CS 6400 A

Database Systems Concepts and Design

Lecture 15

10/21/24

Desirable Properties of Transactions: ACID

- Atomicity: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

- Consistency: A correct execution of the transaction must take the database from one consistent state to another.

- Isolation: A transaction should not make its updates visible to other transactions until it is committed.

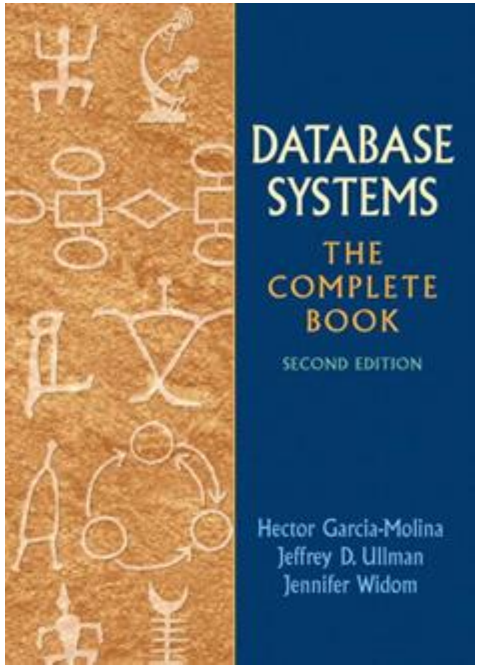
- Durability: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

This class: ensuring isolation via concurrency control

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 18 – Concurrency Control



Supplementary materials

Fundamental of Database Systems (7th Edition)

- Chapter 21 - Concurrency Control Techniques

Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang.

Agenda

1. Locking-based Concurrency Control
2. Optimistic Concurrency Control
3. Multi-version Concurrency Control

1. Lock-based Concurrency Control

Enforce serializability with locks

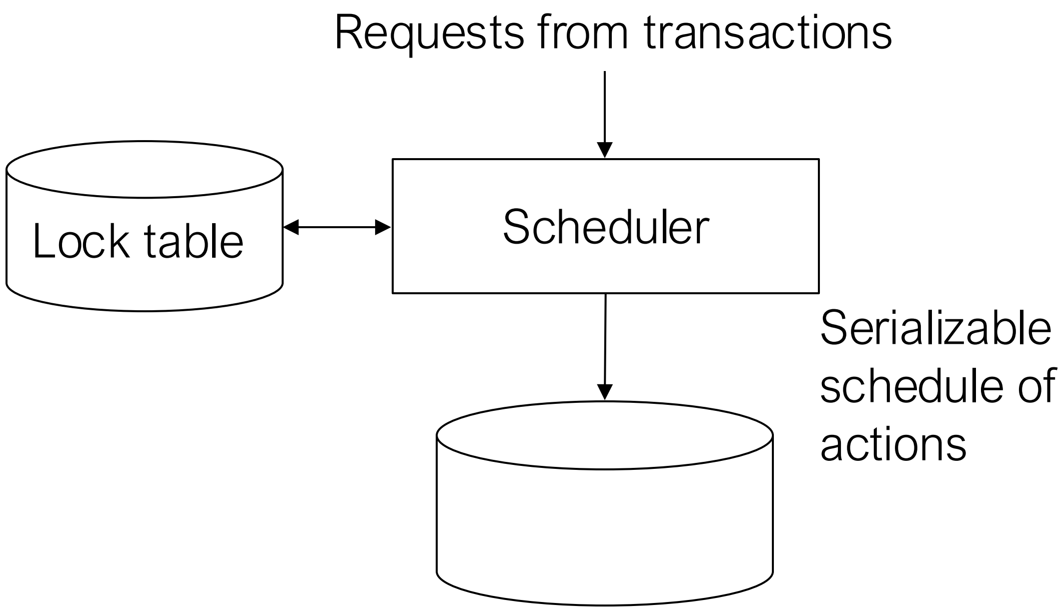
$l_i(X)$: Ti requests lock on X
$u_i(X)$: Ti releases lock on X

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

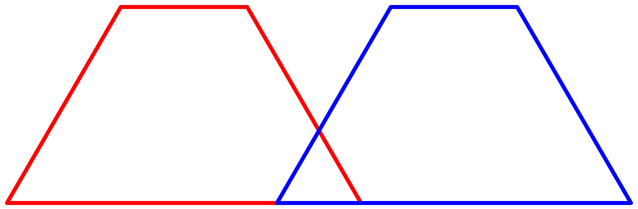
Two-phase locking (2PL)

- 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

- 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

Locking with several modes

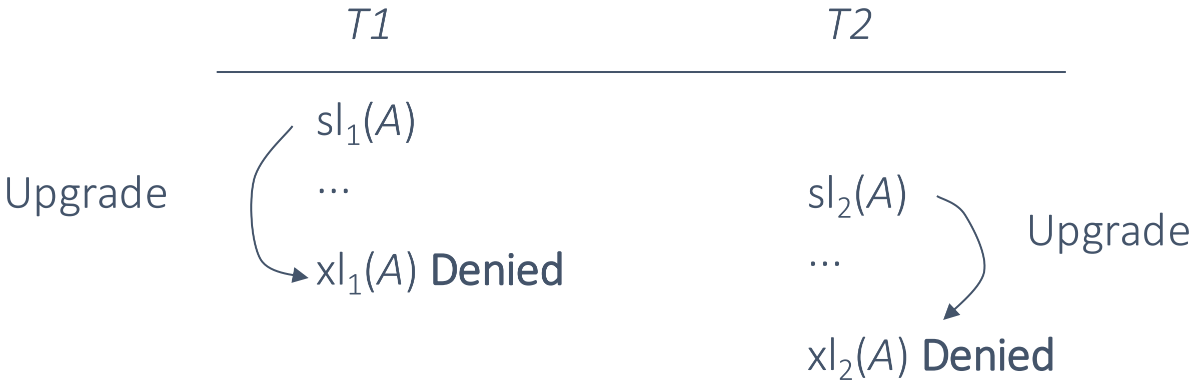
- Compatibility matrix

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

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



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*	
Upgrade	$ul_1(A)$		
	…	$ul_2(A)$ **Denied**	
	$xl_1(A)$		
		$ul_2(A)$	Upgrade
		…	
		$xl_2(A)$	

Compatibility matrix

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

Locks With Multiple Granularity

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

A few options:

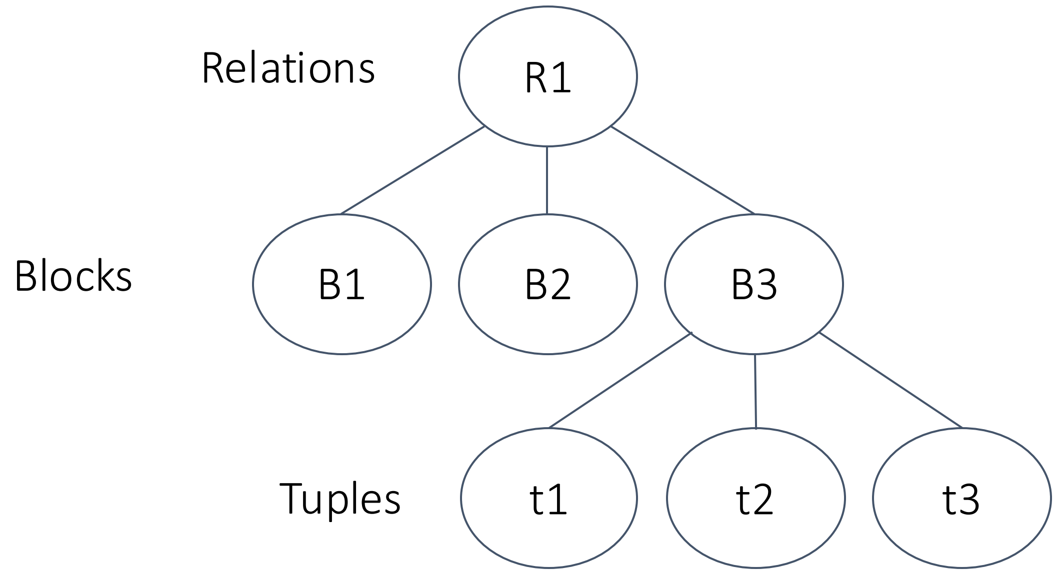
- Relations → Least concurrency
- Pages or data blocks
- Tuples → Most concurrency, but also expensive

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

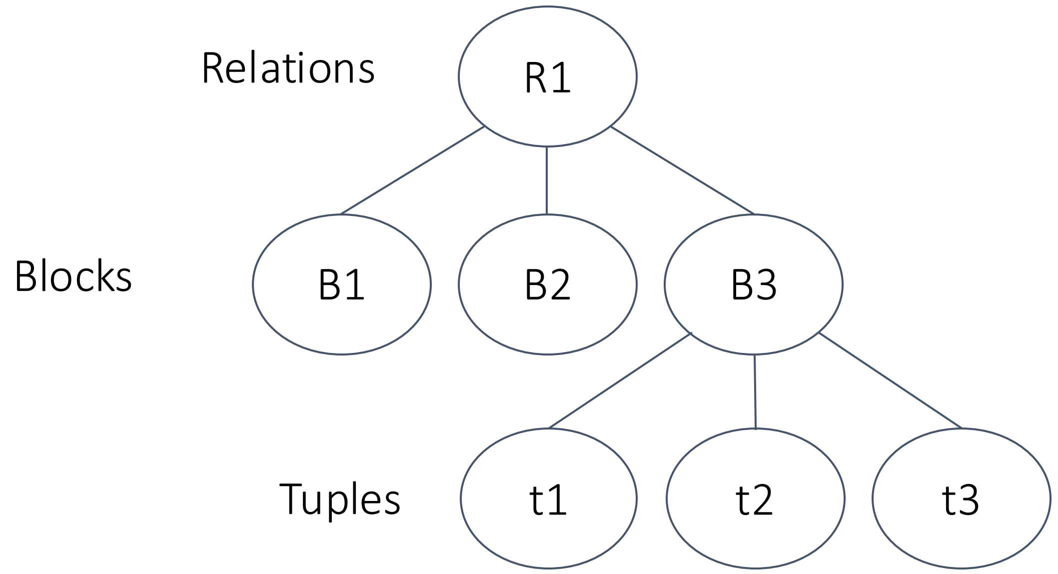
- Ordinary locks: S and X
- Warning locks: I (shows intention to lock)



Warning locks

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

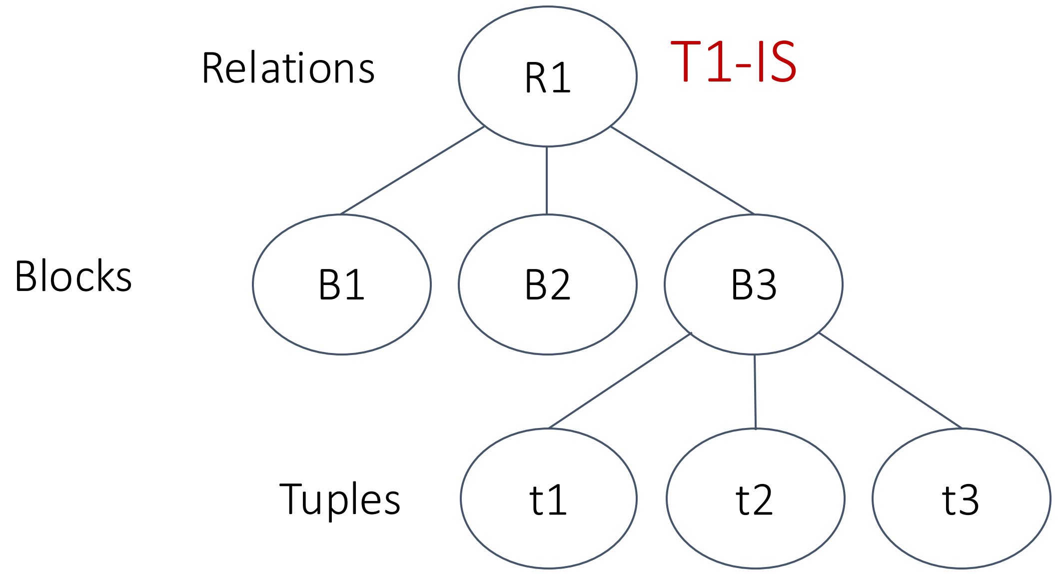
T1 wants to read t3



Warning locks

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

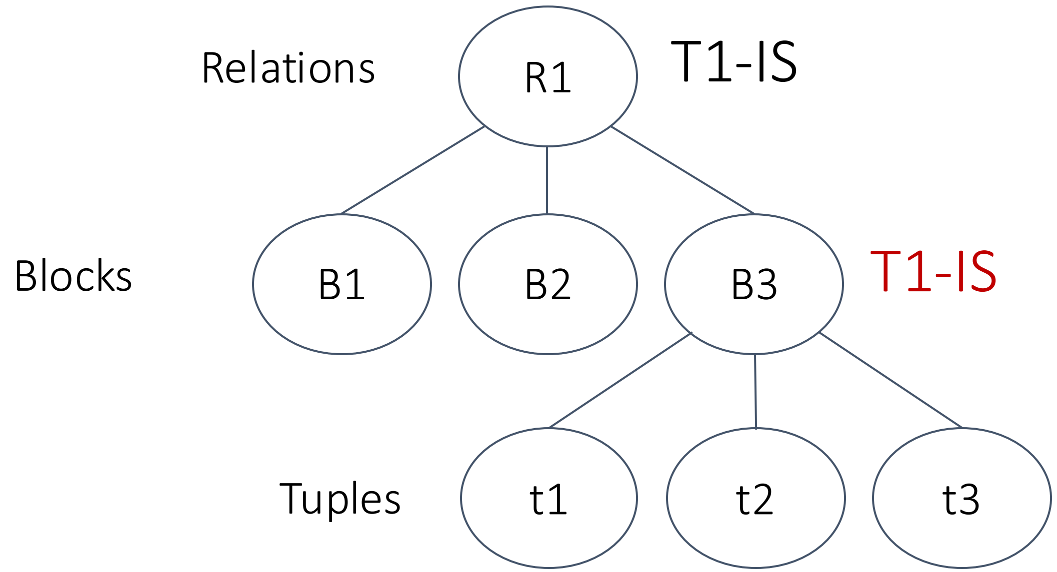
T1 wants to read t3



Warning locks

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

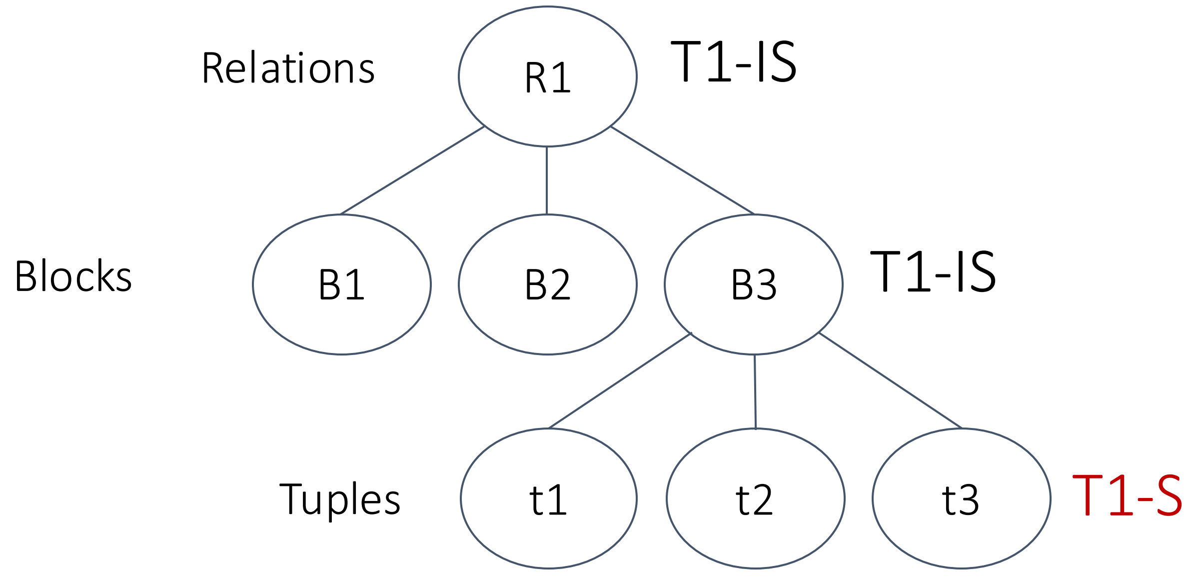
T1 wants to read t3



Warning locks

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

T1 wants to read t3

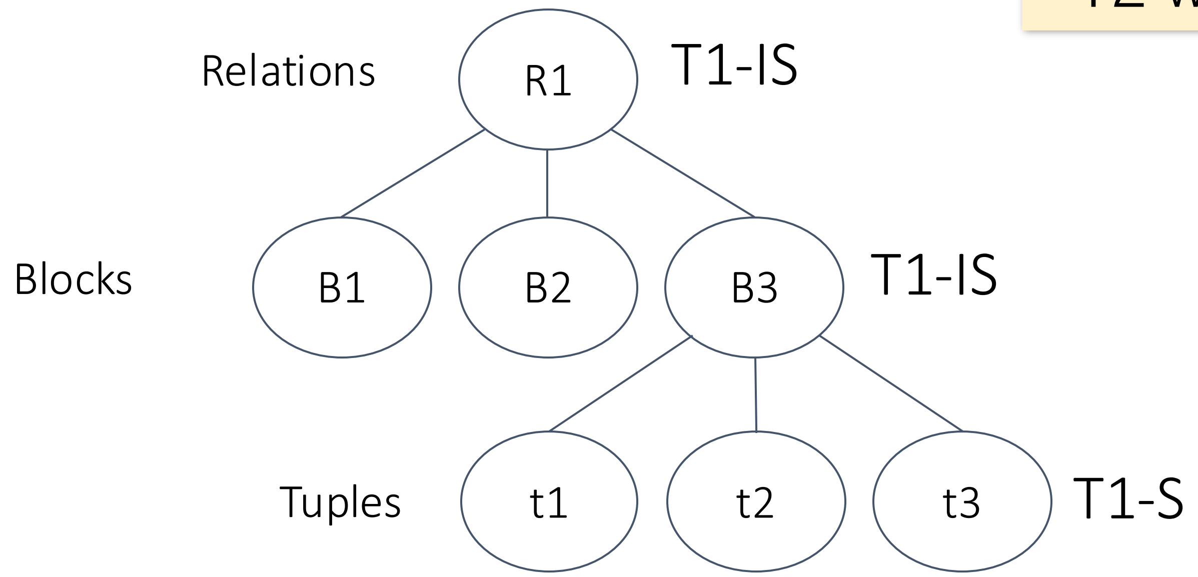


Warning locks

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

T1 wants to read t3

T2 wants to write B2

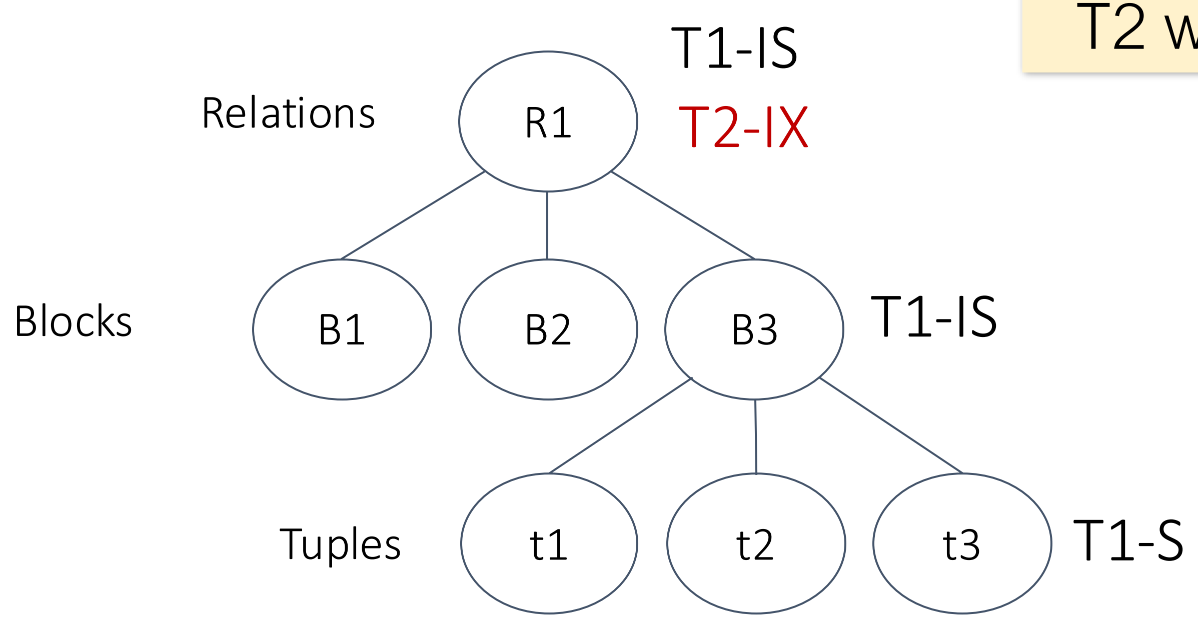


Warning locks

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

T1 wants to read t3

T2 wants to write B2

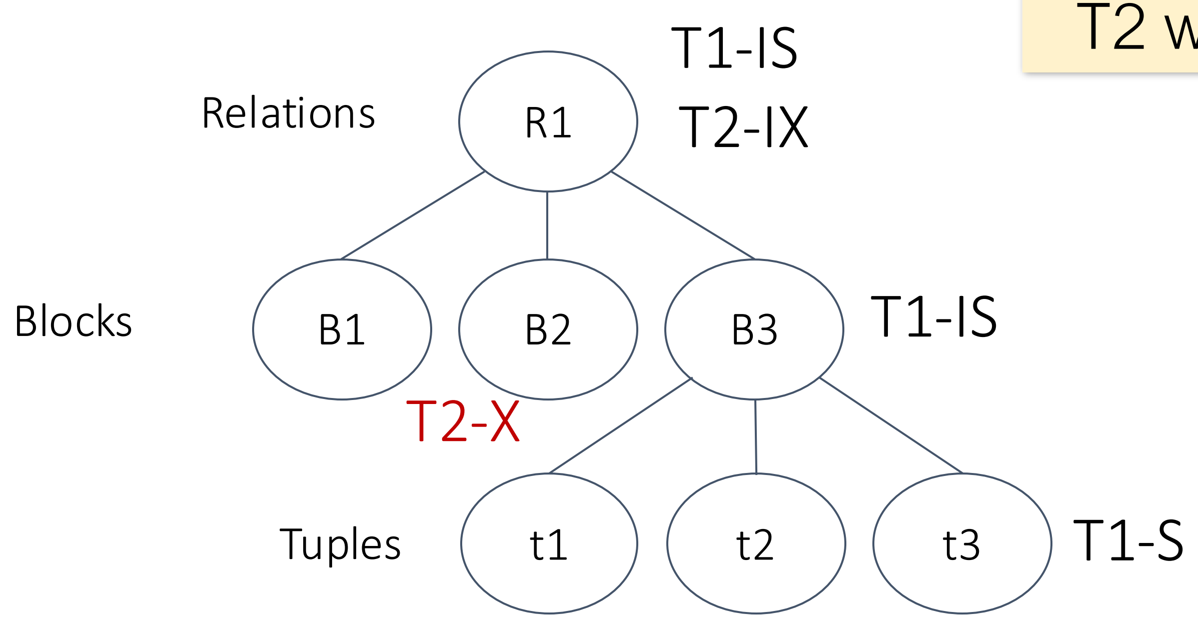


Warning locks

- 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

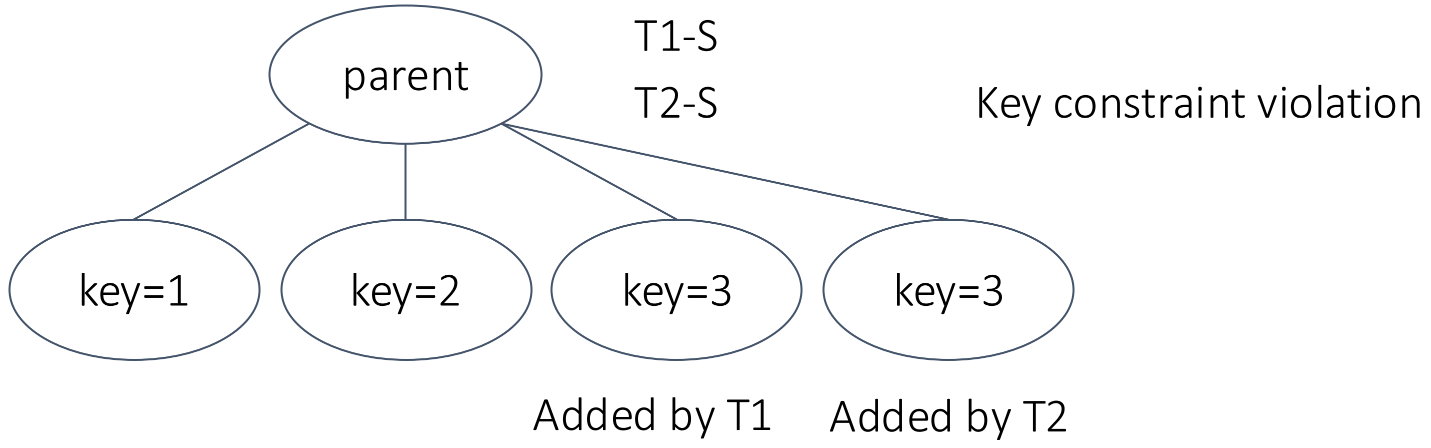
Holder		IS	IX	S	X
	IS	Yes	Yes	Yes	No
	IX	Yes	Yes	No	No
	S	Yes	No	Yes	No
	X	No	No	No	No

Inserts and Deletes

Delete: get exclusive lock on X before deleting it

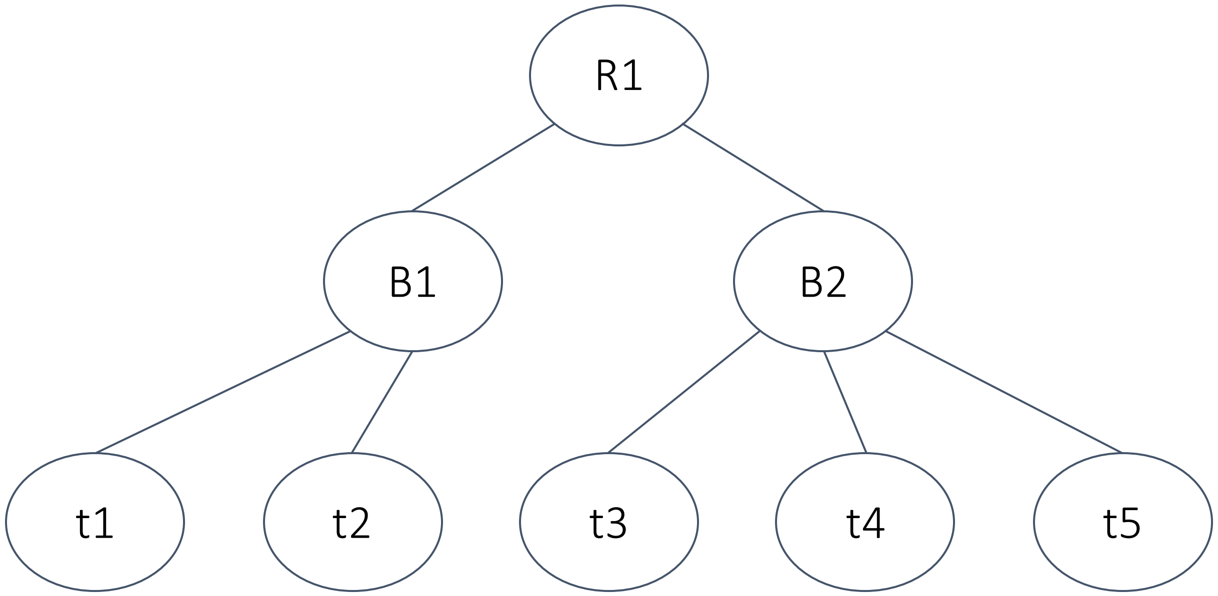
Insert: get exclusive lock on the parent of the new tuple

- If no exclusive lock is held, then database can become inconsistent due to “phantoms”



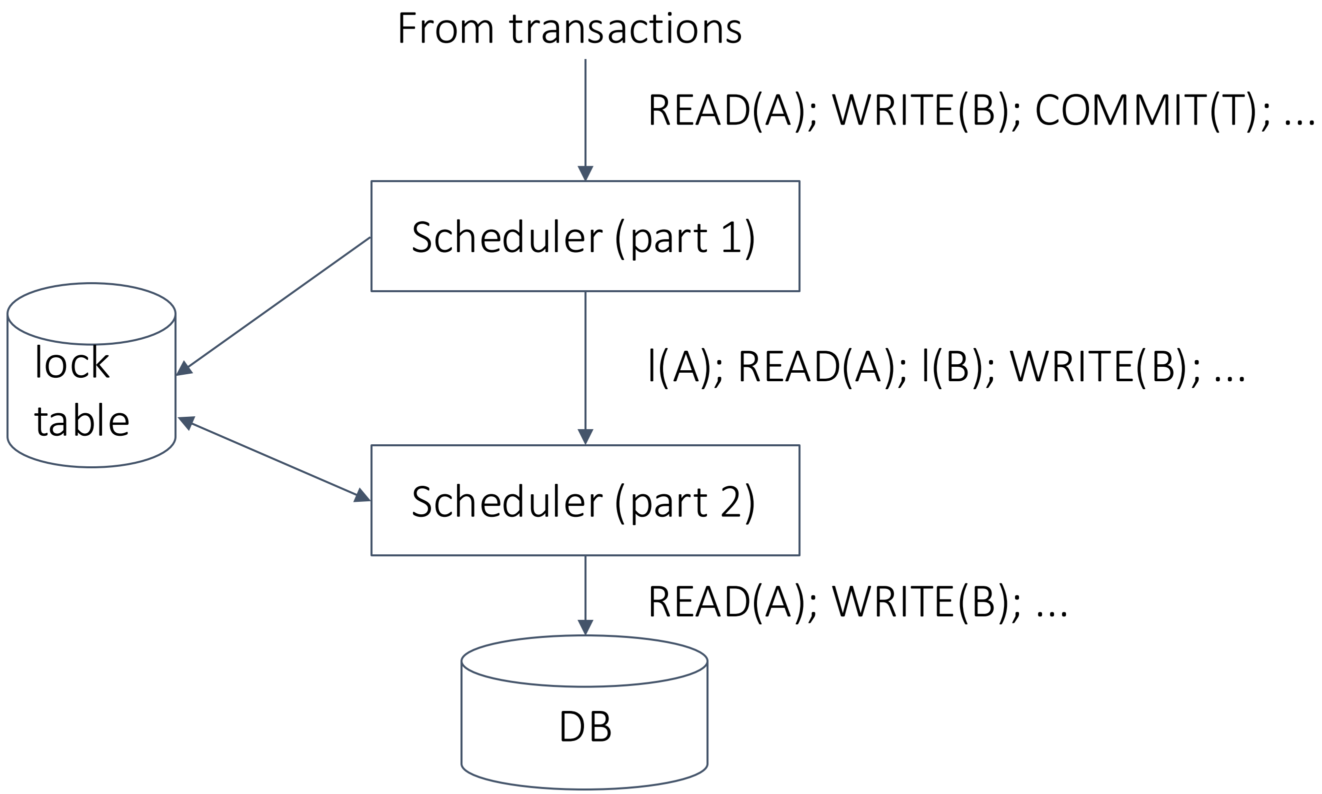
In-class Exercise

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



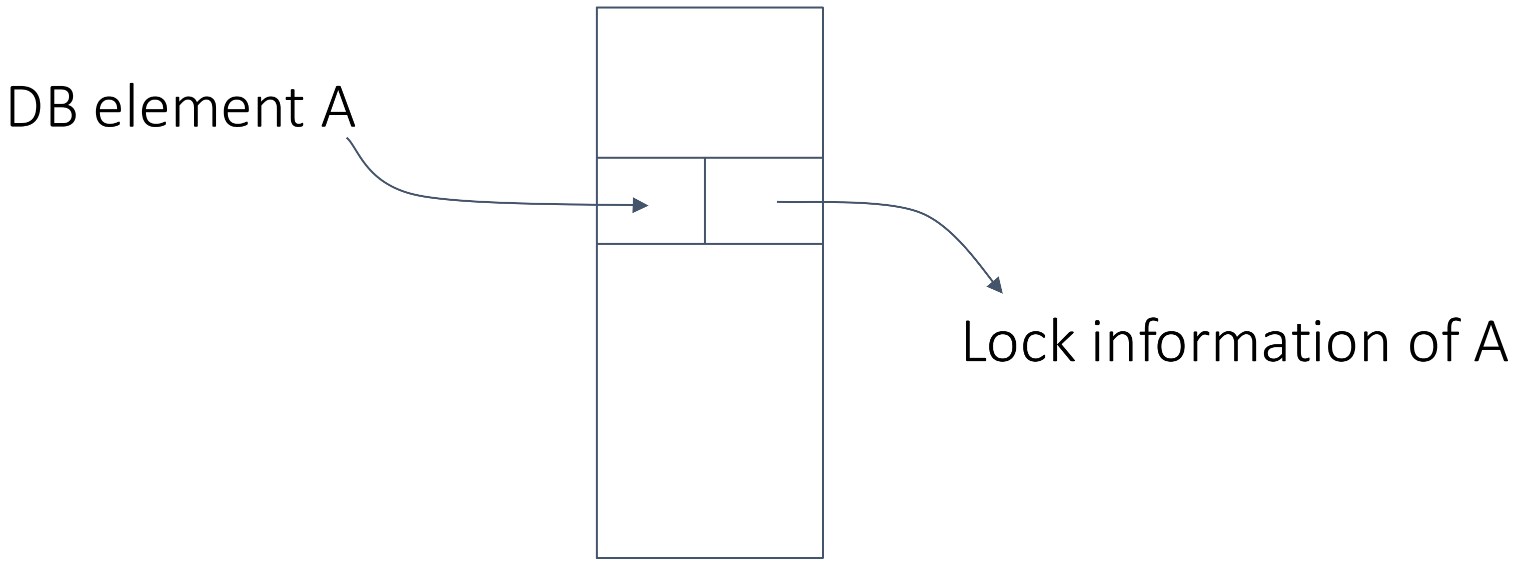
Locking scheduler architecture

- Part 1 takes stream of requests and inserts appropriate lock actions
- Part 2 executes the sequences from Part 1



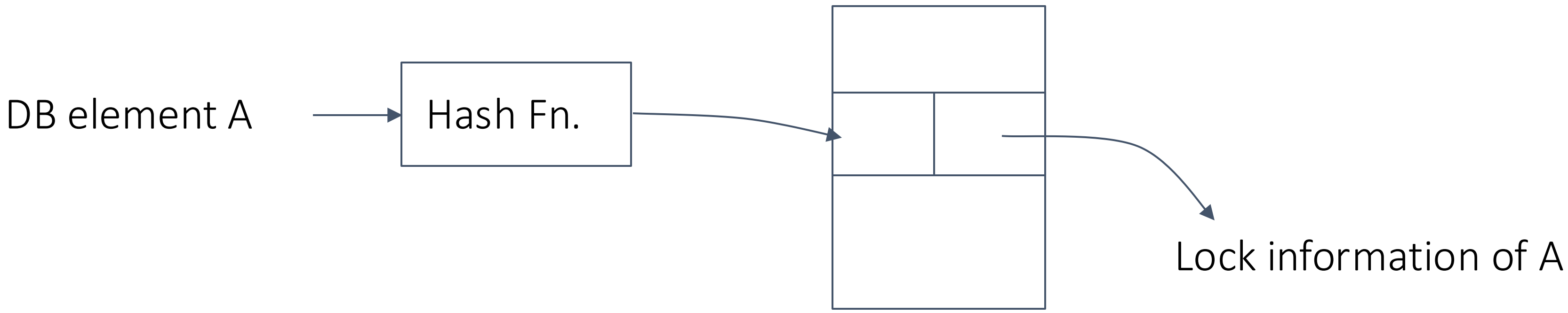
Lock table

- Maps database elements to lock information



Lock table

- Can implement with hash table
- If element is not in table, it is unlocked



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)

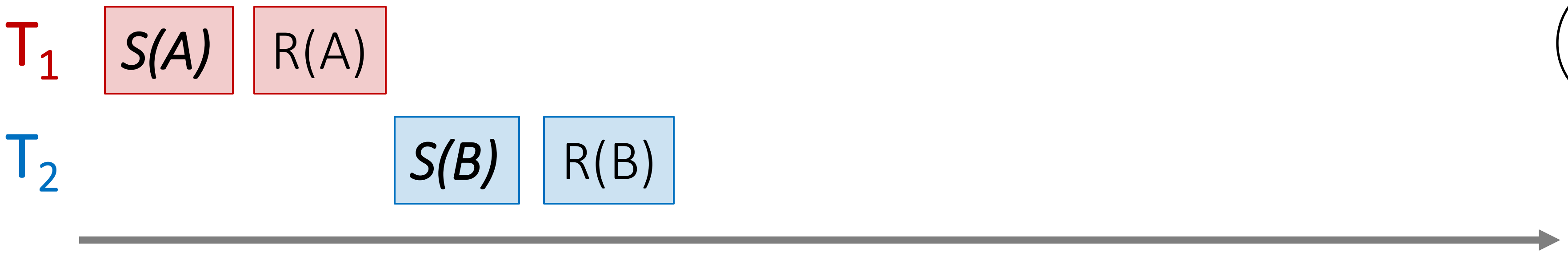
Deadlock Detection: Example

Waits-for graph:

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

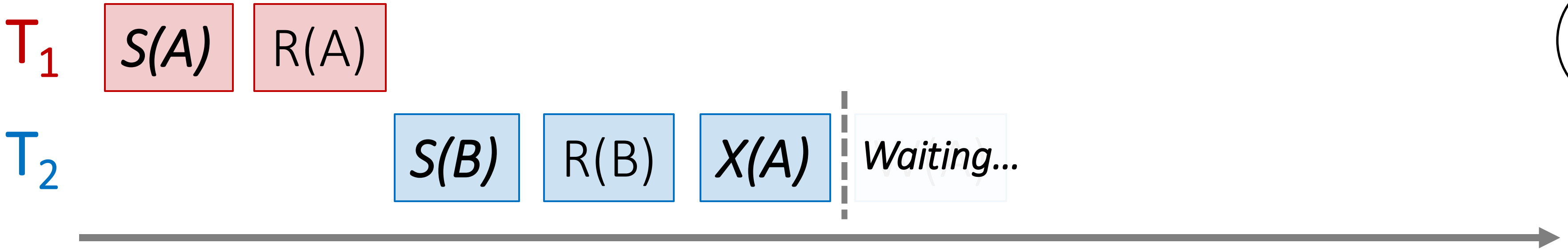
Deadlock Detection: Example

Waits-for graph:



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

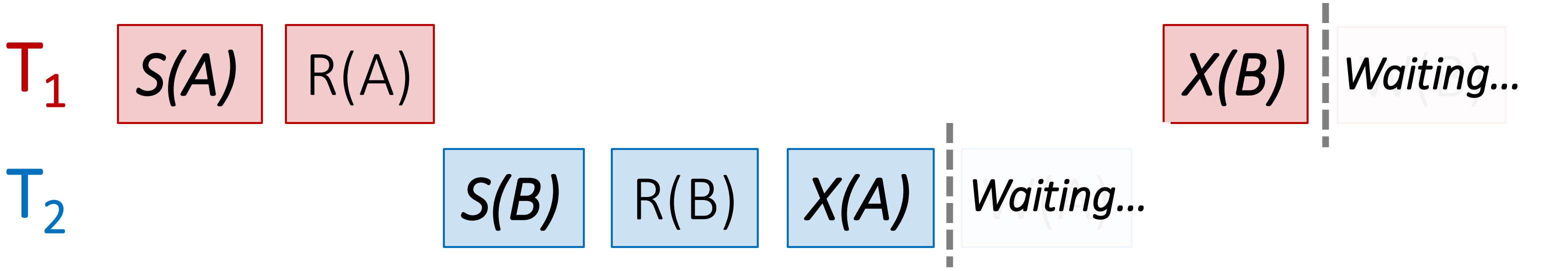
Deadlock Detection: Example



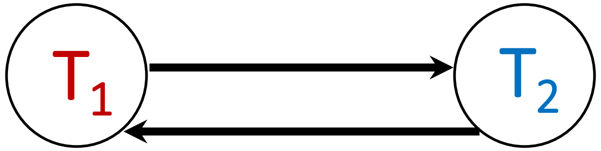
Waits-for graph:

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

2. 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

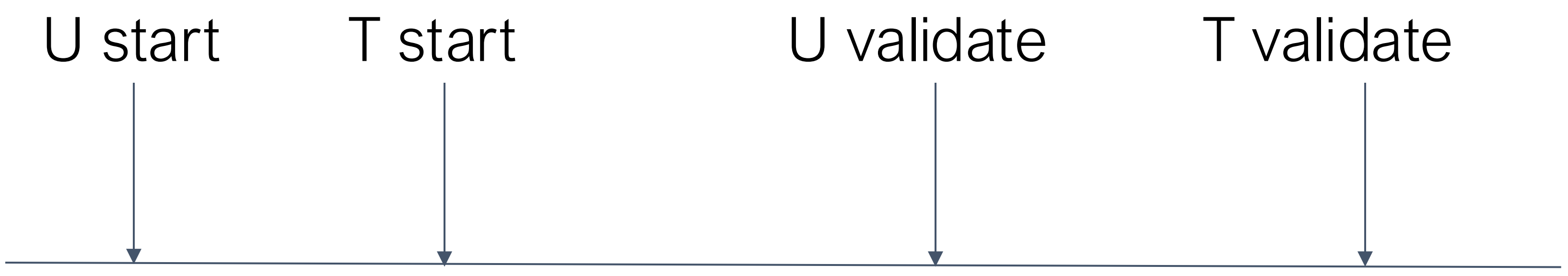
- FIN(T), the time at which T finished.

Validation rules (assume U validated)

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

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

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



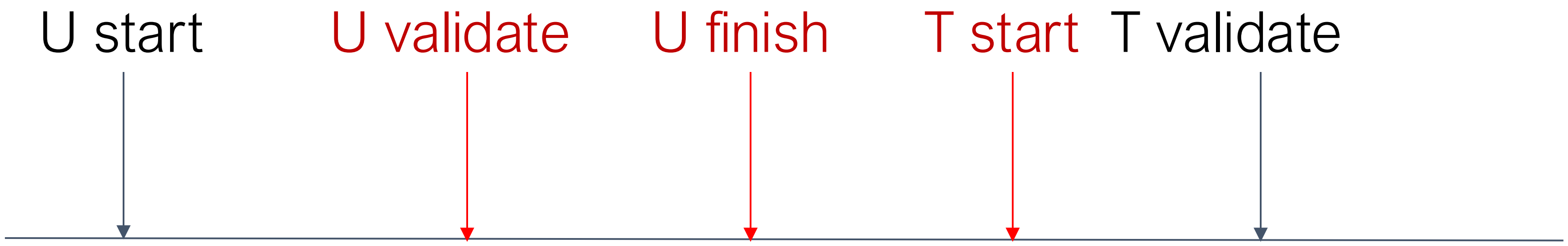
Validation rules (assume U validated)

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

WS(U) = {A, B}

RS(T) = {B, C}

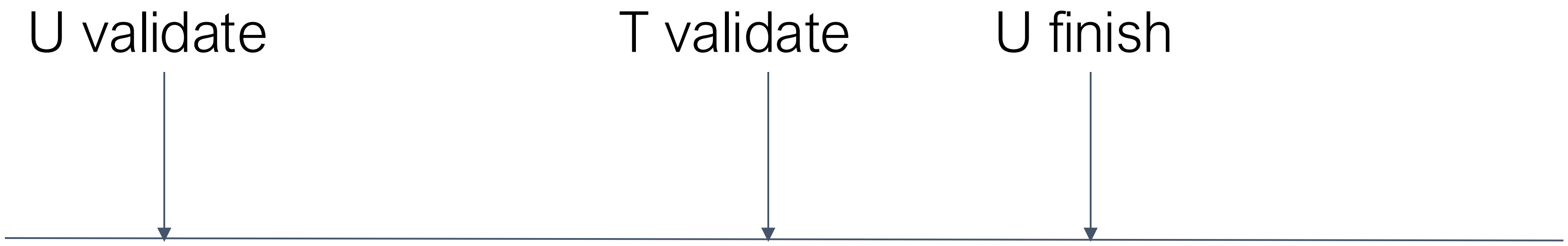
This satisfies rule 1



Validation rules (assume U validated)

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

WS(U) = {A, B}　　　　WS(T) = {B, C}

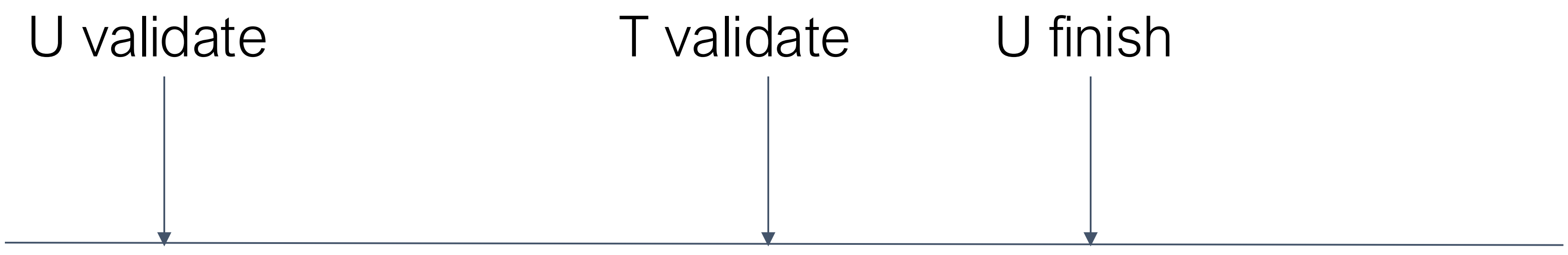


Validation rules (assume U validated)

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



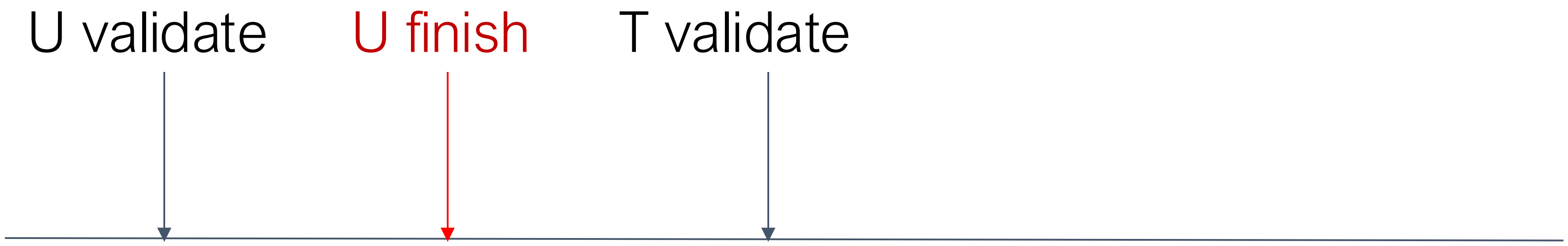
Validation rules (assume U validated)

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

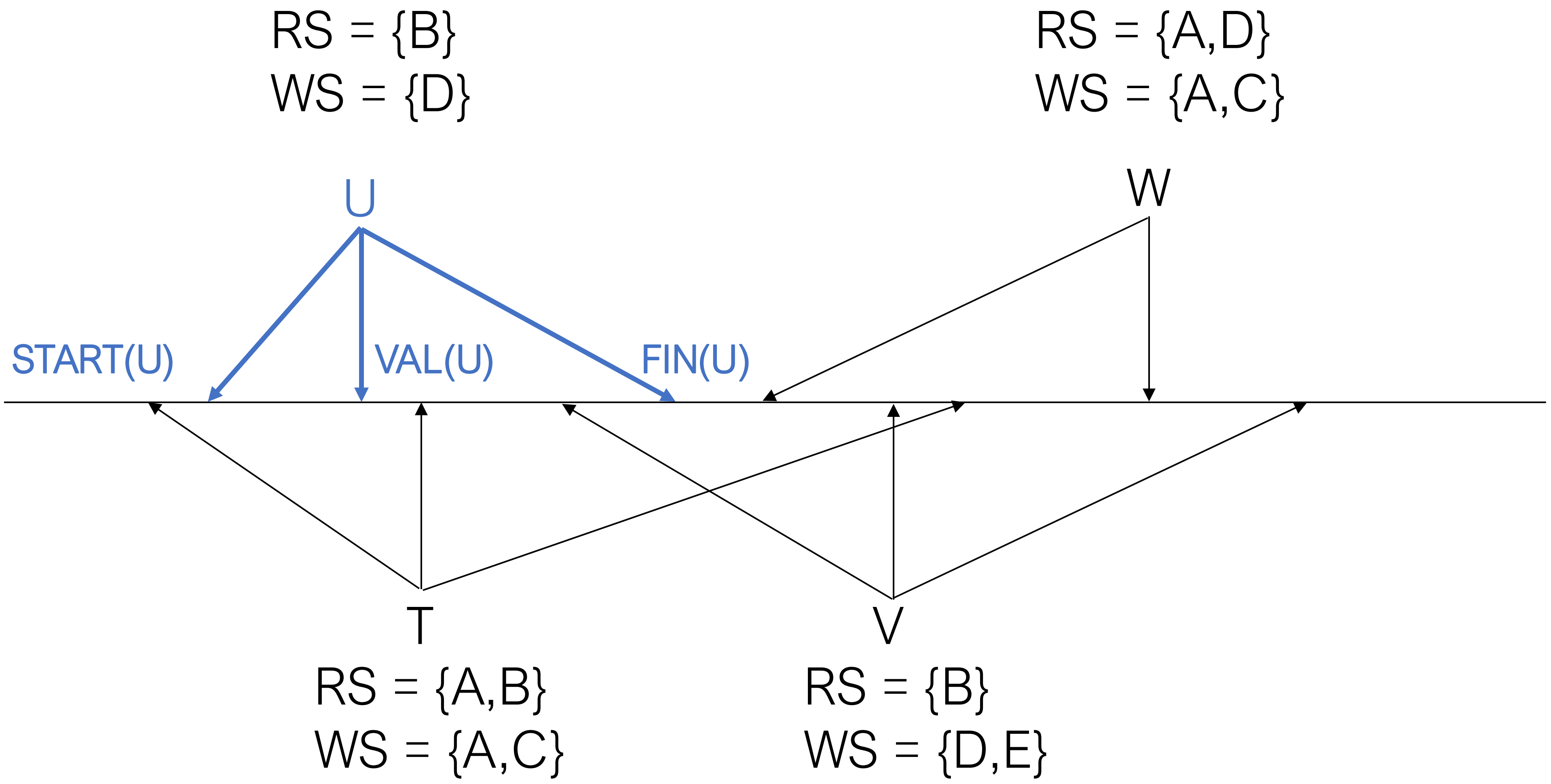
WS(U) = {A, B}

WS(T) = {B, C}

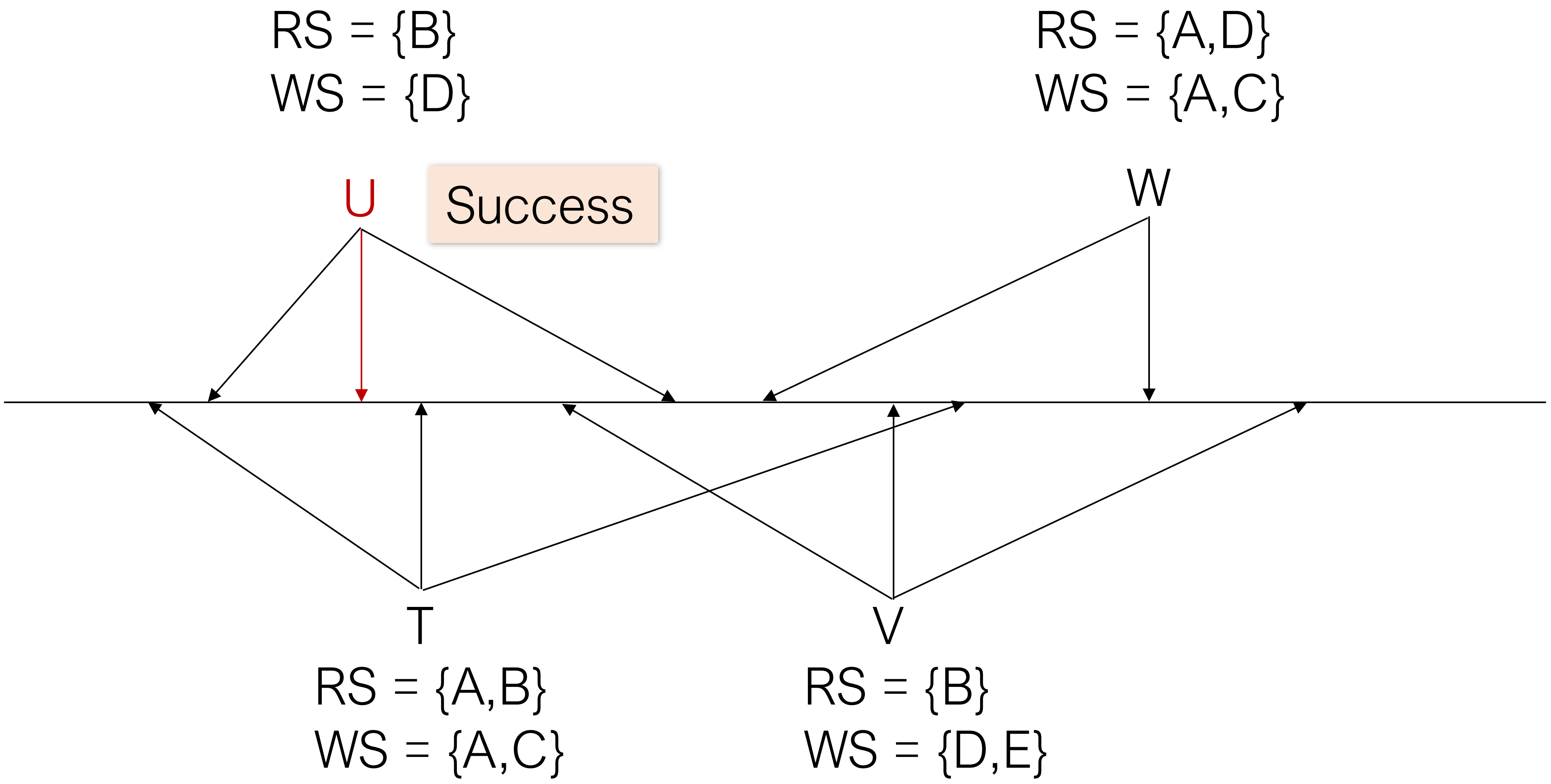
This satisfies rule 2



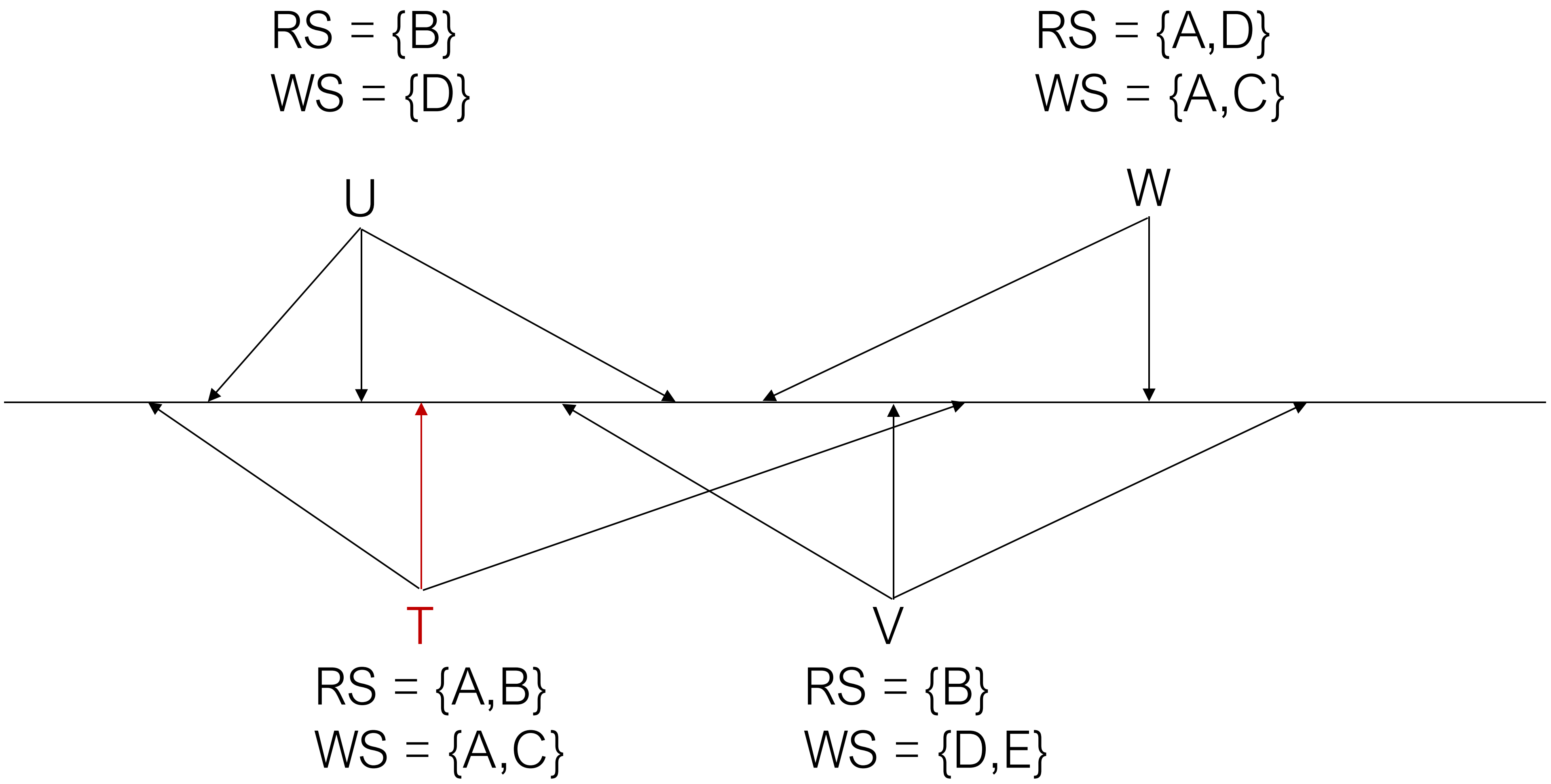
Example: CC by Validation



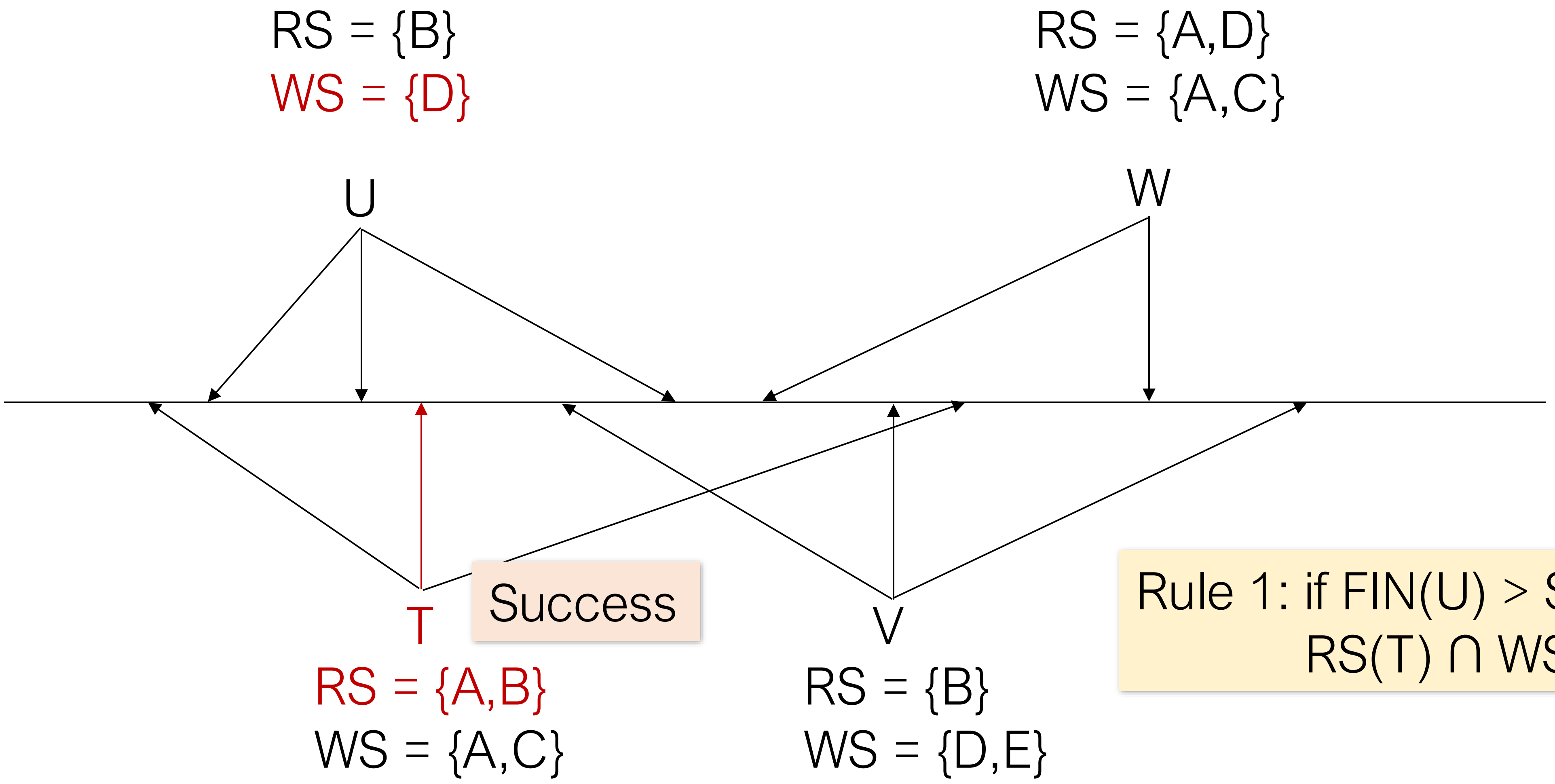
Example: CC by Validation



Example: CC by Validation

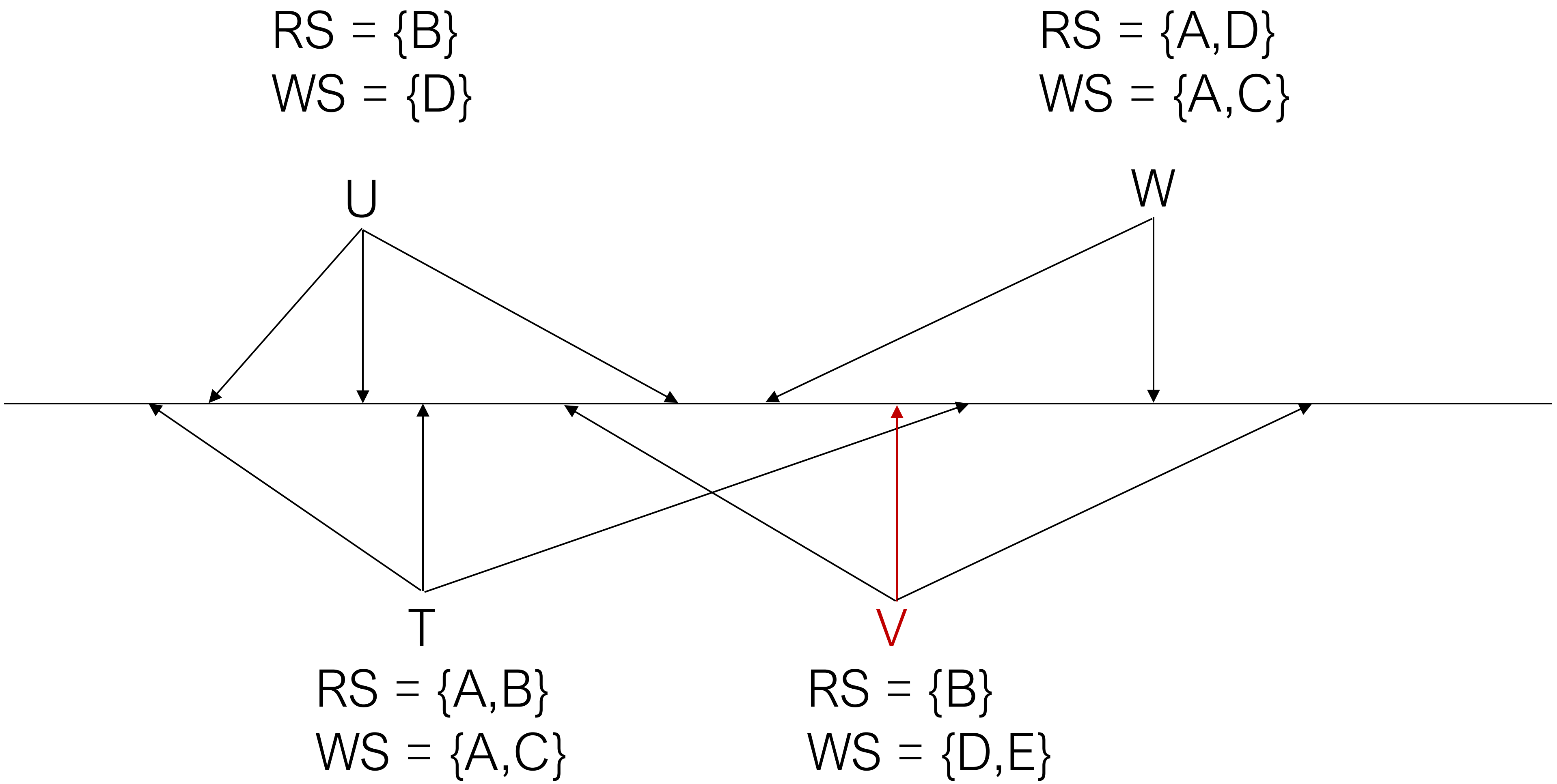


Example: CC by Validation



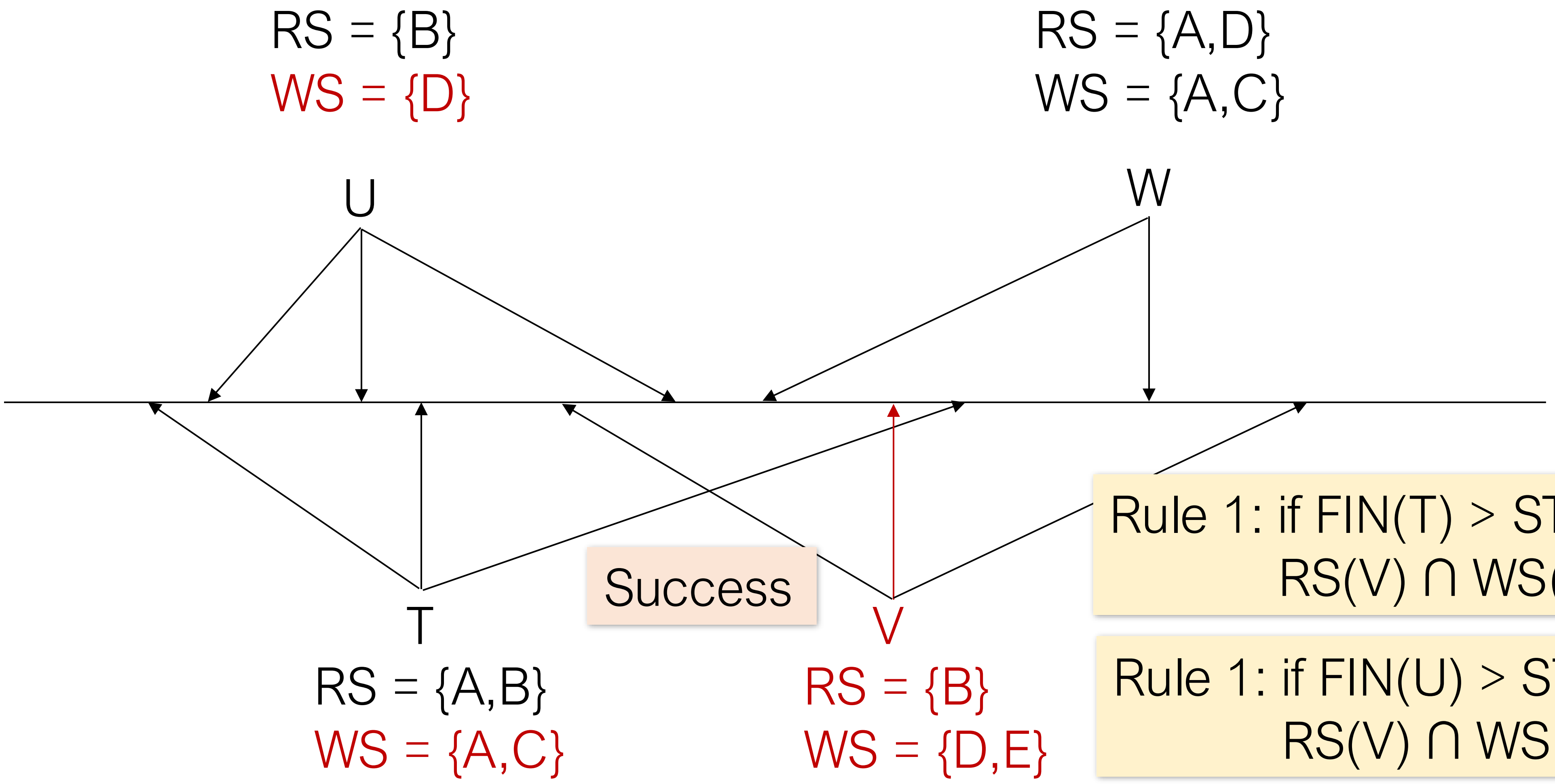
Rule 1: if FIN(U) > START(T),
RS(T) ∩ WS(U) = ∅

Example: CC by Validation



Example: CC by Validation

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

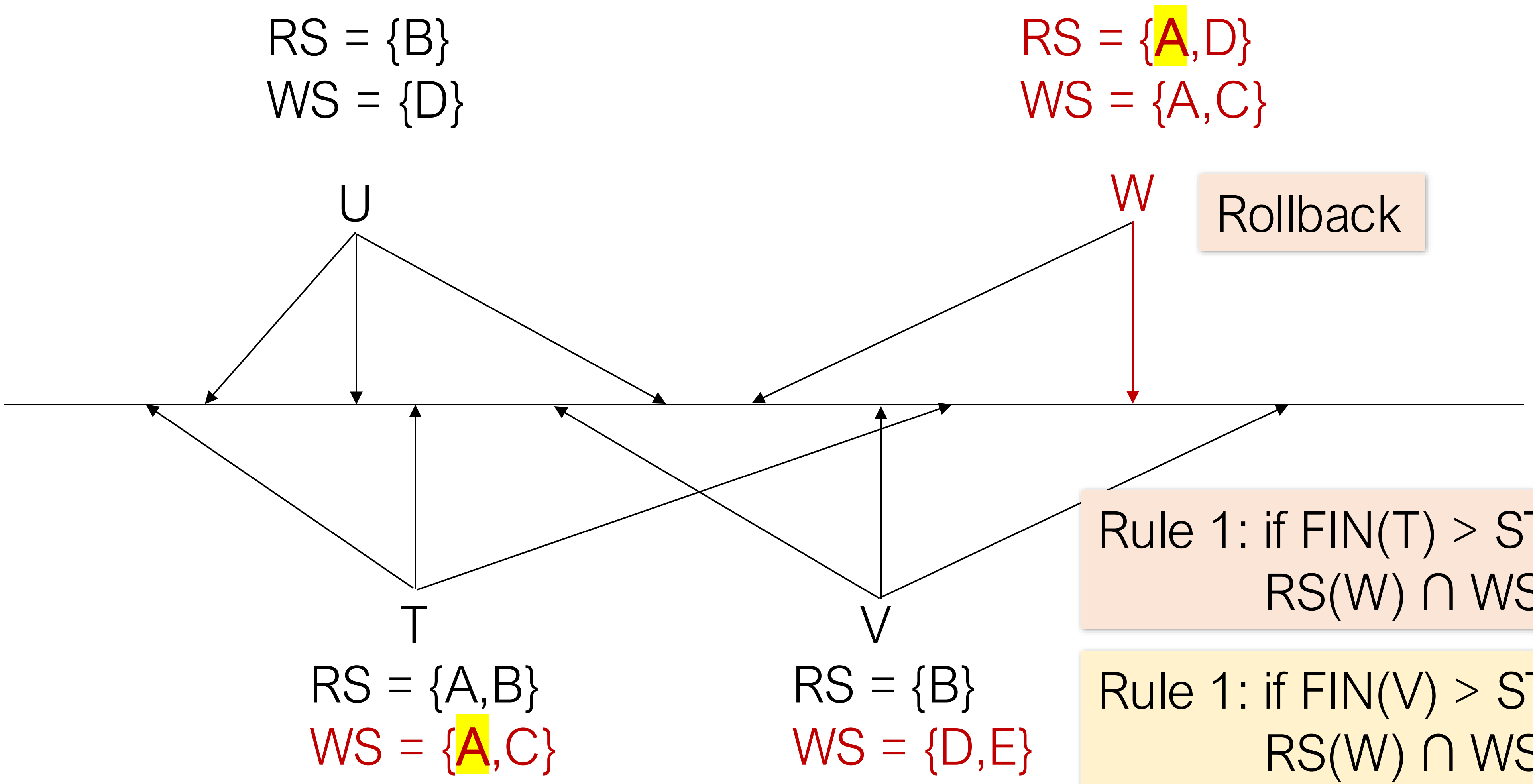


Rule 1: if FIN(T) > START(V),
RS(V) ∩ WS(T) = ø

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

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) = ∅

3. Multi-version Concurrency Control

MVCC Overview

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.

MVCC Overview

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.

MVCC

For each transaction T:

- a unique timestamp TS(T) when it begins
- Later transactions get higher timestamps

For each object O:

- a write-timestamp WT(O)
- a read-timestamp RT(O)

Each version of an object has

- its writer's TS as its WT (WT is associated with versions of an element, and they never change.)
- the timestamp of the transaction that most recently read this version as its RT

Example

$TS(T_1) = 1$
$TS(T_2) = 2$
$TS(T_3) = 3$

Schedule

Time ↓

T_1	T_2	T_3
BEGIN		
$R_1(A)$		
	BEGIN	
	$W_2(A)$	
$R_1(A)$		
COMMIT		
	COMMIT	
		BEGIN
		$R_3(A)$
		COMMIT

Database

Version	Value	RT	WT
A_0	1000	1	0

- A_0 existed before the transactions started

Example

$TS(T_1) = 1$
$TS(T_2) = 2$
$TS(T_3) = 3$

Schedule

Time

T_1	T_2	T_3
BEGIN		
$R_1(A)$		
	BEGIN	
	$W_2(A)$	
$R_1(A)$		
COMMIT		
	COMMIT	
		BEGIN
		$R_3(A)$
		COMMIT

Database

Version	Value	RT	WT
A_0	1000	1	0

- A_0 is the newest version with WT <= $TS(T_1)$
- Read A_0

Example

TS(T_1) = 1
TS(T_2) = 2
TS(T_3) = 3

Schedule

Time ↓

T_1	T_2	T_3
BEGIN		
$R_1(A)$		
	BEGIN	
	➡ $W_2(A)$	
$R_1(A)$		
COMMIT		
	COMMIT	
		BEGIN
		$R_3(A)$
		COMMIT

Database

Version	Value	RT	WT
A_0	1000	1	0
A_1	800	2	2

- RT(A_0) <= TS(T_2)
- T_2 creates a new version A_1
- Set its WT, RT to TS(T_2) = 2

Example

$TS(T_1) = 1$
$TS(T_2) = 2$
$TS(T_3) = 3$

Schedule

Time ↓

T_1	T_2	T_3
BEGIN		
$R_1(A)$		
	BEGIN	
	$W_2(A)$	
➡ $R_1(A)$		
COMMIT		
	COMMIT	
		BEGIN
		$R_3(A)$
		COMMIT

Database

Version	Value	RT	WT
A_0	1000	1	0
A_1	800	2	2

- A_0 is the newest version with WT <= $TS(T_1)$
- Read A_0
- Note that T_1 operates on the snapshot from when it started

Example

$TS(T_1) = 1$
$TS(T_2) = 2$
$TS(T_3) = 3$

Schedule

Time

T_1	T_2	T_3
BEGIN		
$R_1(A)$		
	BEGIN	
	$W_2(A)$	
$R_1(A)$		
COMMIT		
	COMMIT	
		BEGIN
		$R_3(A)$
		COMMIT

Database

Version	Value	RT	WT
A_0	1000	1	0
A_1	800	**3**	2

- A_1 is the newest version with WT <= $TS(T_3)$
- Read A_1
- Update RT to $TS(T_3)$

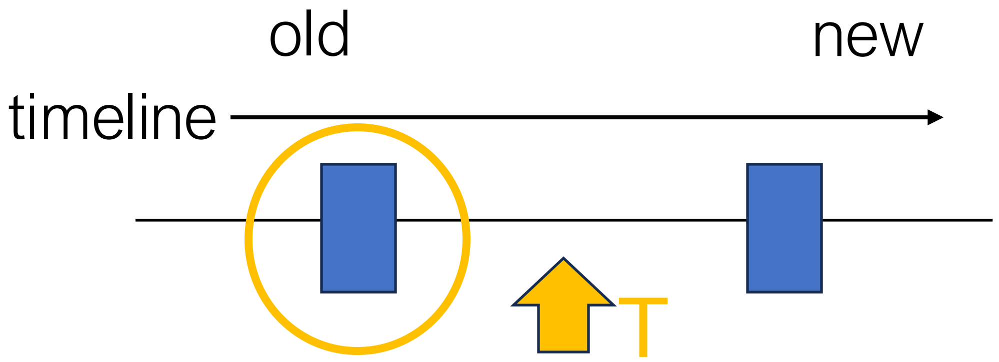
Reader Transaction Protocol

For each object to be read:

- Finds newest version with WT < TS(T)
- Update RT if necessary (i.e., if TS(T) > RT, then RT = TS(T))

Assuming that some version of every object exists from the beginning of time, Reader transactions are never restarted

- However, might block until writer of the appropriate version commits



Writer Transaction Protocol

To read an object, follows reader protocol

To write an object:

- must make sure that the object has not been read by a ”later” transaction
- Finds newest version V s.t. WT(V) <= TS(T)

If RT(V) <= TS(T) :

- T makes a copy V’ of V, with a pointer to V, with WT(V’) = TS(T), RT(V’) = TS(T)
- Write is buffered until T commits; other transactions can see TS values but can’t read version V’

Else

- reject write