covers all reads; IX/X locks are needed for selective writes.

Compatibility matrix

Requestor

Holder	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N

3. Optimistic Concurrency Control

Optimistic Concurrency Control

Optimistic methods

- Key idea: Assume conflicts are rare. Let transactions execute without acquiring read/write locks during their speculative work, then validate before committing.
- Validation succeeds: Commit the transaction's changes.
- Validation fails: Abort and retry.

In comparison, locking methods are pessimistic

- Assume things will go wrong
- Prevent nonserializable behavior
- Delays transactions but reduces rollbacks

Optimistic approaches are often better than lock when transactions have low interference (e.g., read-only)

Concurrency Control by Validation

Each transaction T has:

- RS(T): Set of data items read by T.
- WS(T): Set of data items written by T.

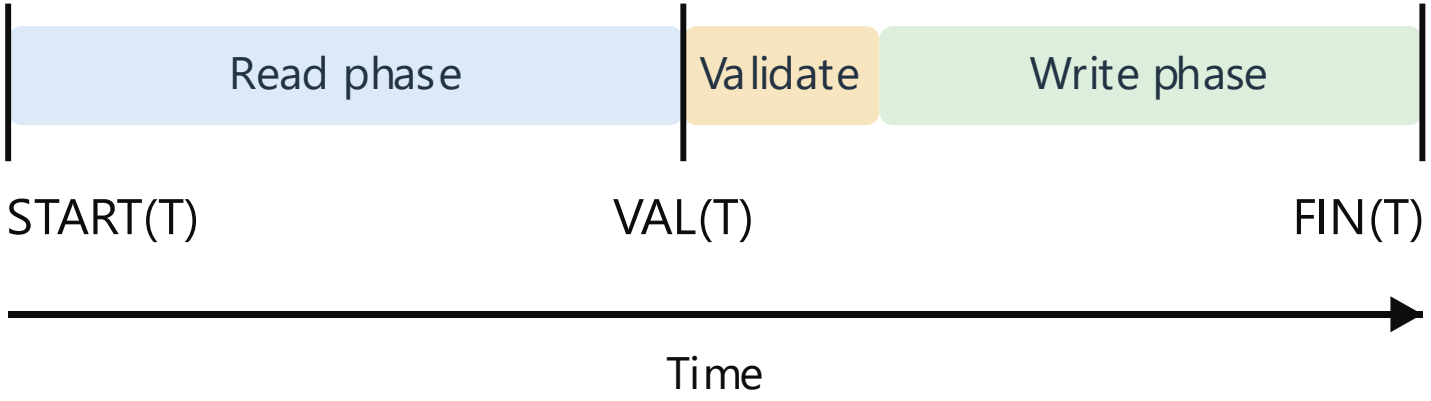
Three phases:

- **Read**: Read database values and compute updates privately (not yet visible to other transactions).
- **Validate**: Check whether concurrent transactions caused conflicts. If validation fails, abort (and potentially retry)..
- **Write**: Apply the updates to the shared database if validation succeeds.

Three Key Timestamps in OCC

For each transaction T, we track three timestamps:

- START(T): When T begins execution (read phase).
- VAL(T): When T begins validation.
- FIN(T): When T finishes its write phase.



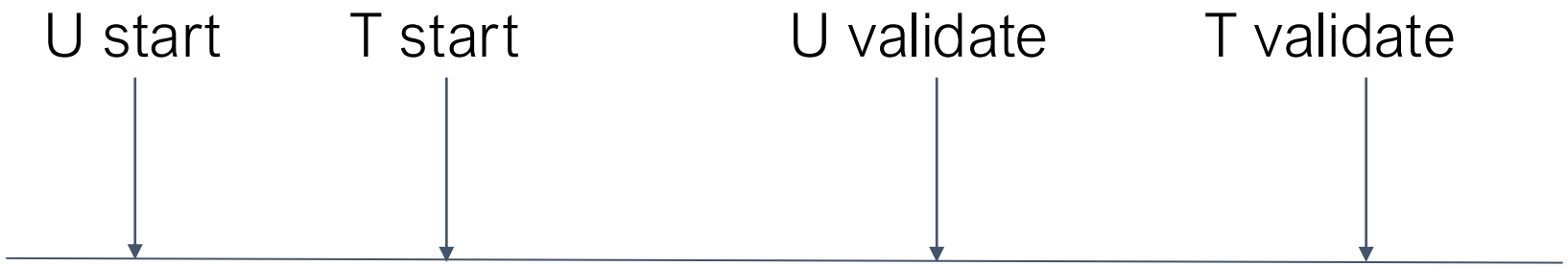
Transactions are ordered by their validation timestamps, VAL(T), to determine the equivalent serial execution order.

Validation rules (assume U validated)

Rule 1: if FIN(U) > START(T), RS(T) ∩ WS(U) = ∅

WS(U) = {A, B} RS(T) = {B, C}

This violates rule 1 because T may be reading B before U writes B



Intuition: Intended serial order is U → T. T must not read an old value that U will overwrite.

Validation rules (assume U validated)

Rule 1: if FIN(U) > START(T), RS(T) ∩ WS(U) = ∅

WS(U) = {A, B}

RS(T) = {B, C}

This satisfies rule 1

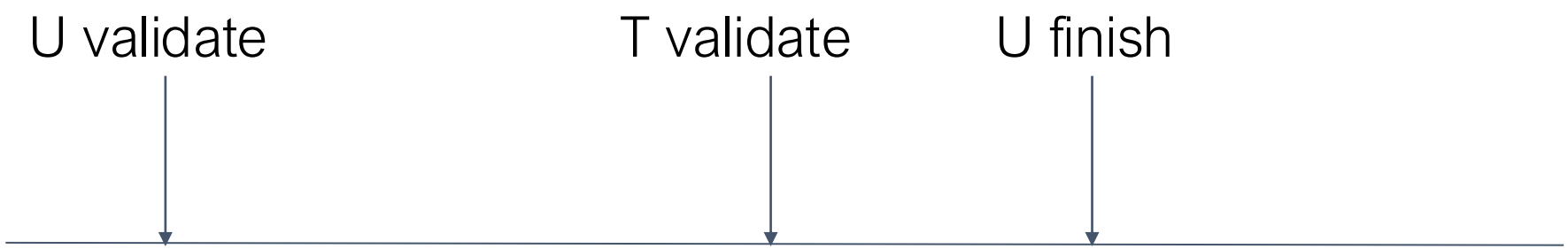


Validation rules (assume U validated)

Rule 2: if FIN(U) > VAL(T), WS(T) ∩ WS(U) = ∅

WS(U) = {A, B}

WS(T) = {B, C}



Intuition: Intended serial order is U → T. T must not publish its writes before U finishes writing the same object.

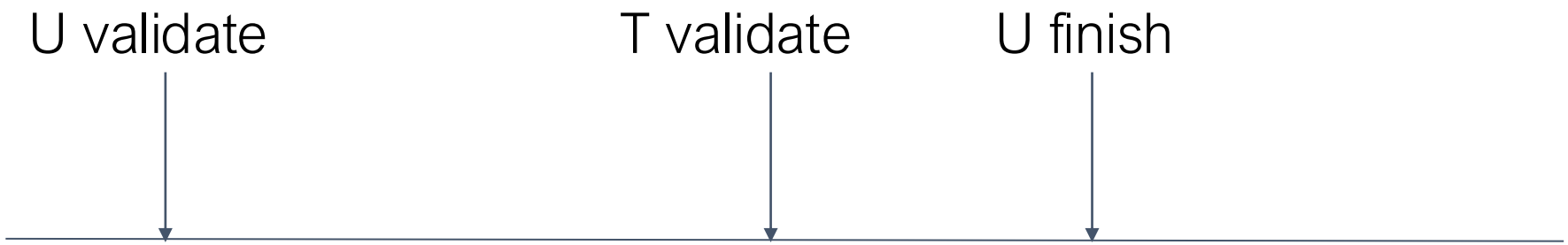
Validation rules (assume U validated)

Rule 2: if FIN(U) > VAL(T), WS(T) ∩ WS(U) = ø

WS(U) = {A, B}

WS(T) = {B, C}

This violates rule 2 because T may write B before U writes B



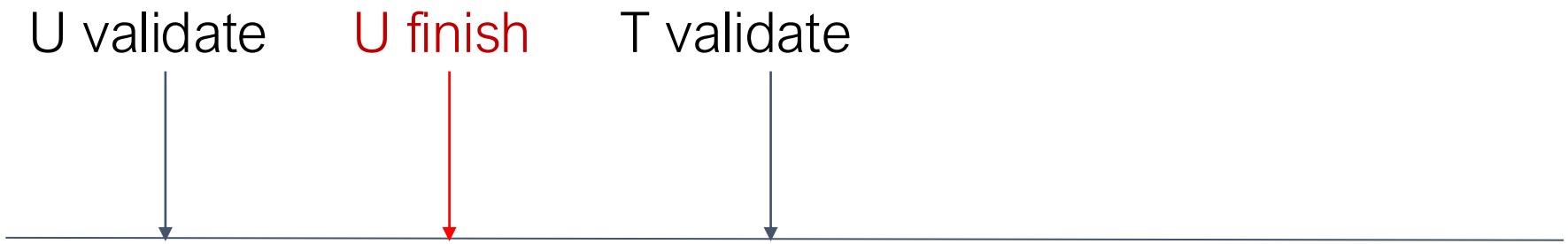
Validation rules (assume U validated)

Rule 2: if FIN(U) > VAL(T), WS(T) ∩ WS(U) = ø

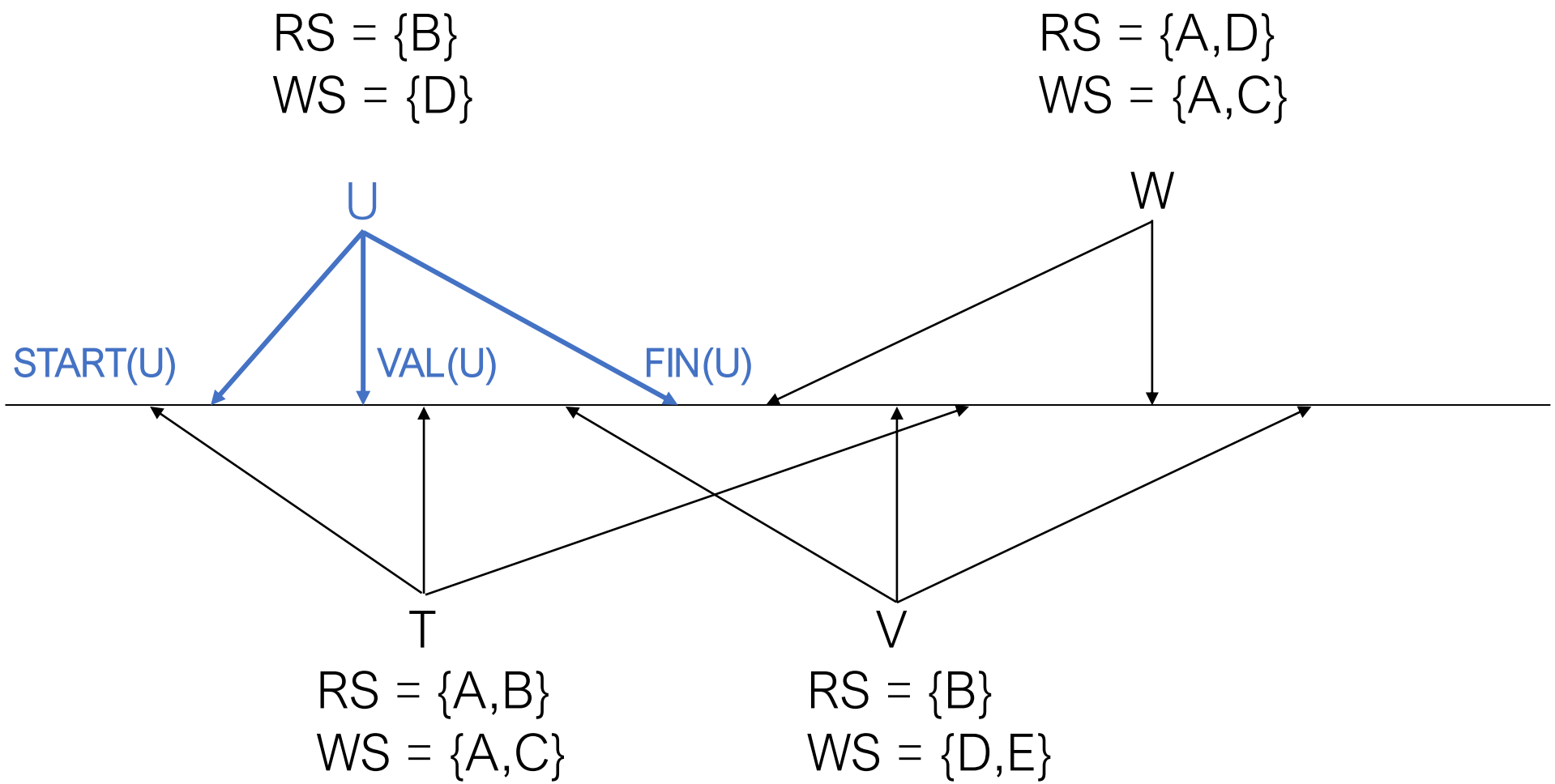
WS(U) = {A, B}

WS(T) = {B, C}

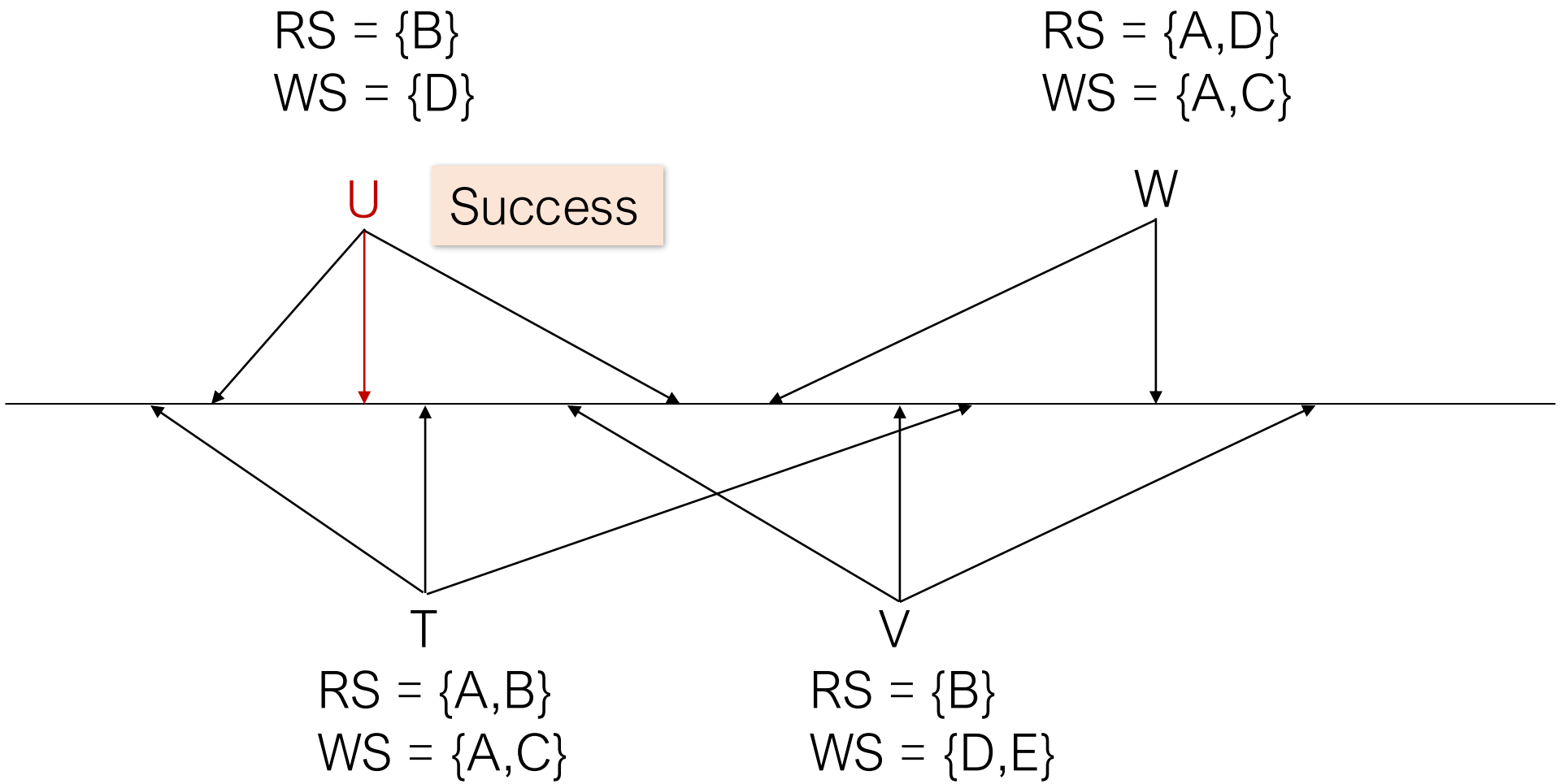
This satisfies rule 2



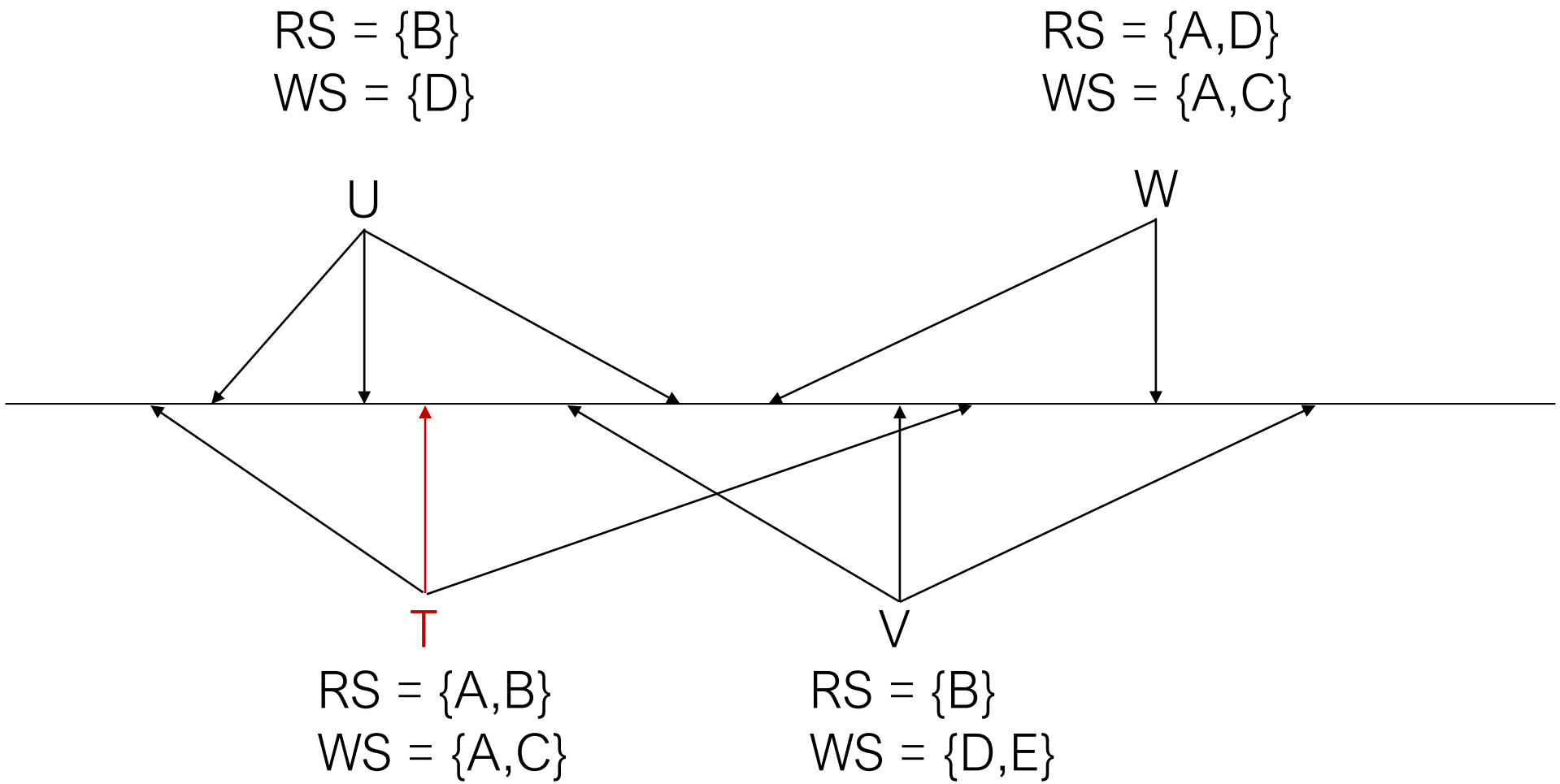
Example: CC by Validation



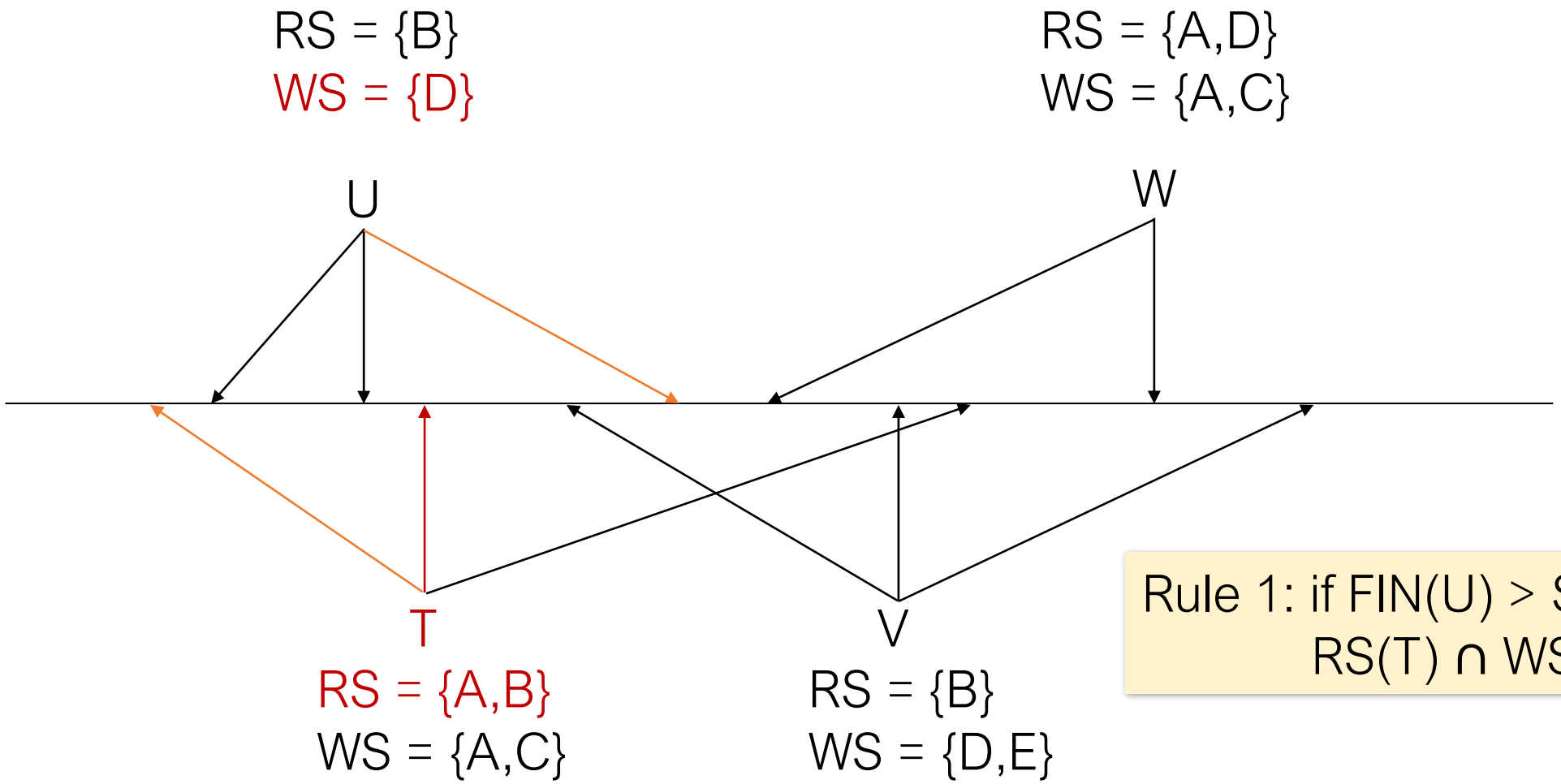
Example: CC by Validation



Example: CC by Validation

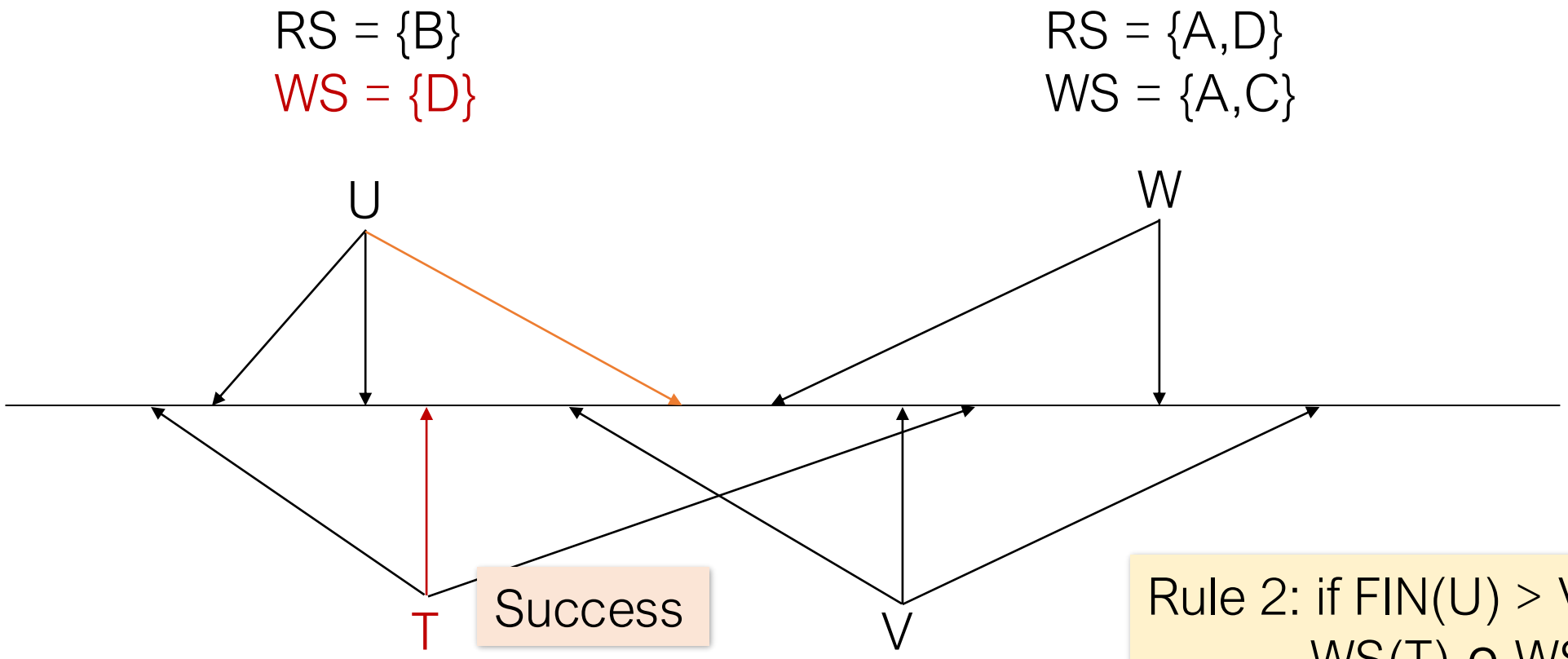


Example: CC by Validation



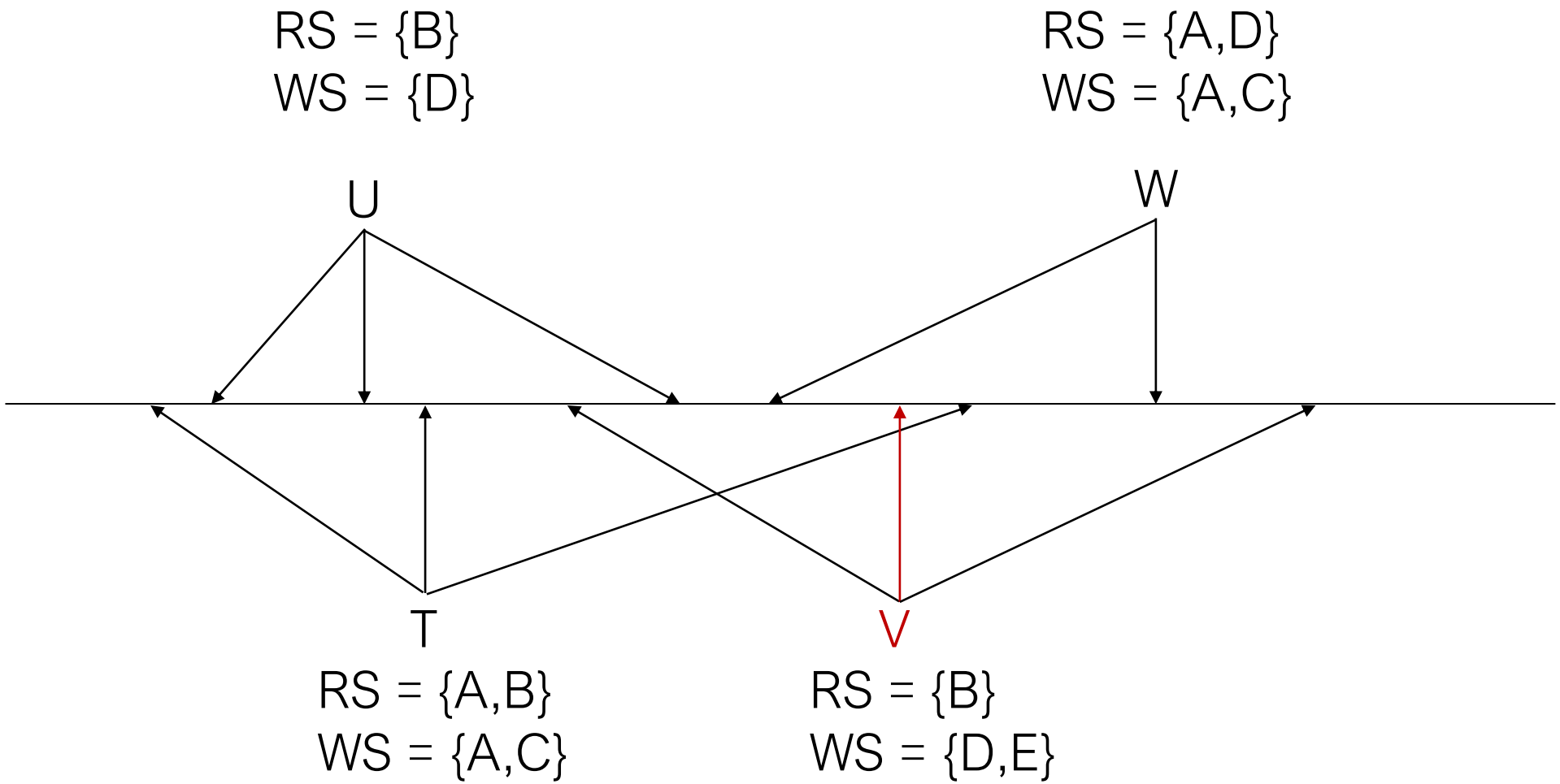
Rule 1: if FIN(U) > START(T),
RS(T) ∩ WS(U) = ∅

Example: CC by Validation

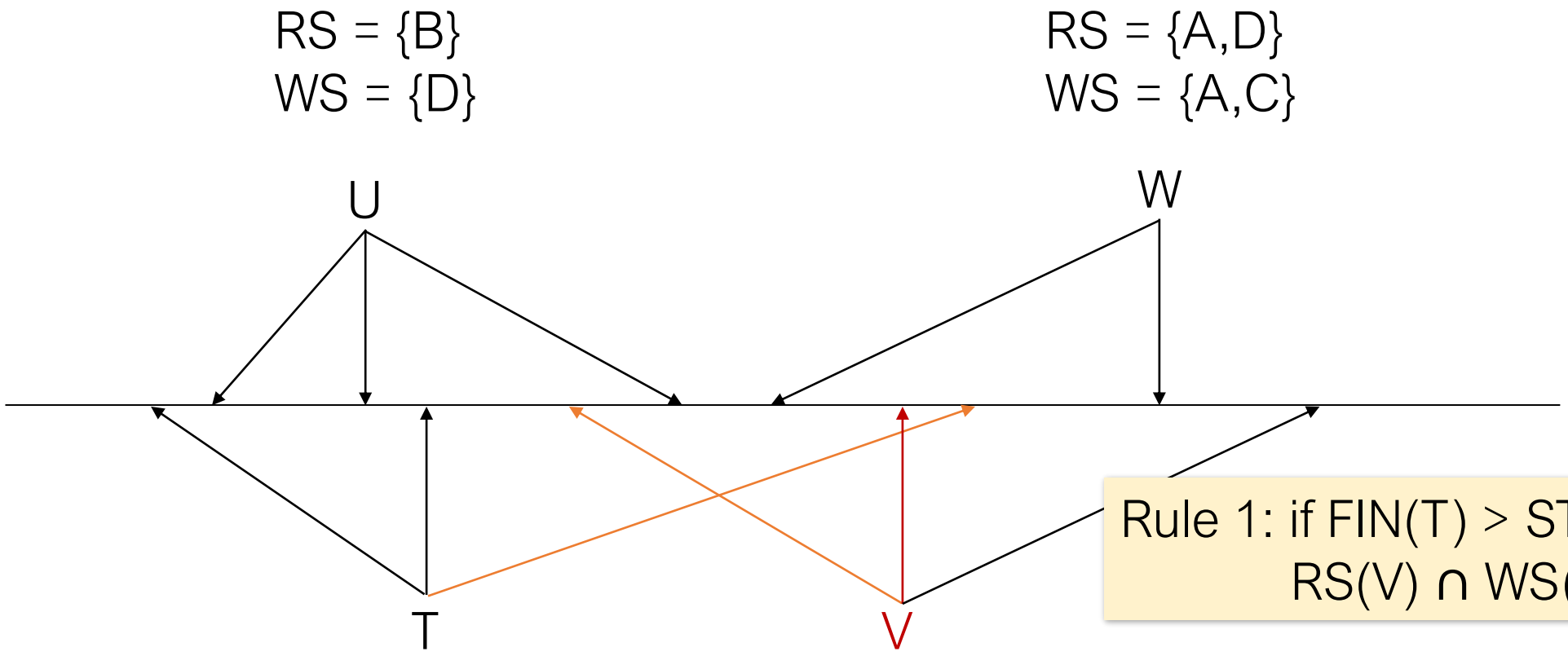


Rule 2: if FIN(U) > VAL(T),
WS(T) ∩ WS(U) = ∅

Example: CC by Validation

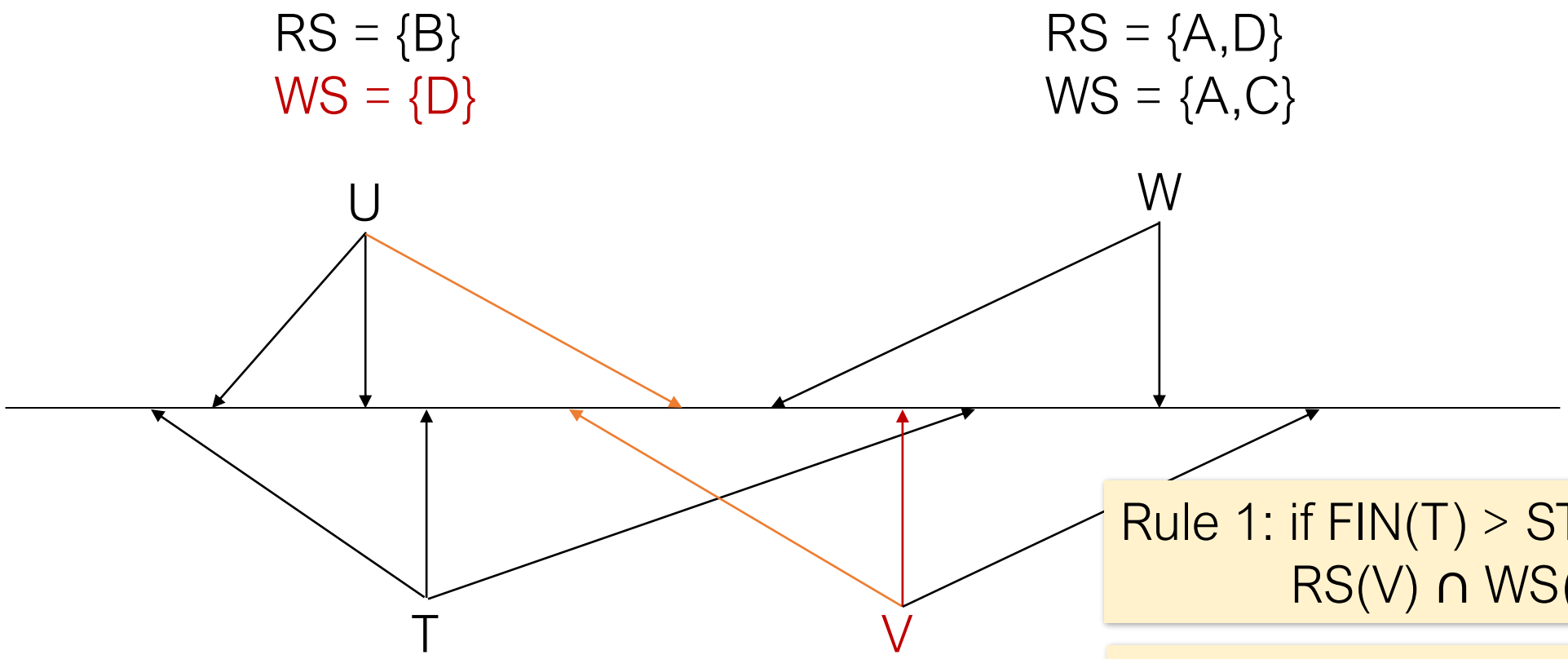


Example: CC by Validation



Rule 1: if FIN(T) > START(V),
RS(V) ∩ WS(T) = ∅

Example: CC by Validation

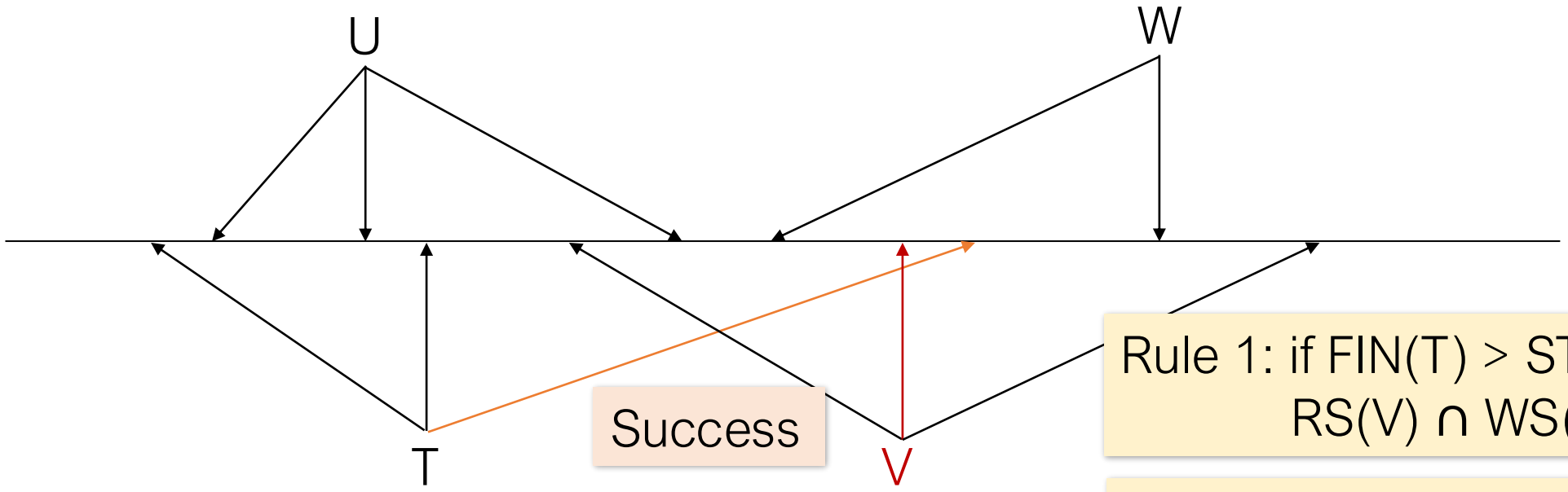


Rule 1: if FIN(T) > START(V),
RS(V) ∩ WS(T) = ∅

Rule 1: if FIN(U) > START(V),
RS(V) ∩ WS(U) = ∅

Example: CC by Validation

Rule 2: if FIN(T) > VAL(V),
WS(V) ∩ WS(T) = ∅



Rule 1: if FIN(T) > START(V),
RS(V) ∩ WS(T) = ∅

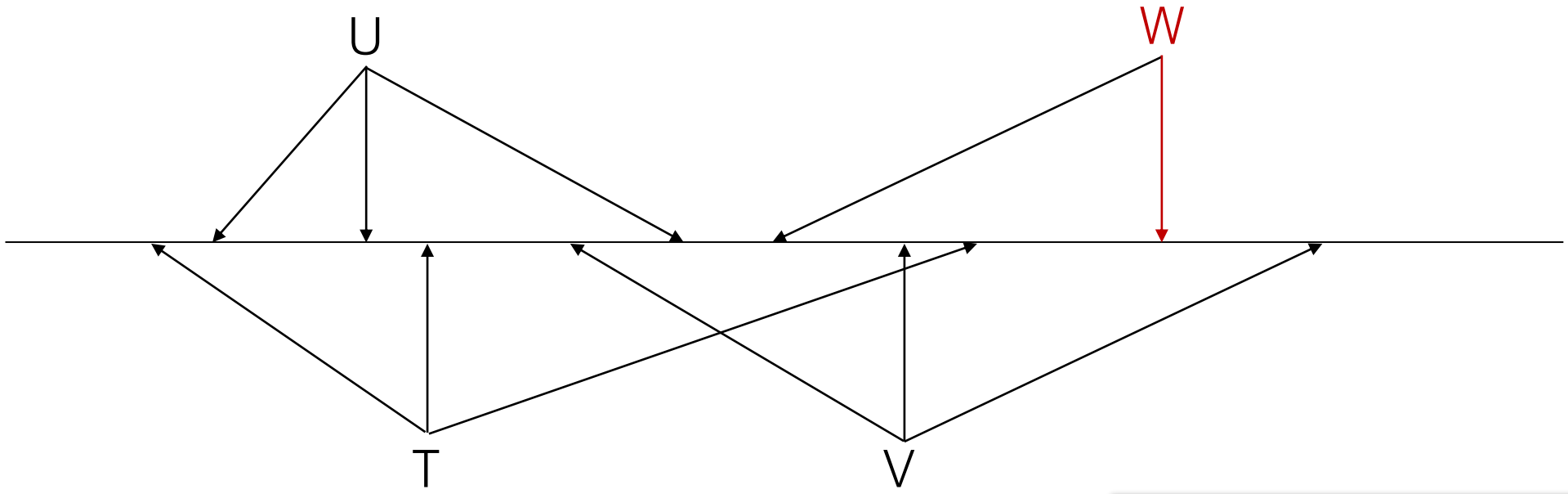
Rule 1: if FIN(U) > START(V),
RS(V) ∩ WS(U) = ∅

Example: CC by Validation

Rule 2: if FIN(V) > VAL(W),
WS(V) ∩ WS(W) = ∅

RS = {B}
WS = {D}

RS = {A,D}
WS = {A,C}



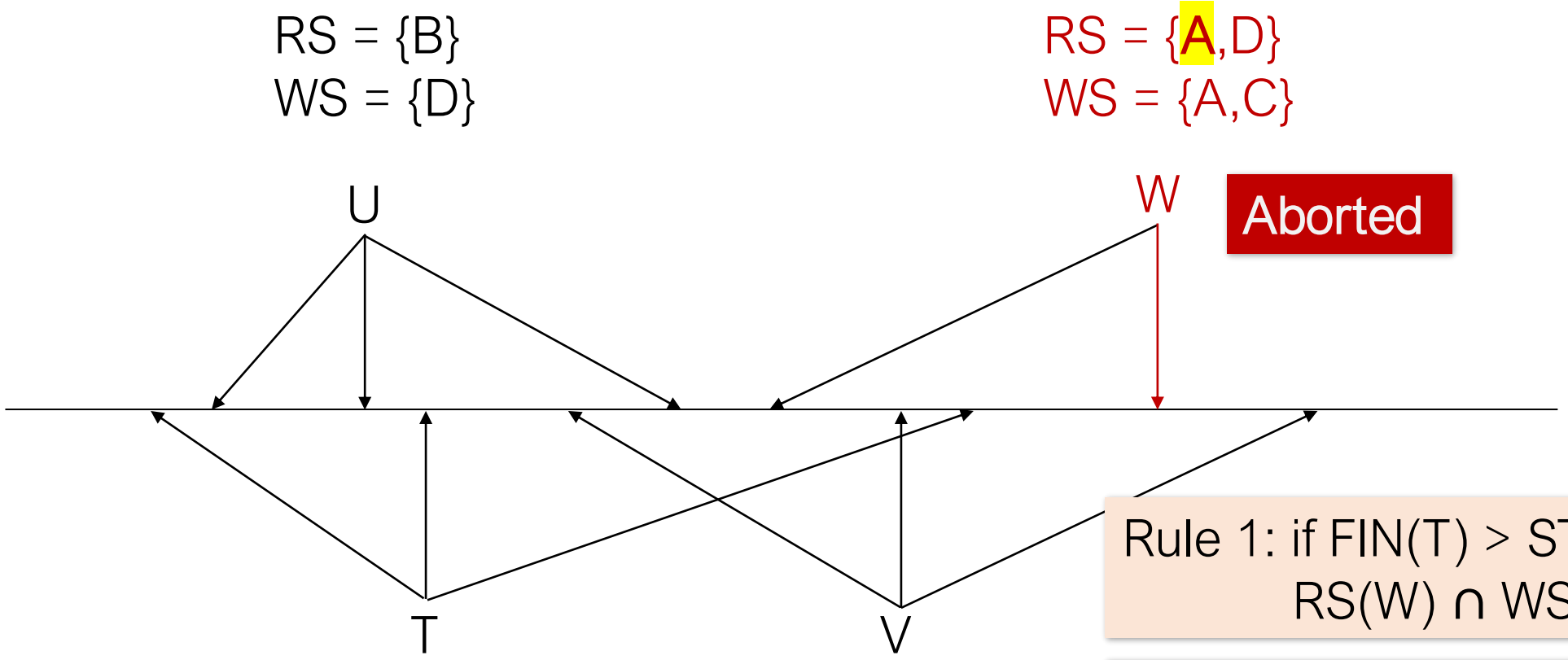
RS = {A,B}
WS = {A,C}

RS = {B}
WS = {D,E}

Rule 1: if FIN(V) > START(W),
RS(W) ∩ WS(V) = ∅

Example: CC by Validation

Rule 2: if FIN(V) > VAL(W),
WS(V) ∩ WS(W) = ∅



Rule 1: if FIN(T) > START(W),
RS(W) ∩ WS(T) ≠ ∅

Rule 1: if FIN(V) > START(W),
RS(W) ∩ WS(V) = ∅

Comparison of CC Techniques

	Locking (2PL)	Validation (OCC)
Key idea	Prevent conflicting operations using locks	Execute optimistically; check for conflicts later
When conflicts occur	Transactions may wait for locks	Transactions may abort and retry
Main tradeoff	Waiting and potential deadlocks	Wasted work from aborted transactions

Other techniques: Multi-Version Concurrency Control (MVCC) maintains multiple versions of data, allowing readers and writers to proceed concurrently in many cases.