Concurrency Control

Instructor: Matei Zaharia

cs245.stanford.edu

Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking

- » Shared and exclusive locks
- » Lock tables and multi-level locking

Optimistic concurrency with validation

Concurrency control + recovery

Lock Modes Beyond S/X

Examples:

(1) increment lock

(2) update lock

Example 1: Increment Lock

Atomic addition action: $IN_i(A)$

$\{Read(A);\ A \leftarrow A+k;\ Write(A)\}$

$IN_i(A)$, $IN_j(A)$ do not conflict, because addition is commutative!

Compatibility Matrix

compat

	S	X	I
S	T	F	F
X	F	F	F
I	F	F	T

Update Locks

A common deadlock problem with upgrades:

T1	T2
I-S_1(A)	
	I-S_2(A)
I-X_1(A)	
	I-X_2(A)

--- Deadlock ---

Solution

If Ti wants to read A and knows it may later want to write A, it requests an **update lock** (not shared lock)

Compatibility Matrix

New request

compat	S	X	U
S	T	F	
X	F	F	
U			

Lock already held in

Compatibility Matrix

compat

New request

Lock already held in

	S	X	U
S	T	F	T
X	F	F	F
U	F	F	F

Note: asymmetric table!

How Is Locking Implemented In Practice?

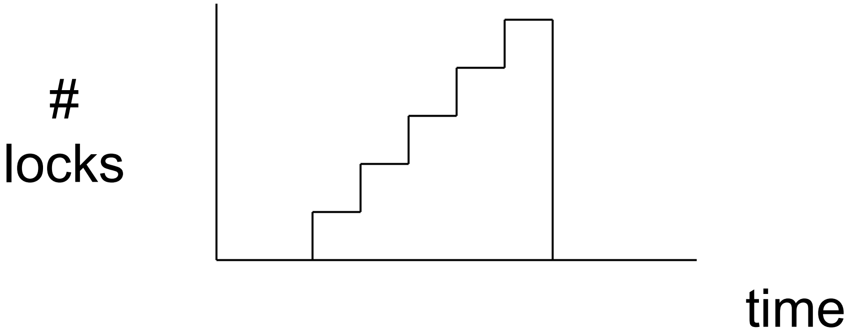
Every system is different (e.g., may not even provide conflict serializable schedules)

But here is one (simplified) way ...

Sample Locking System

1. Don't ask transactions to request/release locks: just get the weakest lock for each action they perform

2. Hold all locks until transaction commits



Sample Locking System

Under the hood: lock manager that keeps track of which objects are locked

- » E.g. hash table

Also need a good way to block transactions until locks are available, and find deadlocks

Which Objects Do We Lock?

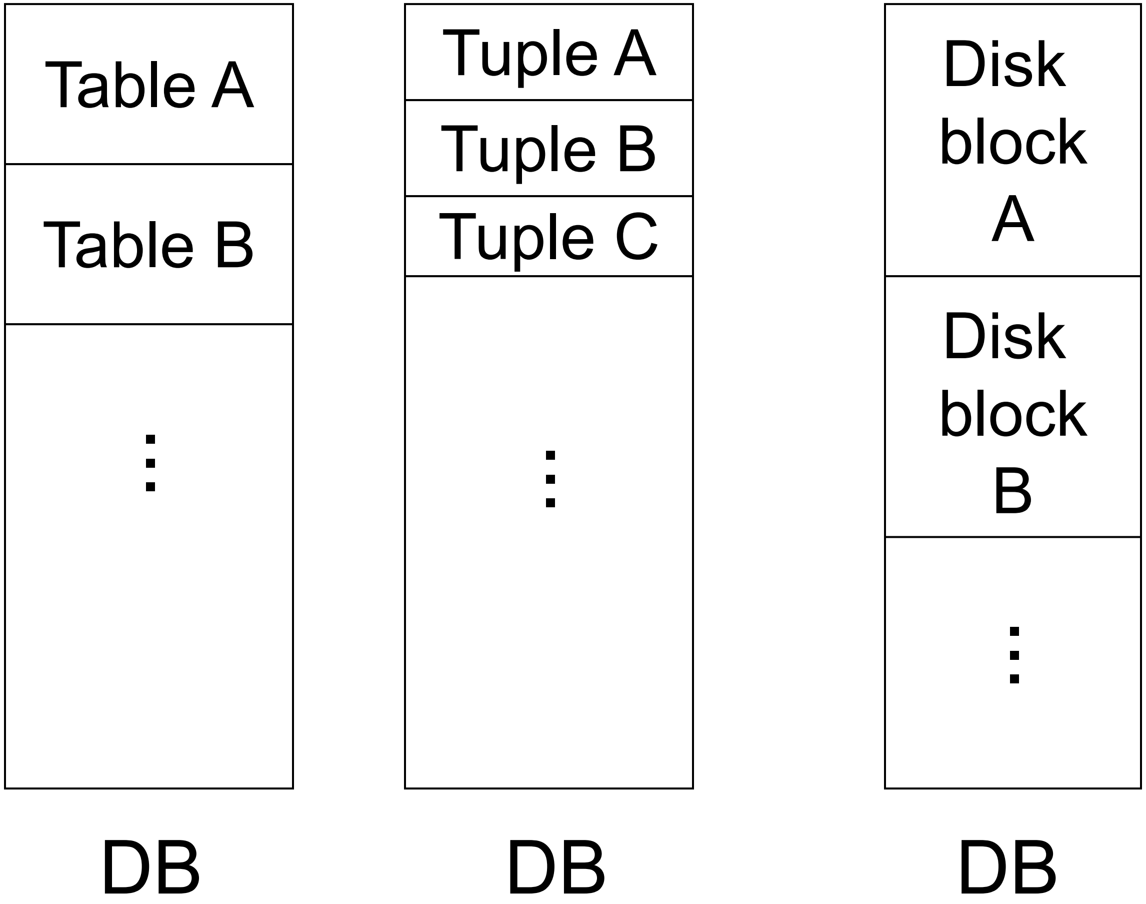


Table A

Table B

⋮

DB

Tuple A

Tuple B

Tuple C

⋮

DB

Disk block A

Disk block B

⋮

DB

Which Objects Do We Lock?

Locking works in any case, but should we choose **small** or **large** objects?

Which Objects Do We Lock?

Locking works in any case, but should we choose **small** or **large** objects?

If we lock **large** objects (e.g., relations)

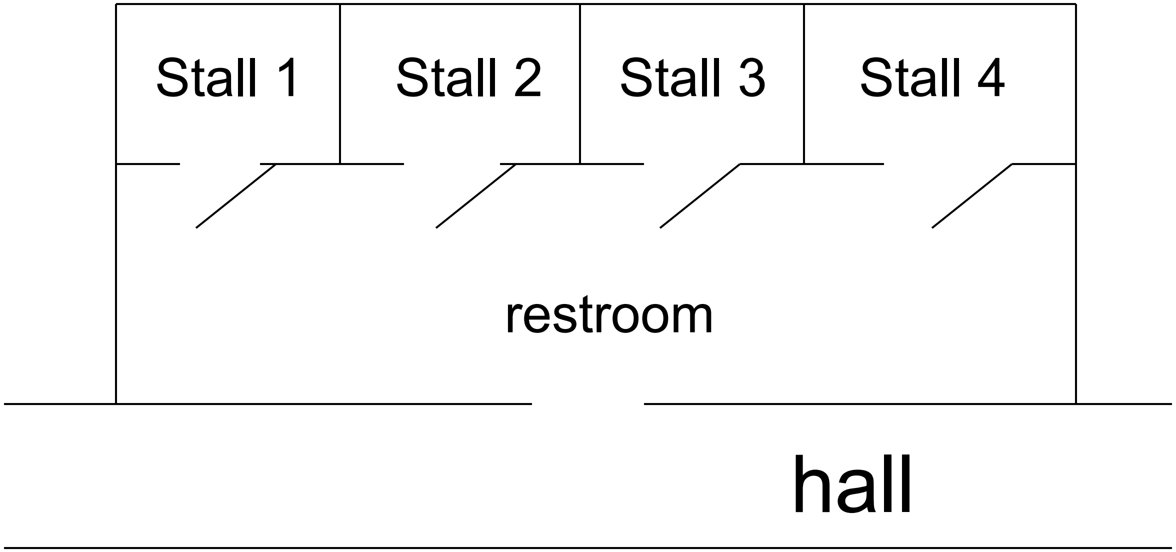
- Need few locks
- Low concurrency

If we lock **small** objects (e.g., tuples, fields)

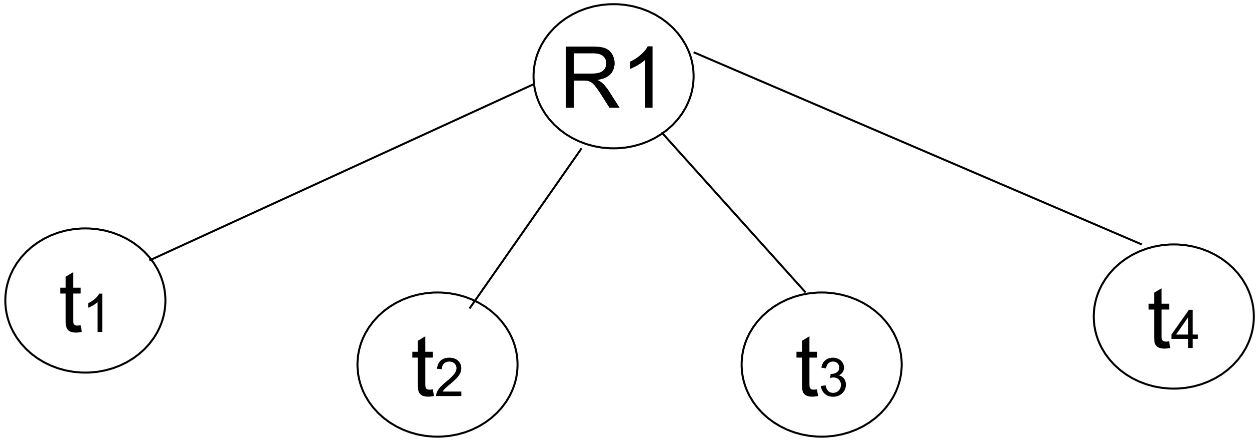
- Need more locks
- More concurrency

We Can Have It Both Ways!

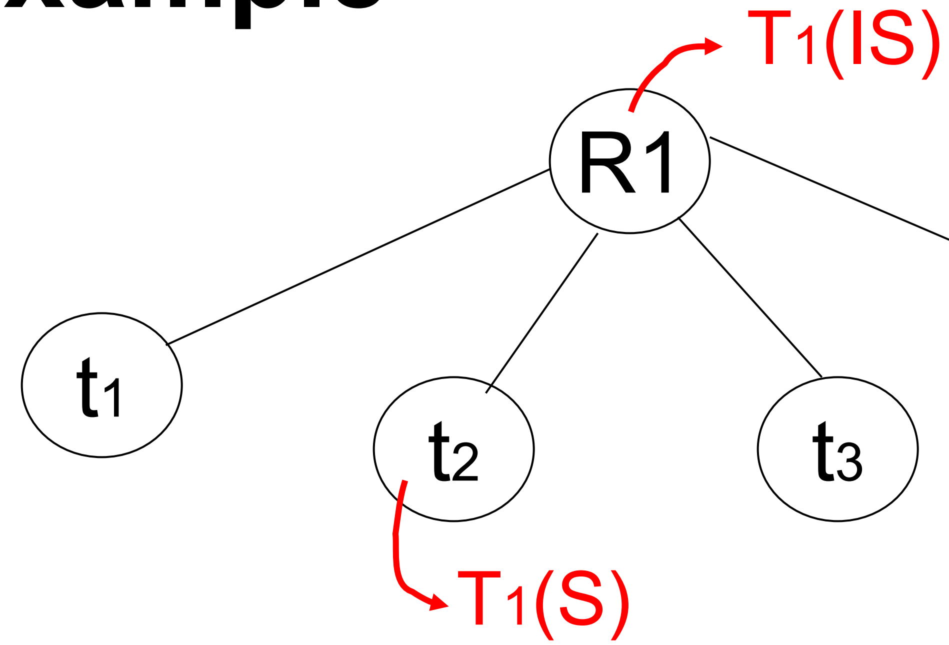
Ask any janitor to give you the solution...



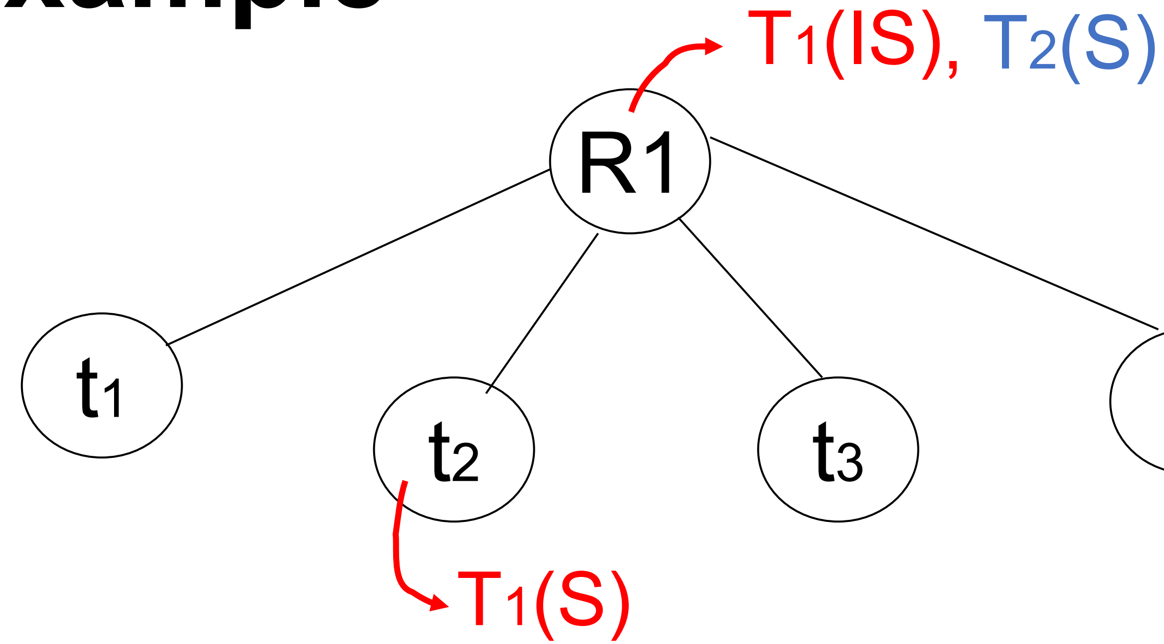
Example



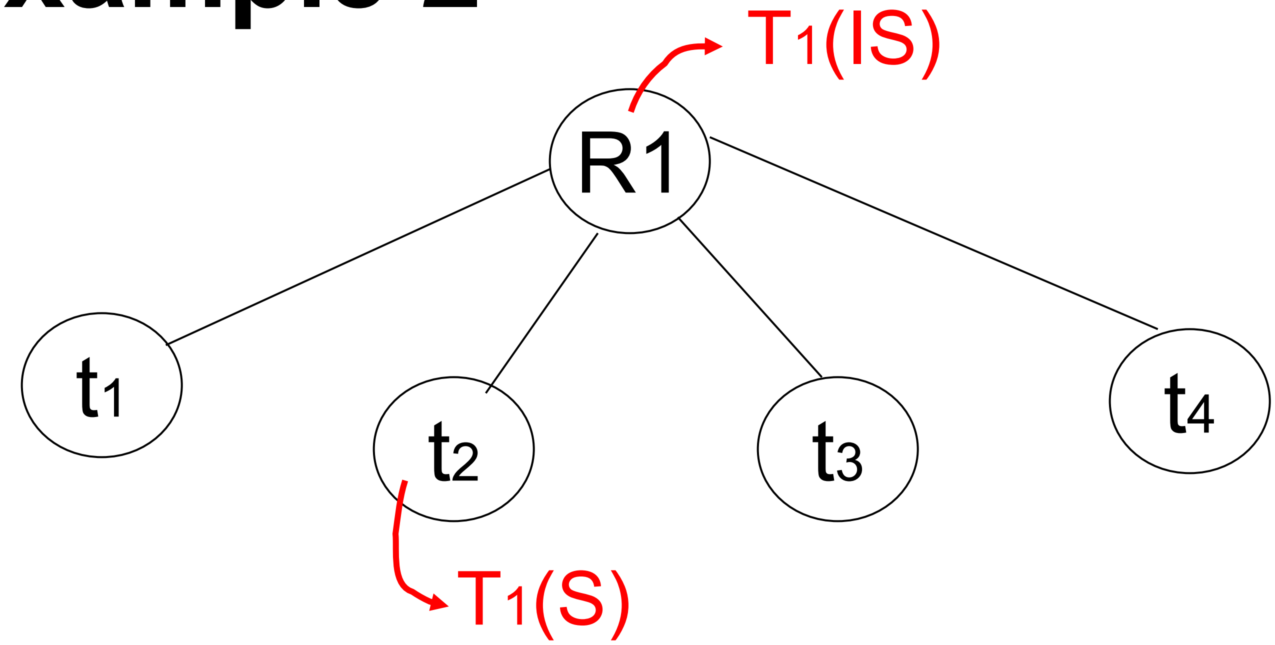
Example



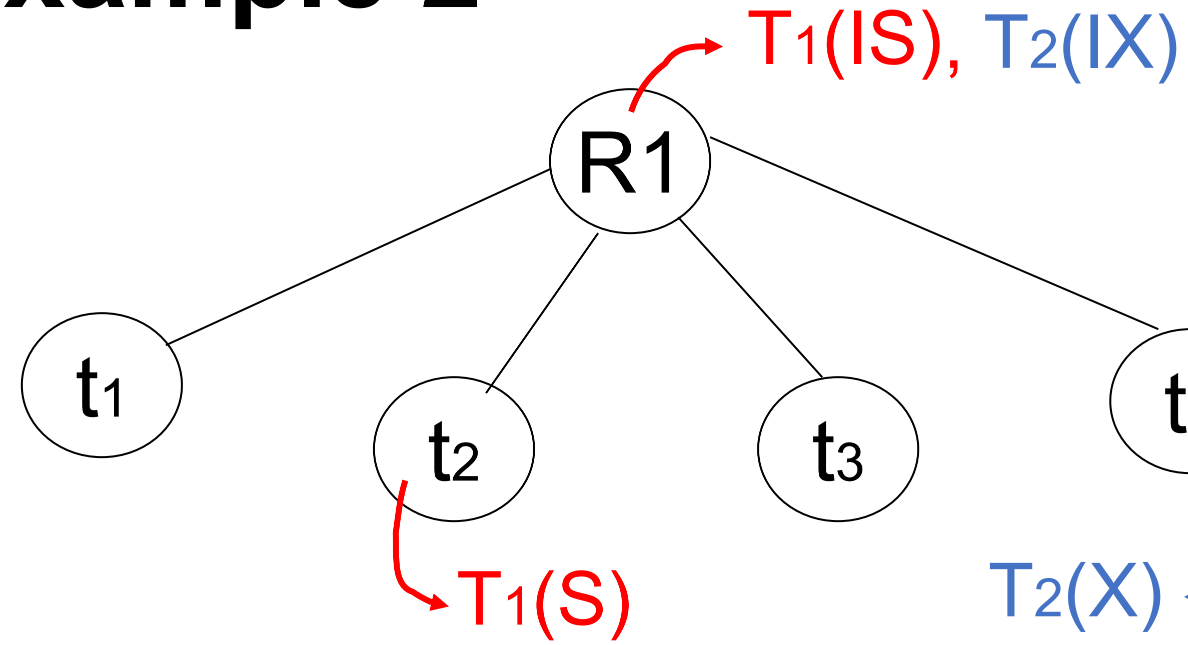
Example



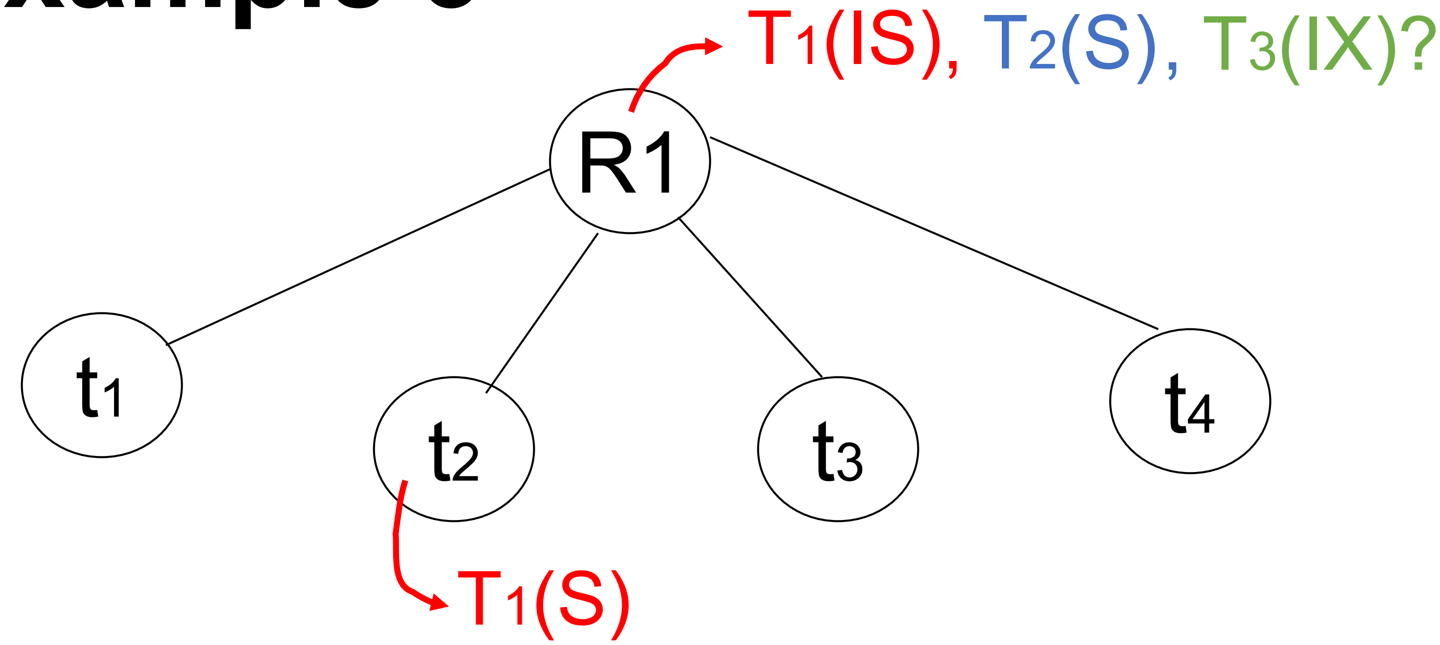
Example 2



Example 2



Example 3



T_1(IS), T_2(S), T_3(IX)?

T_1(S)

Multiple Granularity Locks

compat		Requestor				
		IS	IX	S	SIX	X
	IS					
Holder	IX					
	S					
	SIX					
	X					

Multiple Granularity Locks

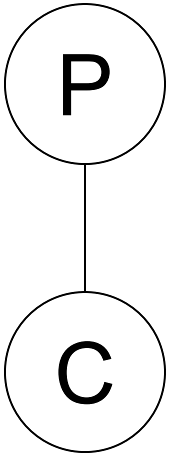
compat Requestor

Holder

compat	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

Rules Within A Transaction

Parent locked in	Child can be locked by same transaction in
IS	IS, S
IX	IS, S, IX, X, SIX
S	none
SIX	X, IX, SIX
X	none

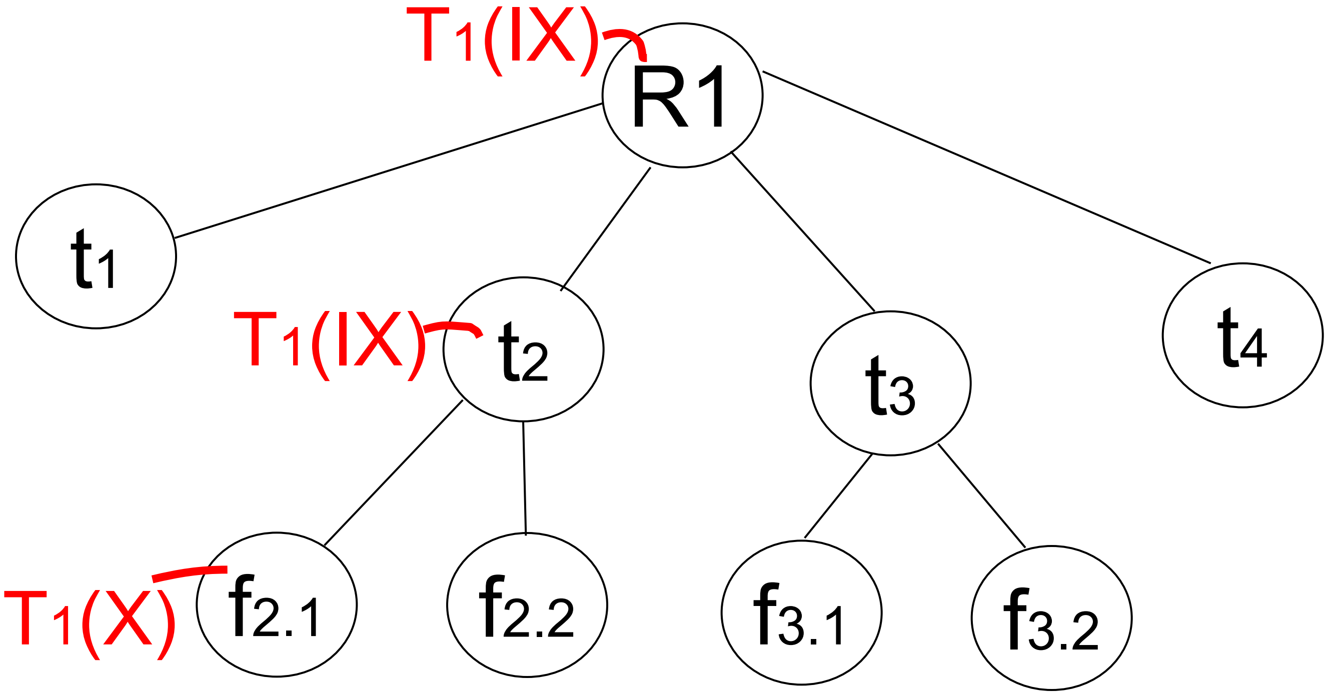


Multi-Granularity 2PL Rules

1. Follow multi-granularity compat function
2. Lock root of tree first, any mode
3. Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
4. Node Q can be locked by Ti in X, SIX, IX only if parent(Q) locked by Ti in IX, SIX
5. Ti is two-phase
6. Ti can unlock node Q only if none of Q’s children are locked by Ti

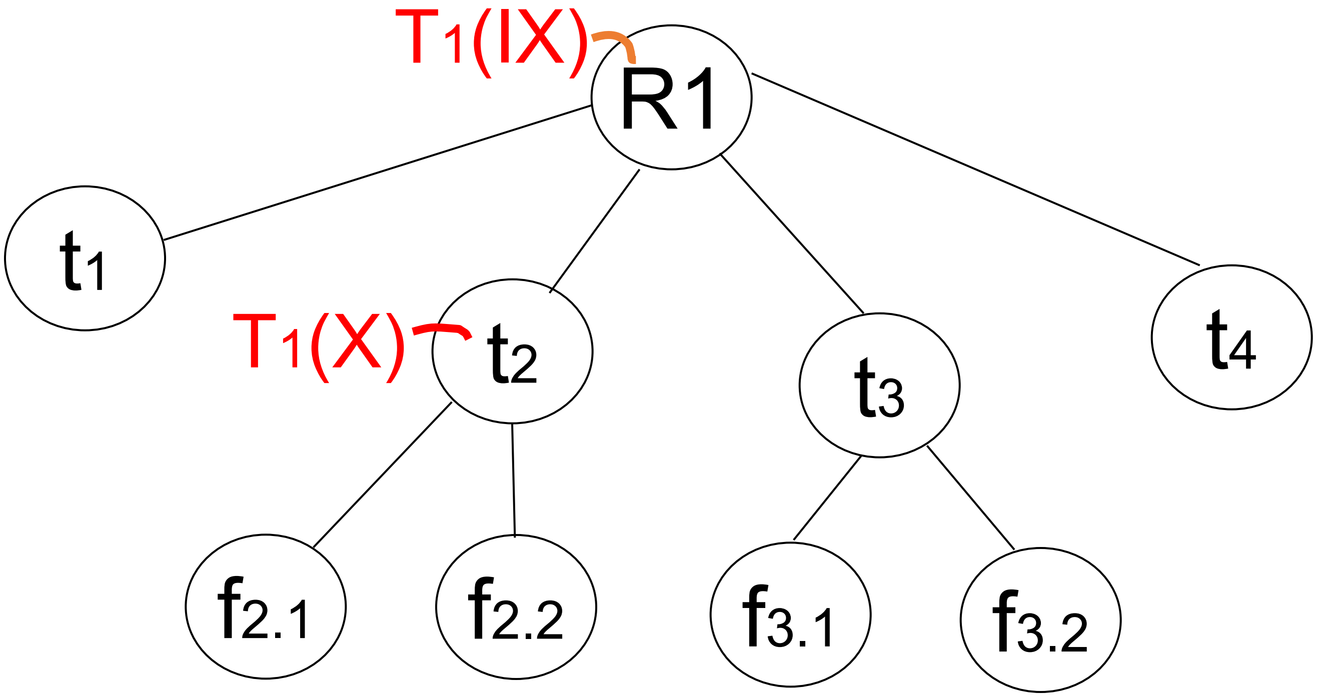
Exercise:

Can T_2 access object f2.2 in X mode? What locks will T_2 get?



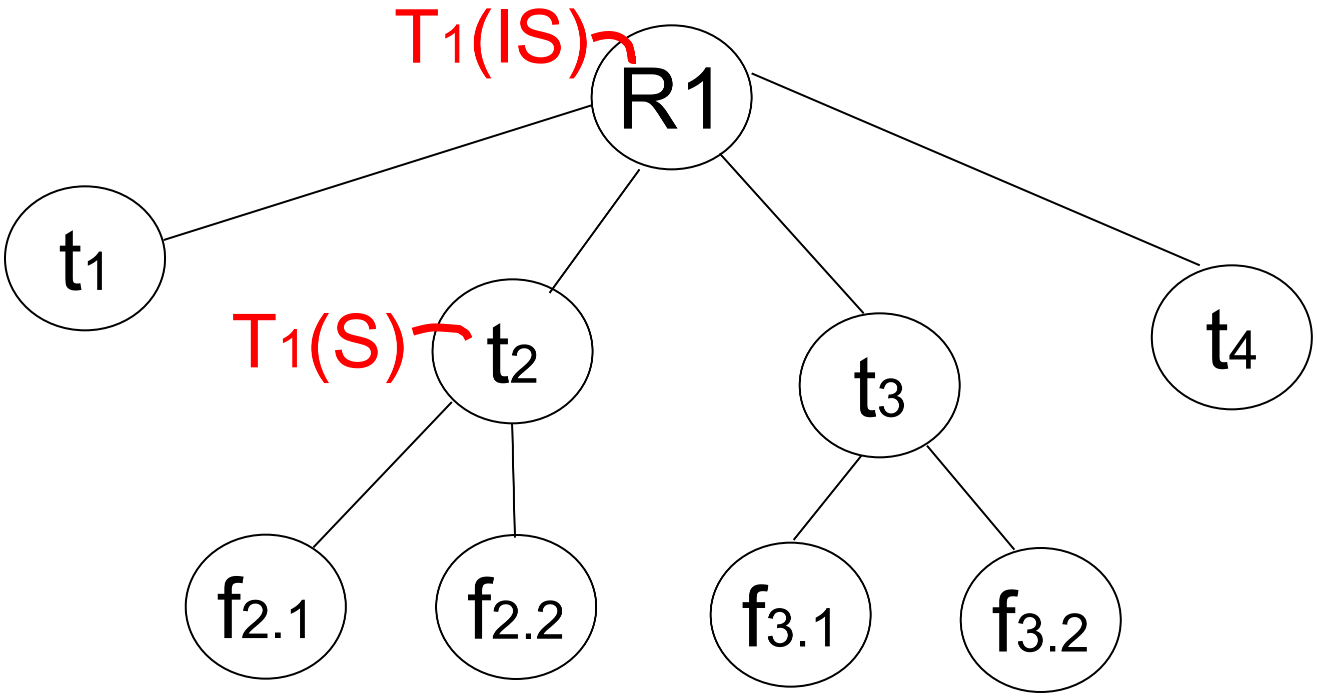
Exercise:

Can T_2 access object f2.2 in X mode? What locks will T_2 get?



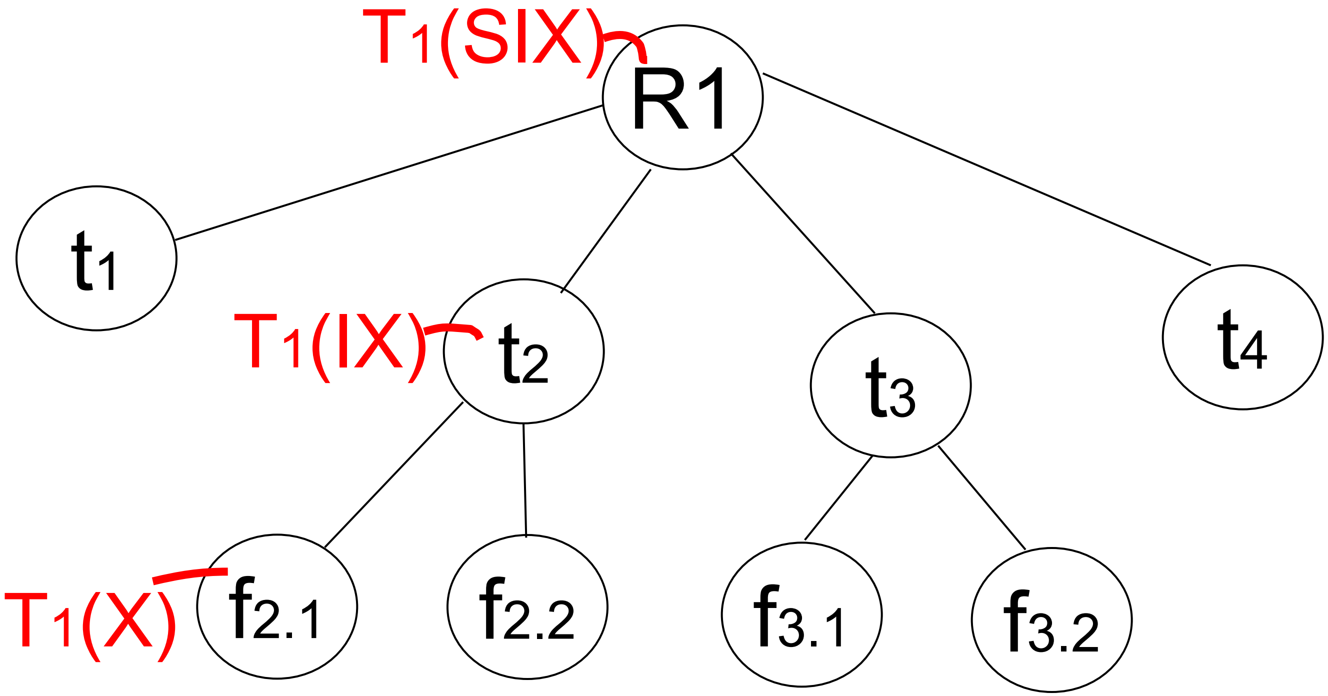
Exercise:

Can T_2 access object f3.1 in X mode? What locks will T_2 get?



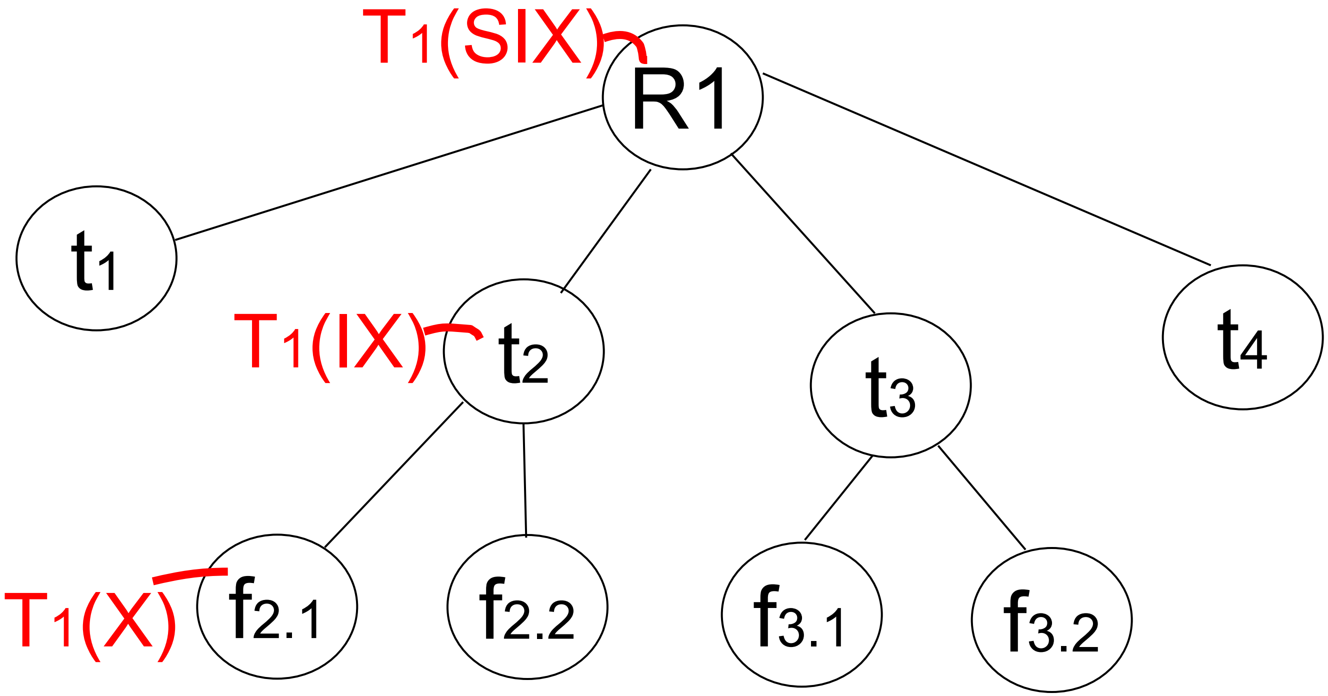
Exercise:

Can T_2 access object f2.2 in S mode? What locks will T_2 get?

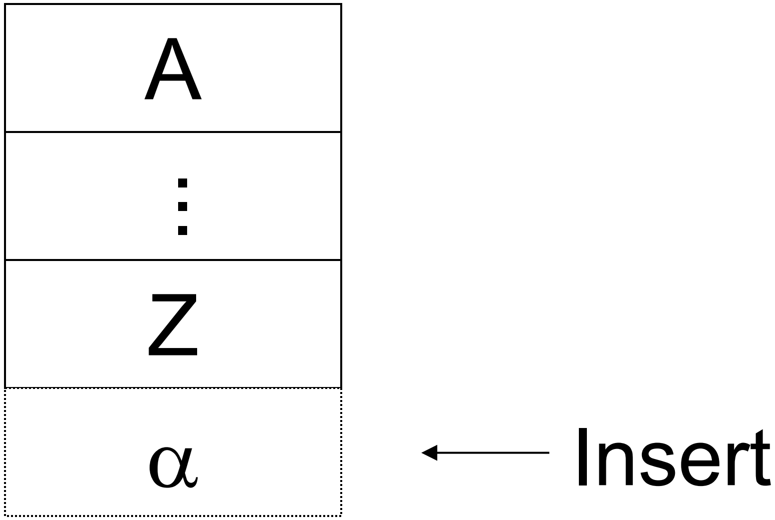


Exercise:

Can T_2 access object f2.2 in X mode? What locks will T_2 get?



Insert + Delete Operations



Changes to Locking Rules:

1. Get exclusive lock on A before deleting A
2. At insert A operation by Ti, Ti is given exclusive lock on A

Still Have Problem: Phantoms

Example: relation R (id, name,…)

constraint: id is unique key

use tuple locking

R	id	Name	….
o1	55	Smith	
o2	75	Jones	

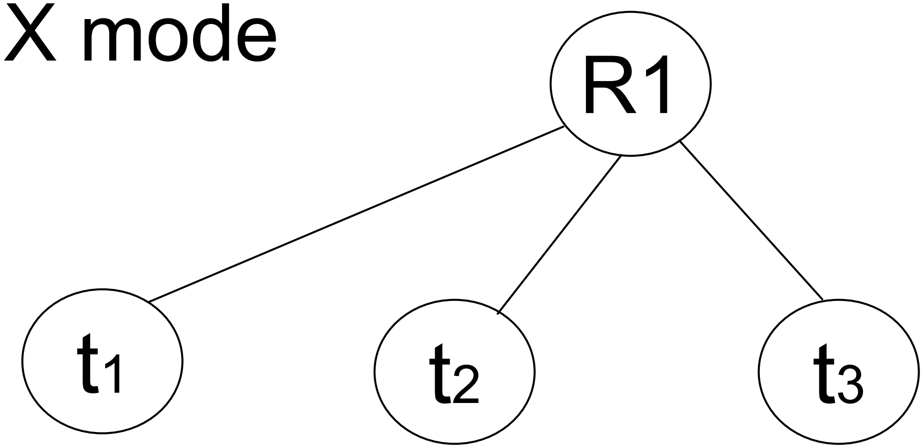
T1: Insert <12,Mary,…> into R
T2: Insert <12,Sam,…> into R

T1	T2
$S_1(o_1)$	$S_2(o_1)$
$S_1(o_2)$	$S_2(o_2)$
Check Constraint	Check Constraint
⋮	⋮
Insert o_3[12,Mary,..]	
	Insert o_4[12,Sam,..]

Solution

Use multiple granularity tree

Before insert of node N,
lock parent(N) in X mode

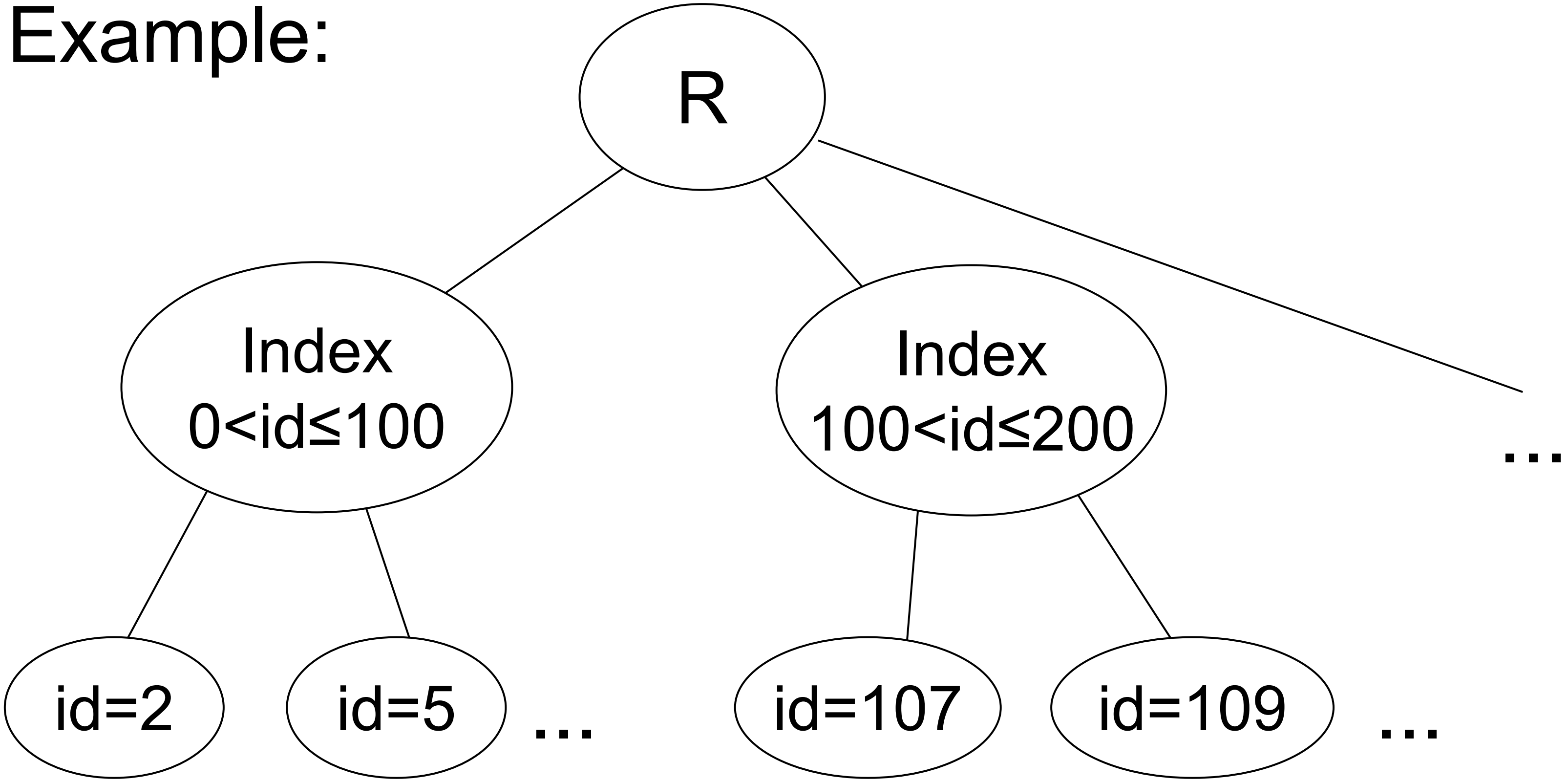


Back to Example

T_1: Insert<12,Mary>	T_2: Insert<12,Sam>
T_1	T_2
$X_1(R)$	
	$X_2(R)$ ← delayed
Check constraint	
Insert<12,Mary>	
$U_1(R)$	
	$X_2(R)$
	Check constraint
	Oops! e# = 12 already in R!

Instead of Using R, Can Use Index Nodes for Ranges

Example:



Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking

- » Shared and exclusive locks
- » Lock tables and multi-level locking

Optimistic concurrency with validation

Concurrency control + recovery

Validation Approach

Transactions have 3 phases:

1. Read
   - » Read all DB values needed
   - » Write to temporary storage
   - » No locking
2. Validate
   - » Check whether schedule so far is serializable
3. Write
   - » If validate OK, write to DB

Key Idea

Make validation atomic

If T_1, T_2, T_3, … is the validation order, then resulting schedule will be conflict equivalent to $S_s = T_1, T_2, T_3, …$

Implementing Validation

System keeps track of two sets:

FIN = transactions that have finished phase 3 (write phase) and are all done

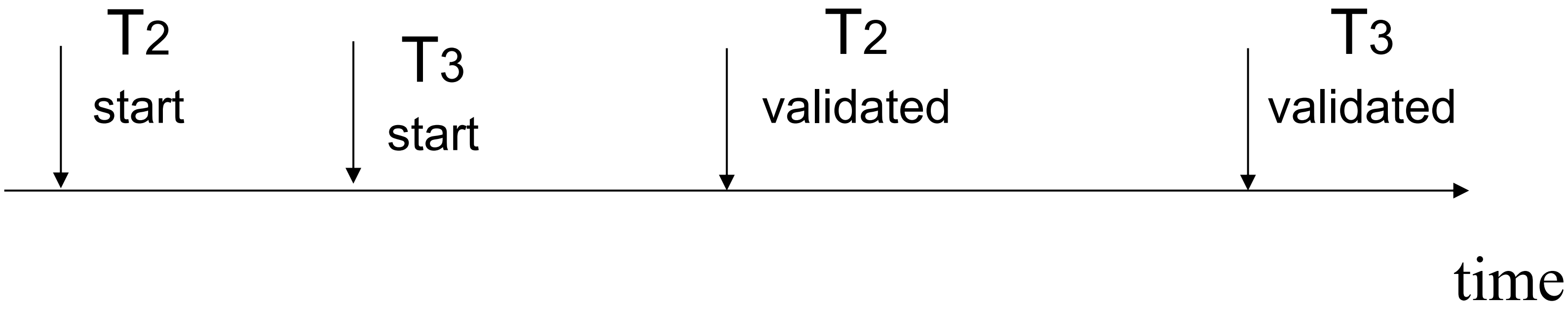
VAL = transactions that have successfully finished phase 2 (validation)

Example That Validation Must Prevent:

RS(T2)={B}   RS(T3)={A,B} ≠ ∅

∩

WS(T2)={B,D}   WS(T3)={C}

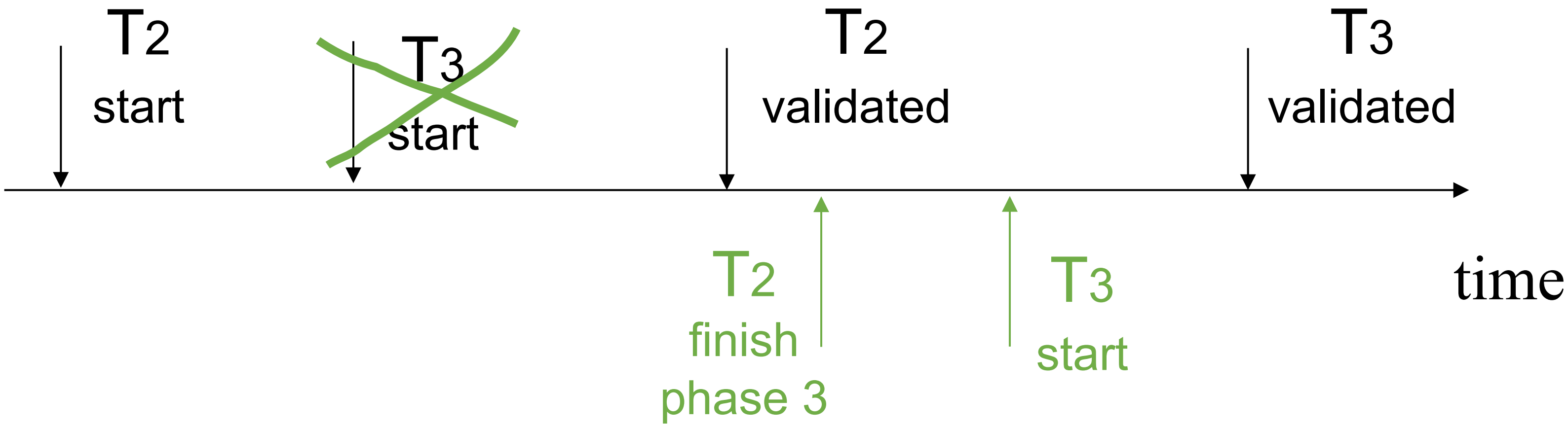


Example That Validation Must ~~Prevent~~ Allow:

RS(T2)={B}   RS(T3)={A,B} ≠ ∅

∩

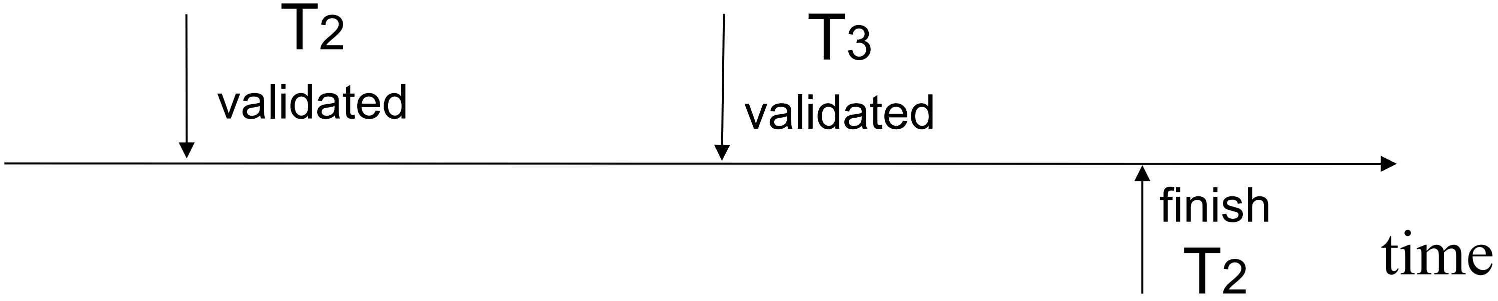
WS(T2)={B,D}   WS(T3)={C}



Another Thing Validation Must Prevent:

$RS(T_2)=\{A\}$ $RS(T_3)=\{A,B\}$

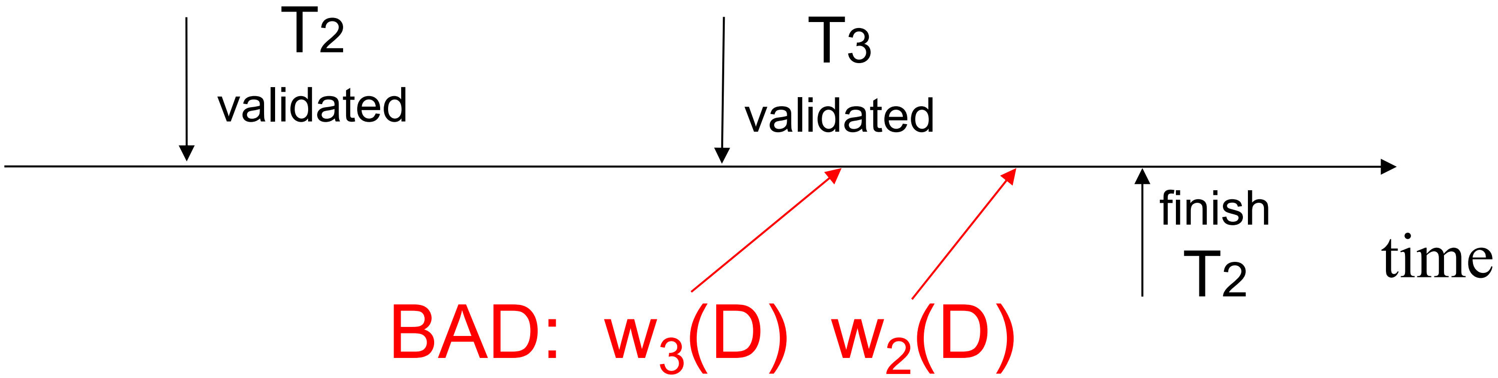
$WS(T_2)=\{D,E\}$ $WS(T_3)=\{C,D\}$



Another Thing Validation Must Prevent:

RS(T2)={A} RS(T3)={A,B}

WS(T2)={D,E} WS(T3)={C,D}

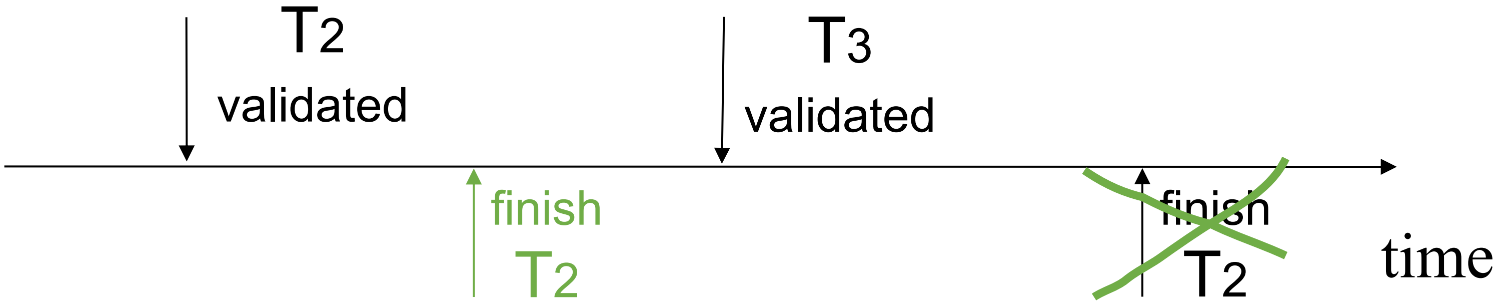


BAD: $w_3(D)$ $w_2(D)$

Another Thing Validation Must ~~Prevent~~ Allow:

$RS(T_2)=\{A\}$ $RS(T_3)=\{A,B\}$

$WS(T_2)=\{D,E\}$ $WS(T_3)=\{C,D\}$



T2 validated

finish T2

T3 validated

~~finish T2~~

time

Validation Rules for Tj:

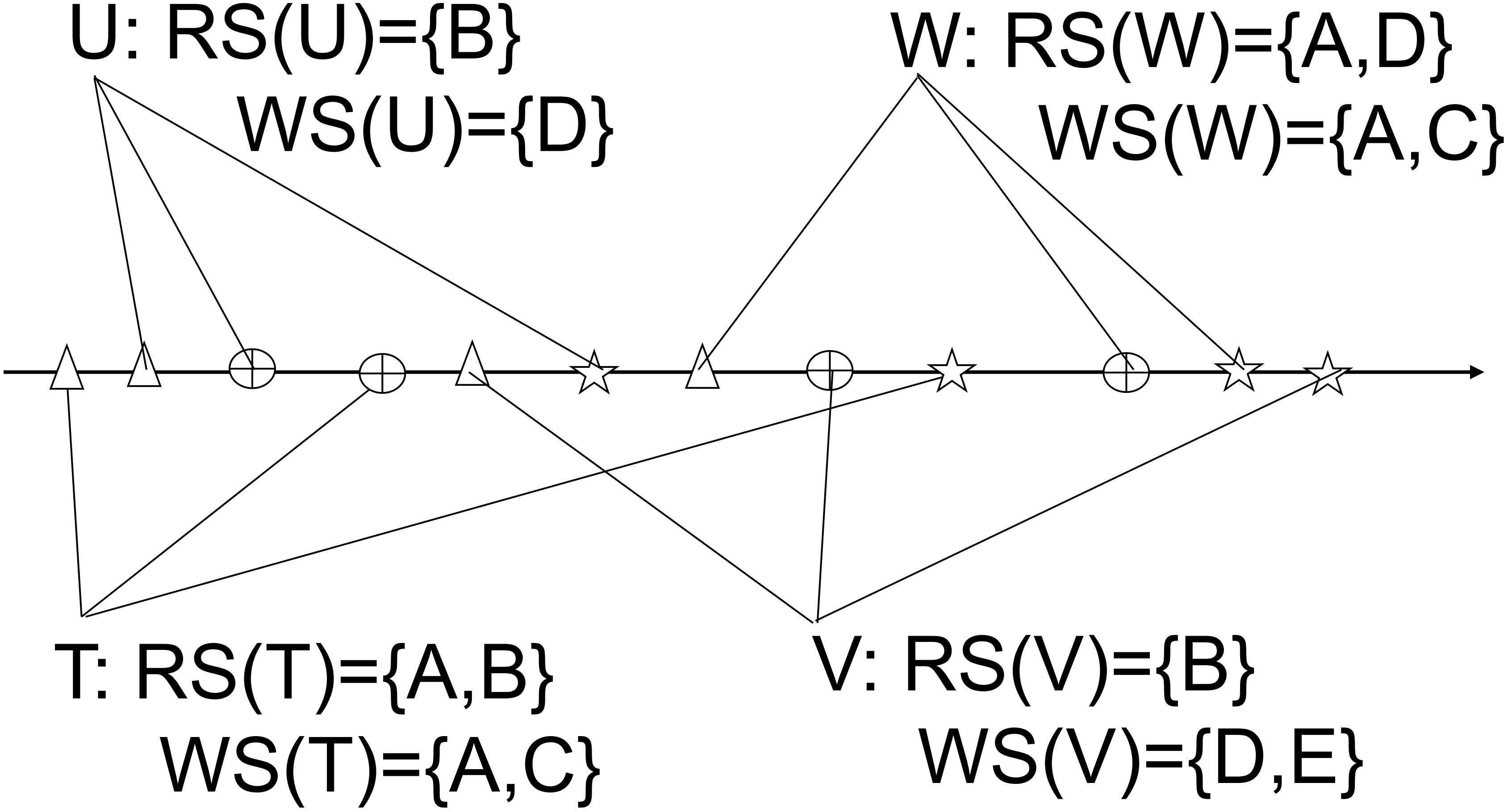
```
when Tj starts phase 1:
    $ignore(Tj) \leftarrow FIN$

at Tj Validation:
    if Check(Tj) then
        $VAL \leftarrow VAL \cup \{Tj\}$
        do write phase
        $FIN \leftarrow FIN \cup \{Tj\}$
```

Check(Tj)

```
for Ti $\in$ VAL – ignore(Tj) do
    if (WS(Ti) $\cap$ RS(Tj) $\neq \emptyset$ or
        (Ti $\notin$ FIN and WS(Ti) $\cap$ WS(Tj) $\neq \emptyset$))
            then return false
return true
```

Exercise

△ start
⊕ validate
☆ finish



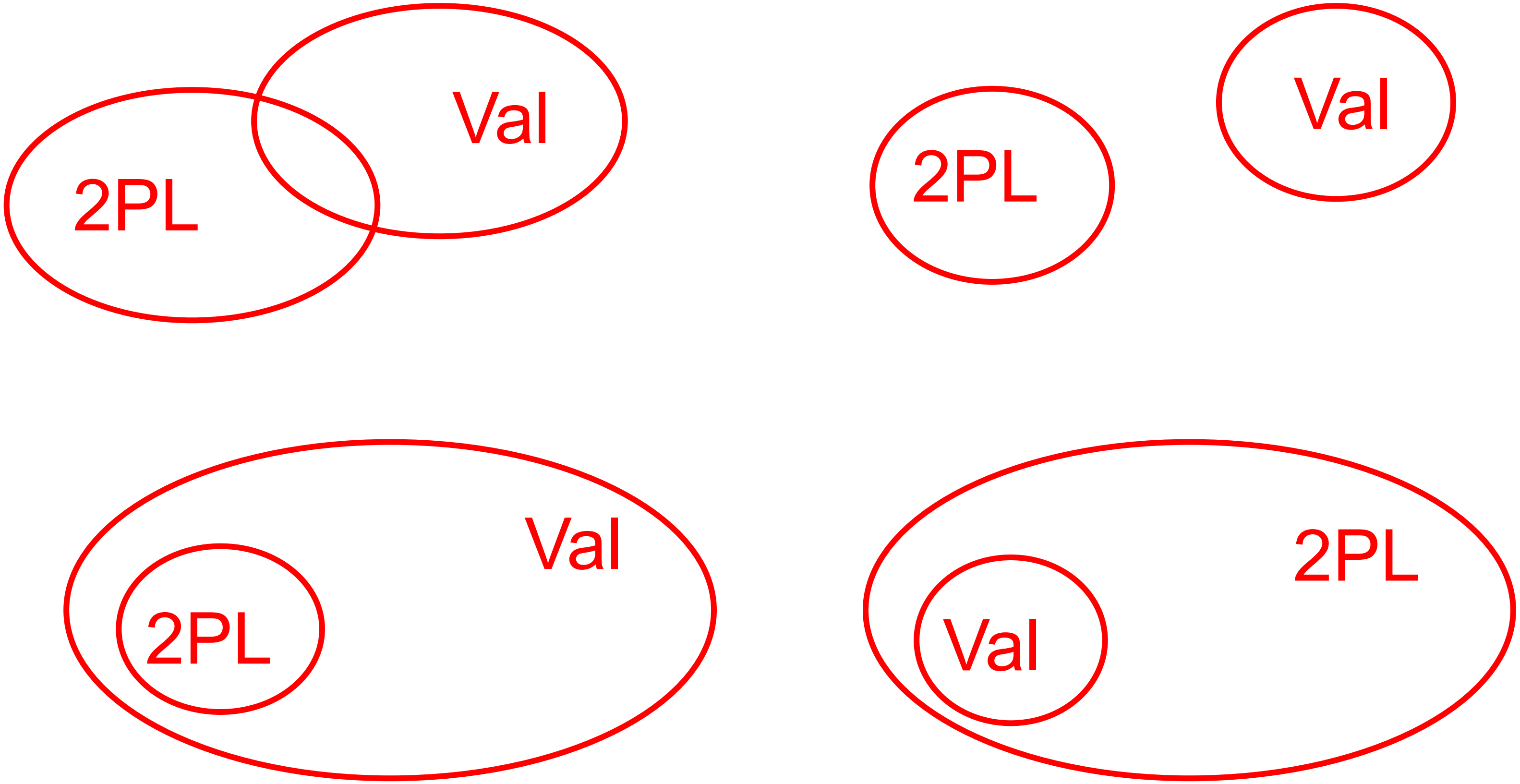
U: RS(U)={B}
WS(U)={D}

W: RS(W)={A,D}
WS(W)={A,C}

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

V: RS(V)={B}
WS(V)={D,E}

Is Validation = 2PL?



S: $w_2(y)$ $w_1(x)$ $w_2(x)$

Achievable with 2PL?

Achievable with validation?

S: $w_2(y)\ w_1(x)\ w_2(x)$

S can be achieved with 2PL:
$l_2(y)\ w_2(y)\ l_1(x)\ w_1(x)\ u_1(x)\ l_2(x)\ w_2(x)\ u_2(x)\ u_2(y)$

S cannot be achieved by validation:
The validation point of T_2, val_2, must occur before $w_2(y)$ since transactions do not write to the database until after validation. Because of the conflict on x, $val_1 < val_2$, so we must have something like:

S: $val_1\ \ val_2\ \ w_2(y)\ \ w_1(x)\ \ w_2(x)$

With the validation protocol, the writes of T_2 should not start until T_1 is all done with writes, which is not the case.

Validation Subset of 2PL?

Possible proof (Check!):

- » Let S be validation schedule
- » For each T in S insert lock/unlocks, get S':
  - At T start: request read locks for all of RS(T)
  - At T validation: request write locks for WS(T); release read locks for read-only objects
  - At T end: release all write locks
- » Clearly transactions well-formed and 2PL
- » Must show S' is legal (next slide)

Validation Subset of 2PL?

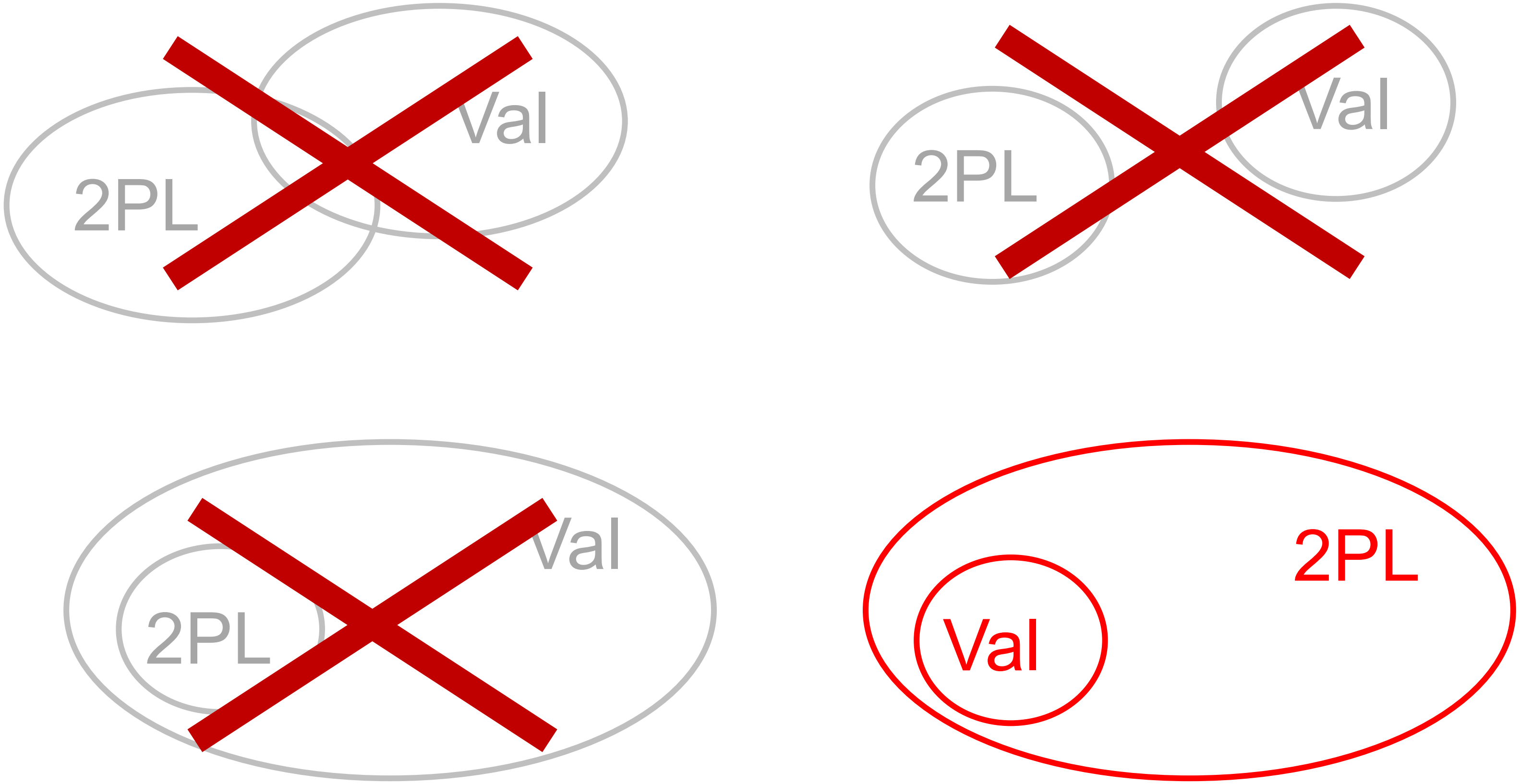
Say S' not legal (due to w-r conflict):
S': ... l1(x)   w2(x) r1(x)   val1 u1(x) ...

- At val1: T2 not in Ignore(T1); T2 in VAL
- T1 does not validate: WS(T2) $\cap$ RS(T1) $\neq \emptyset$
- contradiction!

Say S' not legal (due to w-w conflict):
S': ... val1 l1(x)   w2(x) w1(x)   u1(x) ...

- Say T2 validates first (proof similar if T1 validates first)
- At val1: T2 not in Ignore(T1); T2 in VAL
- T1 does not validate:
  T2 $\notin$ FIN AND WS(T1) $\cap$ WS(T2) $\neq \emptyset$)
- contradiction!

Is Validation = 2PL?



When to Use Validation?

Validation performs better than locking when:

- » Conflicts are rare
- » System resources are plentiful
- » Have tight latency constraints

Summary

Have studied several concurrency control mechanisms used in practice

- » 2 PL
- » Multiple granularity
- » Validation

Next: how does concurrency control interact with failure recovery?

Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking

- Shared and exclusive locks
- Lock tables and multi-level locking

Optimistic concurrency with validation

Concurrency control + recovery