CS 4440 A

Emerging Database Technologies

Lecture 11

02/14/24

Recap

Schedule

- Serial schedule and serializable schedule
- Conflicting actions
- Conflict-serializable schedule

Dependency (or Precedence) graphs

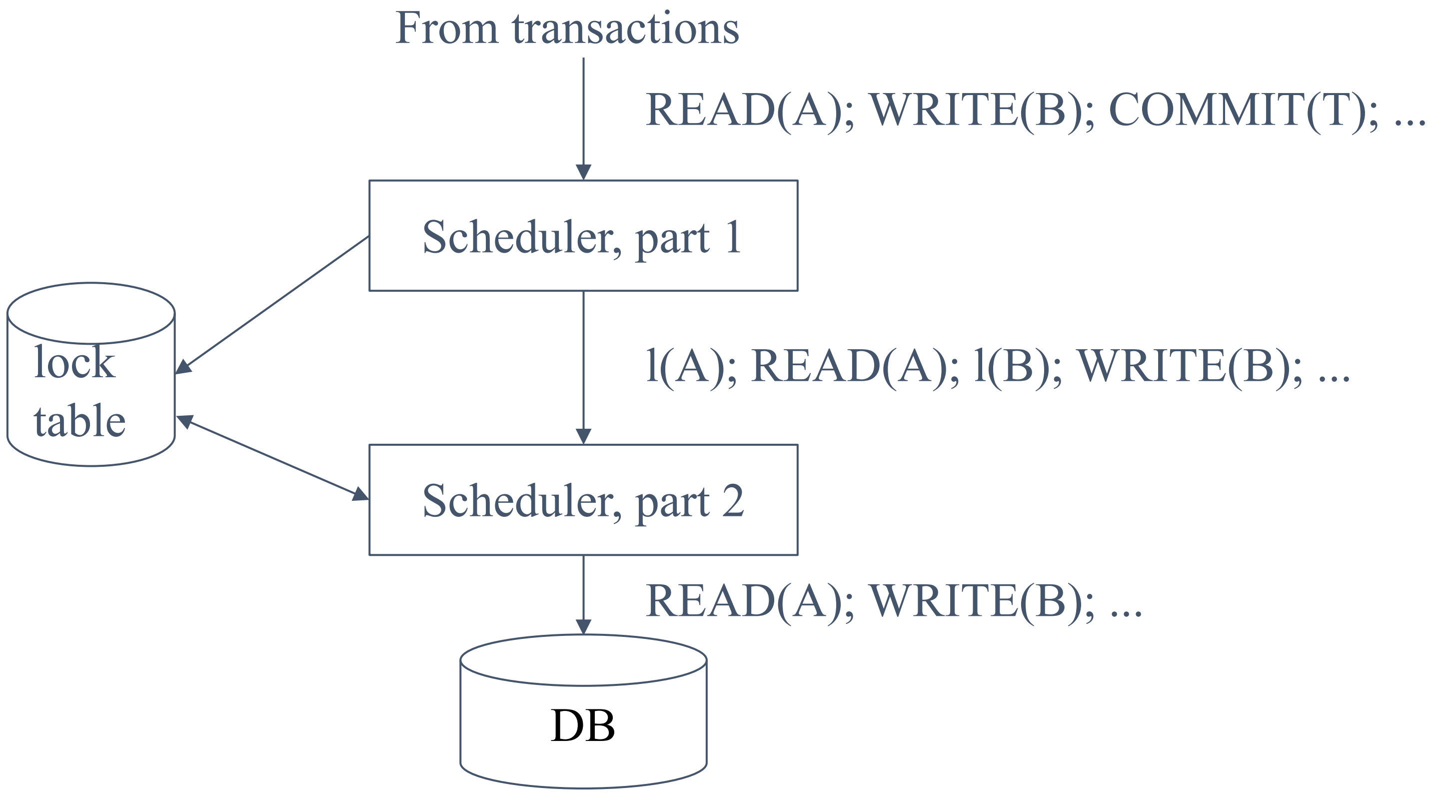
- their relation to conflict serializability (by acyclicity)

Ensuring serializability via locking

- Two phase locking
- Shared lock, exclusive lock, update lock, increment lock

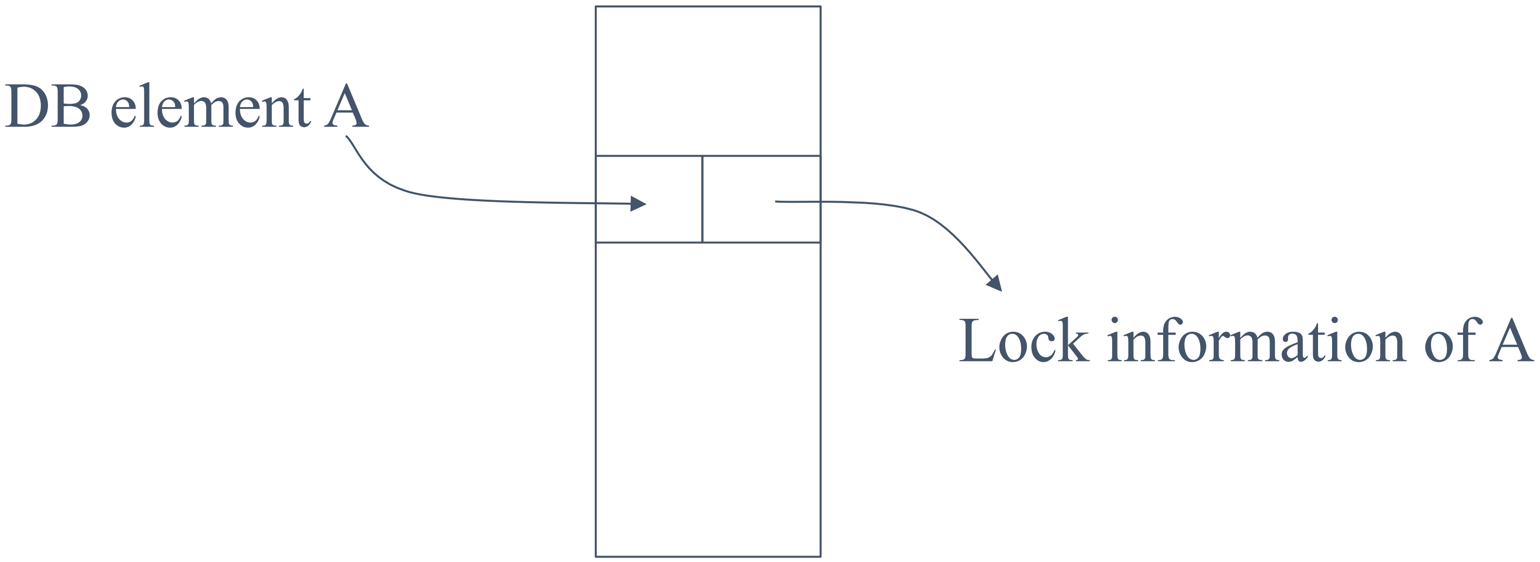
Locking scheduler

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



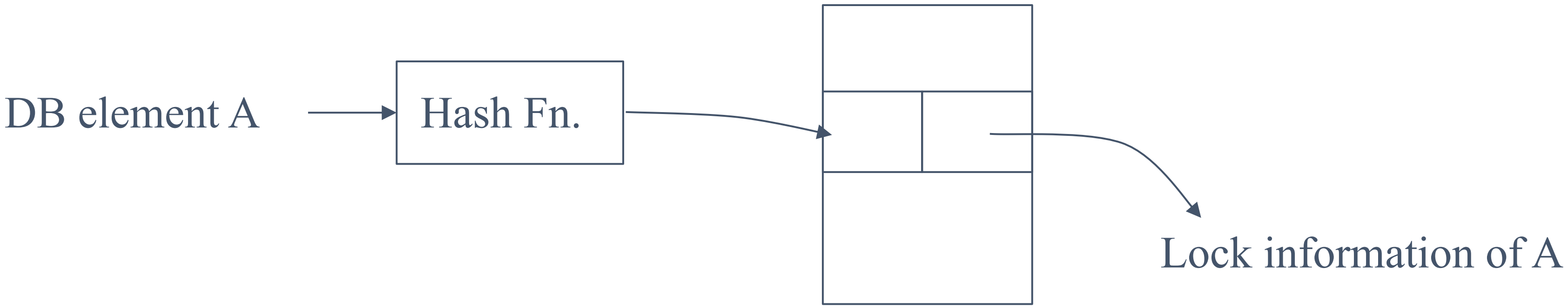
Lock table

- Maps database elements to lock information



Lock table

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



Locks With Multiple Granularity

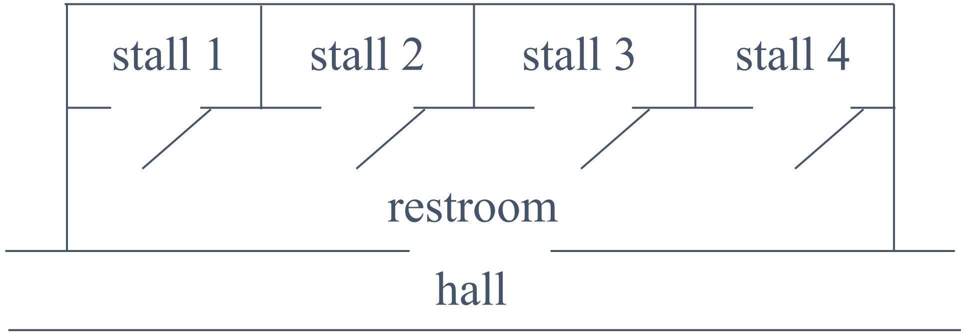
- Relations → Least concurrency
- Pages or data blocks
- Tuples → Most concurrency, but also expensive

Locks With Multiple Granularity

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

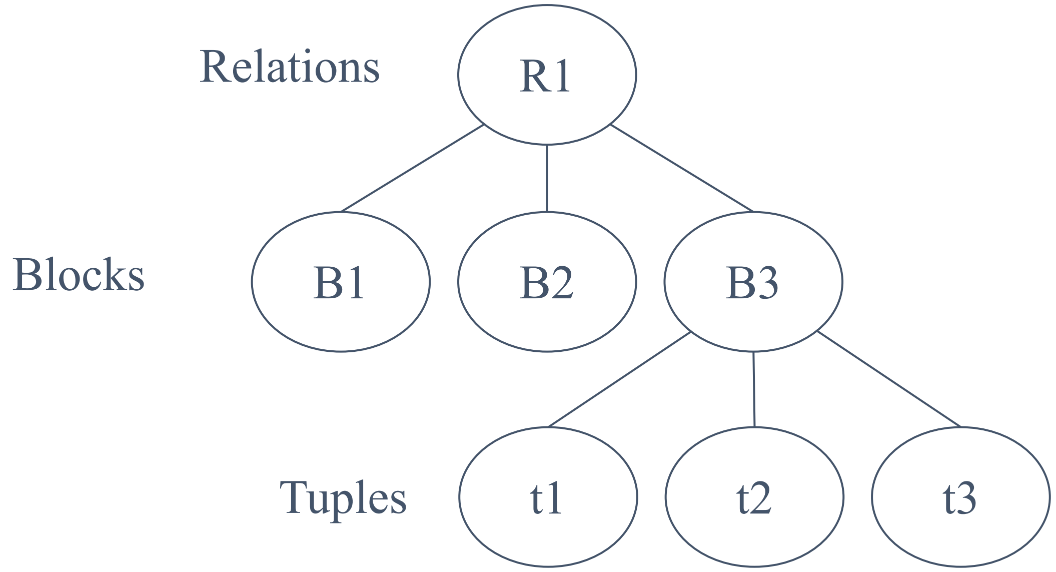
We can have it all ways using warning locks

Just ask any janitor



Warning locks

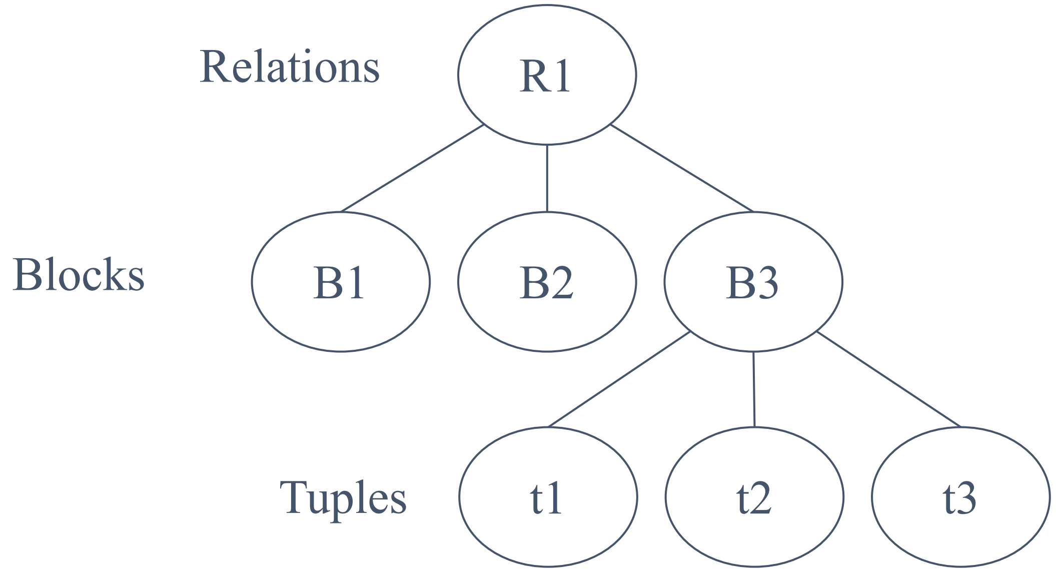
- Ordinary locks: S and X
- Warning locks: I (shows intention to lock)



Warning locks

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

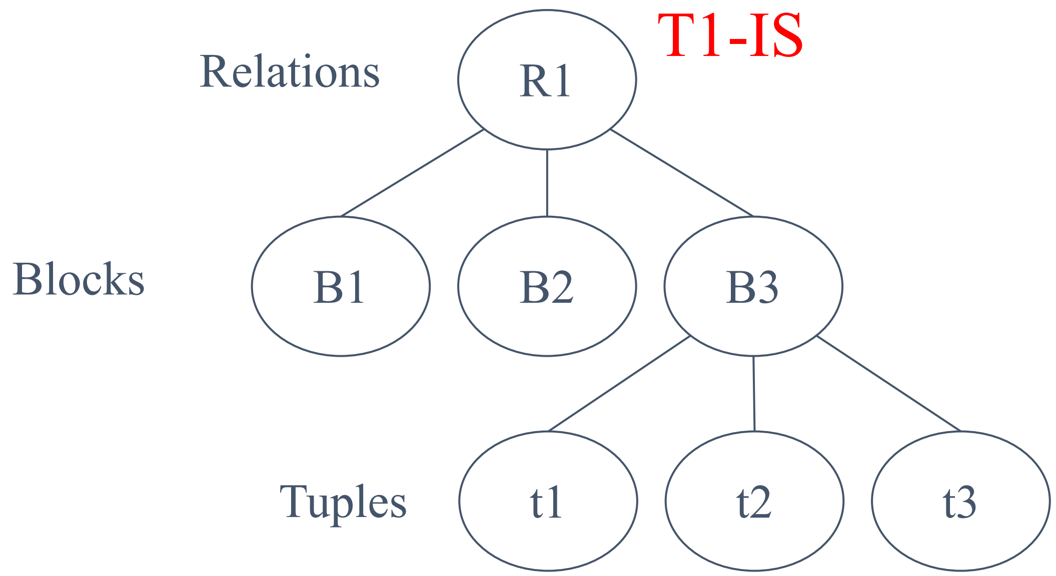
T1 wants to read t3



Warning locks

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

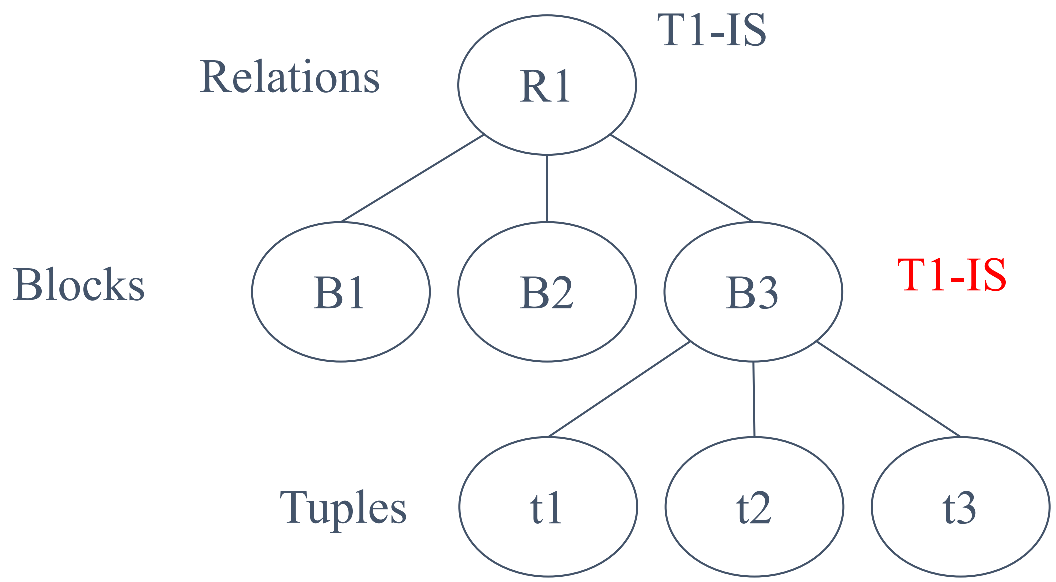
T1 wants to read t3



Warning locks

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

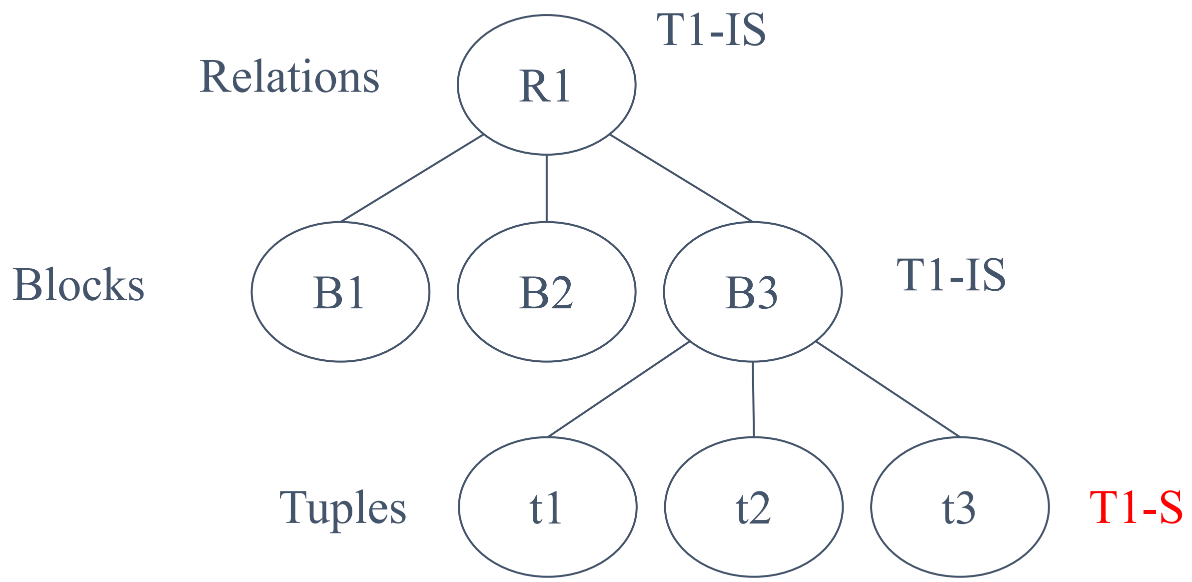
T1 wants to read t3



Warning locks

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

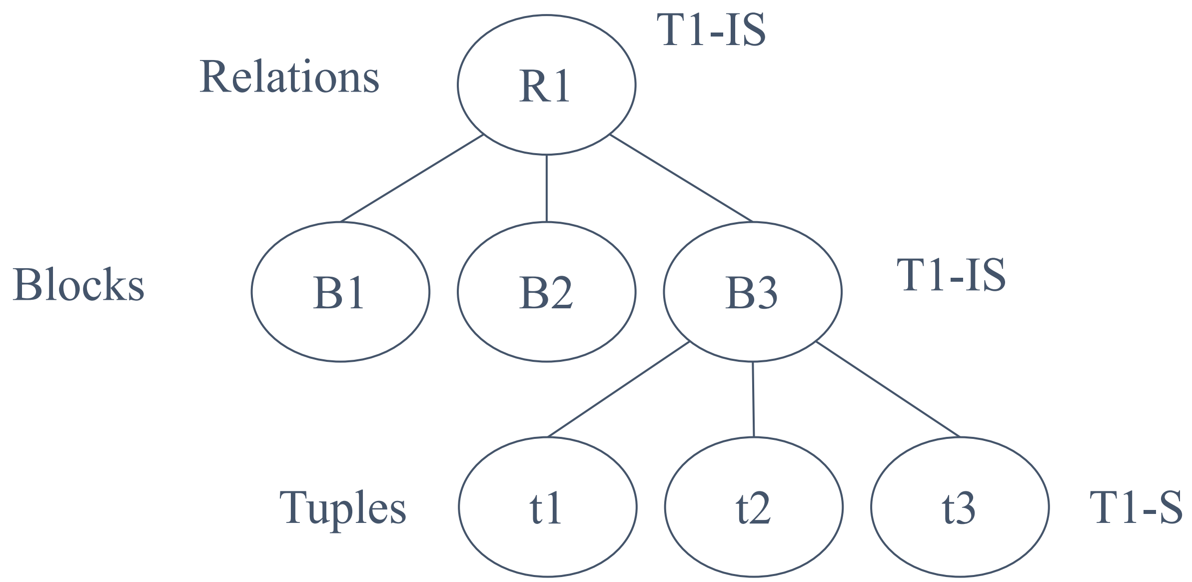
T1 wants to read t3



Warning locks

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

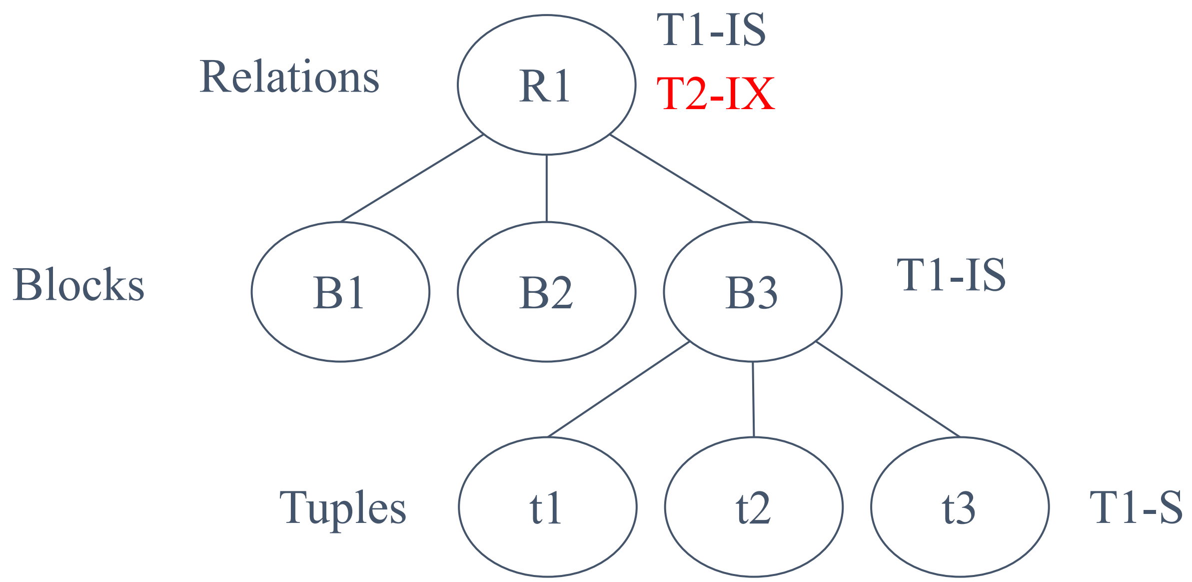
T1 wants to read t3
T2 wants to write B2



Warning locks

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

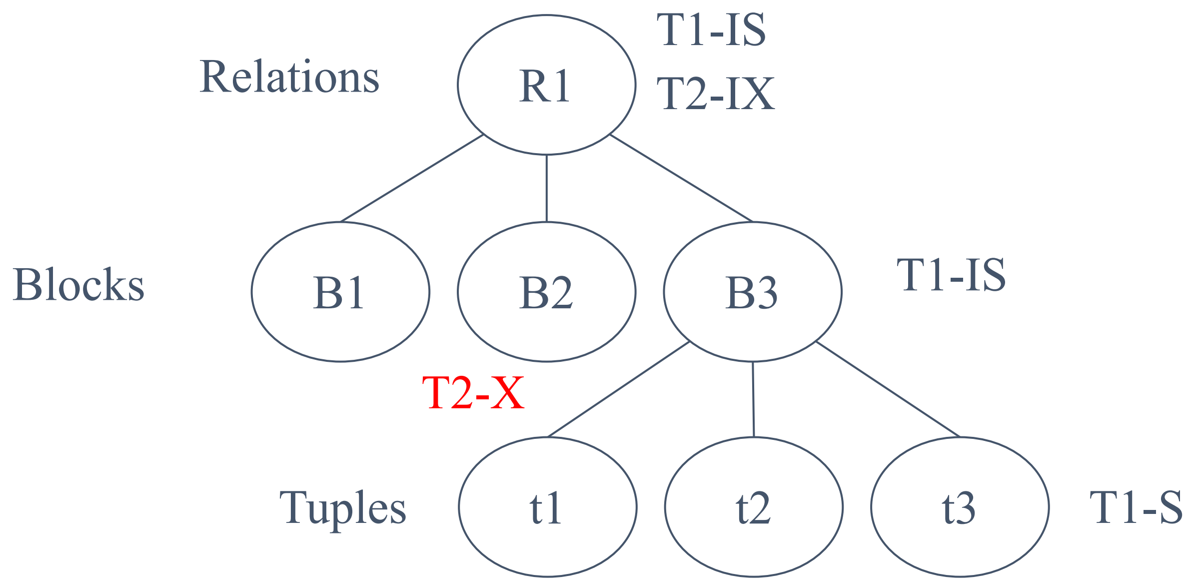
T1 wants to read t3
T2 wants to write B2



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

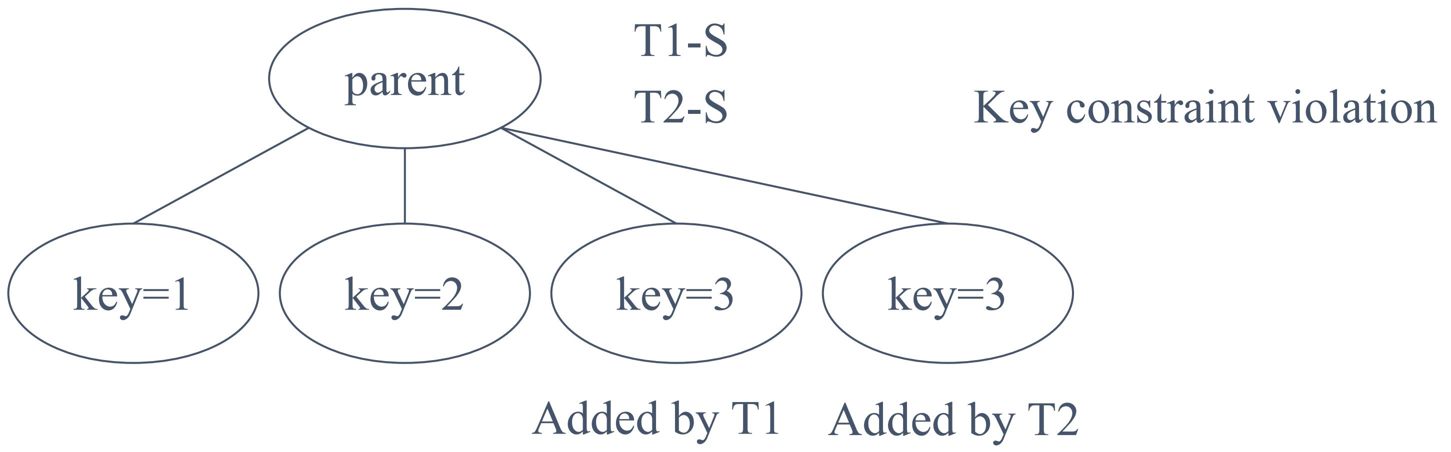
- For shared, exclusive, and intention locks

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

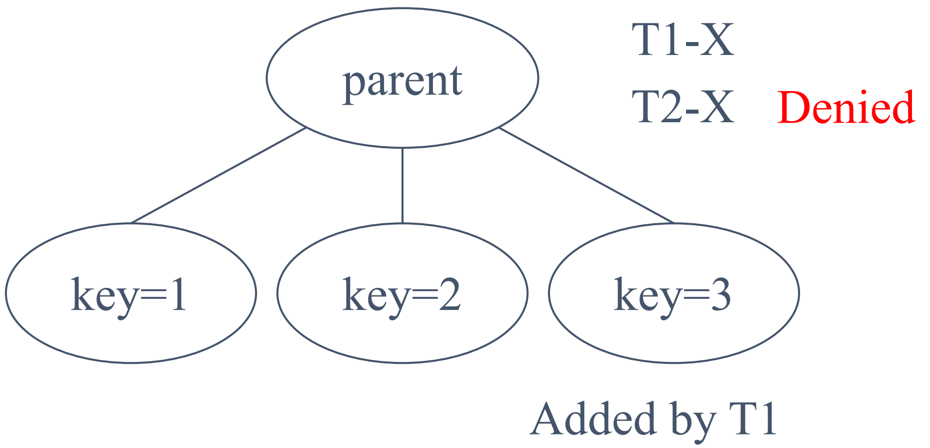
Inserts and Deletes

- Get exclusive lock on X before deleting it
- For inserts, need to be more careful
  - Cannot lock “future” elements
  - If no exclusive lock is held, then database can become inconsistent due to “phantoms”



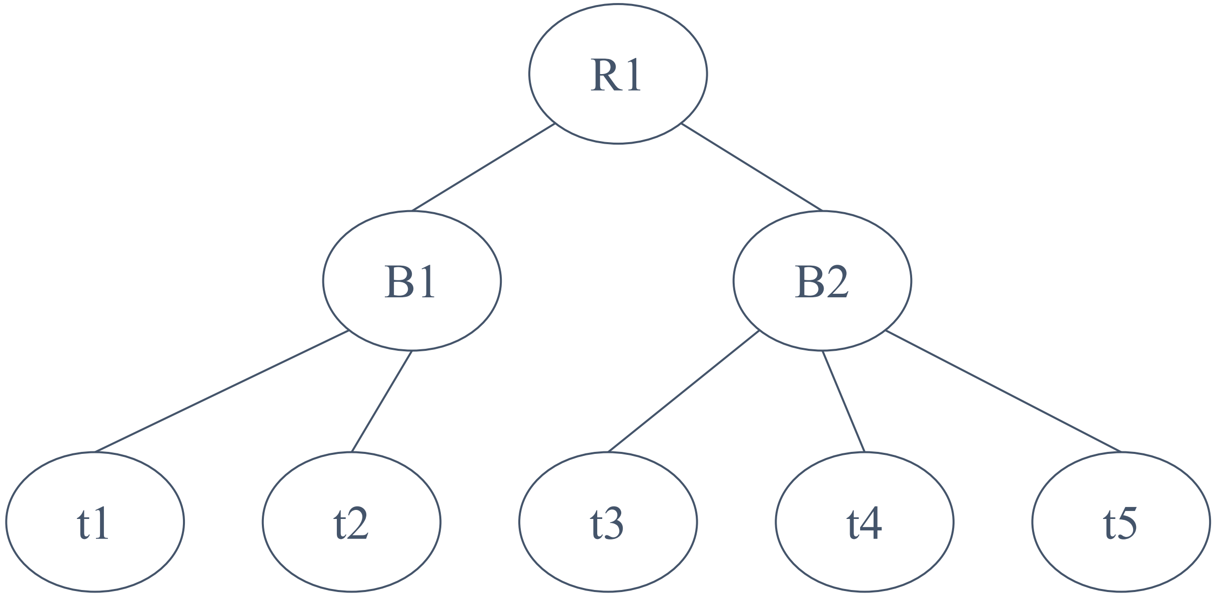
Inserts and Deletes

- Solution for inserts: use exclusive lock on the parent of the new tuple



Exercise #1

- Given the hierarchy of objects, what is the sequence of lock requests by T1 and T2 for the sequence of requests: $r_1(t_5)$; $w_2(t_5)$; $r_2(t_3)$;



Concurrency control by validation

- This method is optimistic
  - Assume no unserializable behavior
  - Abort transactions when violation is apparent

- In comparison, locking methods are pessimistic
  - Assume things will go wrong
  - Prevent nonserializable behavior

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 compute locally everything it is going to write
  - Validate T by comparing RS(T) and WS(T) with other transactions
  - Write elements in WS(T) to disk, if validation is OK
- Validation needs to be done atomically
  - Validation order = hypothetical serial order

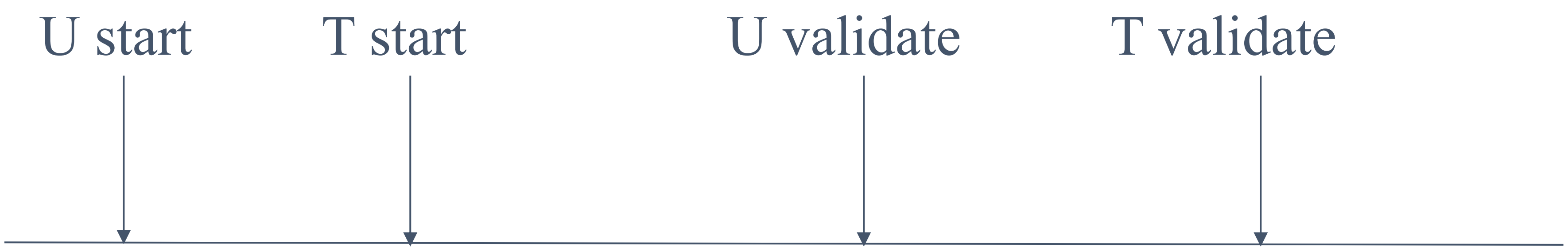
To validate, scheduler maintains three sets

- START: set of transactions that started, but have not validated
- VAL: set of transactions that validated, but not yet finished write phase
- FIN: set of transactions that have completed write phase

Validation rules (assume U validated)

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

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

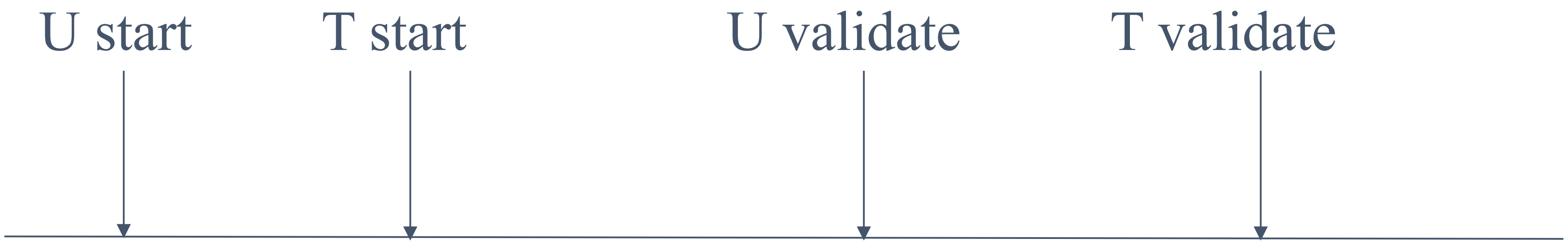


Validation rules (assume U validated)

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

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

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

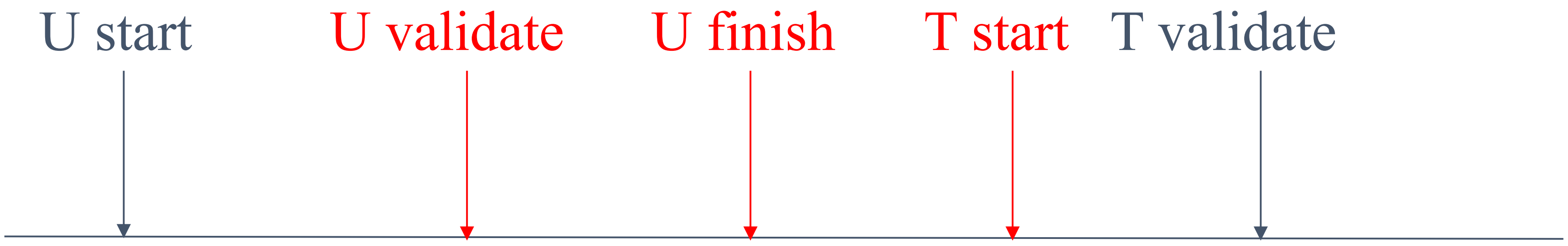


Validation rules (assume U validated)

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

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

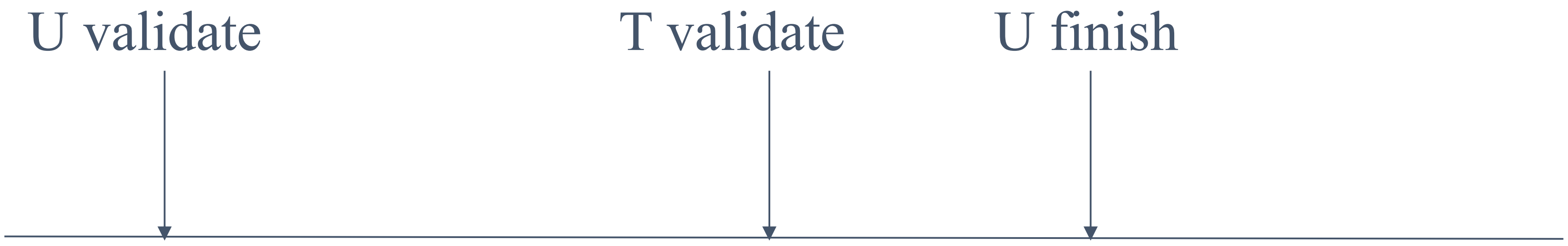
This satisfies rule 1



Validation rules (assume U validated)

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

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

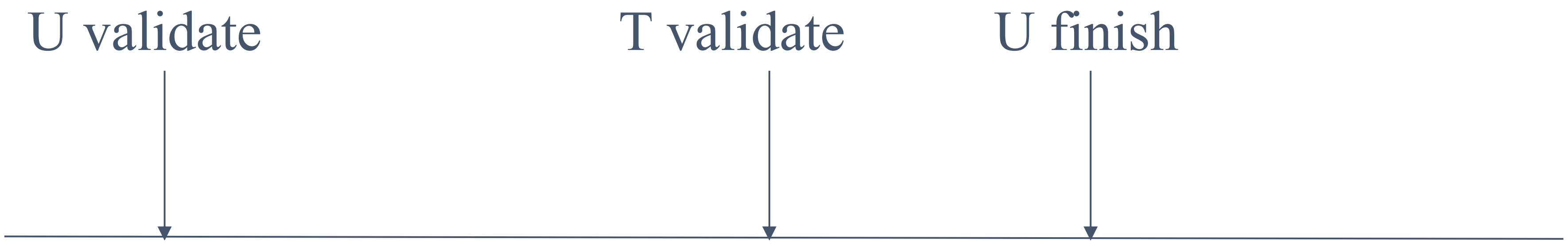


Validation rules (assume U validated)

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

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

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

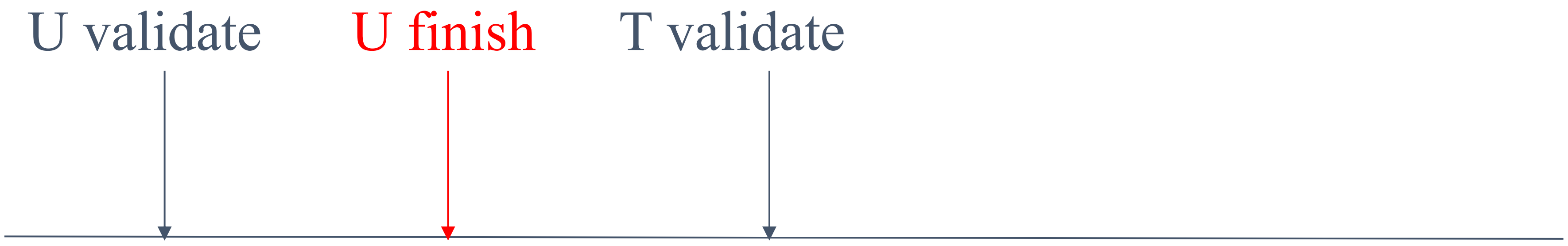


Validation rules (assume U validated)

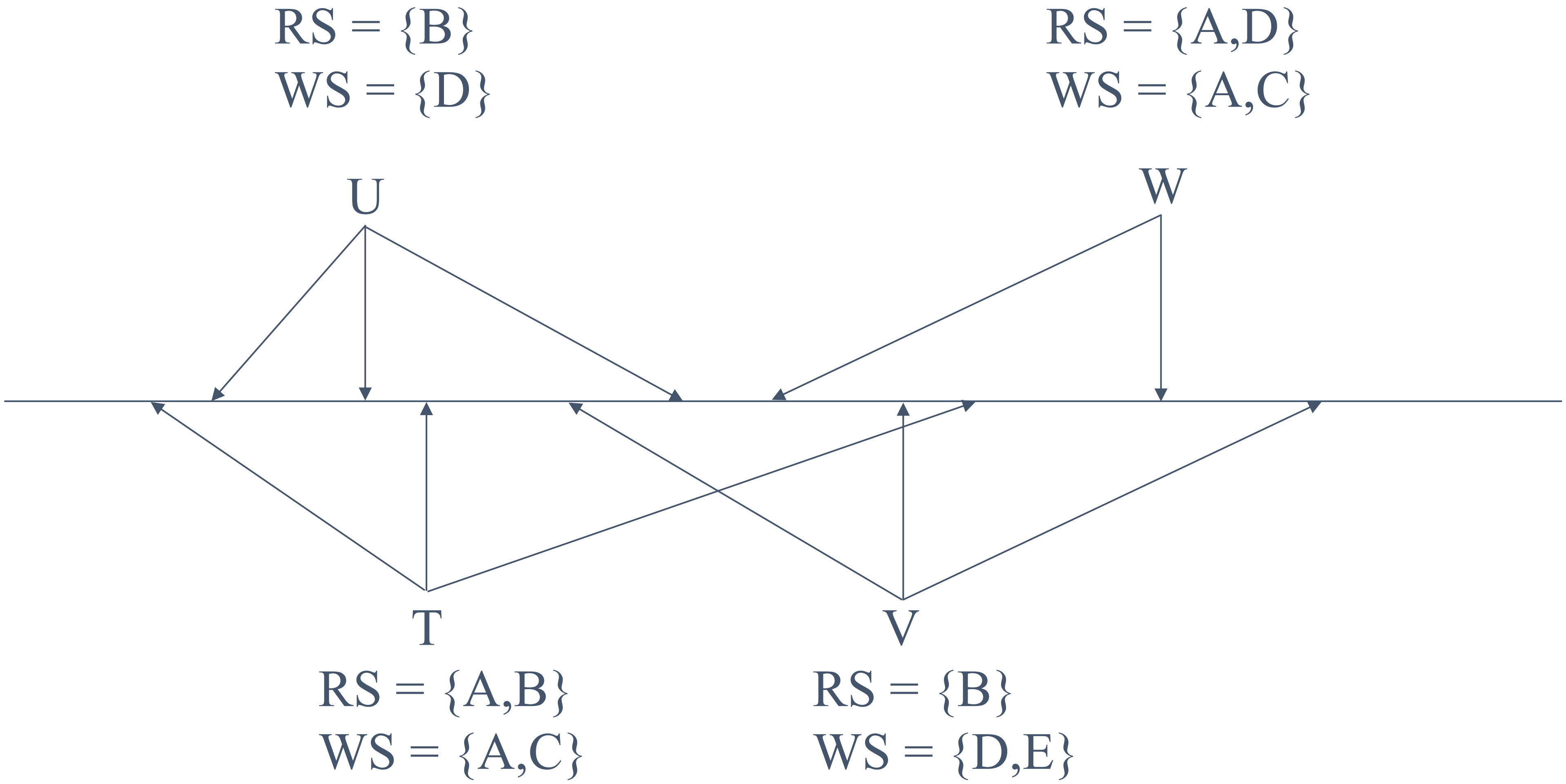
Rule 2: $WS(T) \cap WS(U) = \emptyset$ if $FIN(U) > VAL(T)$

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

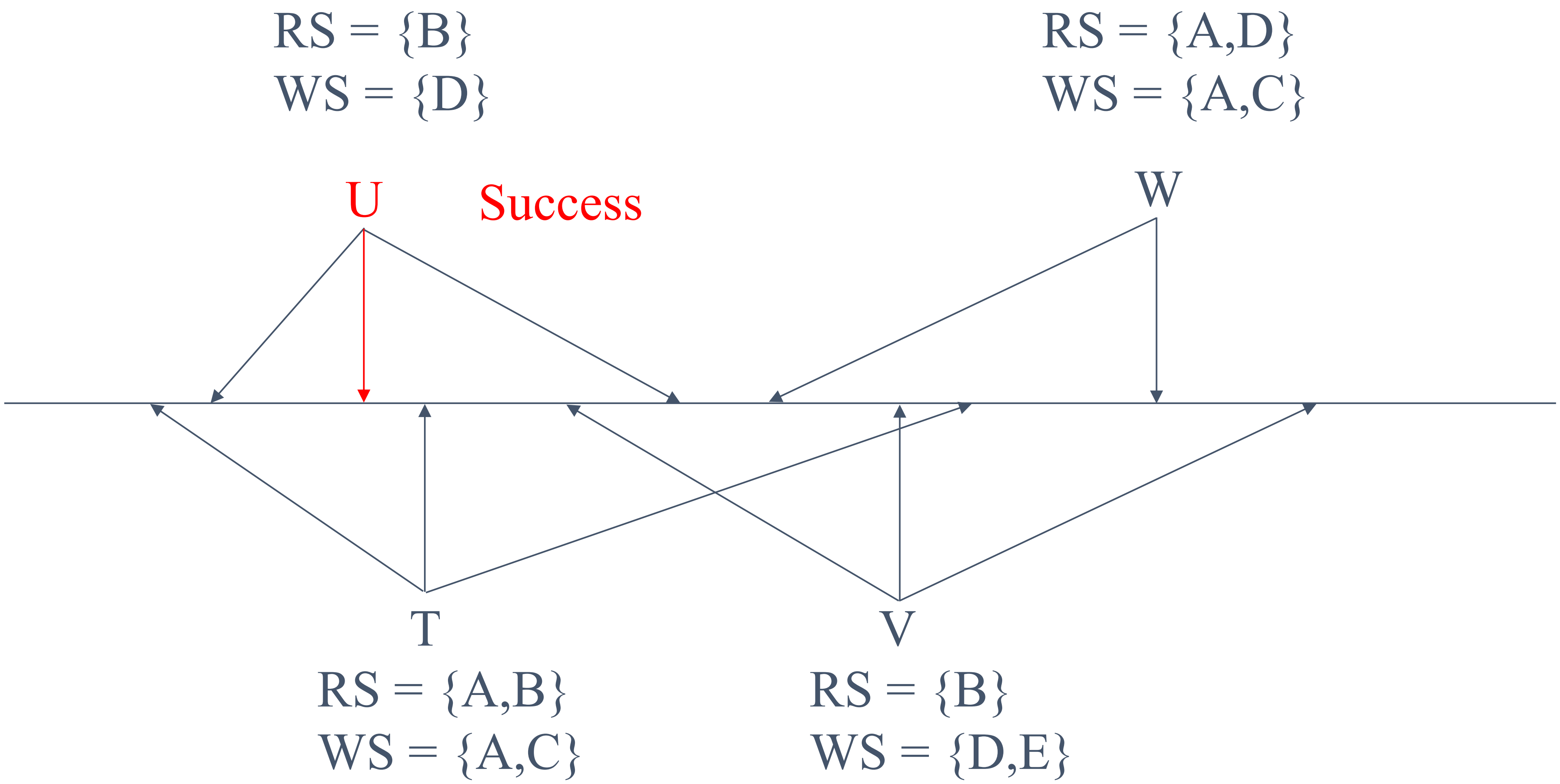
This satisfies rule 2



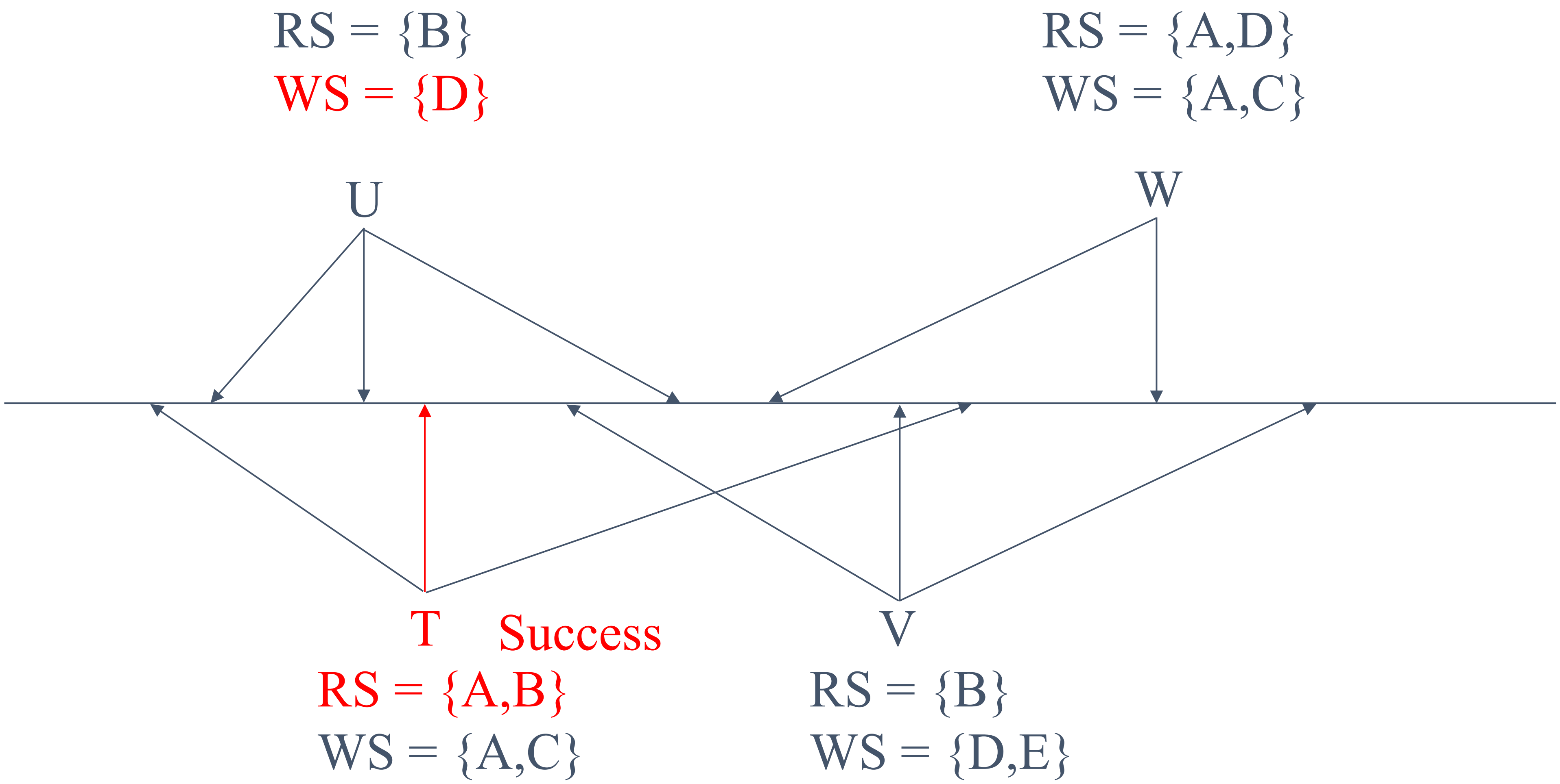
Running example



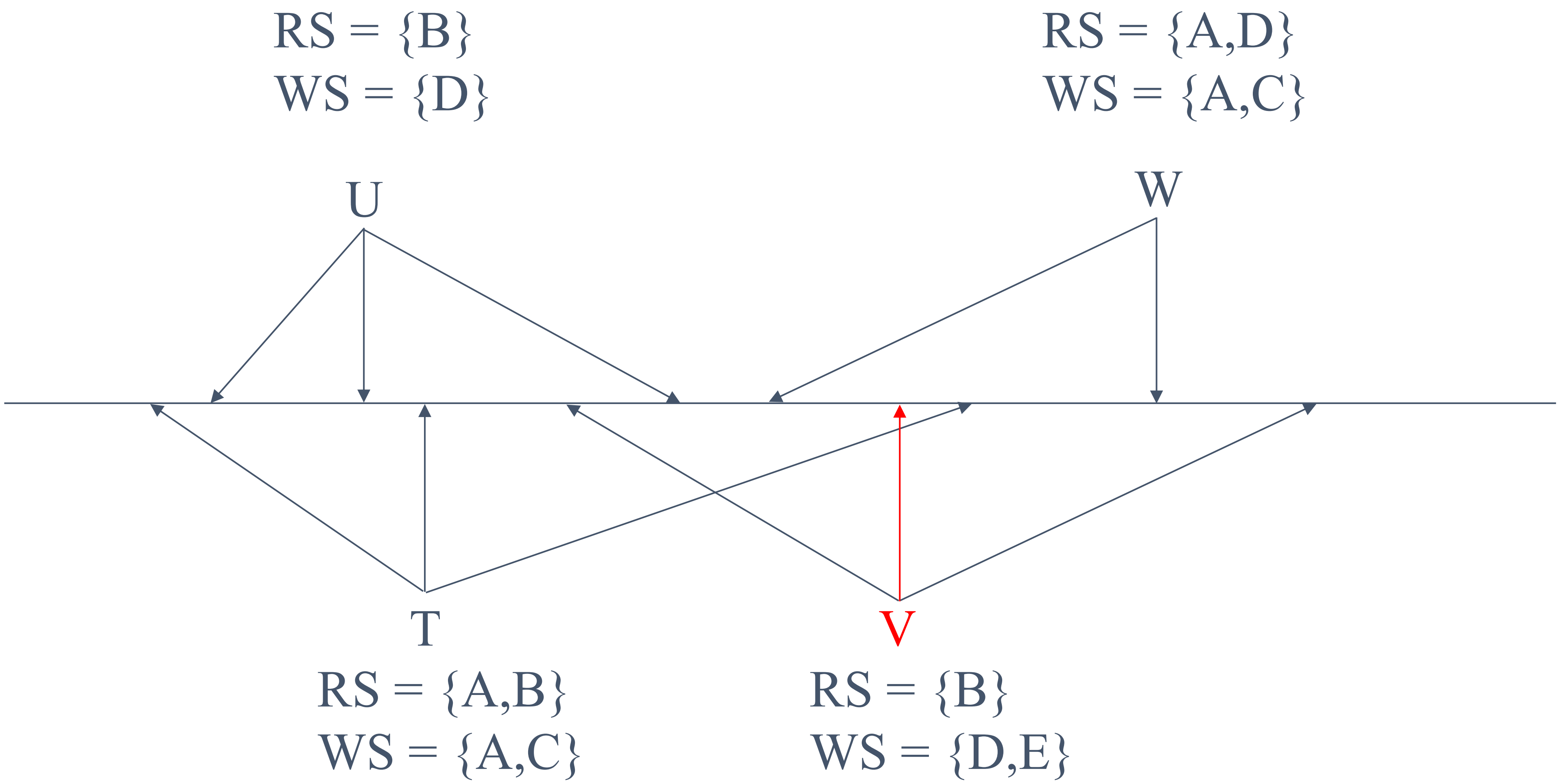
Running example



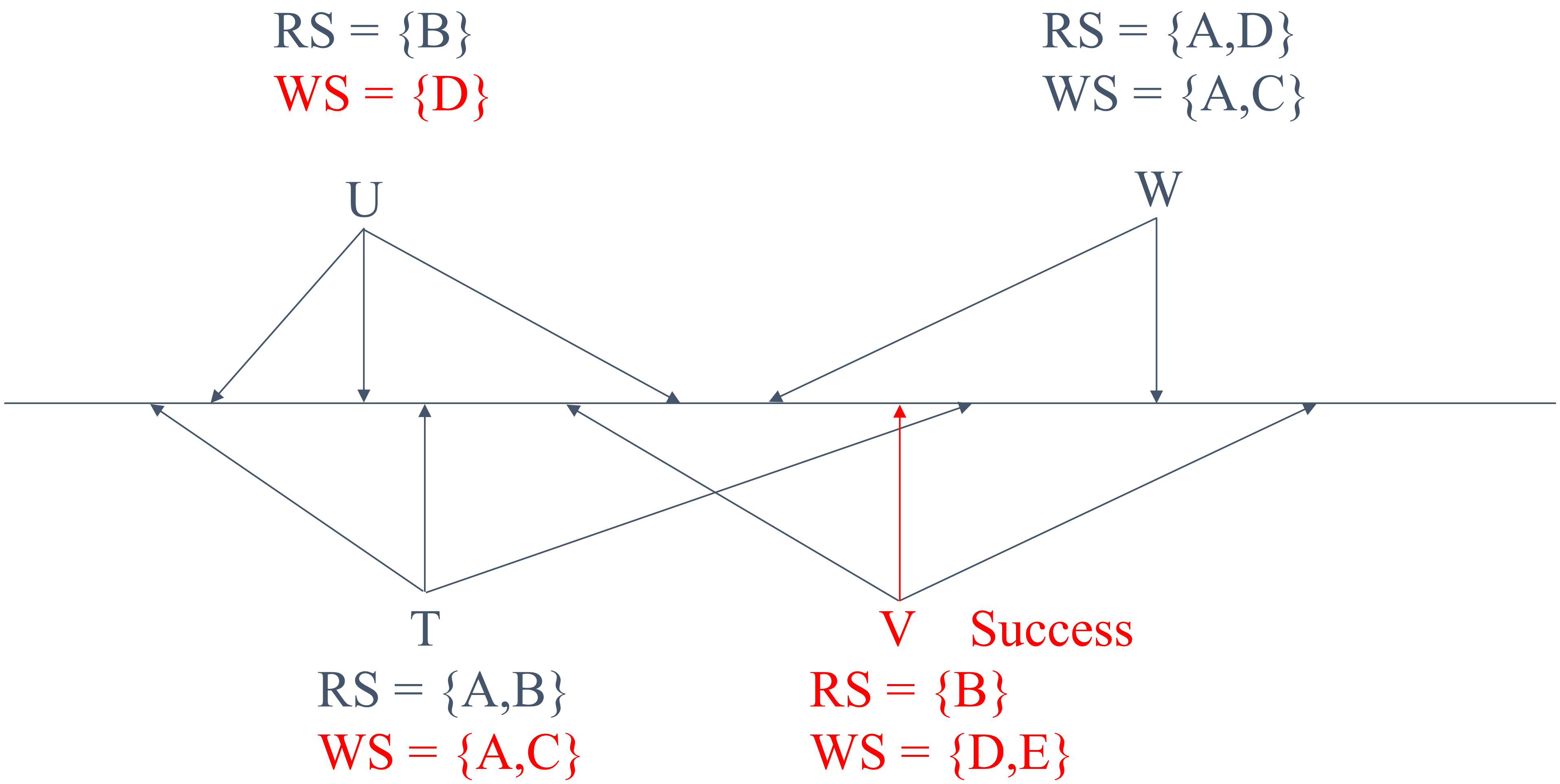
Running example



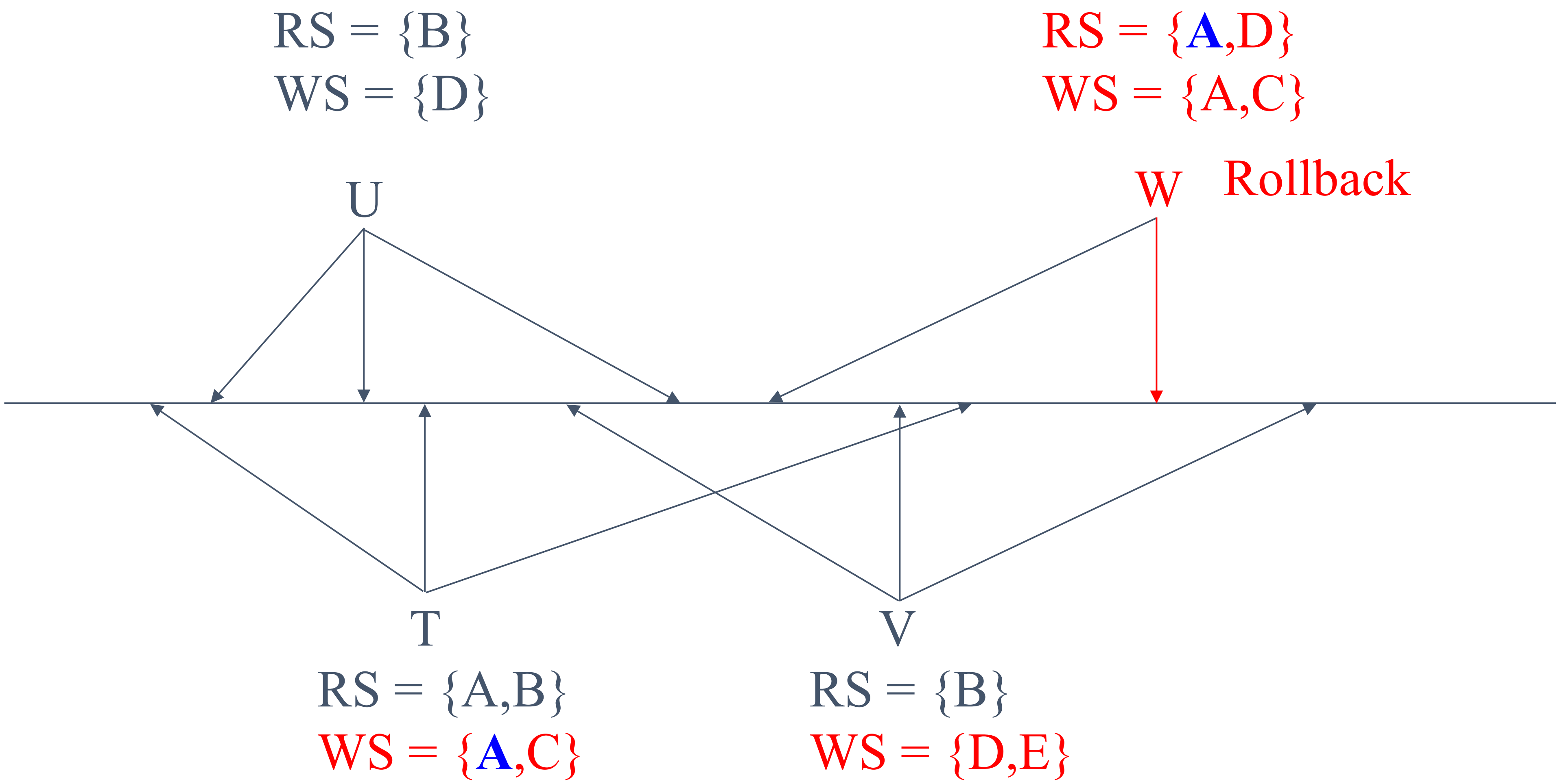
Running example



Running example



Running example



Validation is useful when

- Conflicts are rare
- System resources are plentiful
- Application has real-time constraints

Summary: Approaches to Concurrency Control

Lock-based CC

- 2PL
- Multiple granularity

Optimistic CC

- CC by validation
- Time-stamp-based CC
  - Not covered, Chapter 18.8

Recap: ACID properties

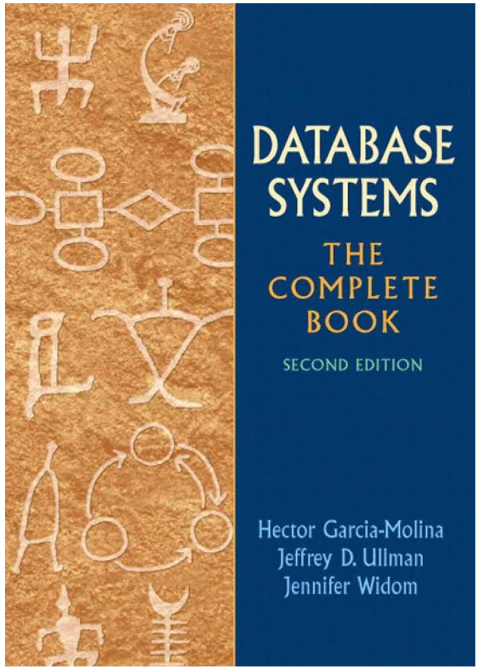
- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates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

Ensuring atomicity and durability with logging and recovery manager

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 17 - Copying with System Failures



Supplementary materials

Fundamental of Database Systems (7th Edition)

- Chapter 22 - Database Recovery Techniques

Failure modes and solutions

- Erroneous data entry
  - Typos
    → Write constraints and triggers
- Media failures
  - Local disk failure, head crashes
    → Parity checks, RAID, archiving and copying
- Catastrophic failures
  - Explosions, fires
    → Archiving and copying
- System failures
  - Transaction state lost due to power loss and software errors
    → Logging

Today's focus

Recovery

Atomicity

- by ”undo”ing actions of “aborted transactions”

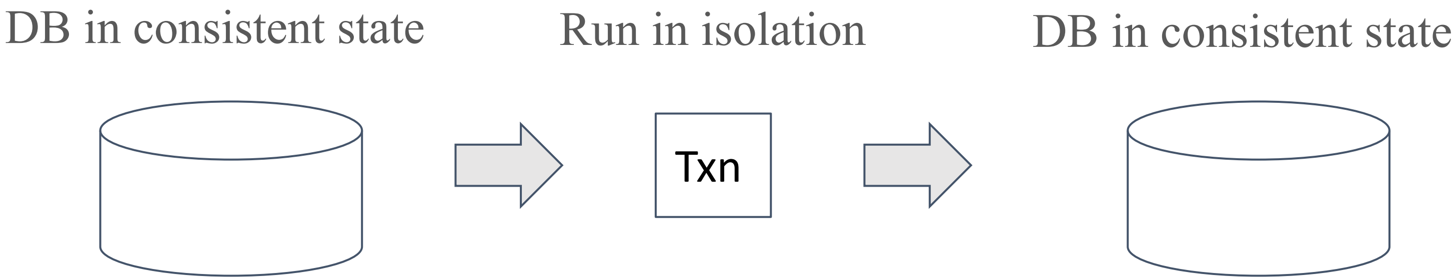
Durability

- by making sure that all actions of committed transactions survive crashes and system failure
- – i.e. by “redo”-ing actions of “committed transactions”

The Correctness Principle

A fundamental assumption about transaction is:

If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transactions ends.



Transaction primitives

- Example transaction
  - Consistent state: $A = B$

Logical steps

$A := A * 2$
$B := B * 2$

Execution

		Memory		Disk	
Action	t	A	B	A	B
READ(A, t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A, t)	16	16		8	8
READ(B, t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B, t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Transaction primitives

- Example transaction
  - Consistent state: $A = B$

Logical steps

$$A := A * 2$$
$$B := B * 2$$

Execution

Action	t	Memory A	Memory B	Disk A	Disk B
READ(A, t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A, t)	16	16		8	8
READ(B, t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B, t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Consistent

Transaction primitives

- Example transaction
  - Consistent state: $A = B$

Logical steps

$$A := A * 2$$
$$B := B * 2$$

Execution

Action	Memory t	Memory A	Memory B	Disk A	Disk B
READ(A, t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A, t)	16	16		8	8
READ(B, t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B, t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Consistent

Transaction primitives

- Example transaction
  - Consistent state: $A = B$

Logical steps

$A := A * 2$
$B := B * 2$

Execution

Action	Memory t	Memory A	Memory B	Disk A	Disk B
READ(A, t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A, t)	16	16		8	8
READ(B, t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B, t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Not consistent!
Either reset $A = 8$
or advance $B = 16$

#1 Undo logging

- Log: a file of log records telling what transaction has done

Action	t	Memory A	Memory B	Disk A	Disk B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

#1 Undo logging

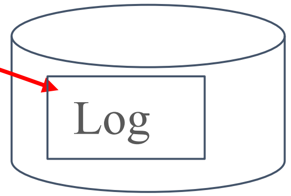
- Log: a file of log records telling what transaction has done

	Memory			Disk			
Action	*t*	*A*	*B*	*A*	*B*	Log	
						<START *T*>	*T* started
READ(*A*, t)	8	8		8	8		
t := *t* * 2	16	8		8	8		
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>	*T* changed *A*, and its former value is 8
READ(*B*, *t*)	8	16	8	8	8		
t := *t* * 2	16	16	8	8	8		
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>	
FLUSH LOG							
OUTPUT(*A*)	16	16	16	16	8		
OUTPUT(*B*)	16	16	16	16	16		
						<COMMIT *T*>	*T* completed successfully
FLUSH LOG							

#1 Undo logging

- Log: a file of log records telling what transaction has done

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						



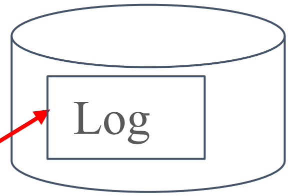
Rule 1:
<*T, A,* 8> must be flushed to disk before new *A* is written to disk (same for *B*)

#1 Undo logging

- Log: a file of log records telling what transaction has done

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Rule 1:
<*T*, *A*, 8> must be flushed to disk before new *A* is written to disk (same for *B*)



Rule 2:
<COMMIT *T*> must be flushed to disk after *A* and *B* are written to disk

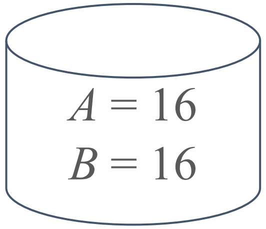
Recovery using undo logging

- Simplifying assumption: use entire log, no matter how long

	Memory			Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



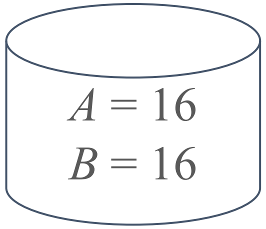
Recovery using undo logging

- Simplifying assumption: use entire log, no matter how long

		Memory		Disk		
Action	t	A	B	A	B	Log
						<START *T*>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<*T*, *A*, 8>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



Observe <COMMIT *T*> record

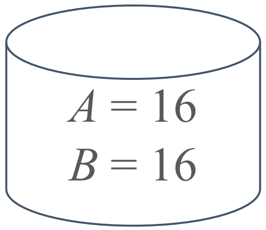
Recovery using undo logging

- Simplifying assumption: use entire log, no matter how long

Action	Memory *t*	Memory *A*	Memory *B*	Disk *A*	Disk *B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



Ignore (*T* was committed)

↑

Observe <COMMIT *T*> record

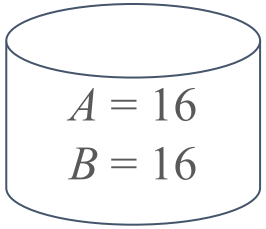
Recovery using undo logging

- Simplifying assumption: use entire log, no matter how long

Action	t	Memory A	Memory B	Disk A	Disk B	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



Ignore (*T* was committed)

↑

Ignore (*T* was committed)

↑

Observe <COMMIT *T*> record

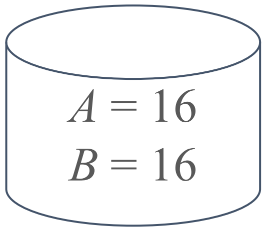
Recovery using undo logging

- Simplifying assumption: use entire log, no matter how long

	Memory			Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



Recovery using undo logging

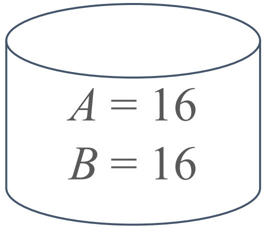
- Simplifying assumption: use entire log, no matter how long

Action	t	Memory A	Memory B	Disk A	Disk B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Crash

<COMMIT T> may or may not have been flushed to disk. If so, same as previous scenario. If not, T is considered incomplete

Recovery



Recovery using undo logging

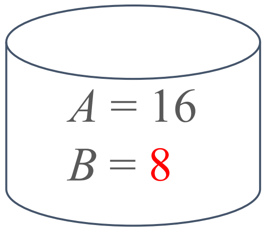
- Simplifying assumption: use entire log, no matter how long

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

If *T* was incomplete, set *B* to previous value 8 on disk

Recovery



Recovery using undo logging

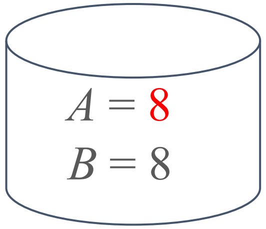
- Simplifying assumption: use entire log, no matter how long

		Memory		Disk		
Action	t	A	B	A	B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Crash

If T was incomplete, set A to previous value 8 on disk

Recovery



Recovery using undo logging

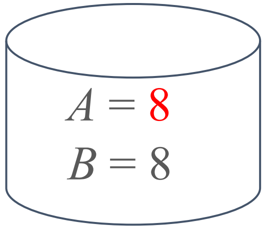
- Simplifying assumption: use entire log, no matter how long

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
~~FLUSH LOG~~						

Crash

Recovery

Write <ABORT *T*> to log and flush to disk

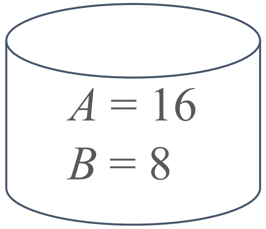


Recovery using undo logging

- Simplifying assumption: use entire log, no matter how long

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
Crash						
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Recovery

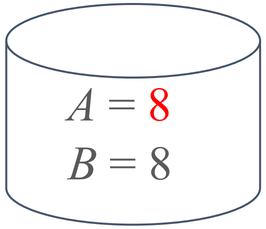


Recovery using undo logging

- Simplifying assumption: use entire log, no matter how long

		Memory		Disk		
Action	t	A	B	A	B	Log
						<START T>
READ(A, t)	8	8		8	8	
$t := t * 2$	16	8		8	8	
WRITE(A, t)	16	16		8	8	<T, A, 8>
READ(B, t)	8	16	8	8	8	
$t := t * 2$	16	16	8	8	8	
WRITE(B, t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
Crash						
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Recovery



Same recovery as before, but only A is set to previous value

What happens if the system crashes during the recovery?

- Undo-log recovery is idempotent, so repeating the recovery is OK

Image source: https://insightrsblog.com/2010/09/17/do-you-want-to-recreate-your-entire-database/

Exercise #2

- Given the undo log, describe the action of the recovery manager

<START T>
<T, *A*, 10>
<START U>
<U, *B*, 20>
<T, *C*, 30>
<U, *D*, 40>
<COMMIT U>

Checkpointing

- Entire log can be too long
- Cannot truncate log after a COMMIT because there are other running transactions

Checkpointing

- Solution: checkpoint log periodically

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>

Checkpointing

- Solution: checkpoint log periodically

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>

Stop accepting new transactions

Checkpointing

- Solution: checkpoint log periodically

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<T2, *C*, 15>
<T1, *D*, 20>
<COMMIT T1>
<COMMIT T2>

Stop accepting new transactions

Wait until all transactions commit or abort

Checkpointing

- Solution: checkpoint log periodically

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<T2, *C*, 15>
<T1, *D*, 20>
<COMMIT T1>
<COMMIT T2>
<CKPT>

Stop accepting new transactions

Wait until all transactions commit or abort

Flush log

Write <CKPT> and flush

Checkpointing

- Solution: checkpoint log periodically

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<T2, *C*, 15>
<T1, *D*, 20>
<COMMIT T1>
<COMMIT T2>
<CKPT>
<START T3>
<T3, E, 25>
<T3, F, 30>

Stop accepting new transactions

Wait until all transactions commit or abort

Flush log

Write <CKPT> and flush

Resume transactions

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

Crash

If we first meet <END CKPT>, only need to recover until <START CKPT (T1, T2)>

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>

Crash

<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>

If we first meet <START CKPT (T1, T2)>, only need to recover until <START T1>