CS 6400 A

Database Systems Concepts and Design

Lecture 20

11/05/25

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.

Serial and serializable schedules are guaranteed to preserve the consistency of database states

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

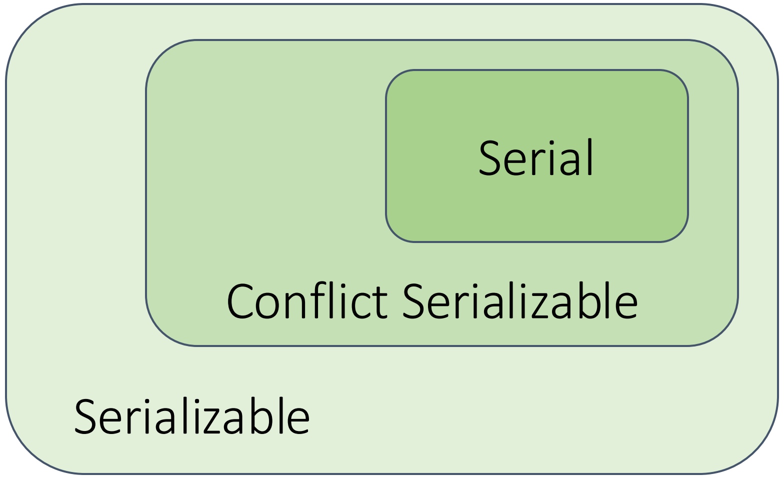
- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.

Conflict-serializable schedule

- A conflict-serializable schedule is always serializable
- But not vice versa (e.g., serializable schedule due to detailed transaction behavior)

S1: $w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$ Serial

S2: $w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$ Serializable, but not conflict serializable



2. Lock-based Concurrency Control

Enforce serializability with locks

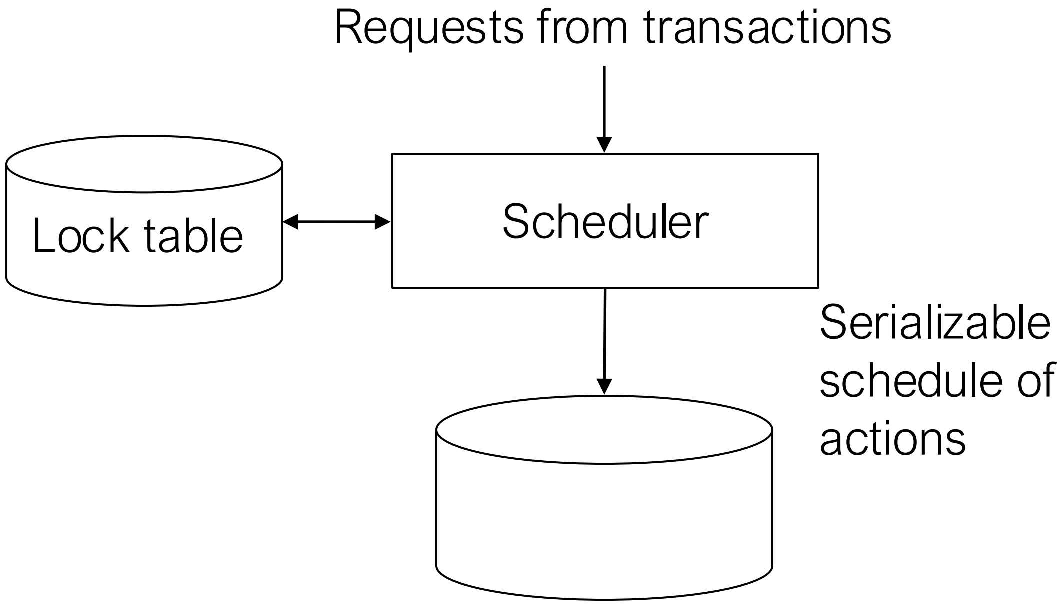
$l_i(X)$: Ti requests lock on X
$u_i(X)$: Ti releases lock on X

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

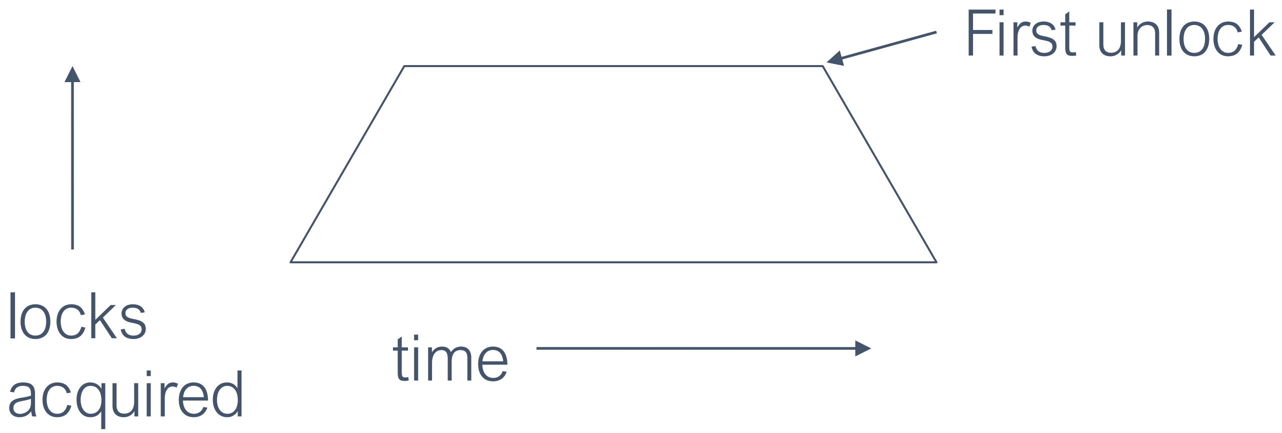
- Legal, but not serializable schedule

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)

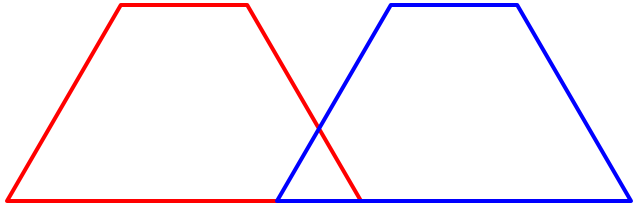
- In every transaction, all lock actions precede all unlock actions
- Guarantees a legal schedule of consistent transactions is conflict serializable



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

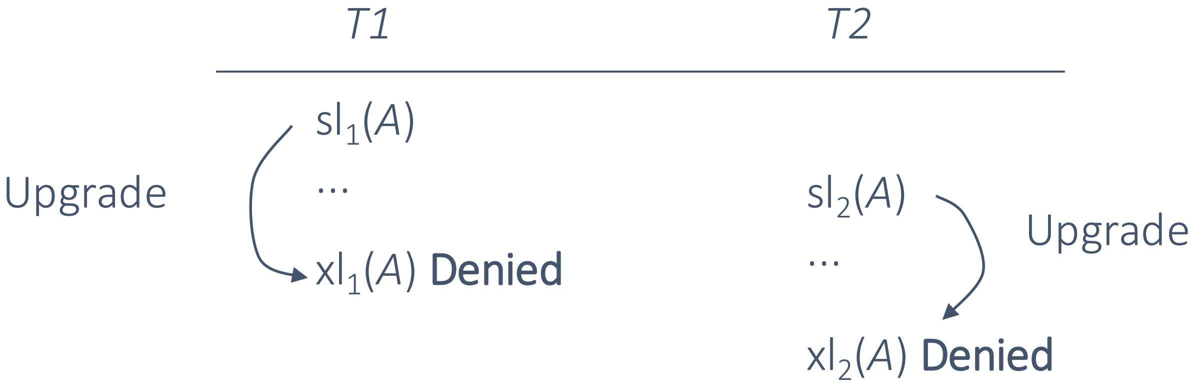
T1	*T2*
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$;
$xl_1(B)$ Denied	
	$u_2(A)$; $u_2(B)$;
$xl_1(B)$; $r_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;	

- 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

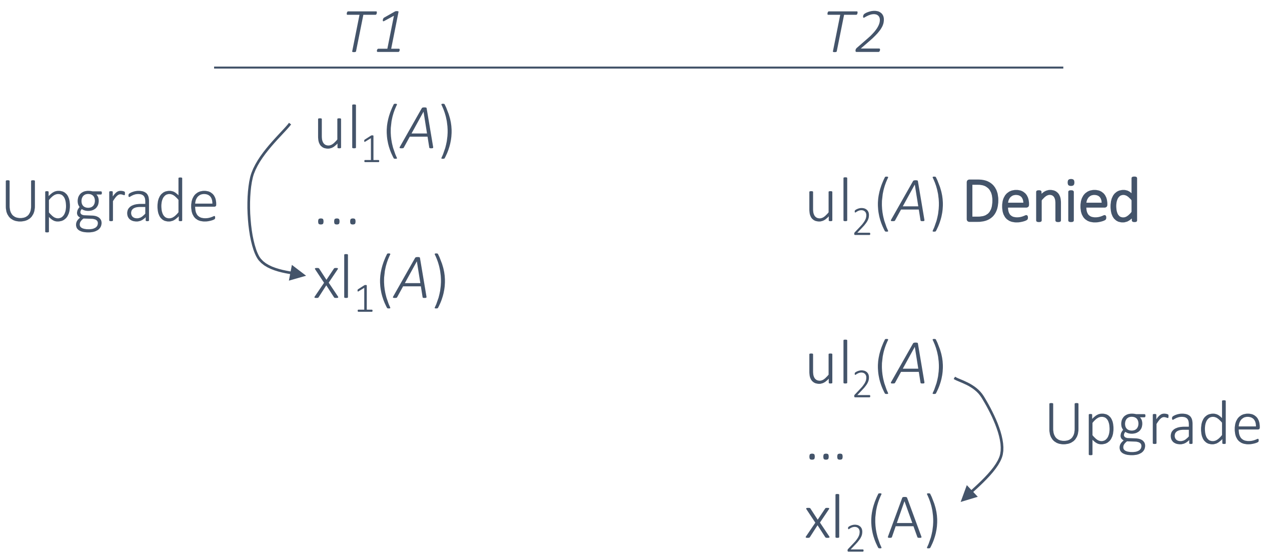


T1	*T2*
$sl_1(A)$	
…	$sl_2(A)$
$xl_1(A)$ **Denied**	…
	$xl_2(A)$ **Denied**

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



T1 *T2*

$ul_1(A)$

…

$xl_1(A)$

Upgrade

$ul_2(A)$ **Denied**

$ul_2(A)$

…

$xl_2(A)$

Upgrade

Compatibility matrix

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

Deadlocks

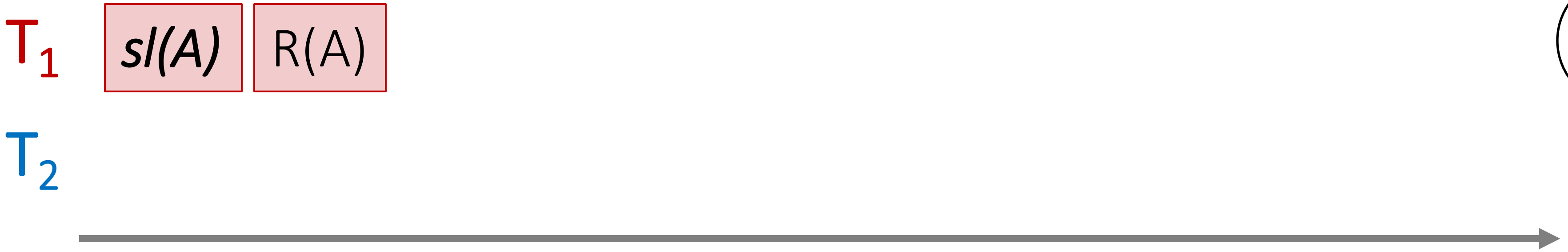
Deadlock: Cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

1. Deadlock detection

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

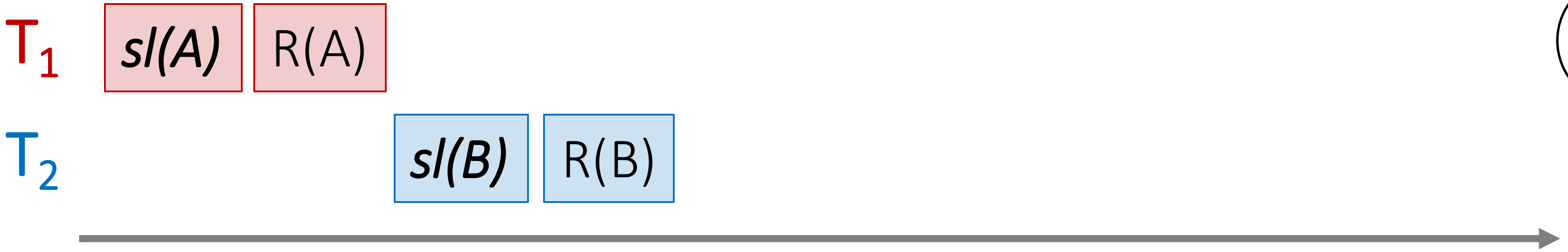
Deadlock Detection: Example



Waits-for graph:

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

Deadlock Detection: Example



T_1 *sl(A)* R(A)

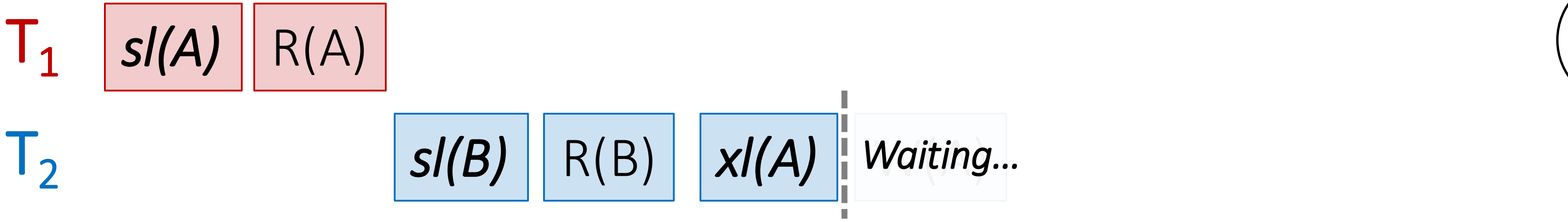
T_2 *sl(B)* R(B)

Waits-for graph:

T_1 T_2

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

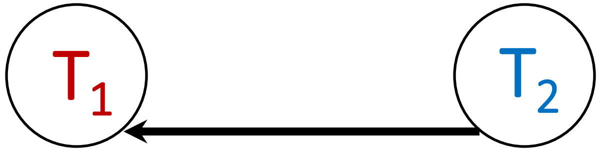
Deadlock Detection: Example



T_1 *sl(A)* R(A)

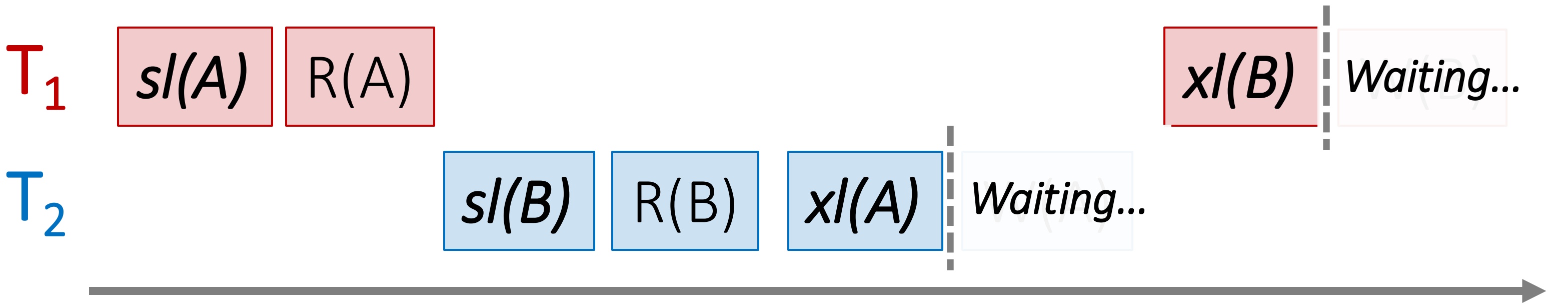
T_2 *sl(B)* R(B) *xl(A)* *Waiting…*

Waits-for graph:

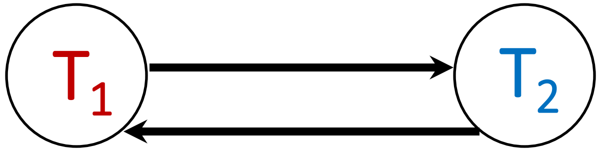


T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1…**

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 → 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

Round 1: Round-Robin

Round 2: Free for All

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:

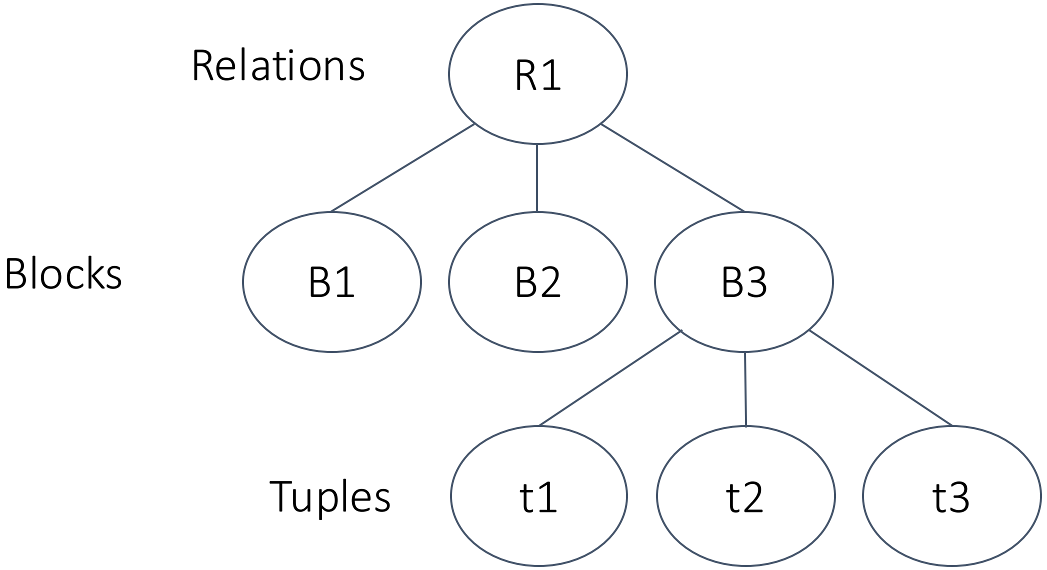
- Relations → Least concurrency
- Pages or data blocks
- Tuples → Most concurrency, but also expensive

Having locks with multiple granularity could lead to unserializable behavior

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

Warning locks

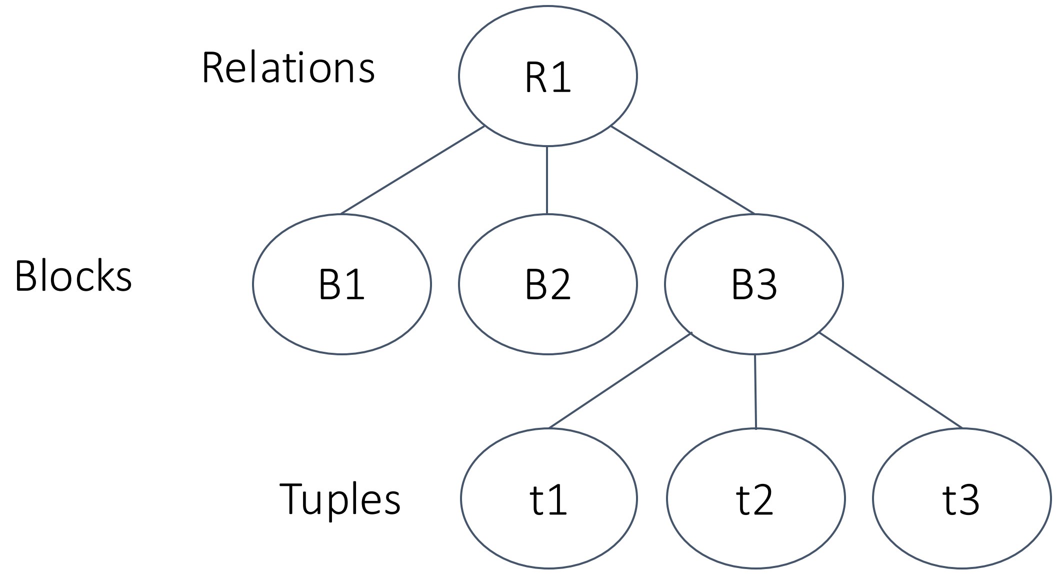
- Ordinary locks: S and X
- Warning locks: I (shows intention to lock)



Warning locks

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

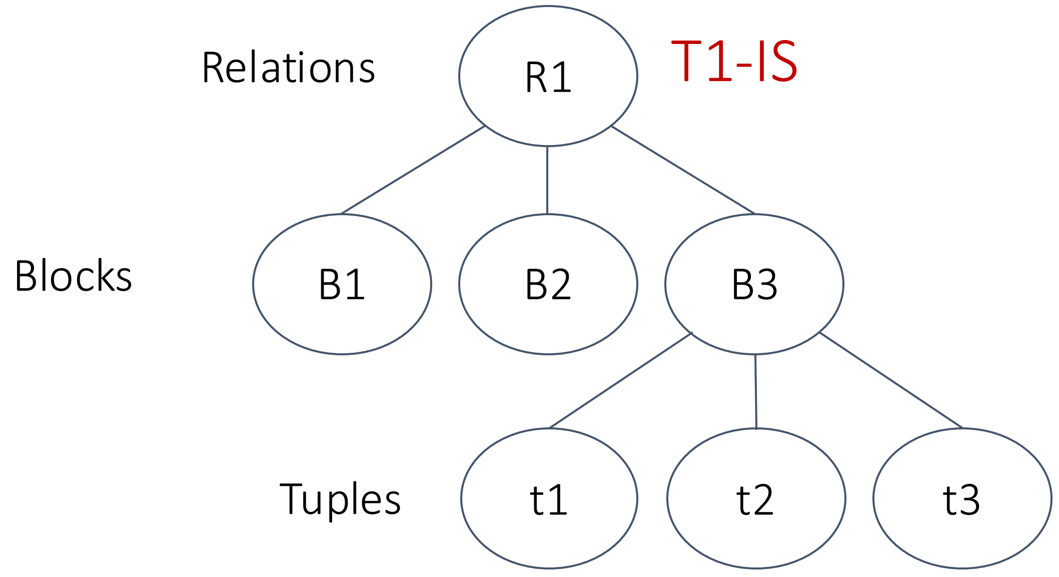
T1 wants to read t3



Warning locks

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

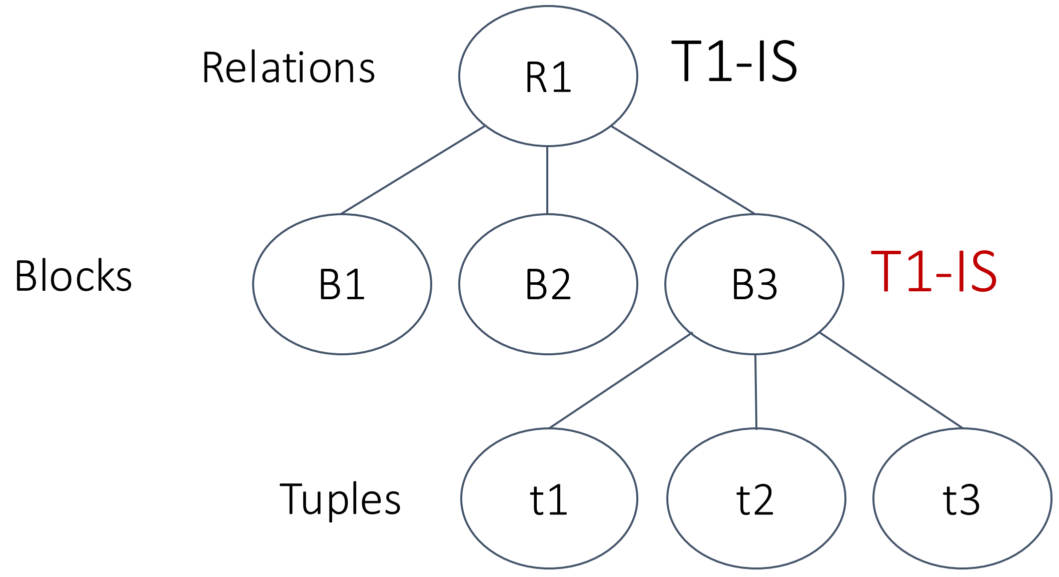
T1 wants to read t3



Warning locks

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

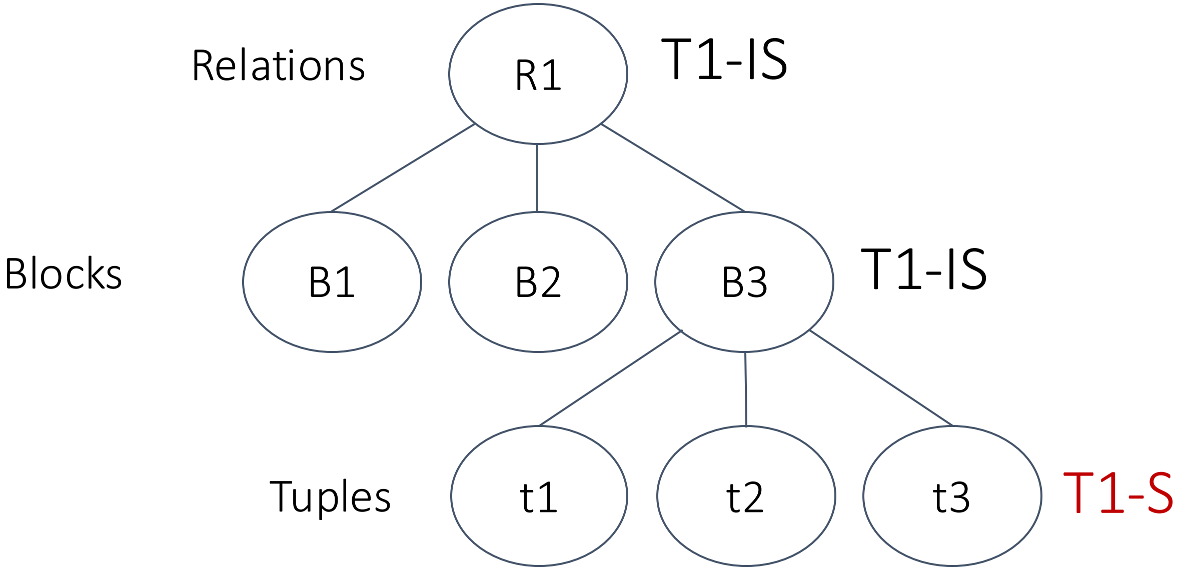
T1 wants to read t3



Warning locks

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

T1 wants to read t3

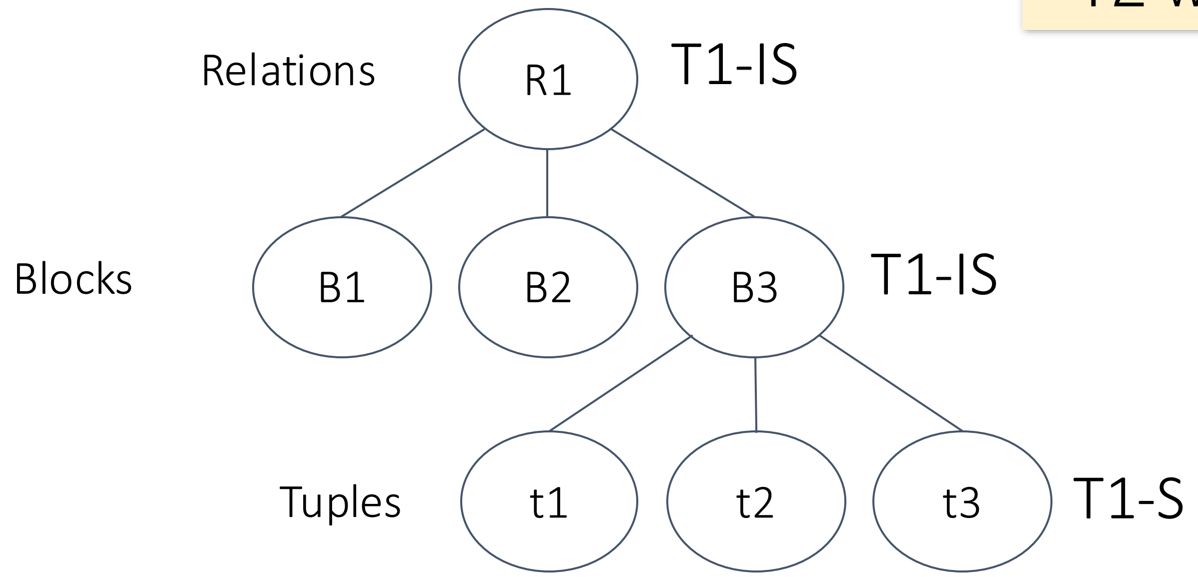


Warning locks

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

T1 wants to read t3

T2 wants to write B2

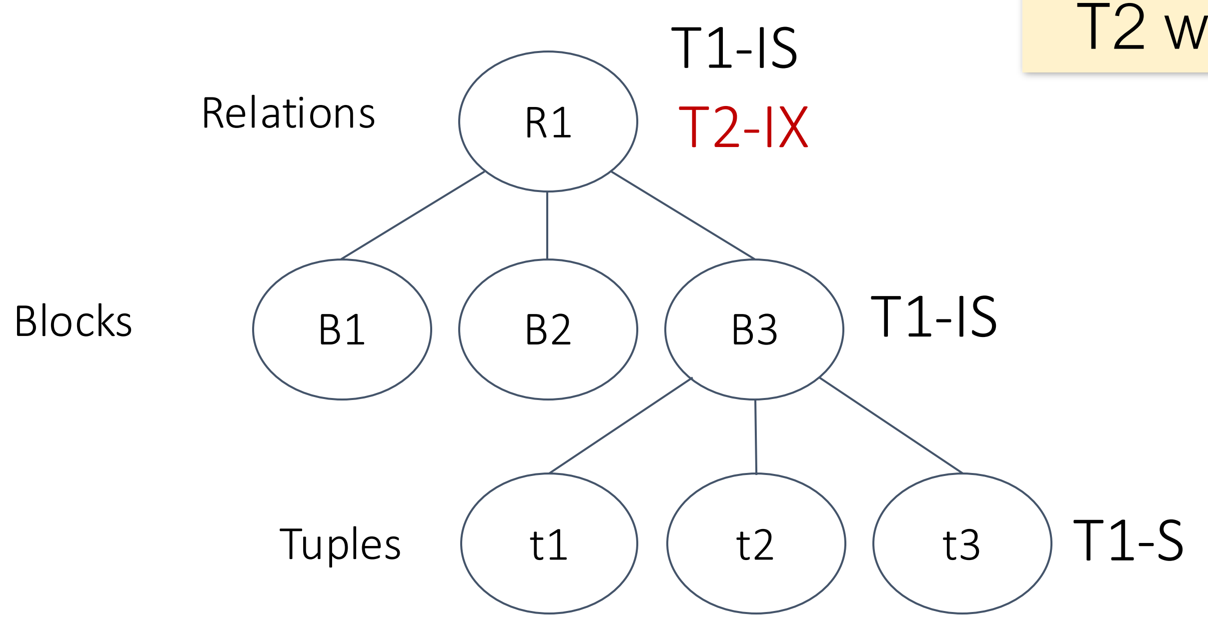


Warning locks

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

T1 wants to read t3

T2 wants to write B2

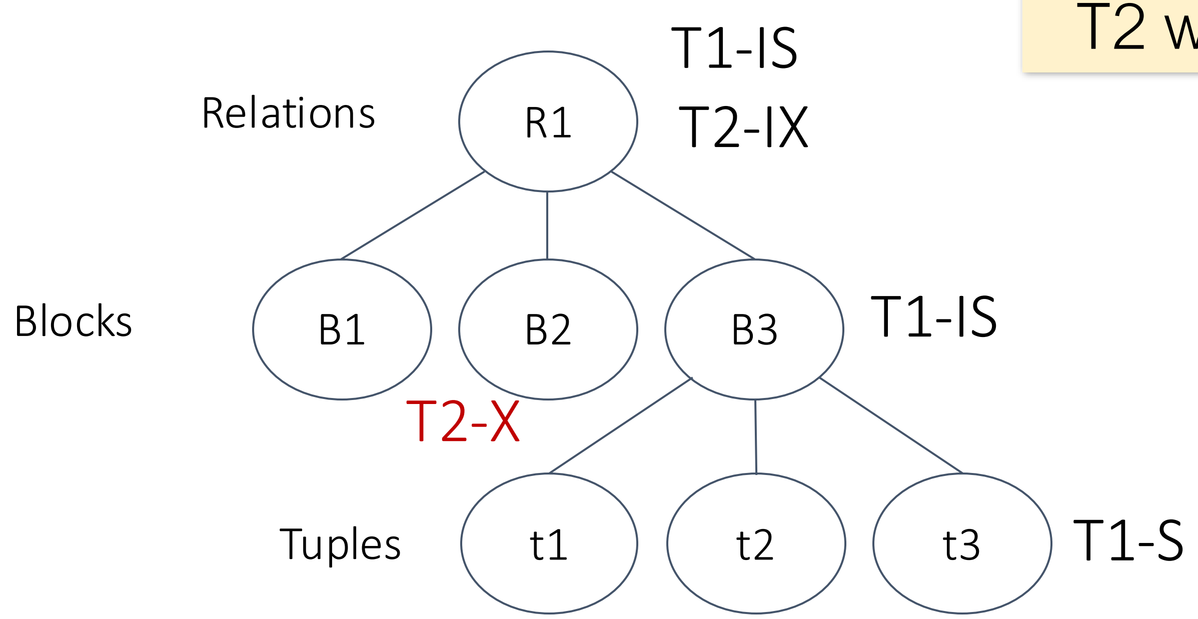


Warning locks

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

T1 wants to read t3

T2 wants to write B2



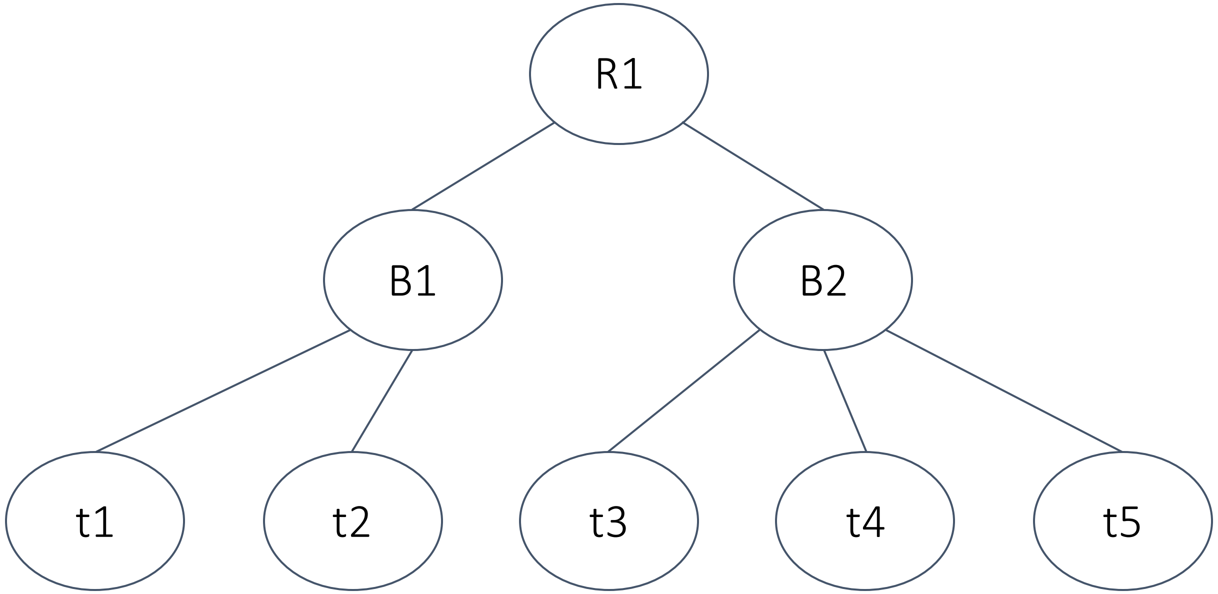
Compatibility matrix

- For shared, exclusive, and intention locks

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

In-class Exercise

- Given the hierarchy of objects, what is the sequence of lock requests by T1 and T2 for the sequence of requests: $r_1(t_5)$; $w_2(t_5)$; $w_1(t_4)$;



3. Optimistic Concurrency Control

Optimistic Concurrency Control

Optimistic methods

- Two methods: validation (covered next), and timestamping
- Assume no unserializable behavior
- Abort transactions when violation is apparent
- may cause transactions to rollback

In comparison, locking methods are pessimistic

- Assume things will go wrong
- Prevent nonserializable behavior
- Delays transactions but avoids 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 a read set RS(T) and write set WS(T)

Three phases of a transaction

- **Read** from DB all elements in RS(T) and store their writes in a private workspace
- **Validate** T by comparing RS(T) and WS(T) with other transactions
- **Write** elements in WS(T) to disk, if validation is OK (make private changes public)

Validation needs to be done atomically

- Validation order = hypothetical serial order

To validate, scheduler maintains three sets

START: set of transactions that started, but have not validated

- START(T), the time at which T started

VAL: set of transactions that validated, but not yet finished write phase

- VAL(T), time at which T is imagined to execute in the hypothetical serial order of execution

FIN: set of transactions that have completed write phase

- FIN(T), the time at which T finished.

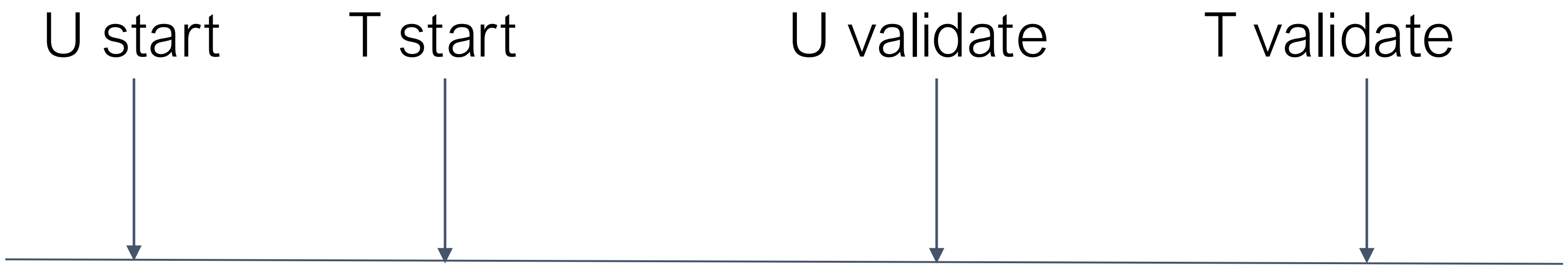
Validation rules (assume U validated)

Rule 1: if $FIN(U) > START(T)$, $RS(T) \cap WS(U) = \emptyset$

$WS(U) = \{A, B\}$

$RS(T) = \{B, C\}$

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



U start

T start

U validate

T validate

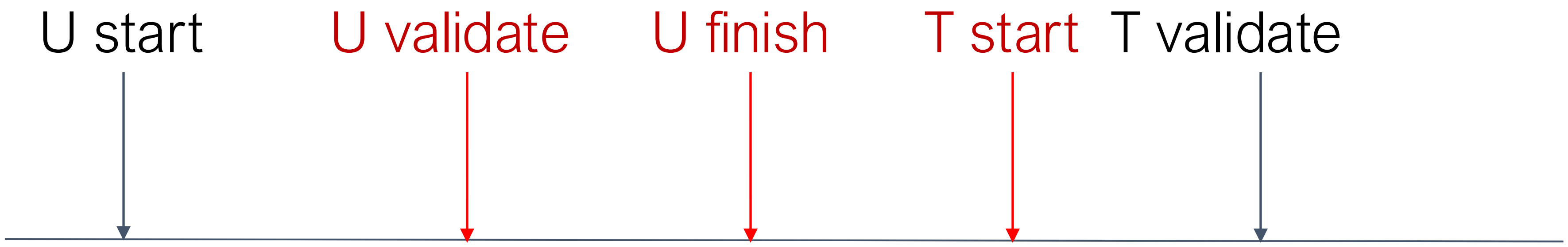
Validation rules (assume U validated)

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

WS(U) = {A, B}

RS(T) = {B, C}

This satisfies rule 1

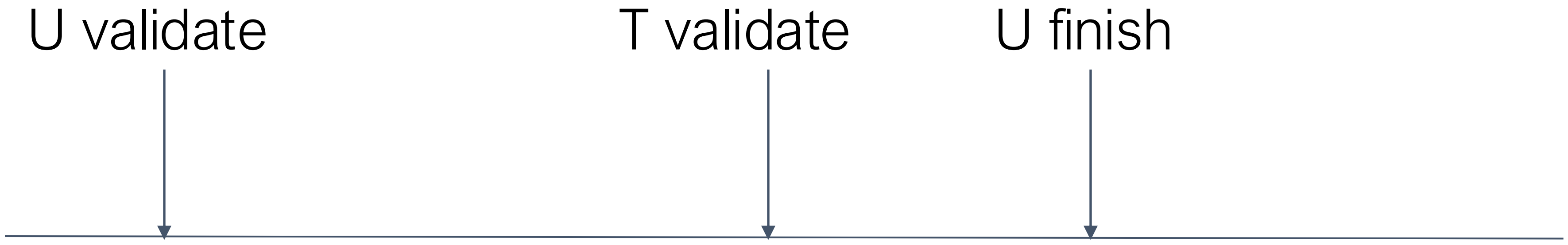


Validation rules (assume U validated)

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

WS(U) = {A, B}

WS(T) = {B, C}

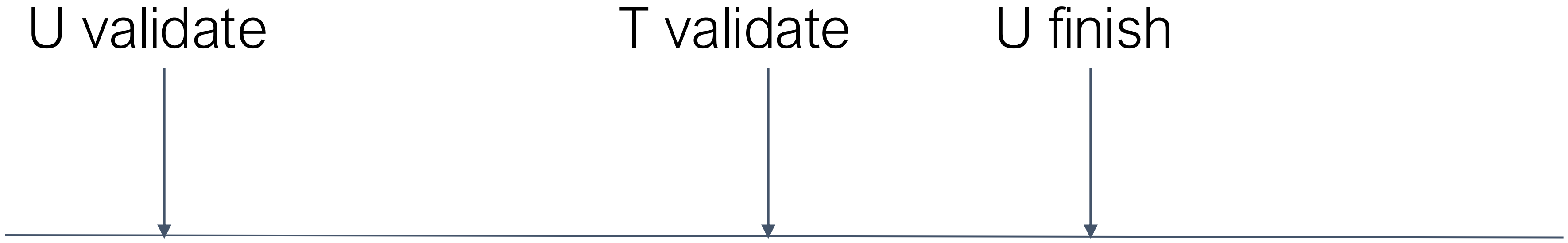


Validation rules (assume U validated)

Rule 2: if $FIN(U) > VAL(T)$, $WS(T) \cap WS(U) = \emptyset$

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

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



U validate

T validate

U finish

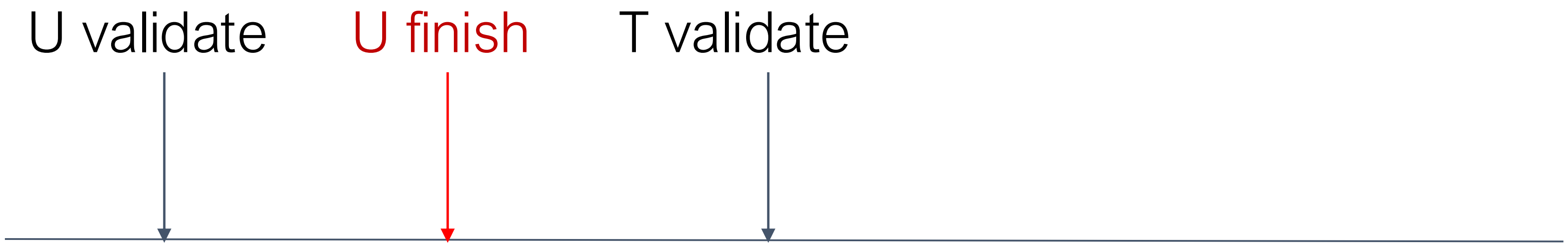
Validation rules (assume U validated)

Rule 2: if $\mathrm{FIN}(U) > \mathrm{VAL}(T)$, $\mathrm{WS}(T) \cap \mathrm{WS}(U) = \emptyset$

$\mathrm{WS}(U) = \{A, B\}$

$\mathrm{WS}(T) = \{B, C\}$

This satisfies rule 2

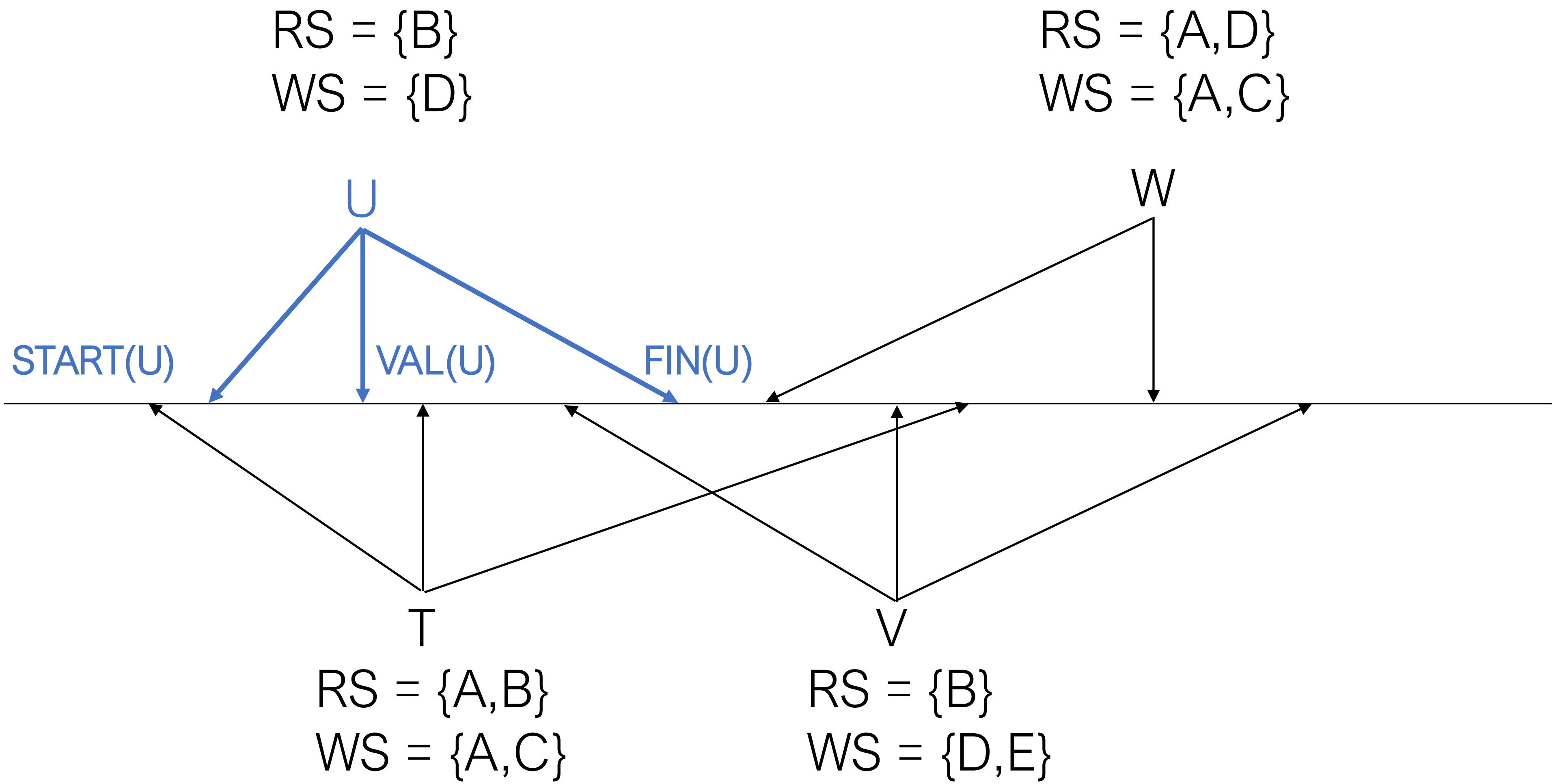


U validate

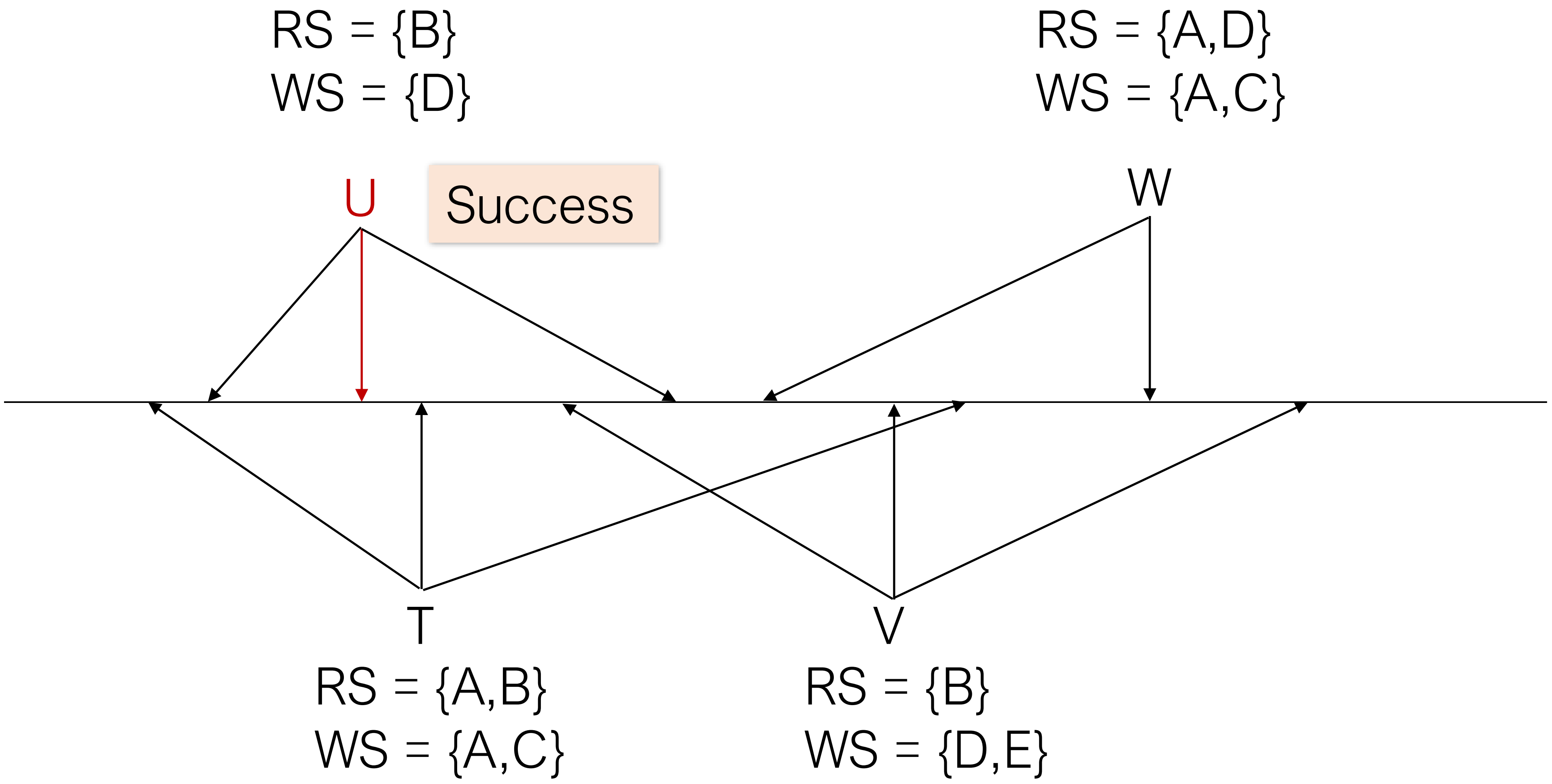
U finish

T validate

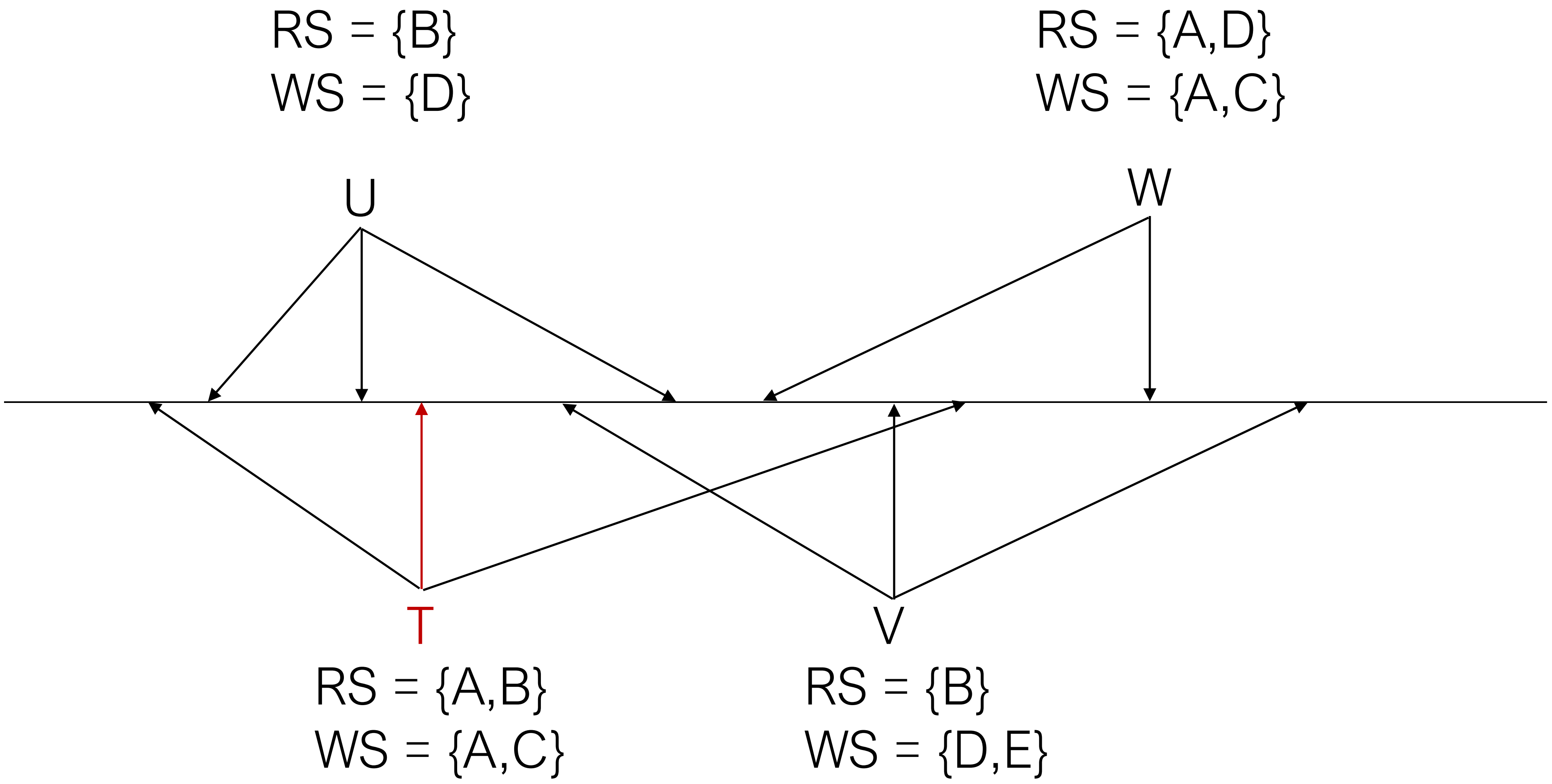
Example: CC by Validation



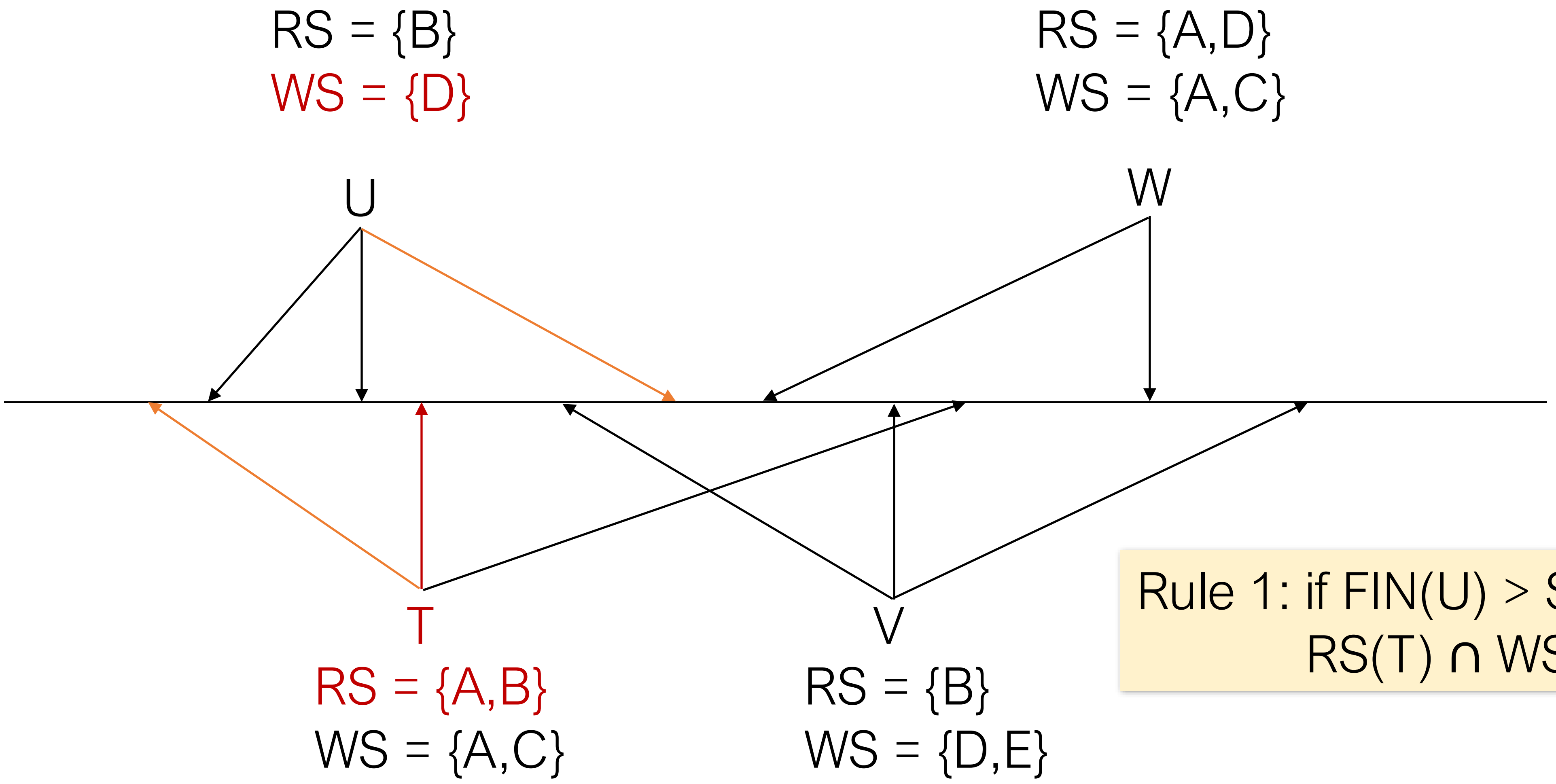
Example: CC by Validation



Example: CC by Validation

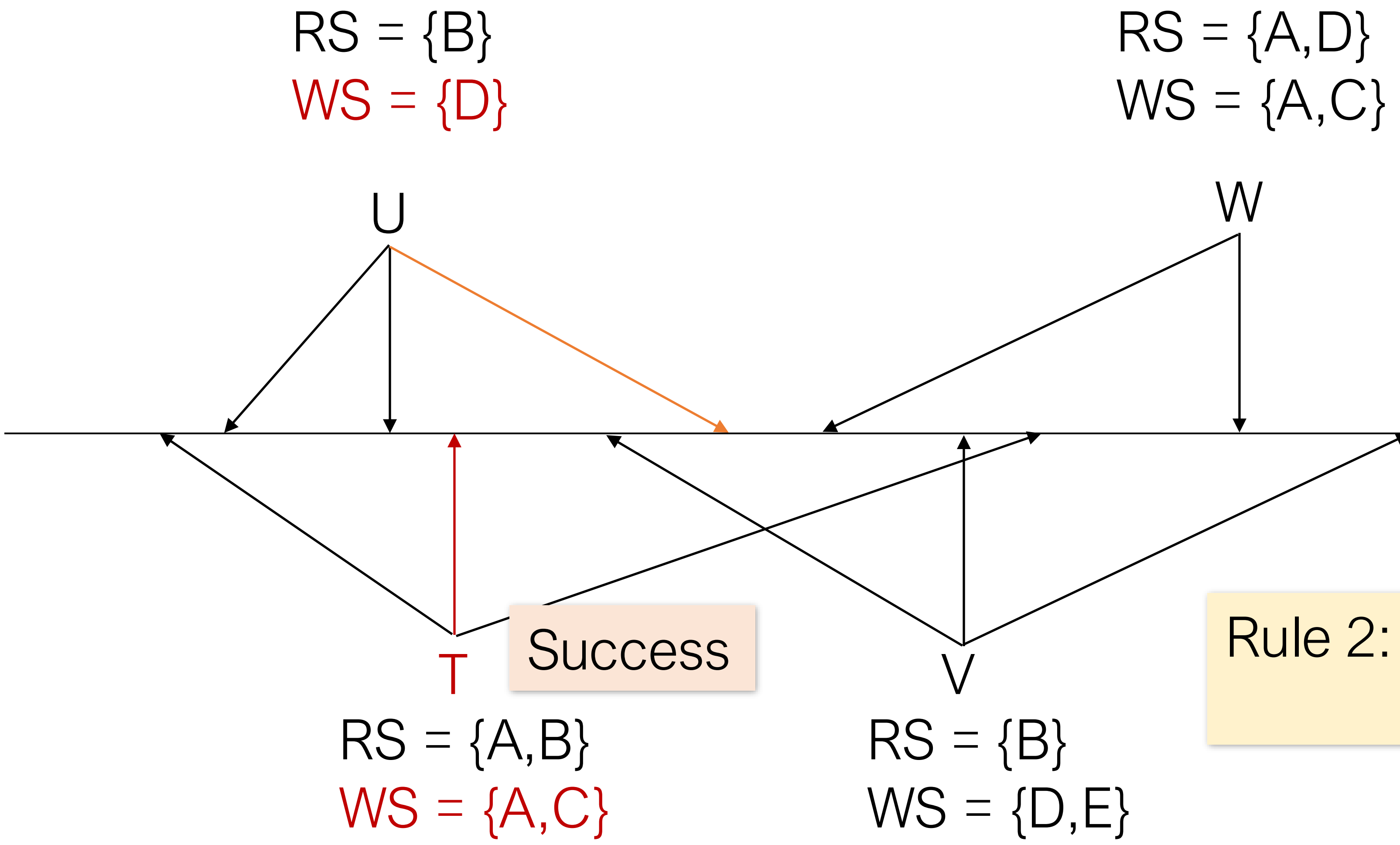


Example: CC by Validation



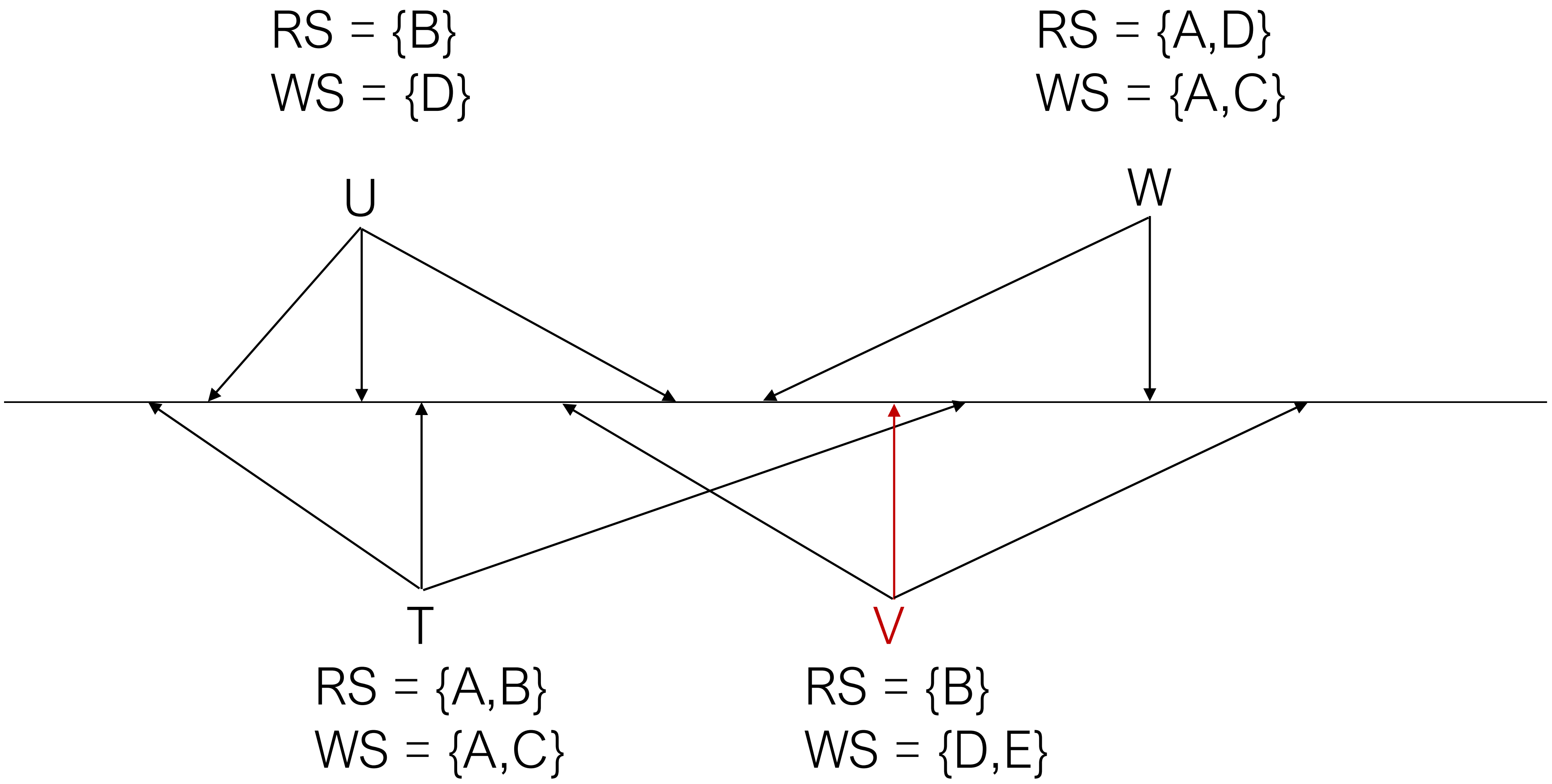
Rule 1: if FIN(U) > START(T),
RS(T) ∩ WS(U) = ∅

Example: CC by Validation



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

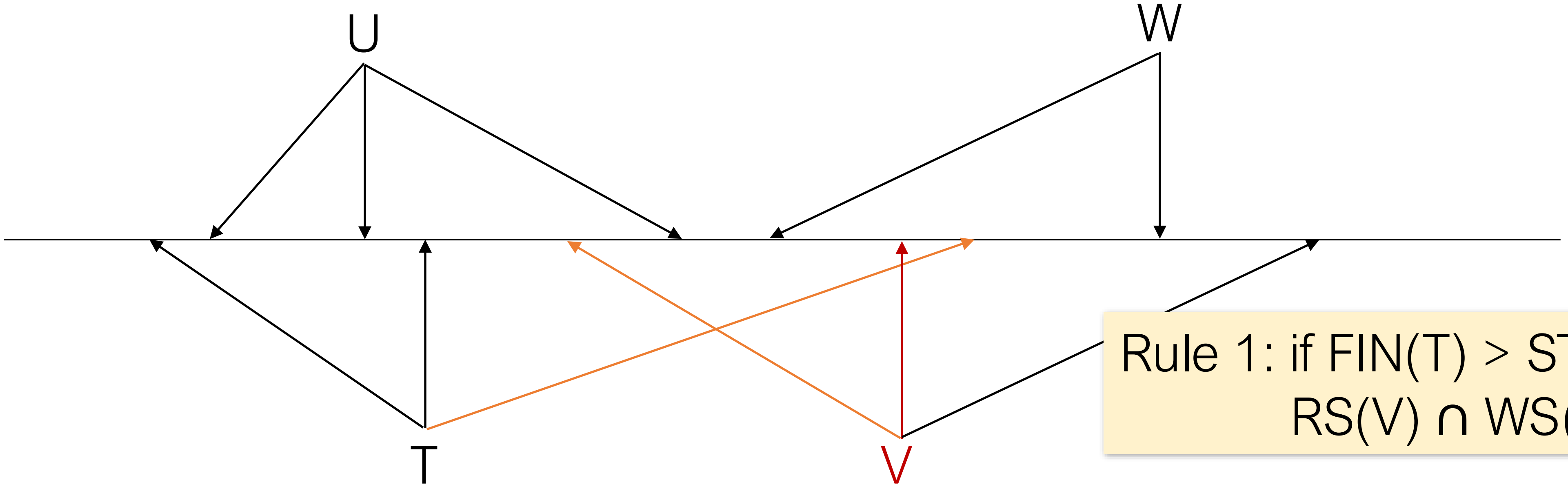
Example: CC by Validation



Example: CC by Validation

RS = {B}
WS = {D}

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

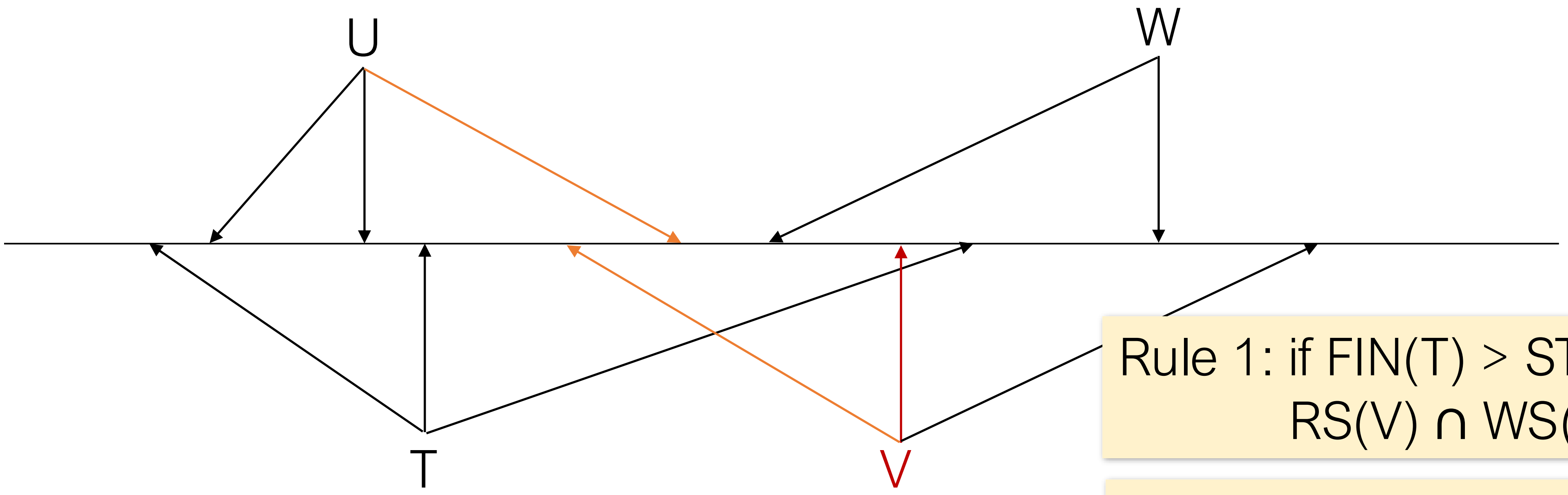


RS = {A,B}
WS = {A,C}

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

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

Example: CC by Validation



RS = {B}
WS = {D}

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

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

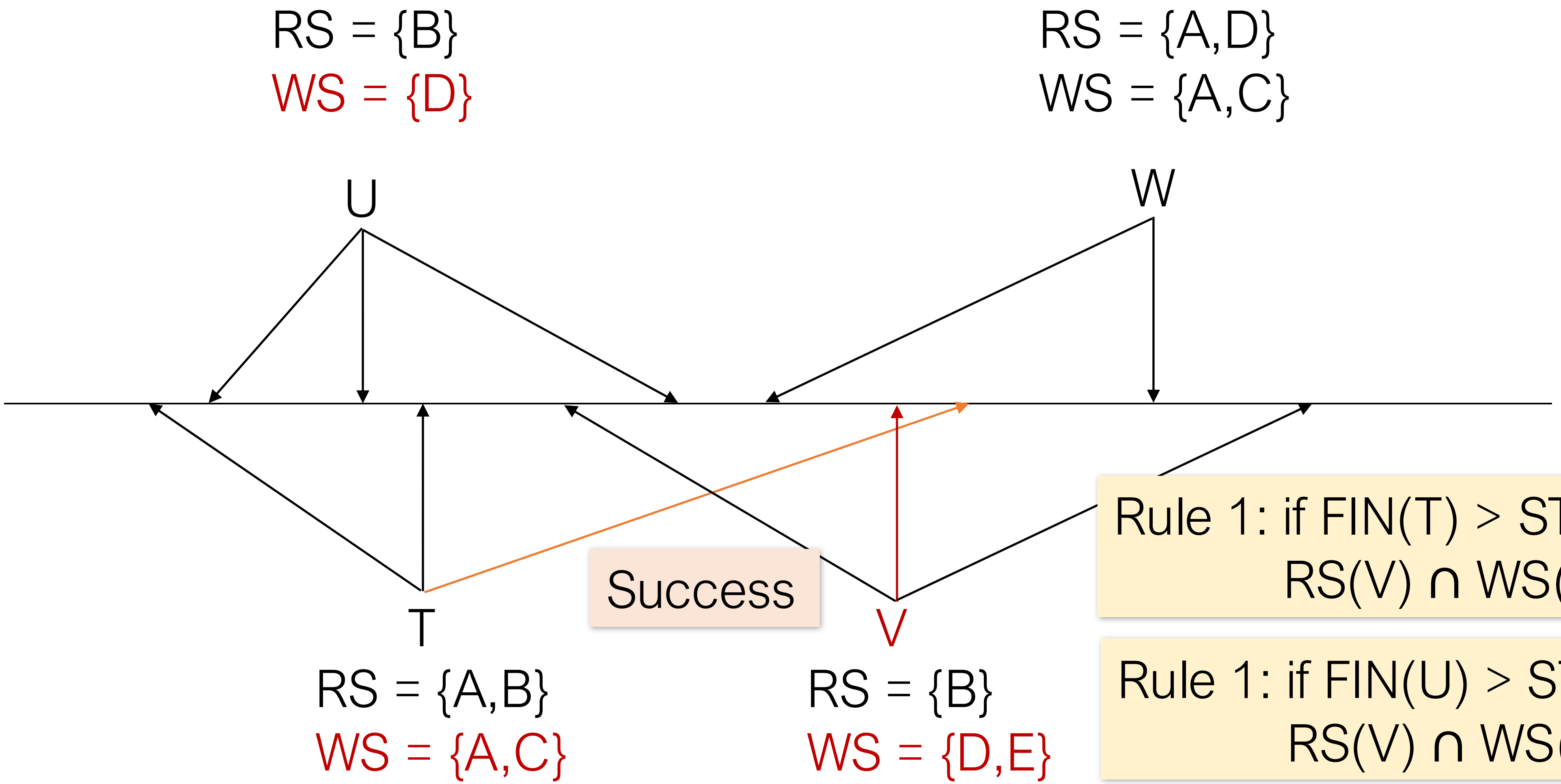
V
RS = {B}
WS = {D,E}

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) = ∅



RS = {B}
WS = {D}

U

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

W

Success

T

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

V

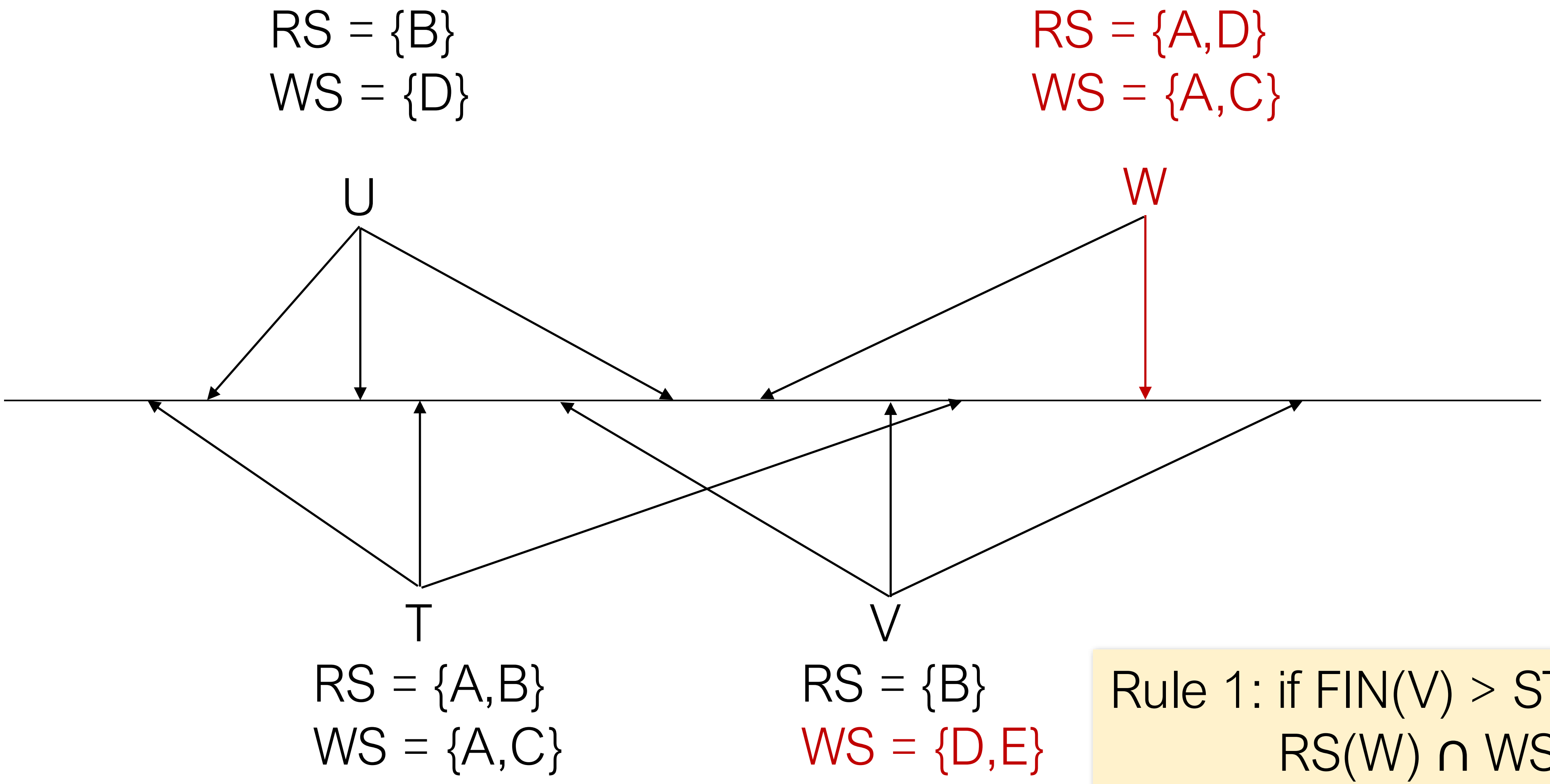
RS = {B}
WS = {D,E}

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) = ∅



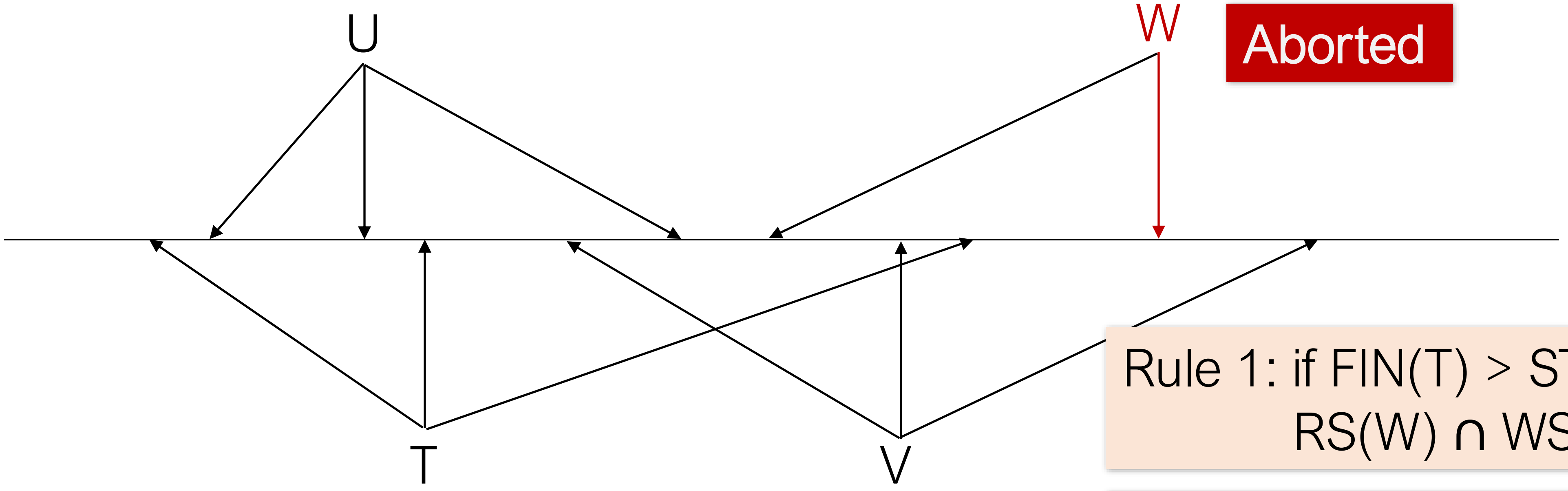
Rule 1: if FIN(V) > START(W),
RS(W) ∩ WS(V) = ∅

Example: CC by Validation

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

RS = {B}
WS = {D}

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



Aborted

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

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

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

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

One more non-locking CC Techniques

Multi-version Concurrency Control (MVCC)

The DBMS maintains multiple physical versions of a single logical object in the database:

- When a TXN writes to an object, the DBMS creates a new version of that object.
- When a TXN reads an object, it reads the newest version that existed when the TXN started.

More on MVCC

Each transaction is classified as reader or writer.

- Readers don’t block writers. Writers don’t block readers.

Read-only txns can read a consistent snapshot without acquiring locks.

- Use timestamps to determine visibility.

Easily support time-travel queries.

Comparison of CC Techniques

Techniques	Conflict Resolution	Behavior	Concurrency
Locking	Prevents conflicts upfront	TXNs may block waiting for locks	Lower
Validation	Detect conflicts at commit	No blocking during execution, but may abort at validation time	Higher
MVCC	Avoid conflicts via versioning	Generally non-blocking for reads, may have conflicts for writes	Higher