CS 4440 A

Emerging Database Technologies

Lecture 13

03/24/25

Announcements

- Assignment 3 released
    - Start early!
    - Due Apr 9

- Exam 2
    - Take home, open book and notes, no time limit
    - Contents covered: up until lectures next Monday
    - Released Apr 2, due Apr 4

- Upcoming guest lectures
    - Apr 2, Apr 7
    - Mandatory attendance

Desirable Properties of Transactions: ACID

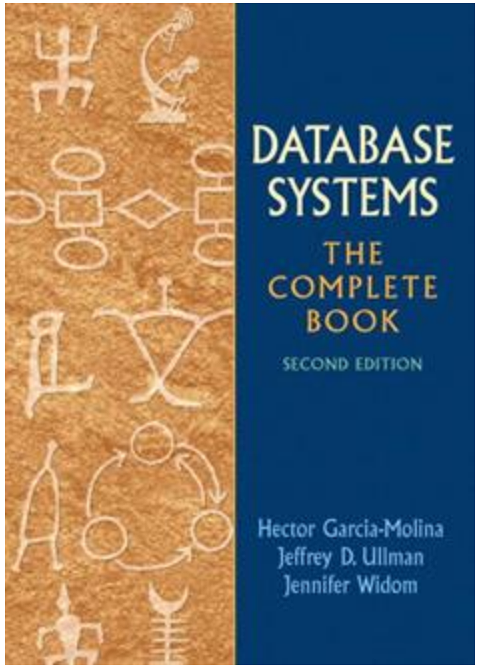
- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates 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 consistency & 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. Schedule

2. Lock-based Concurrency Control

3. Optimistic Concurrency Control

1. Schedule

Schedule

A transaction is seen by DBMS as a list of actions.

- READ, WRITE of database objects
- ABORT, COMMIT

Assumption: Transactions communicate only through READ and WRITE

Schedule is a list of actions from a set of transactions as seen by the DBMS

- Two actions from the same transaction T MUST appear in the schedule in the same order that they appear in T
- Intuitively, a schedule represents an actual or potential execution sequence

Transaction primitives

- INPUT(X): copy block X from disk to memory
- READ(X, t): copy X to transaction's local variable t
  (run INPUT(X) if X is not in memory)
- WRITE(X, t): copy value of t to X (run INPUT(X) if X is not in memory)
- OUTPUT(X): copy X from memory to disk

Schedule

- Actions taken by one or more transactions

$T1$	$T2$
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)

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

Serial schedule

- One transaction is executed at a time

T1	*T2*	*A*	*B*
		25	25
READ(*A*, *t*)			
t := *t*+100			
WRITE(*A*, *t*)			
READ(*B*, *t*)		125	
t := *t*+100			
WRITE(*B*, *t*)			
			125
	READ(*A*, s)		
	s := *s**2		
	WRITE(*A*, *s*)	250	
	READ(*B*, *s*)		
	s := *s**2		
	WRITE(*B*, *s*)		250

Schedule: (T1, T2)

Q: Do serial schedules allow for high throughput?

Serializable schedule

- There exists a serial schedule with the same effect

T1	*T2*	*A*	*B*
		25	25
READ(*A*, *t*)			
t := *t*+100			
WRITE(*A*, *t*)		125	
	READ(*A*, s)		
	s := *s**2		
	WRITE(*A*, *s*)	250	
READ(*B*, *t*)			
t := *t*+100			
WRITE(*B*, *t*)			125
	READ(*B*, *s*)		
	s := *s**2		
	WRITE(*B*, *s*)		250

Same effect as (T1, T2)

Serializable schedule

- This is <u>not</u> serializable

$T1$	$T2$	A	B
		25	25
READ(A, t)			
t := t+100			
WRITE(A, t)		125	
	READ(A, s)		
	s := s*2		
	WRITE(A, s)	250	
	READ(B, s)		
	s := s*2		
	WRITE(B, s)		50
READ(B, t)			
t := t+100			
WRITE(B, t)			150

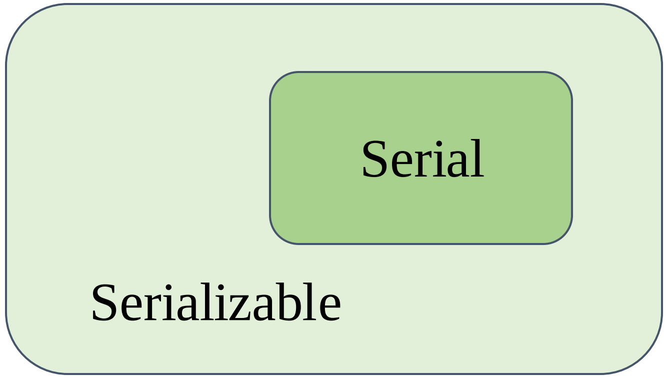
Serializable schedule

- Serializable, but only due to the detailed transaction behavior

T1	*T2*	*A*	*B*
		25	25
READ(*A*, *t*)			
t := *t*+100			
WRITE(*A*, *t*)		125	
	READ(*A*, s)		
	s := *s*+200		
	WRITE(*A*, *s*)	325	
	READ(*B*, *s*)		
	s := *s*+200		
	WRITE(*B*, *s*)		225
READ(*B*, *t*)			
t := *t*+100			
WRITE(*B*, *t*)			325

Same effect as (T1, T2)

Serial vs Serializable Schedule



Serial

Serializable

Being serializable is <u>not</u> the same as being serial

Being serializable implies that the schedule is a <u>correct</u> schedule.

- It will leave the database in a consistent state.

Interleaving improves efficiency due to concurrent execution, e.g.,

- While one transaction is blocked on I/O, the CPU can process another transaction
- Interleaving short and long transactions might allow the short transaction to finish sooner (otherwise it need to wait until the long transaction is done)

Interleaving & Isolation

The DBMS has freedom to interleave TXNs

However, it must pick an interleaving or **schedule** such that isolation and consistency are maintained

- Must be *as if* the TXNs had executed serially!

ACID

DBMS must pick a schedule which maintains isolation & consistency

Abstract view of TXNs: reads and writes

Serializability is hard to check - cannot always know detailed behaviors

DBMS's abstract view of transactions:

$r_i(X)$: Ti reads X
$w_i(X)$: Ti writes X

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

Serializable schedule: $r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(B)$; $w_1(B)$; $r_2(B)$; $w_2(B)$;

Conflicts: Anomalies with Interleaved Execution

Conditions for conflicts:

- The operations must belong to **different transactions** (no conflict within the same transaction).
- The operations must access the **same database object**
- At least one of the operations must be a **write** operation.

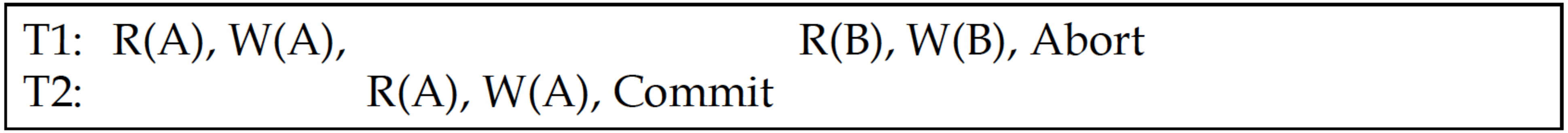
Types of conflicts:

- Write-Read (WR)
- Read-Write (RW)
- Write-Write (WW)

Implication for schedules:

A pair of consecutive actions that cannot be interchanged without changing behavior

WR Conflict

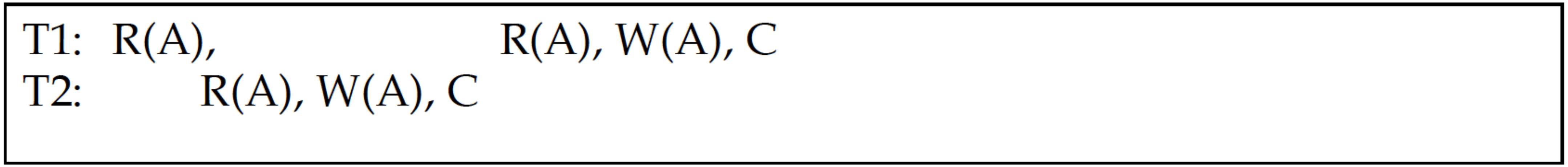


T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A), Commit	

Reading Uncommitted Data (WR Conflicts, “dirty reads”):

- transaction T2 reads an object that has been modified by T1 but not yet committed

RW Conflict



```
T1:  R(A),                      R(A), W(A), C
T2:        R(A), W(A), C
```

Unrepeatable Reads (RW Conflicts):

- T2 changes the value of an object A that has been read by transaction T1, which is still in progress
- If T1 tries to read A again, it will get a different result

WW Conflict

```
T1:  W(A),                    W(B), C
T2:            W(A), W(B), C
```

Overwriting Uncommitted Data (WW Conflicts, “lost update”):

- T2 overwrites the value of A, which has been modified by T1, still in progress
- Suppose we need the salaries of two employees (A and B) to be the same
    - T1 sets them to $1000
    - T2 sets them to $2000

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

- Conflict-equivalent to serial schedule

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

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

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

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

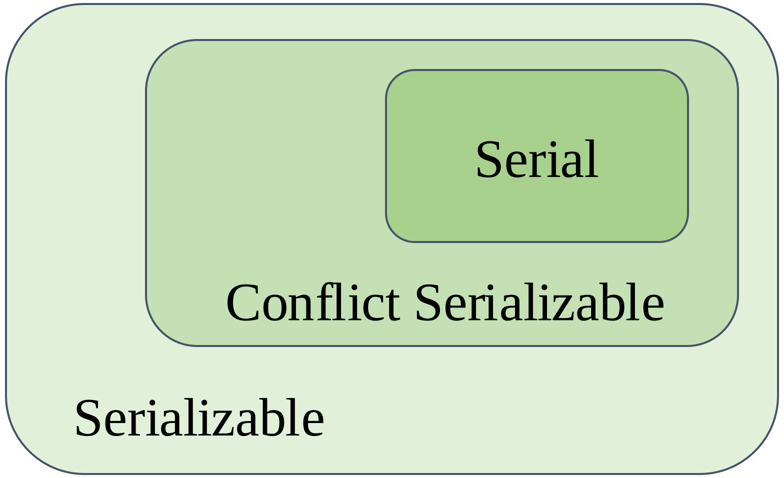
Serial $r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

Conflict-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



In-class Exercise

- What are schedules that are conflict-equivalent to (T1, T2)?

T1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;

T2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $w_2(A)$;

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 serializable if and only if the precedence graph has no cycles.

Precedence graph

Can use to decide conflict serializability

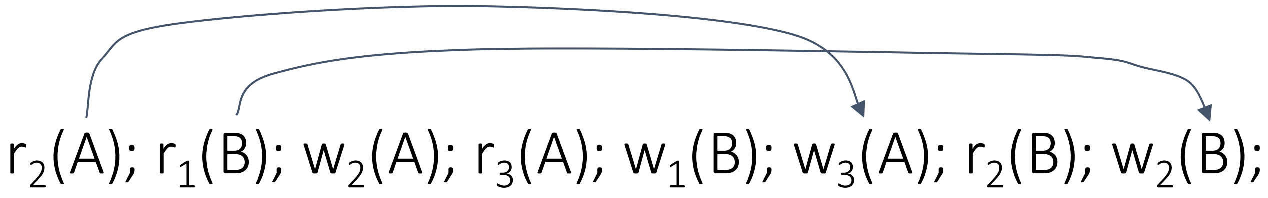
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

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

** Also called dependency graph, conflict graph, or serializability graph*

Precedence graph

Can use to decide conflict serializability



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

T1 → T2 → T3

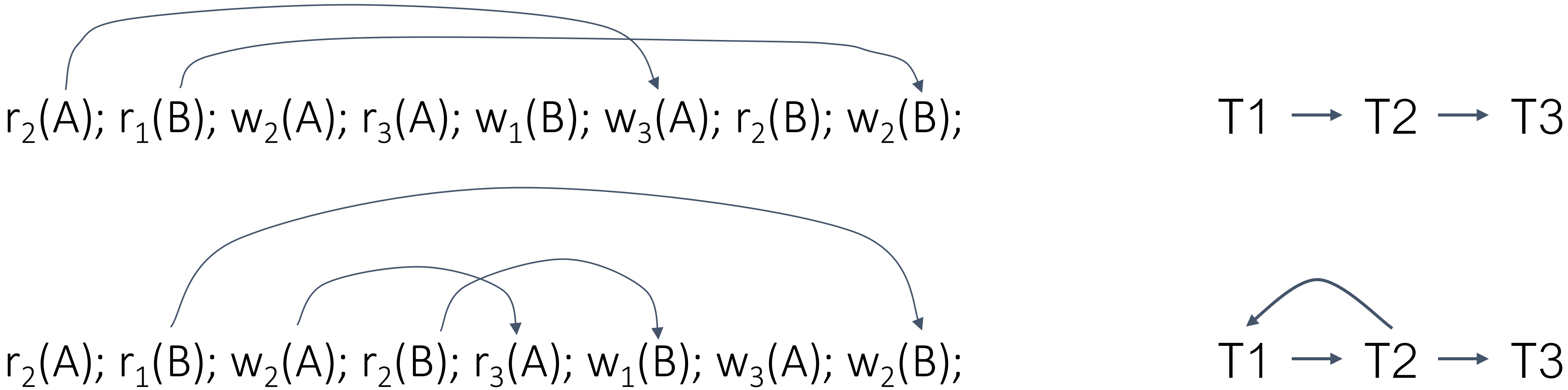
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

T1 T2 T3

- One node per committed transaction
- Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj’s actions
  - – Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

Precedence graph

Can use to decide conflict serializability



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

T1 → T2 → T3

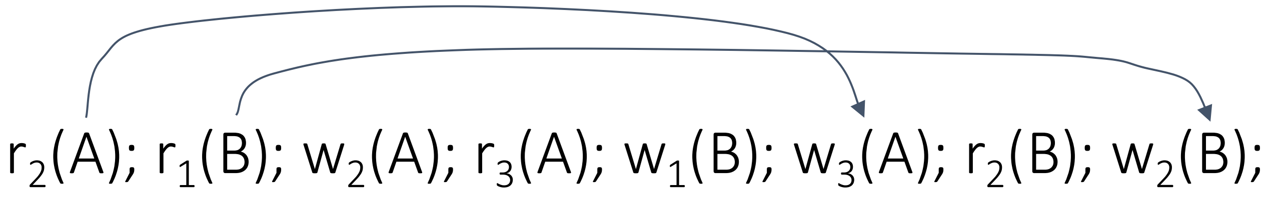
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

T1 → T2 → T3

- One node per committed transaction
- Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj’s actions
  - – Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

Precedence graph

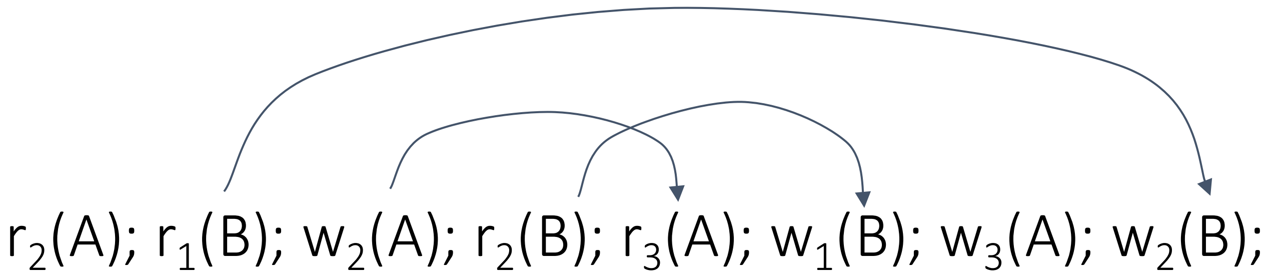
Can use to decide conflict serializability



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

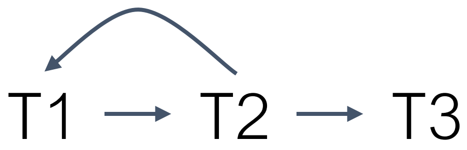
This is conflict serializable

T1 → T2 → T3



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

This is not because of cycle



- One node per committed transaction
- Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions
  - Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

In-class Exercise

- What is the precedence graph for the schedule:

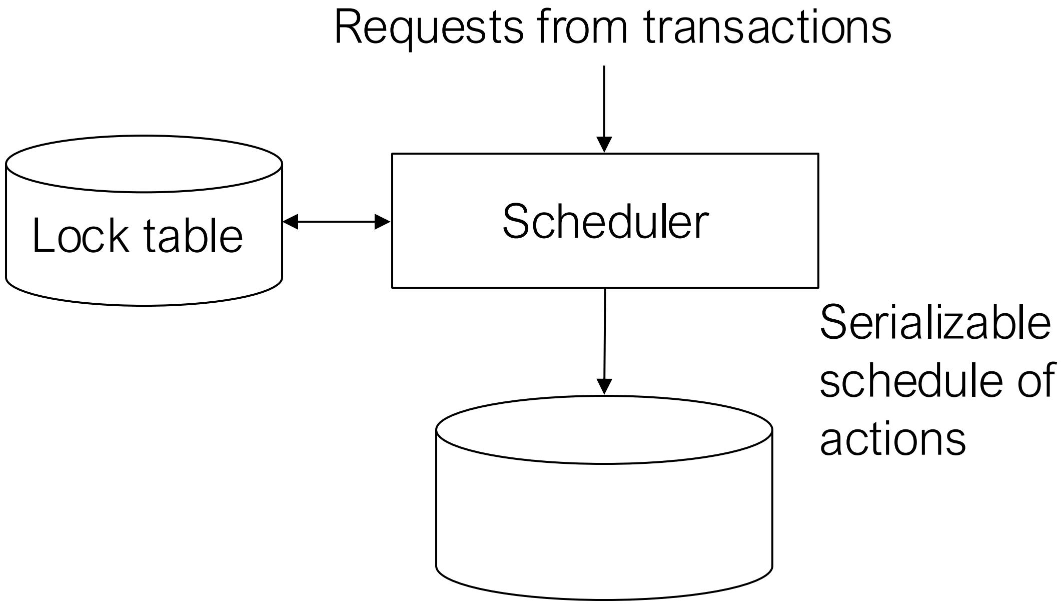
$r_1(A); r_2(A); r_1(B); r_2(B); r_3(A); r_4(B); w_1(A); w_2(B);$

- One node per committed transaction
- Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions
  - Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

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

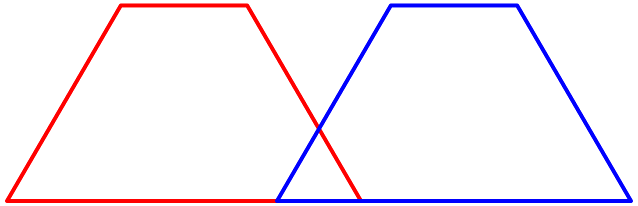
Two-phase locking (2PL)

- 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



One problem with 2PL: deadlocks

- Several transactions wait for lock by another transaction forever
- We will address this problem later

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

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

Locks With Multiple Granularity

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

A few options:

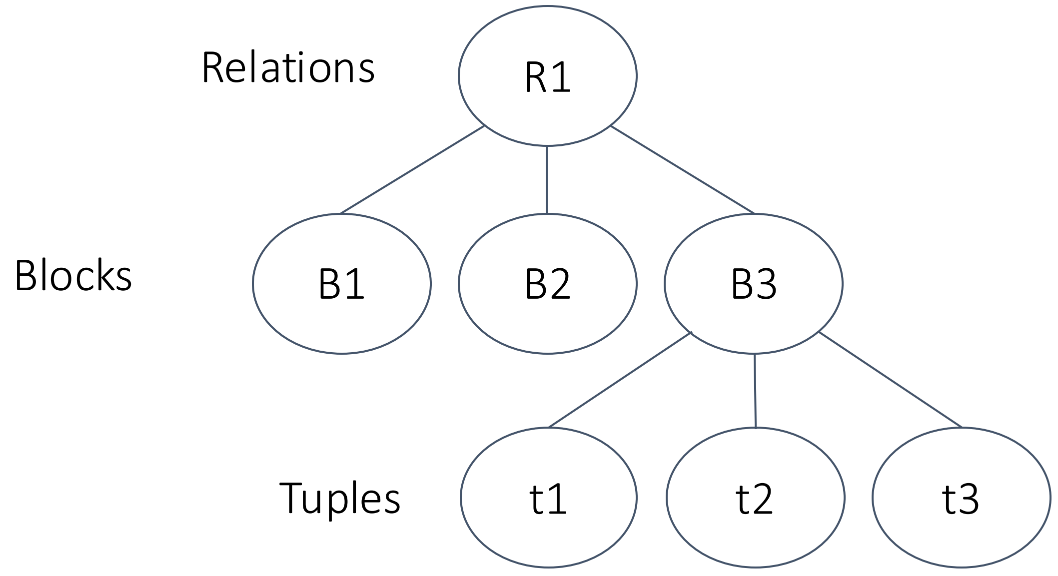
- Relations → Least concurrency
- Pages or data blocks
- Tuples → Most concurrency, but also expensive

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

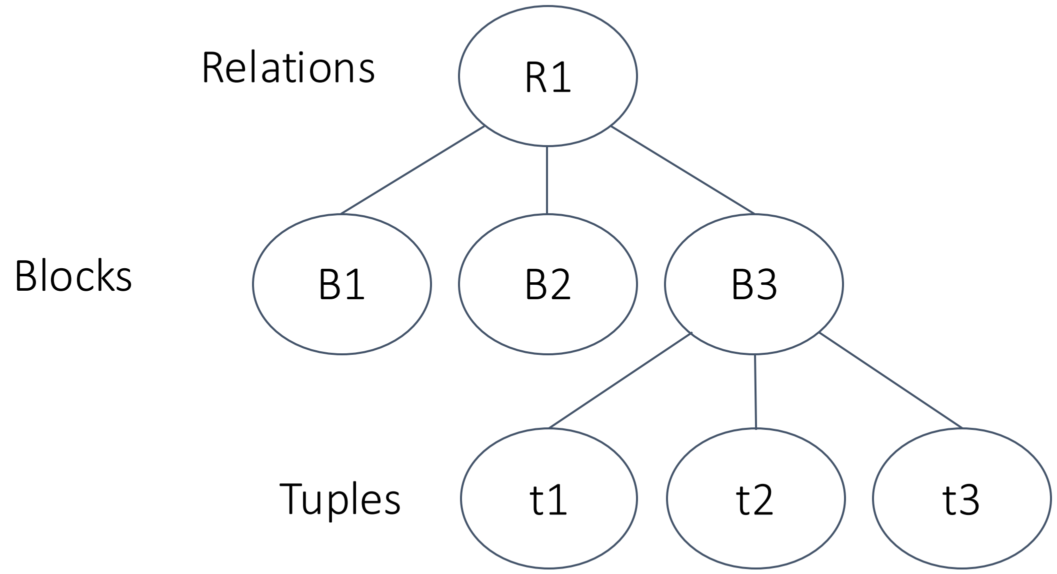
- Ordinary locks: S and X
- Warning locks: I (shows intention to lock)



Warning locks

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

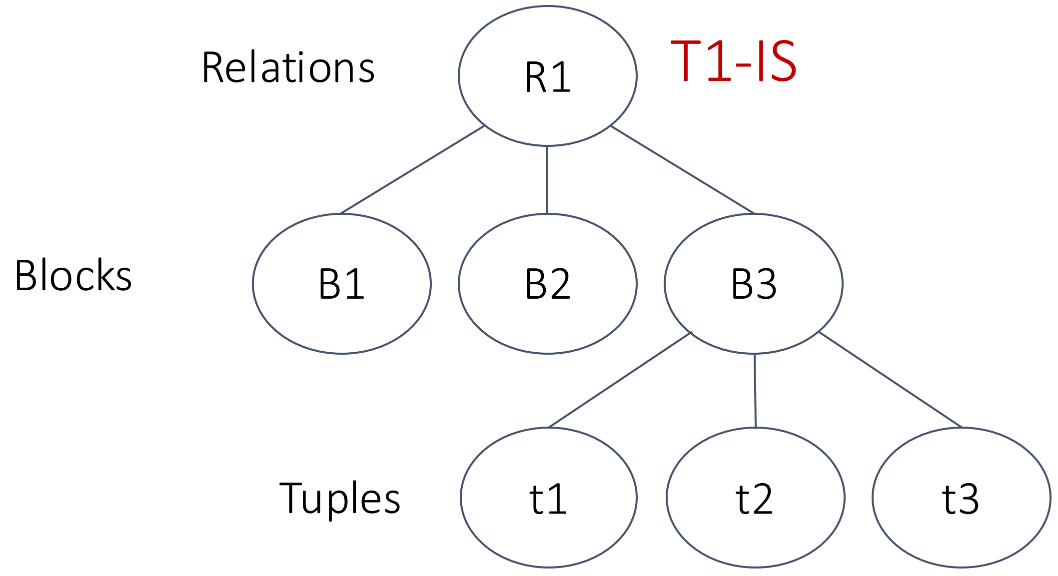
T1 wants to read t3



Warning locks

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

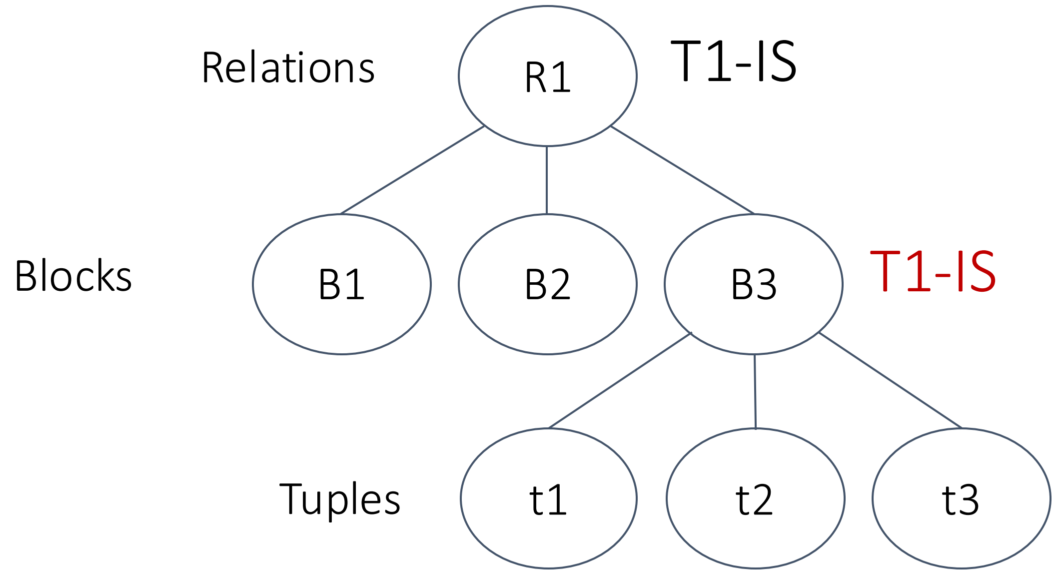
T1 wants to read t3



Warning locks

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

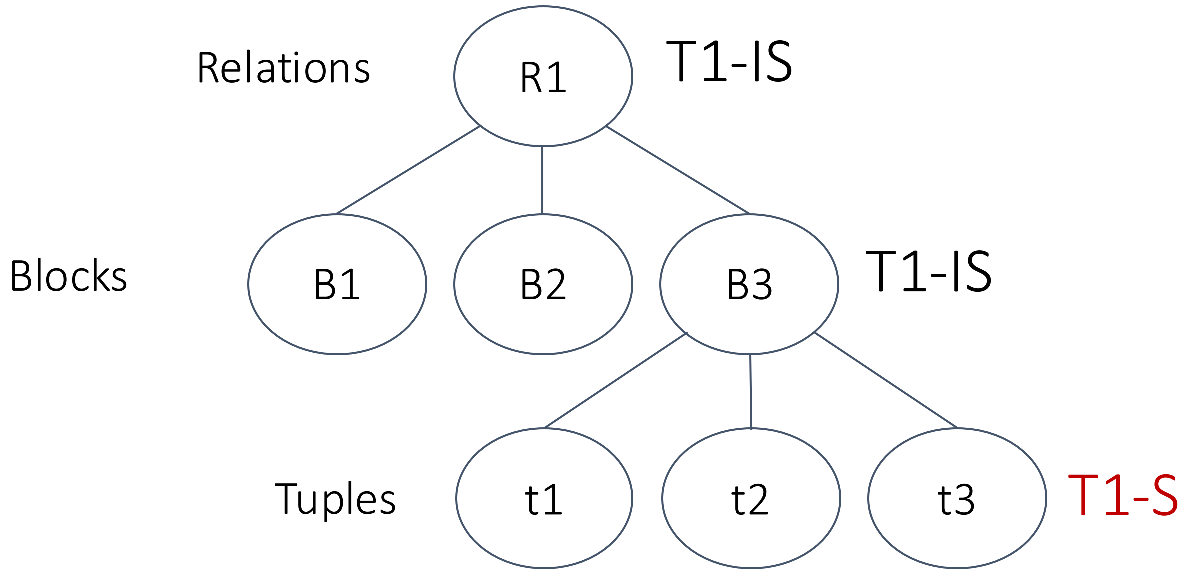
T1 wants to read t3



Warning locks

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

T1 wants to read t3

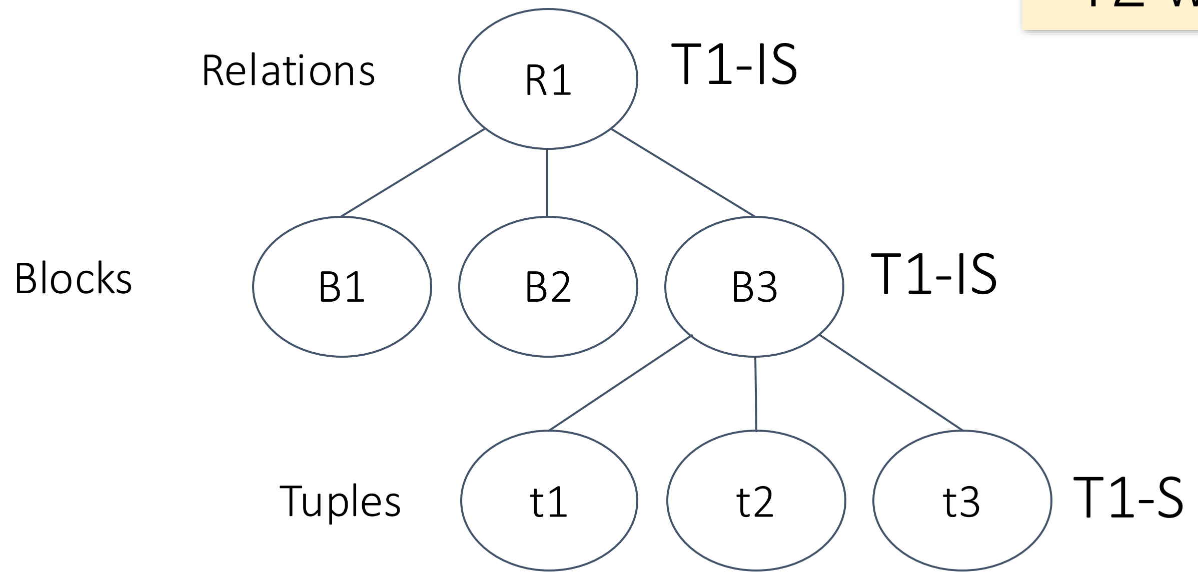


Warning locks

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

T1 wants to read t3

T2 wants to write B2

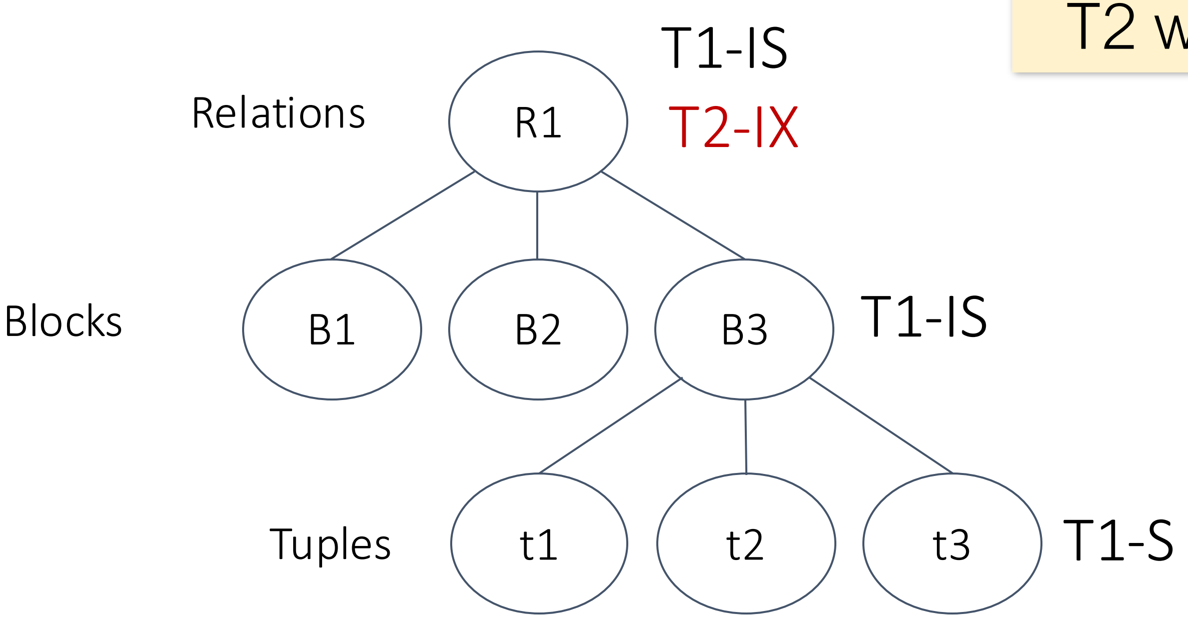


Warning locks

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

T1 wants to read t3

T2 wants to write B2

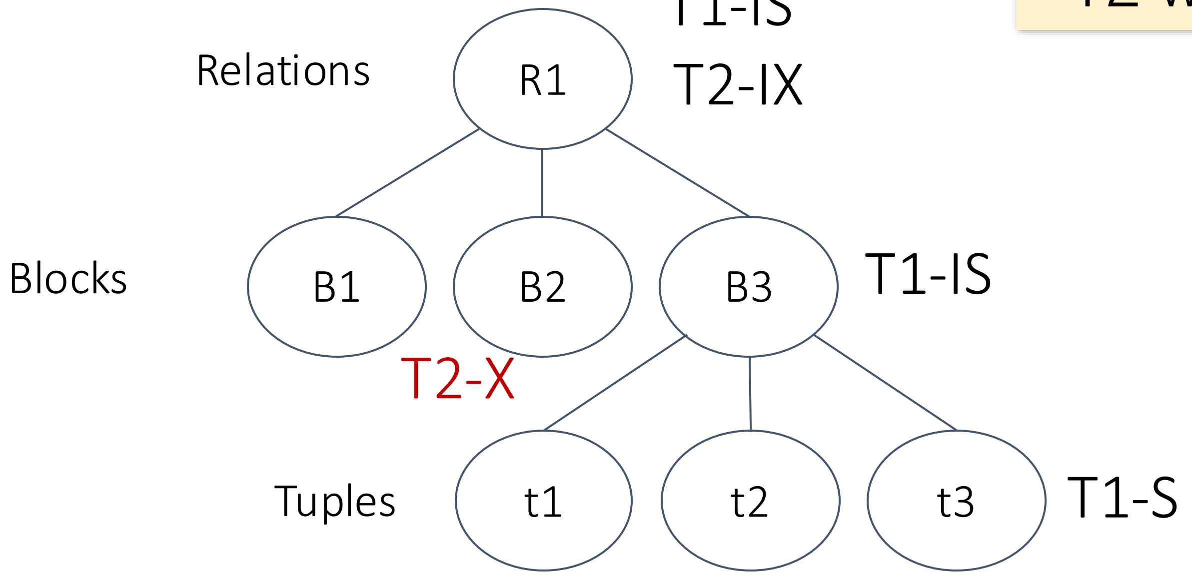


Warning locks

- 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

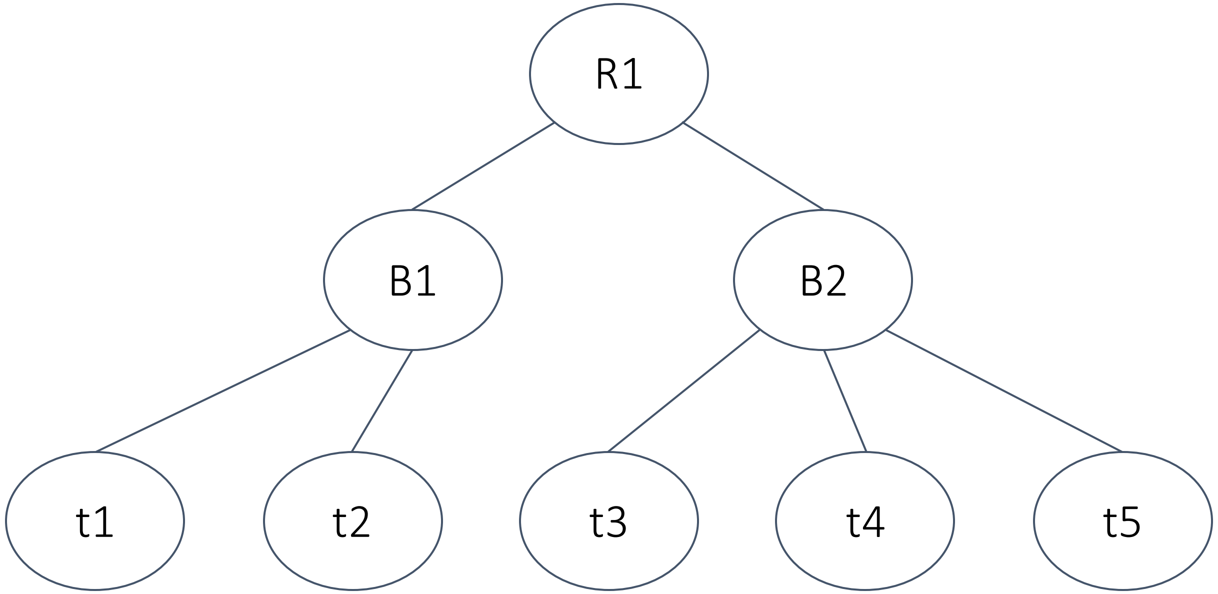
Requestor

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

In-class Exercise

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



Deadlocks

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

Two ways of dealing with deadlocks:

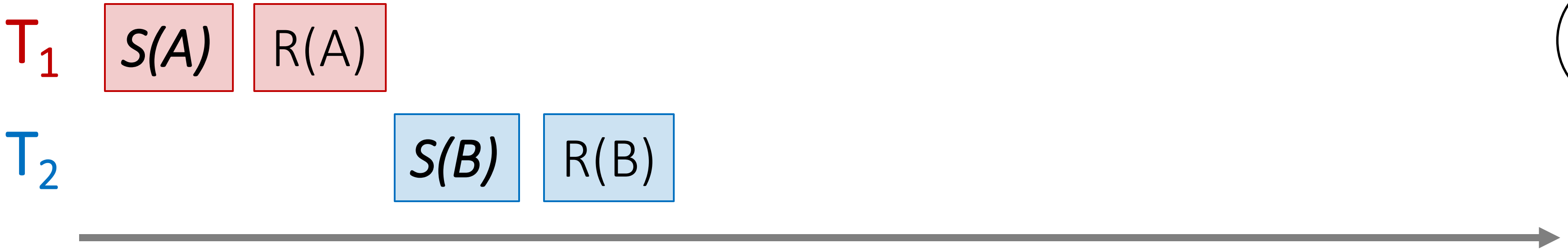
1. Deadlock detection
2. Deadlock prevention (see Database Systems Book Ch19.2)

Deadlock Detection: Example

Waits-for graph:

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

Deadlock Detection: Example


T_1 *S(A)* R(A)

T_2 *S(B)* R(B)

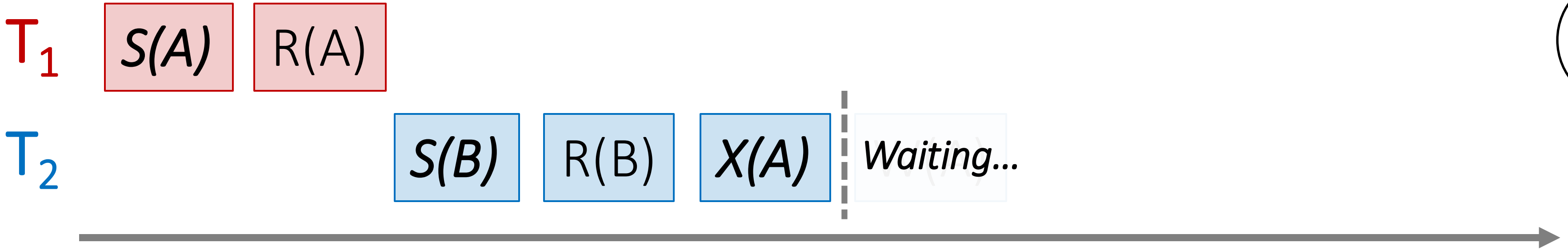
Waits-for graph:


T_1

T_2

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

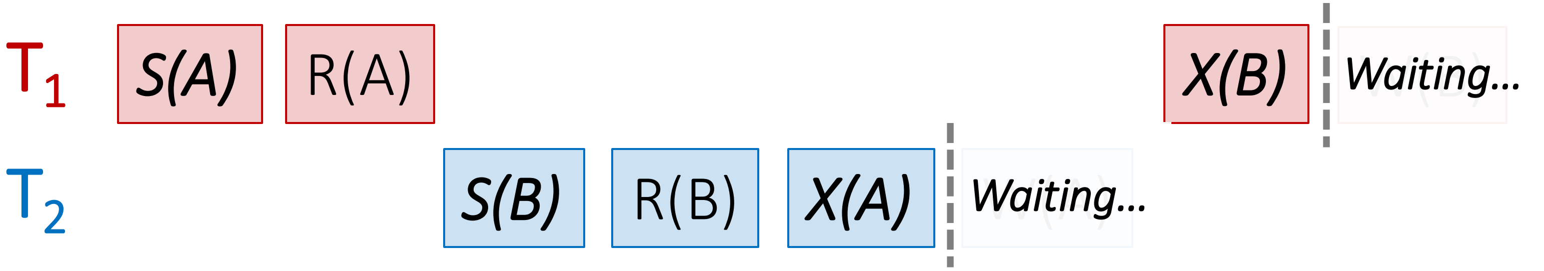
Deadlock Detection: Example



Waits-for graph:

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

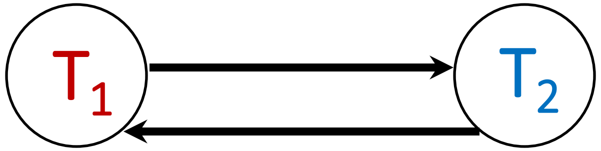
Deadlock Detection: Example



T_1 *S(A)* R(A) *X(B)* *Waiting…*

T_2 *S(B)* R(B) *X(A)* *Waiting…*

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

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.

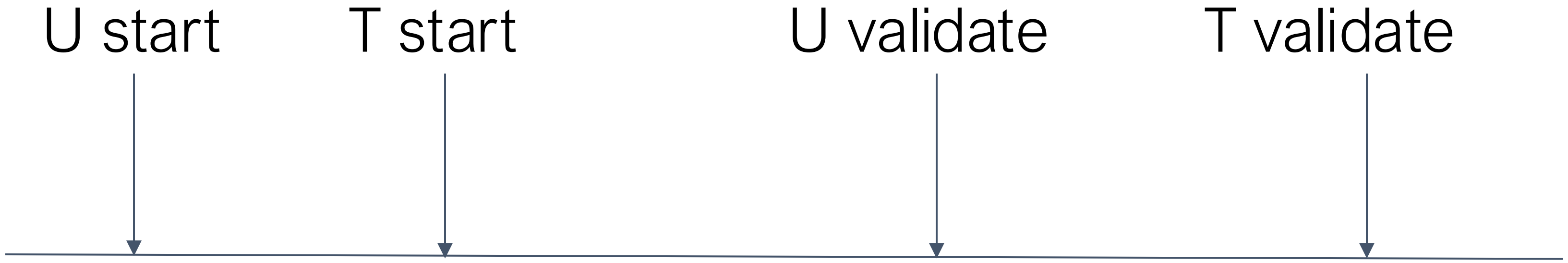
Validation rules (assume U validated)

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

WS(U) = {A, B}

RS(T) = {B, C}

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



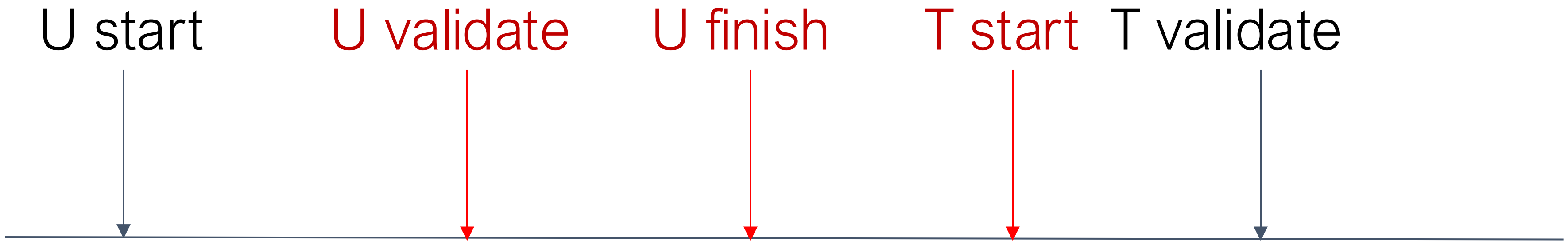
Validation rules (assume U validated)

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

WS(U) = {A, B}

RS(T) = {B, C}

This satisfies rule 1


U start

U validate

U finish

T start

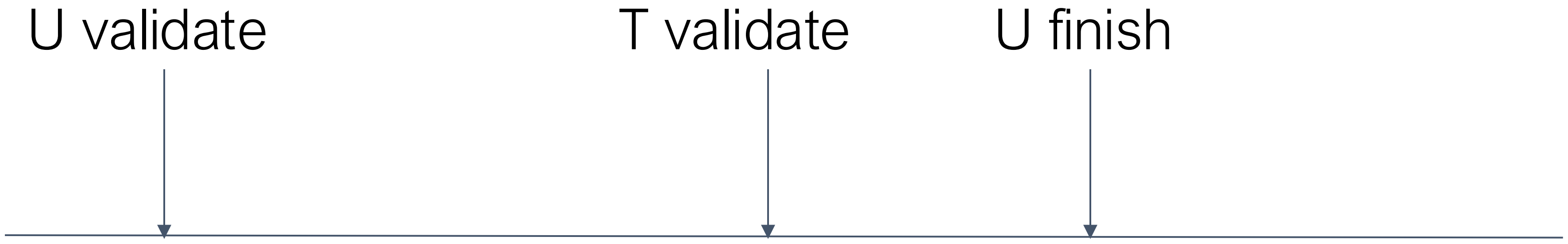
T validate

Validation rules (assume U validated)

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

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

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



U validate

T validate

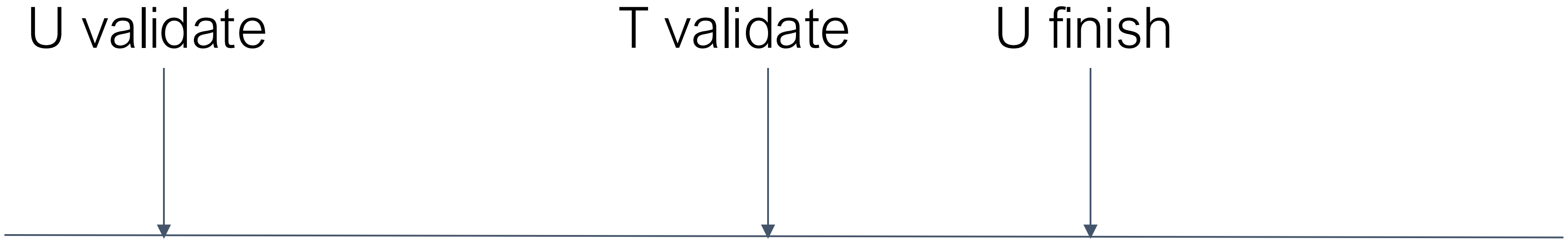
U finish

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

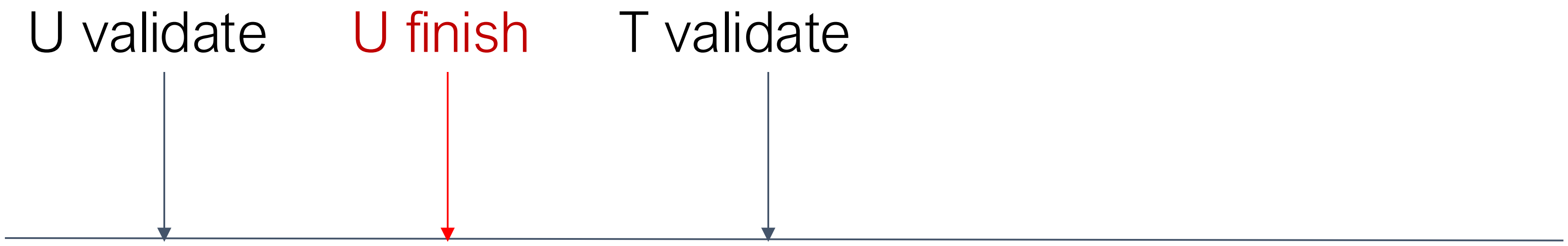
Validation rules (assume U validated)

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

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

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

This satisfies rule 2

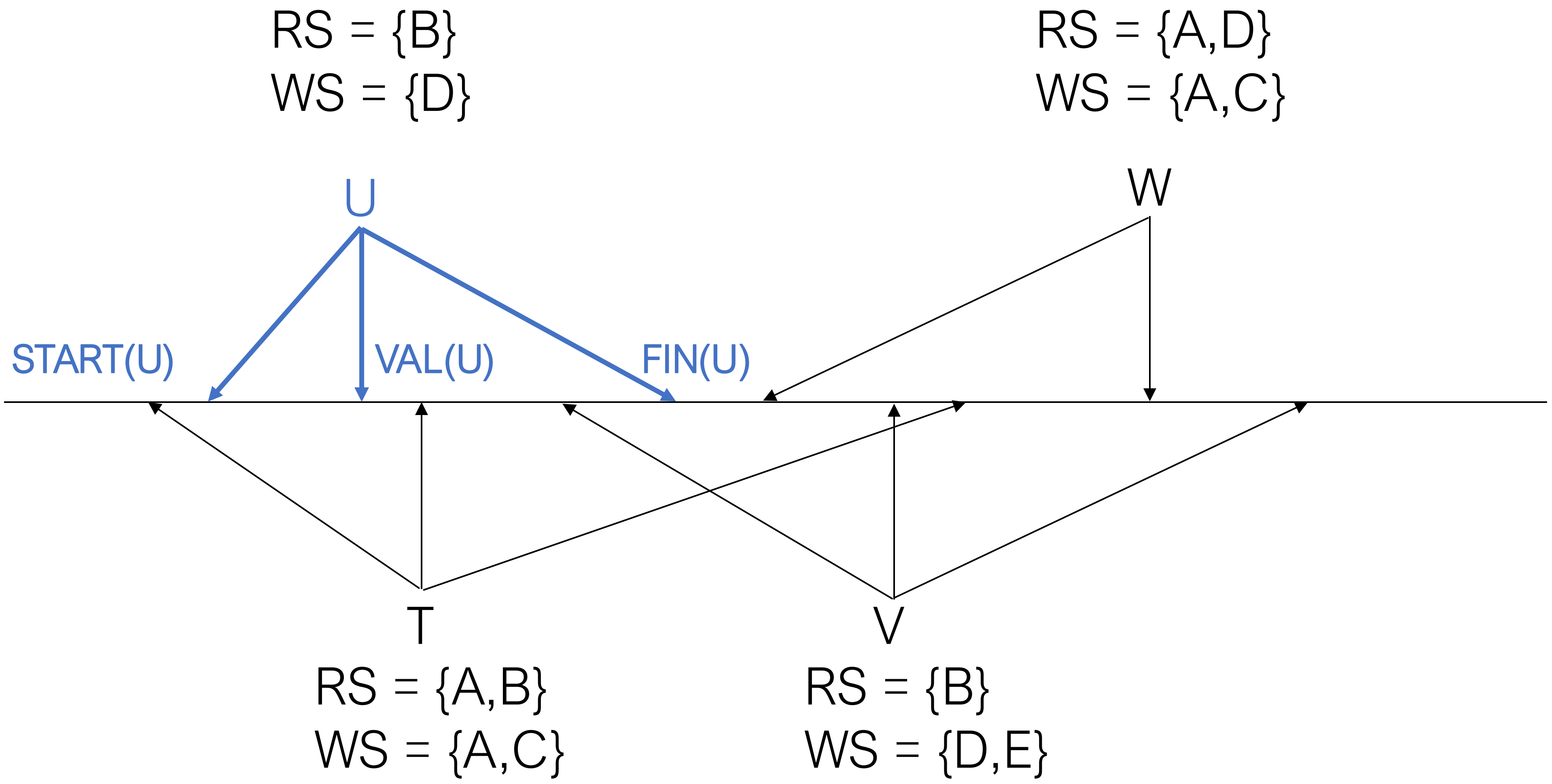


U validate

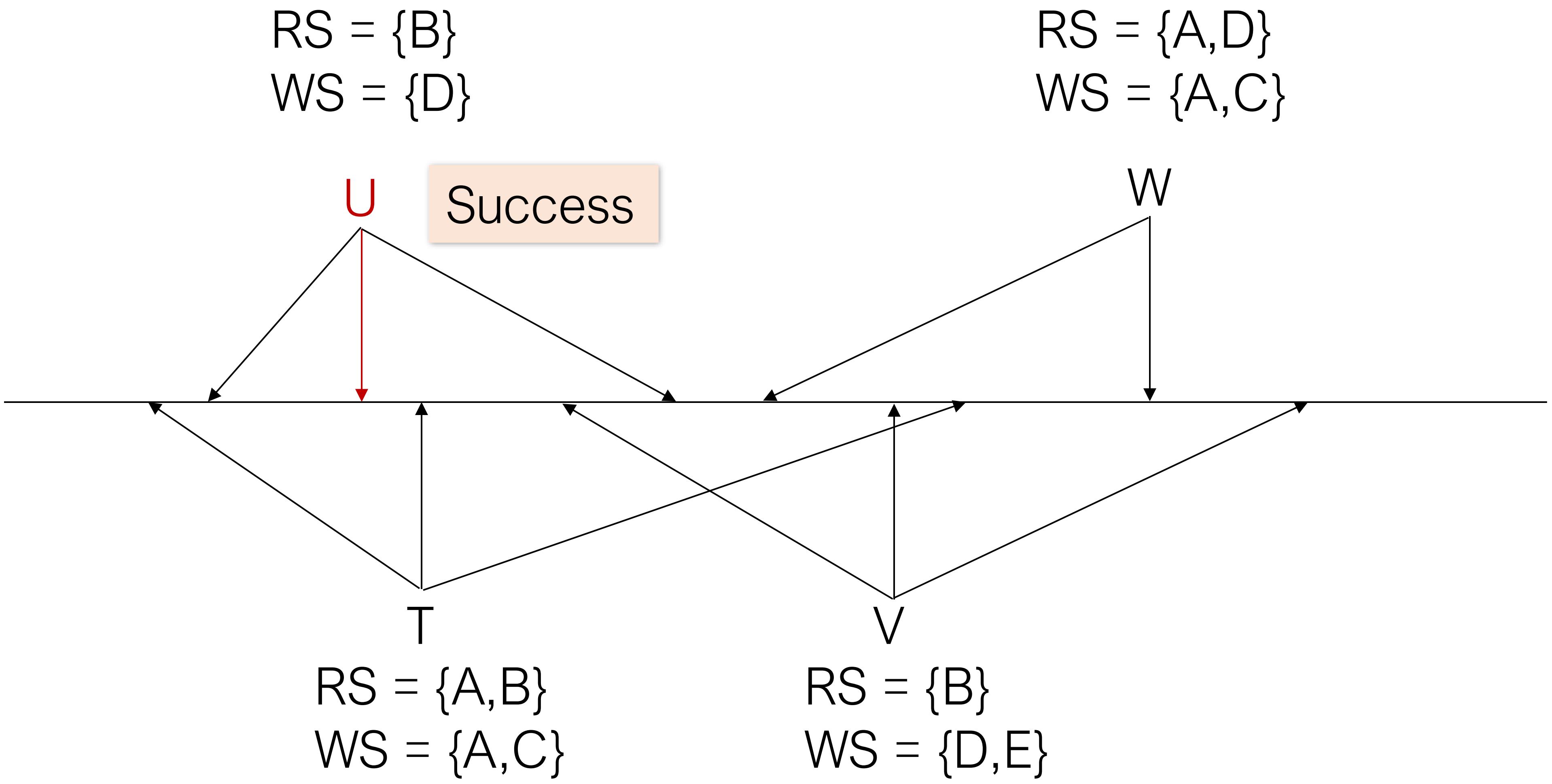
U finish

T validate

Example: CC by Validation



Example: CC by Validation



RS = {B}
WS = {D}

U

Success

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

W

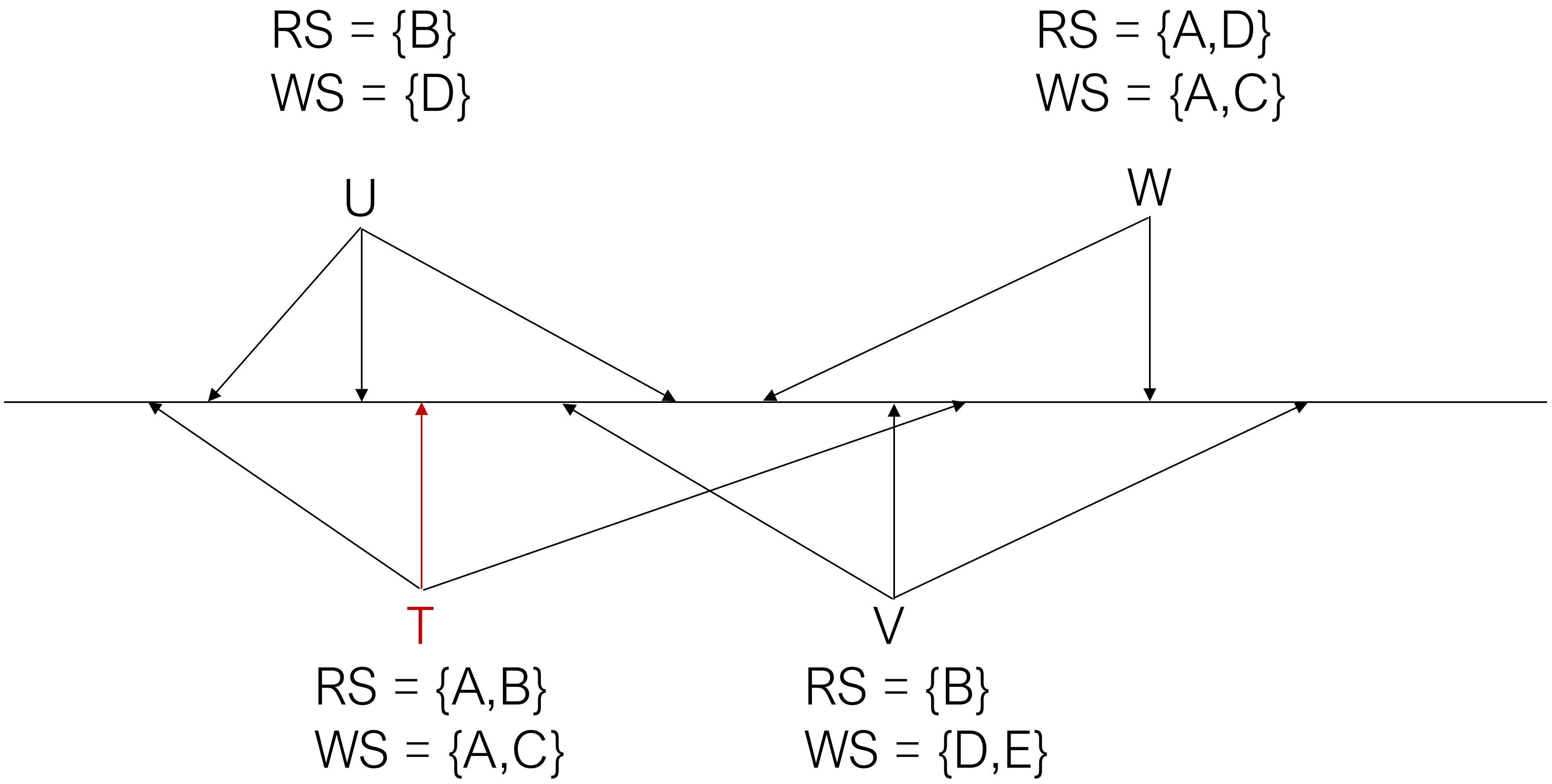
T

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

V

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

Example: CC by Validation



U

RS = {B}
WS = {D}

W

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

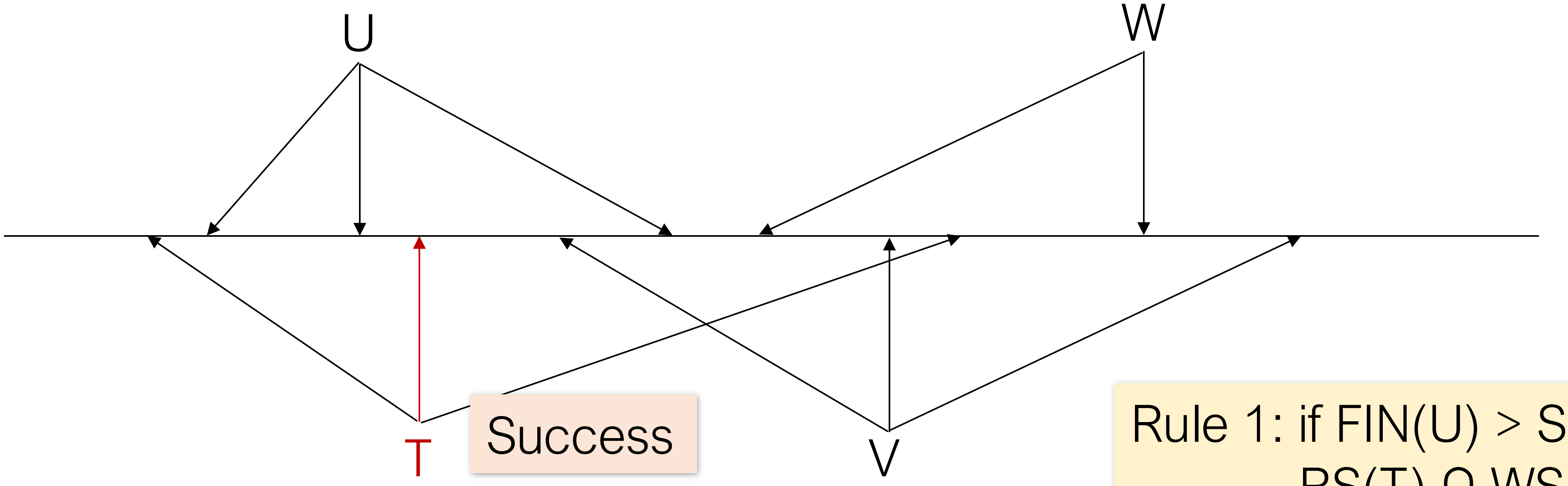
T

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

V

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

Example: CC by Validation



RS = {B}
WS = {D}

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

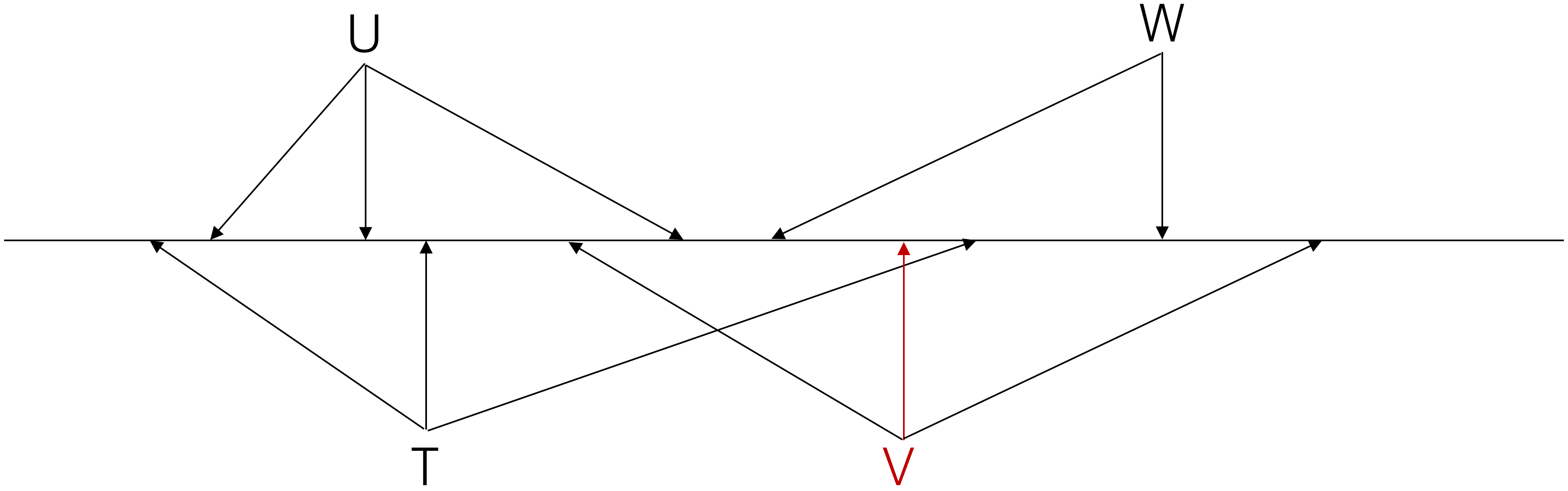
Success

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

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

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

Example: CC by Validation



RS = {B}
WS = {D}

U

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

W

T

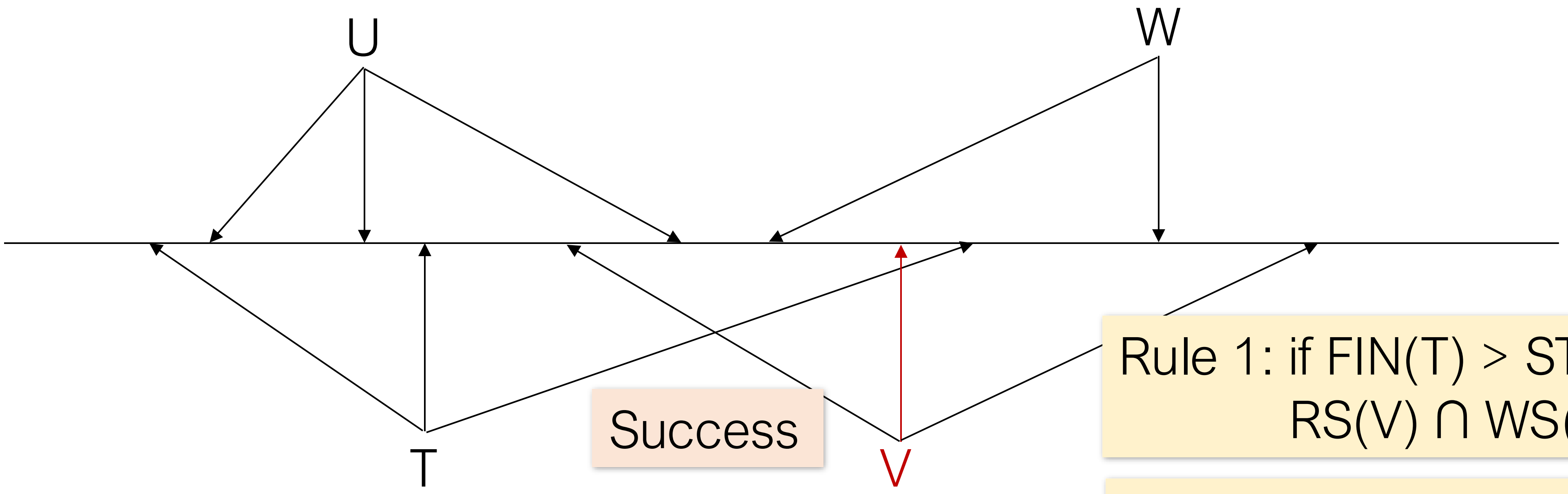
RS = {A,B}
WS = {A,C}

V

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

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

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

Success

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

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

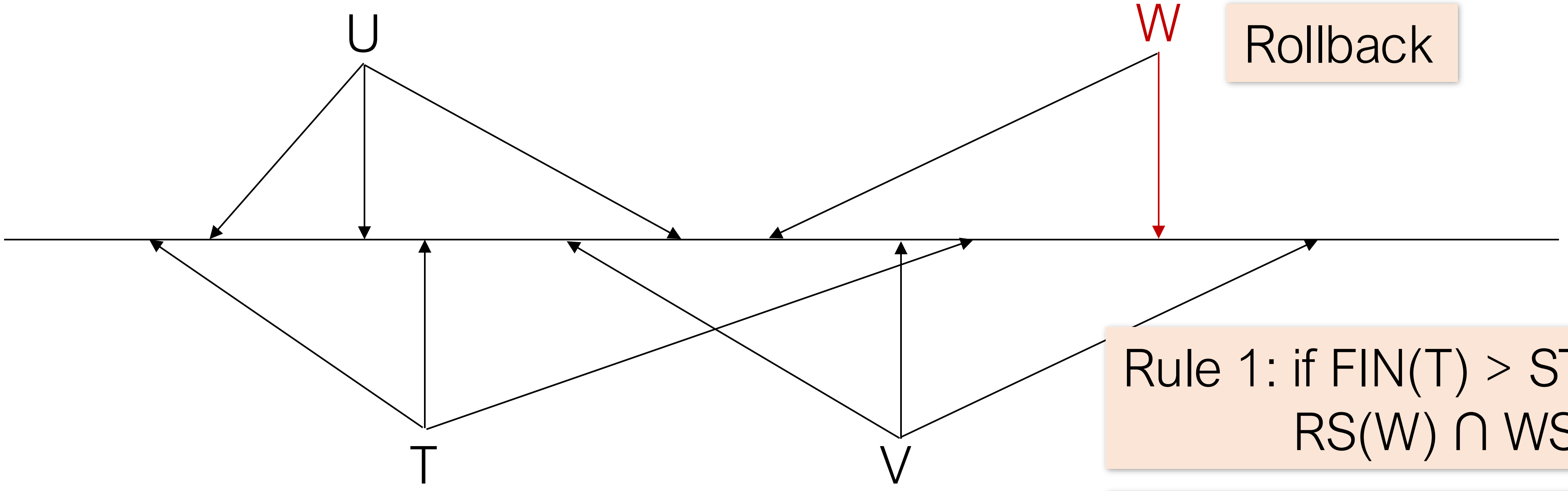
Rule 1: if FIN(U) > START(V),
RS(V) ∩ WS(U) = ∅

Example: CC by Validation

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

RS = {B}
WS = {D}

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



Rollback

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

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

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

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