

# Concurrency Control

Instructor: Matei Zaharia

# 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

Beyond serializability

# Recap: Schedules

Sequence of actions from transactions, e.g.:

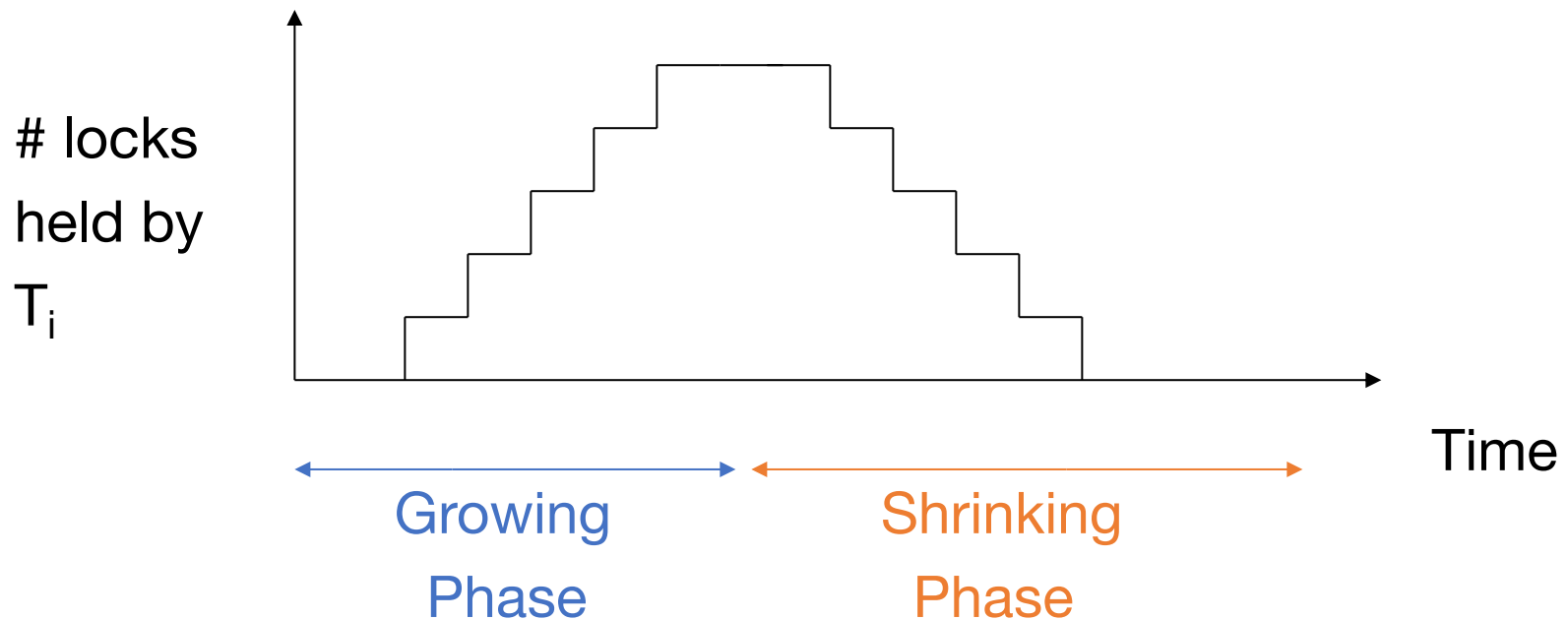
$$r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$$

**Serializable schedule:** result is the same as some serial schedule

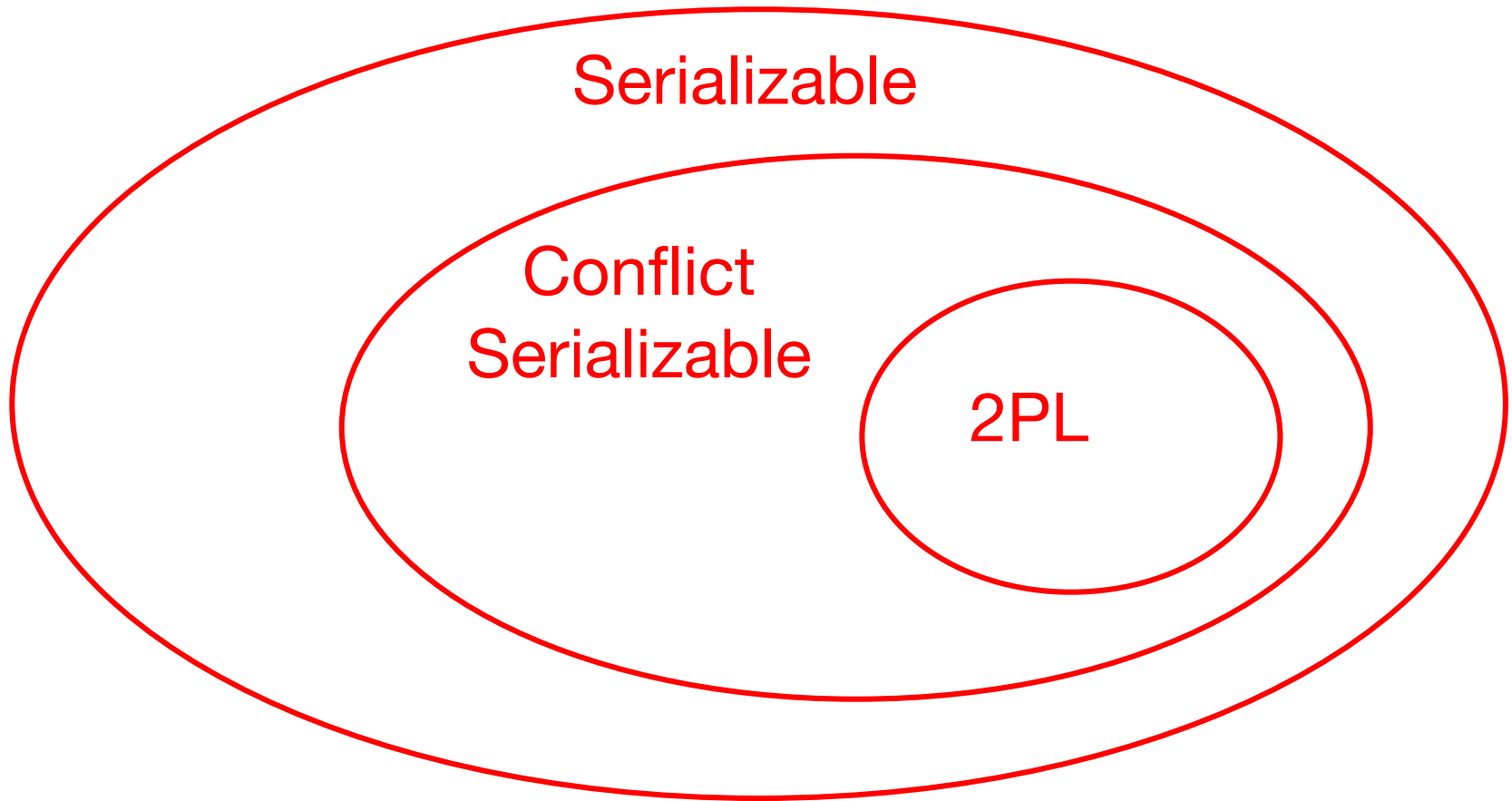
# Recap: 2-Phase Locking (2PL)

Schedules that satisfy three rules:

1. Well-formed transactions (lock items before use)
2. Legal scheduler
3. 2-phase locking pattern



# 2PL Schedules are a Subset of Serializable

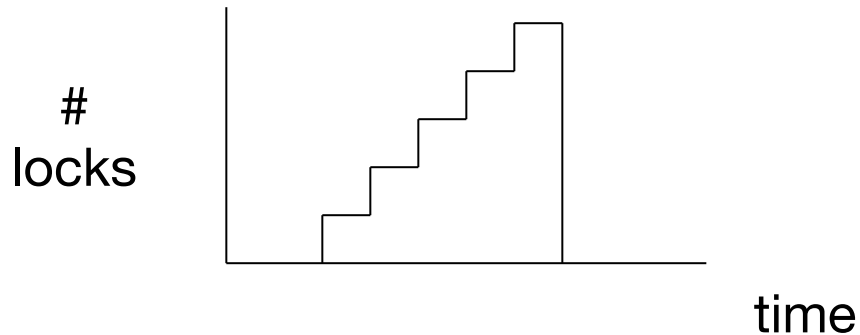


# How Is 2PL Implemented In Practice?

Every system is different, but we'll show one simplified way

# Sample Locking System

1. Don't ask transactions to request/release locks: just get a lock for each action they do
2. Hold all locks until a transaction commits



# Sample Locking System

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

» E.g., hash table

Also need ways to block transactions until locks are available, and to find deadlocks



# Optimizing Performance

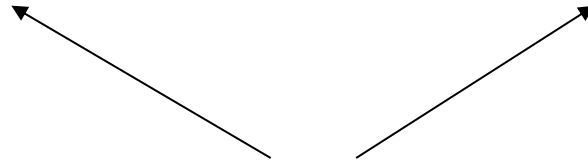
Beyond the base 2PL protocol, many ways to improve performance & concurrency:

- » Shared locks
- » Multiple granularity
- » Inserts, deletes and phantoms
- » Other types of C.C. mechanisms

# Shared Locks

So far:

$S = \dots l_1(A) \ r_1(A) \ u_1(A) \ \dots \ l_2(A) \ r_2(A) \ u_2(A) \ \dots$

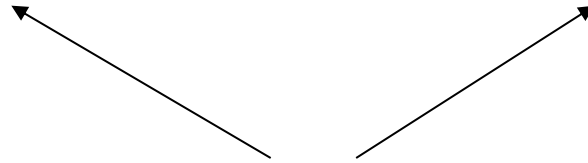


Do not conflict

# Shared Locks

So far:

$S = \dots l_1(A) \ r_1(A) \ u_1(A) \ \dots \ l_2(A) \ r_2(A) \ u_2(A) \ \dots$



Do not conflict

Instead:

$S = \dots l-S_1(A) \ r_1(A) \ l-S_2(A) \ r_2(A) \ \dots \ u_1(A) \ u_2(A)$

# Multiple Lock Modes

Lock actions

$l-m_i(A)$ : lock  $A$  in mode  $m$  ( $m$  is  $S$  or  $X$ )

$u-m_i(A)$ : unlock mode  $m$  ( $m$  is  $S$  or  $X$ )

Shorthand:

$u_i(A)$ : unlock whatever modes  $T_i$  has locked  $A$

# Rule 1: Well-Formed Transactions

$$T_i = \dots I-S_1(A) \dots r_1(A) \dots u_1(A) \dots$$
$$T_i = \dots I-X_1(A) \dots w_1(A) \dots u_1(A) \dots$$

Transactions must acquire the right lock type for their actions (S for read only, X for r/w).

# Rule 1: Well-Formed Transactions

What about transactions that read and write same object?

**Option 1:** Request exclusive lock

$$T_1 = \dots l-X_1(A) \dots r_1(A) \dots w_1(A) \dots u(A) \dots$$

# Rule 1: Well-Formed Transactions

What about transactions that read and write same object?

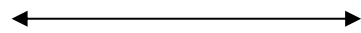
**Option 2:** Upgrade lock to X on write

$T_1 = \dots I-S_1(A) \dots r_1(A) \dots \textcolor{red}{I-X_1(A)} \dots w_1(A) \dots u_1(A) \dots$

(Think of this as replacing S lock with X lock.)

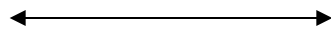
## Rule 2: Legal Scheduler

$S = \dots I-S_i(A) \dots \quad \dots u_i(A) \dots$



no  $I-X_j(A)$

$S = \dots I-X_i(A) \dots \quad \dots u_i(A) \dots$



no  $I-X_j(A)$

no  $I-S_j(A)$



# A Way to Summarize Rule #2

Lock mode compatibility matrix

compat =

		New request	
		S	X
Lock already held in	S	true	false
	X	false	false

# Rule 3: 2PL Transactions

No change except for upgrades: allow upgrades from S to X only in growing phase

# Rules 1,2,3 $\Rightarrow$ Conf. Serializable Schedules for S/X Locks

**Proof:** similar to X locks case

**Detail:**

$l-m_i(A), l-n_j(A)$  do not conflict if  $\text{compat}(m,n)$

$l-m_i(A), u-n_j(A)$  do not conflict if  $\text{compat}(m,n)$

# Lock Modes Beyond S/X

Examples:

(1) increment lock

(2) update lock

(3) hierarchical locks (paper!)

# Increment Locks

Atomic addition action:  $IN_i(A)$

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

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

# Compatibility Matrix

		New request		
compat		S	X	I
Lock already held in	S	T	F	F
	X	F	F	F
	I	F	F	T

# Update Locks

A common deadlock problem with upgrades:

T1	T2
I-S <sub>1</sub> (A)	
I-X <sub>1</sub> (A)	I-S <sub>2</sub> (A)
	I-X <sub>2</sub> (A)

--- Deadlock ---

# Solution

If  $T_i$  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
Lock already held in	S	T	F	
	X	F	F	
	U			

# Compatibility Matrix

compat

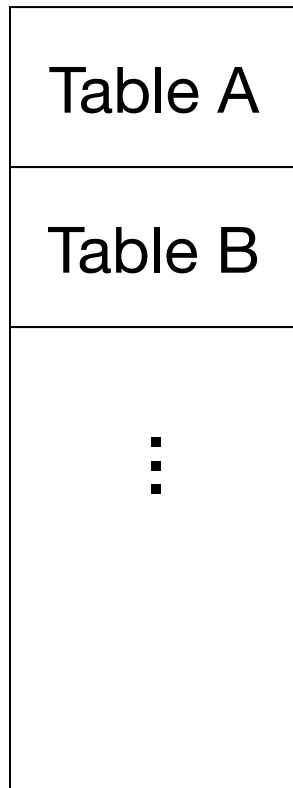
New request

Lock already held in

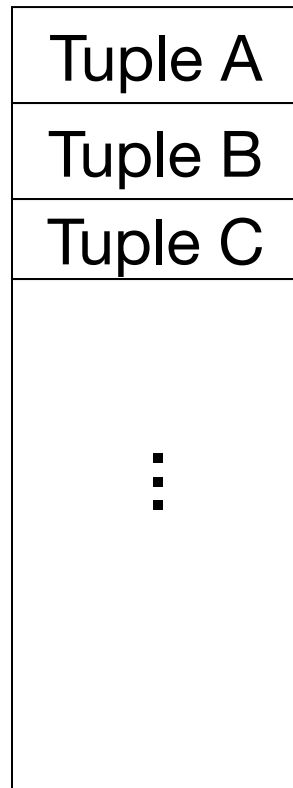
	S	X	U
S	T	F	T
X	F	F	F
U	F	F	F

Note: asymmetric table!

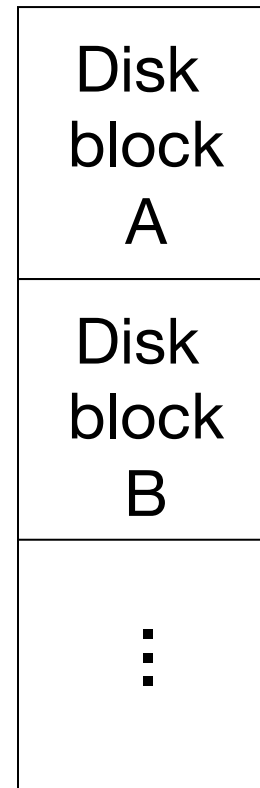
# Which Objects Do We Lock?



DB



DB



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)

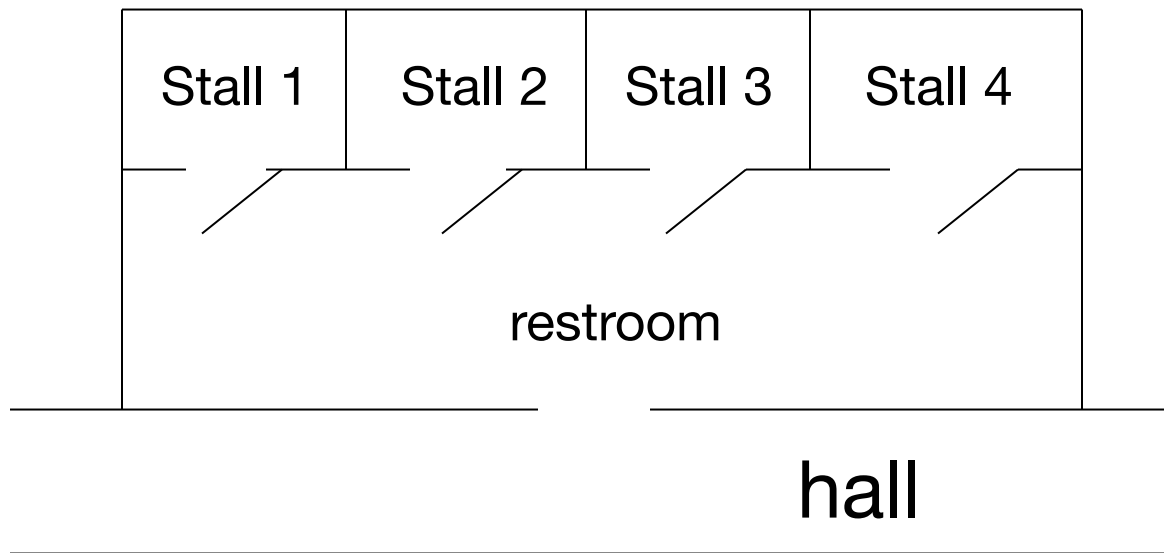
- Need few locks
- Low concurrency

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

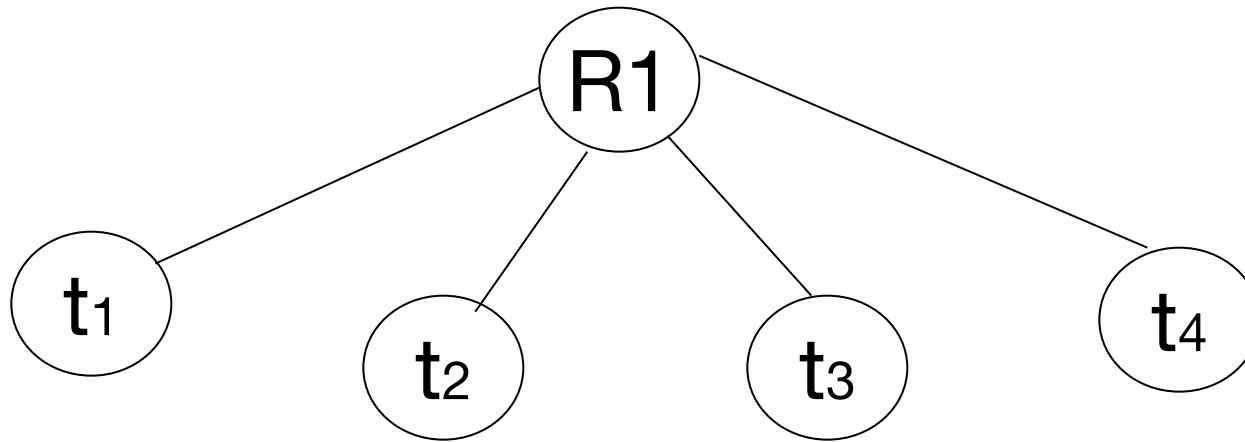
- Need more locks
- More concurrency

# We Can Have It Both Ways!

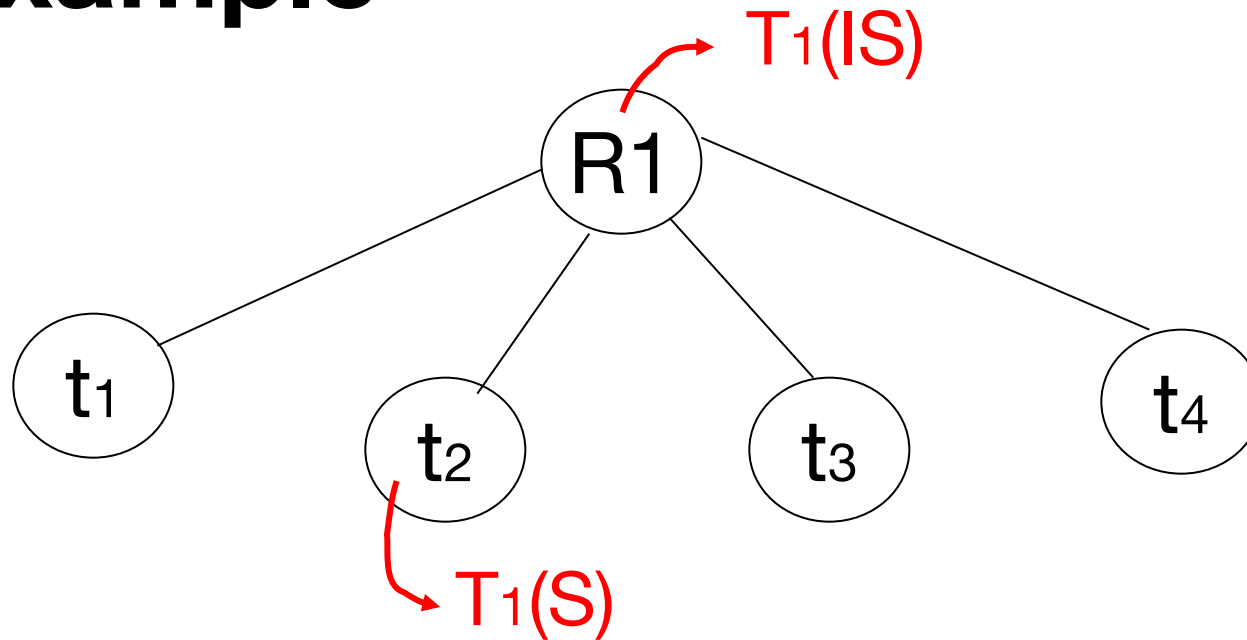
Ask any janitor to give you the solution...



# Example

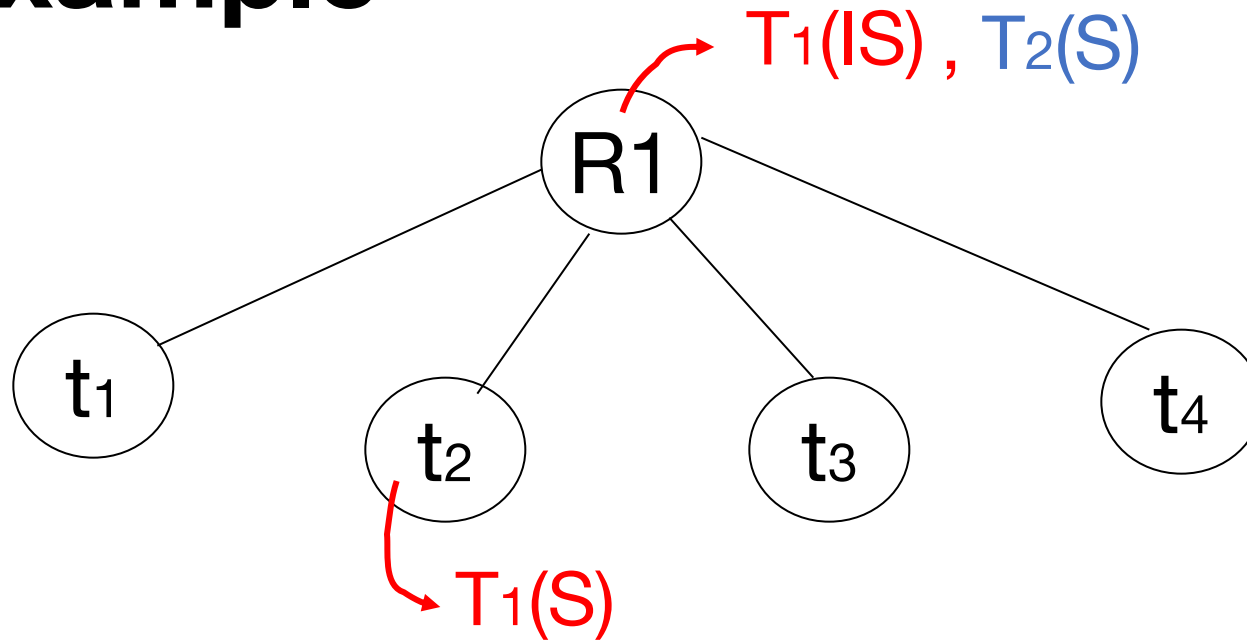


# Example

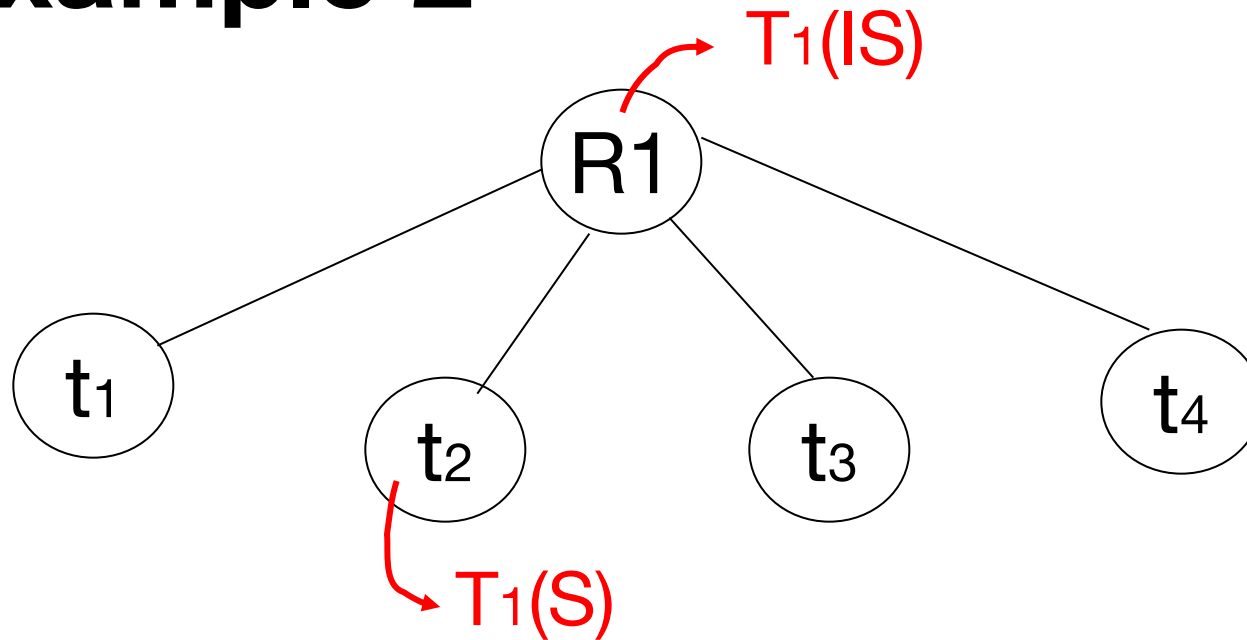




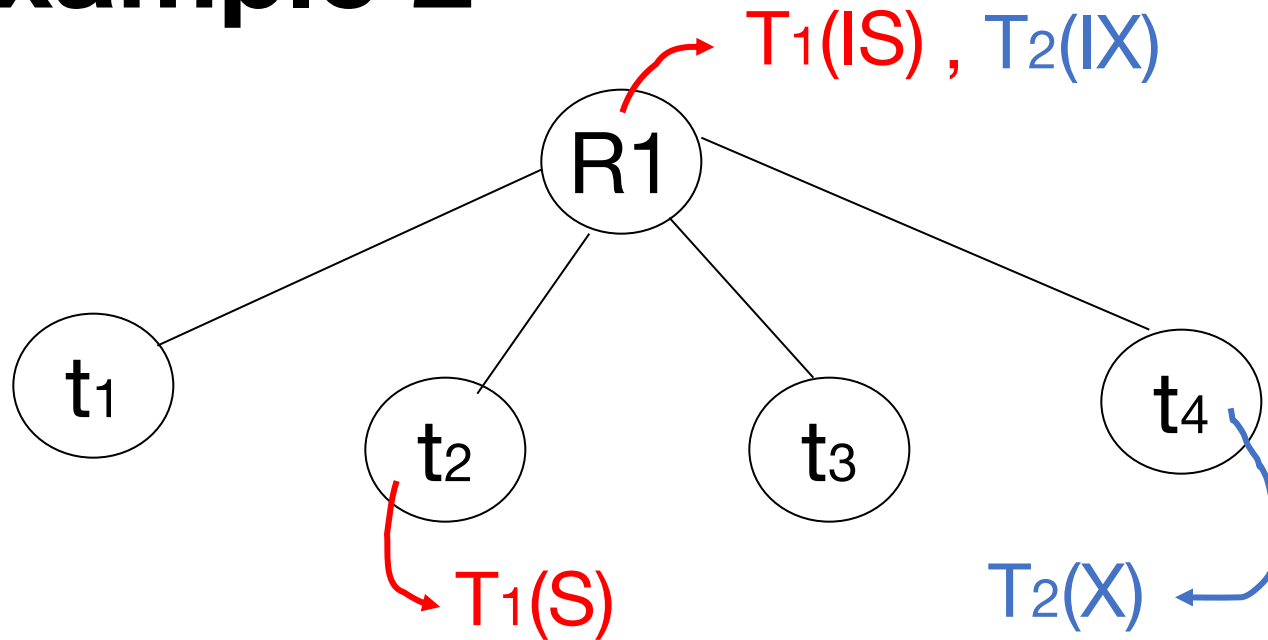
# Example



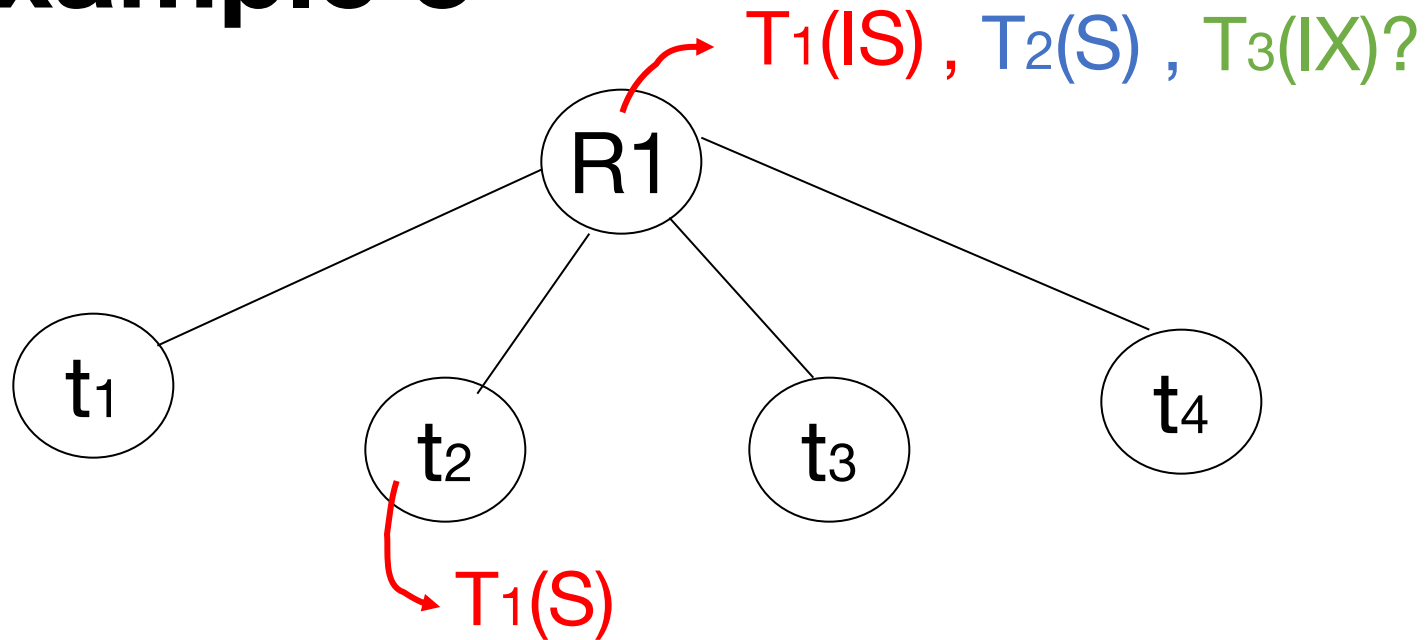
# Example 2



# Example 2



# Example 3



# Multiple Granularity Locks

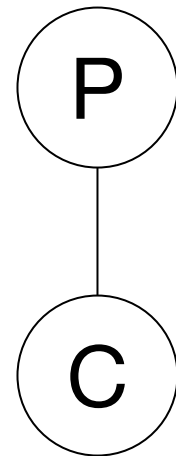
compat

Requester

		IS	IX	S	SIX	X
Holder	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

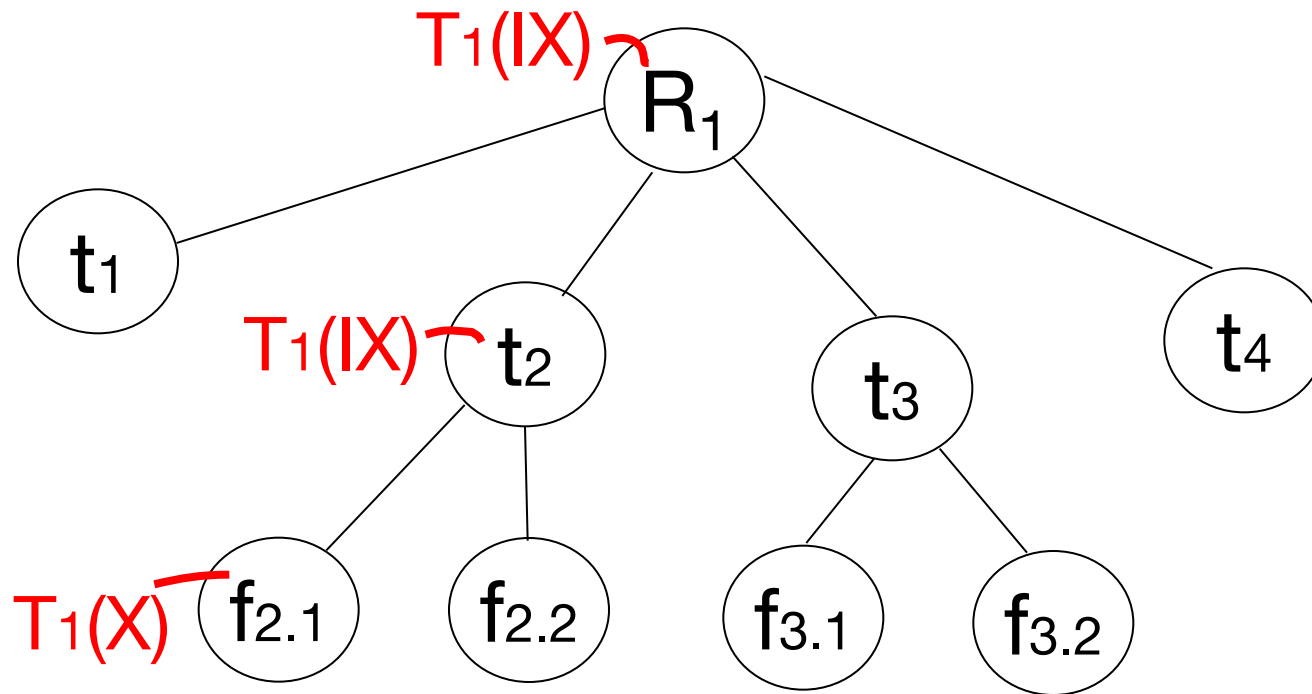


# Multi-Granularity 2PL Rules

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

# Exercise:

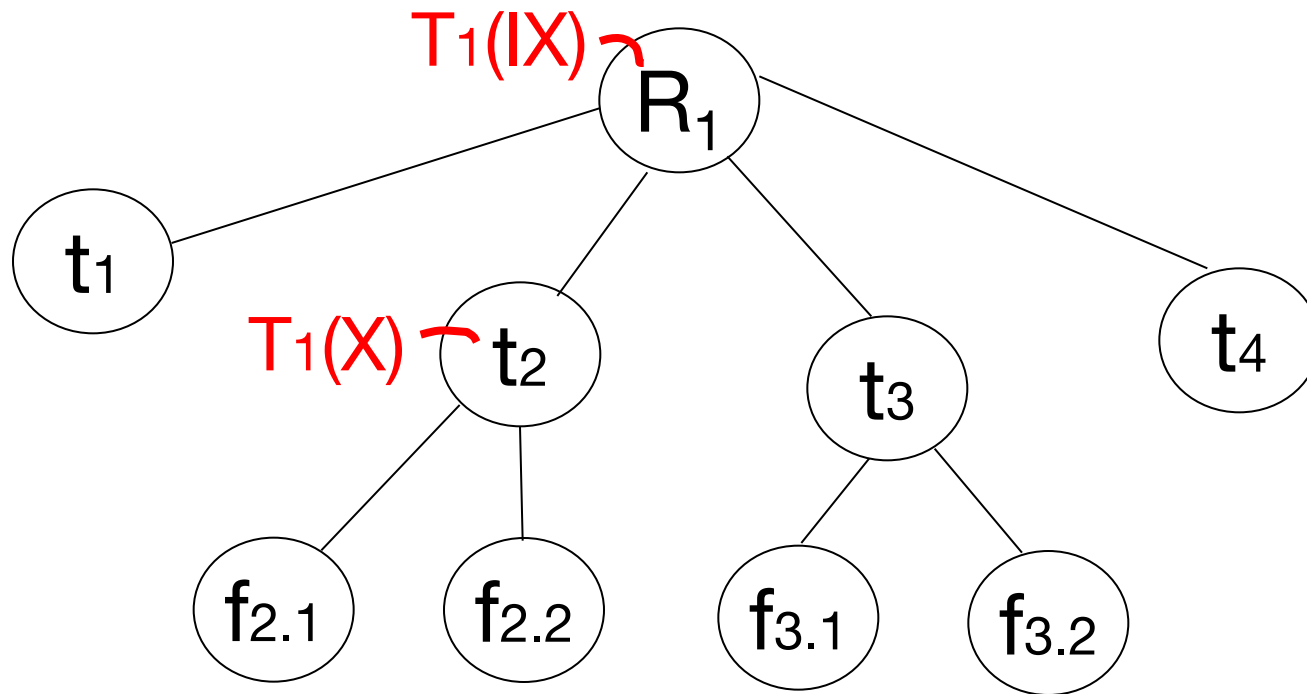
Can  $T_2$  access object  $f_{2.2}$  in X mode? What locks will  $T_2$  get?





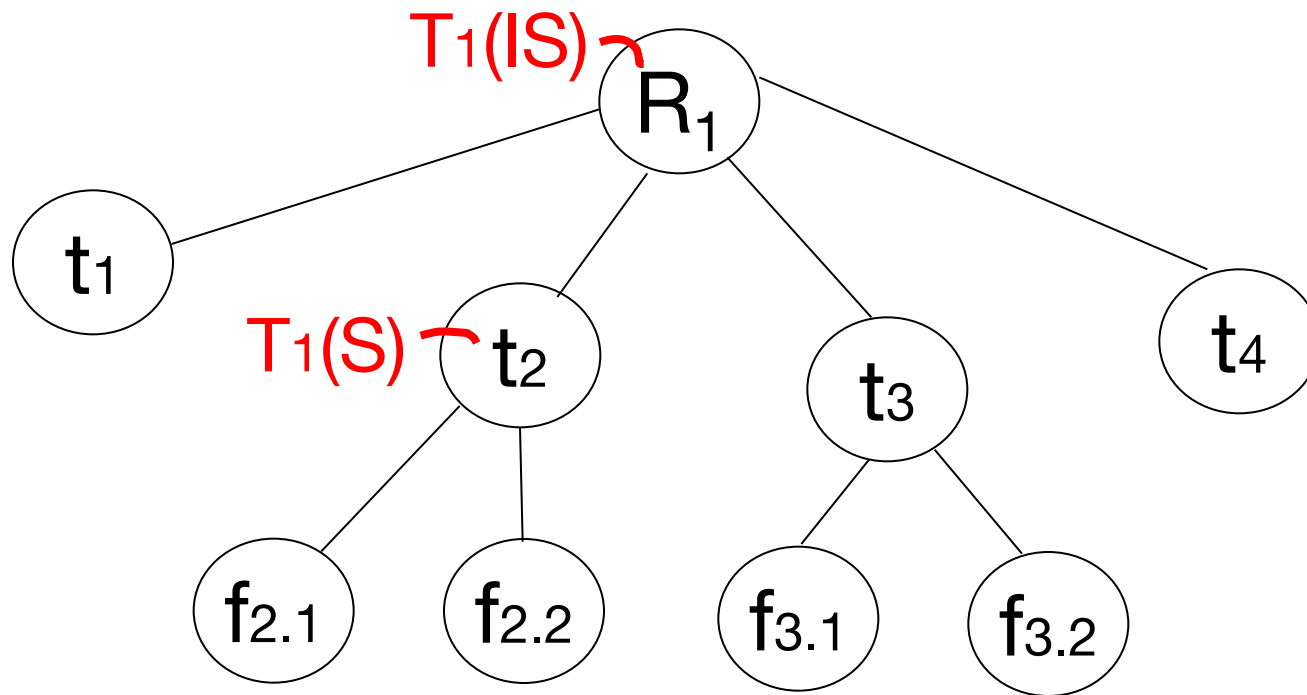
# Exercise:

Can  $T_2$  access object  $f_{2.2}$  in X mode? What locks will  $T_2$  get?



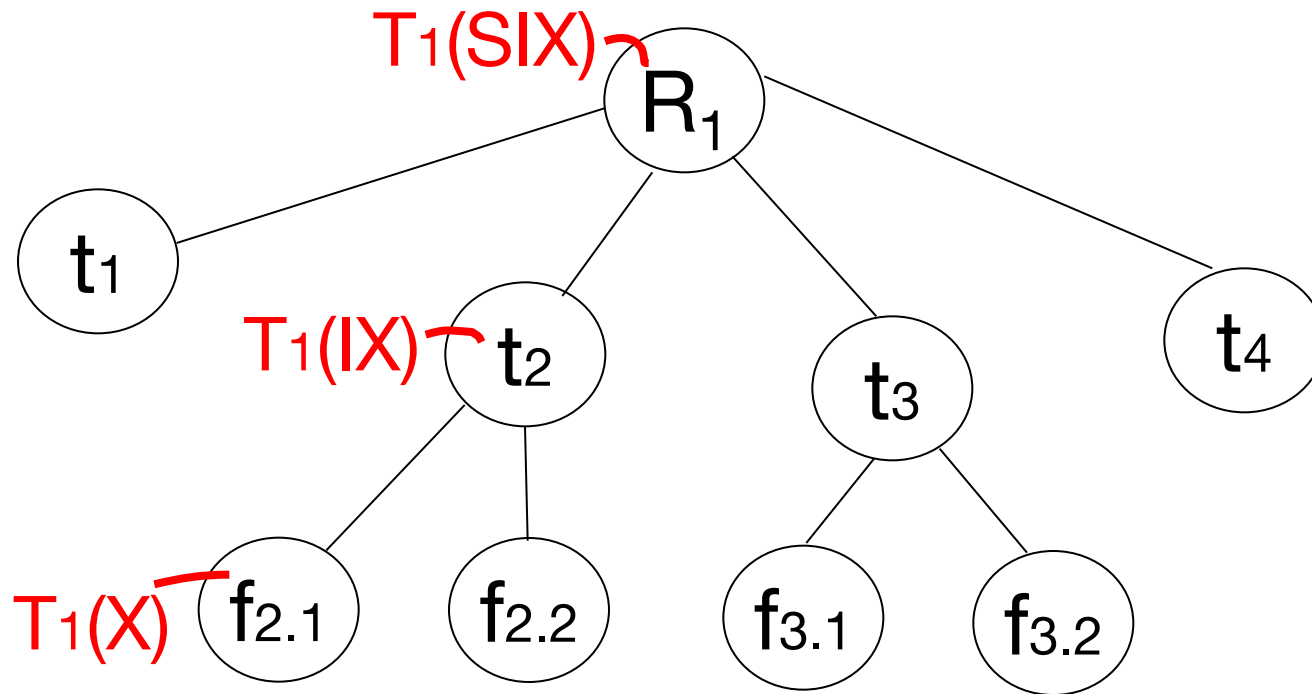
# Exercise:

Can  $T_2$  access object  $f_{3.1}$  in X mode? What locks will  $T_2$  get?



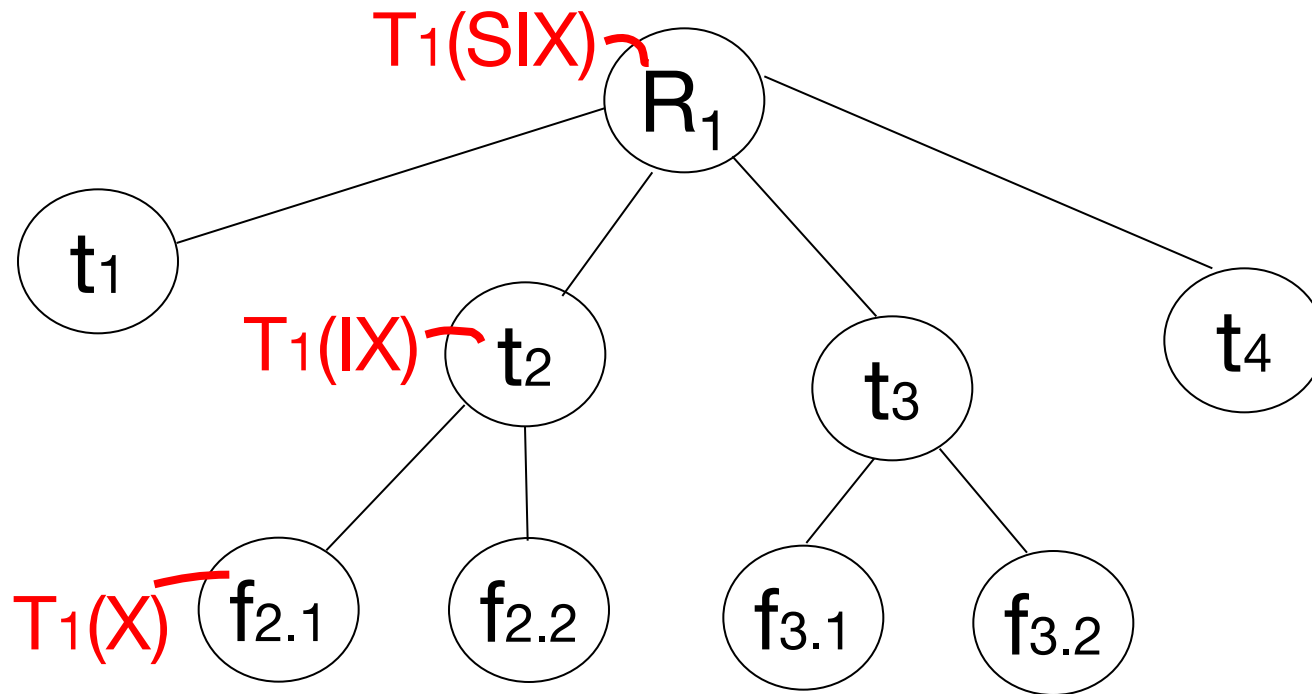
# Exercise:

Can  $T_2$  access object  $f_{2.2}$  in S mode? What locks will  $T_2$  get?

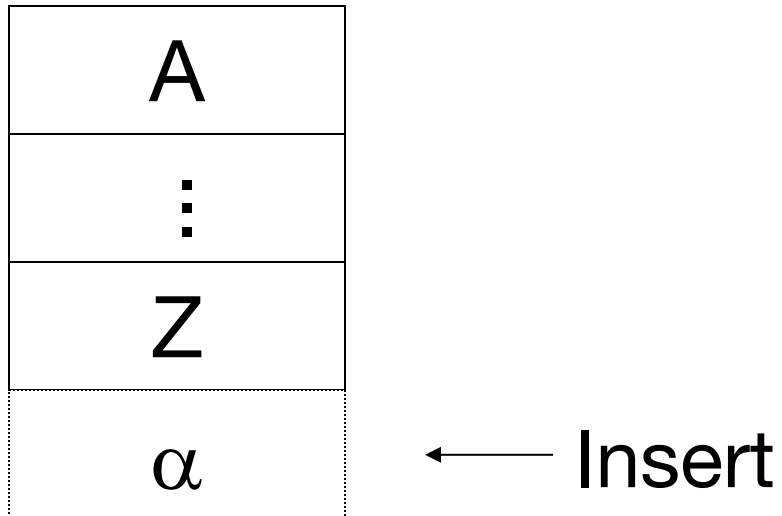


# Exercise:

Can  $T_2$  access object  $f_{2.2}$  in X mode? What locks will  $T_2$  get?



# Insert + Delete Operations



# Changes to Locking Rules:

1. Need exclusive lock on A to delete A
2. When  $T_i$  inserts an object A,  $T_i$  receives an exclusive lock on A

# Still Have Problem: Phantoms

Example: relation R (id, name,...)  
constraint: id is unique key  
use tuple locking

R	id	name	....
$o_1$	55	Smith	
$o_2$	75	Jones	

**$T_1$ : Insert  $\langle 12, \text{Mary}, \dots \rangle$  into R**

**$T_2$ : Insert  $\langle 12, \text{Sam}, \dots \rangle$  into R**

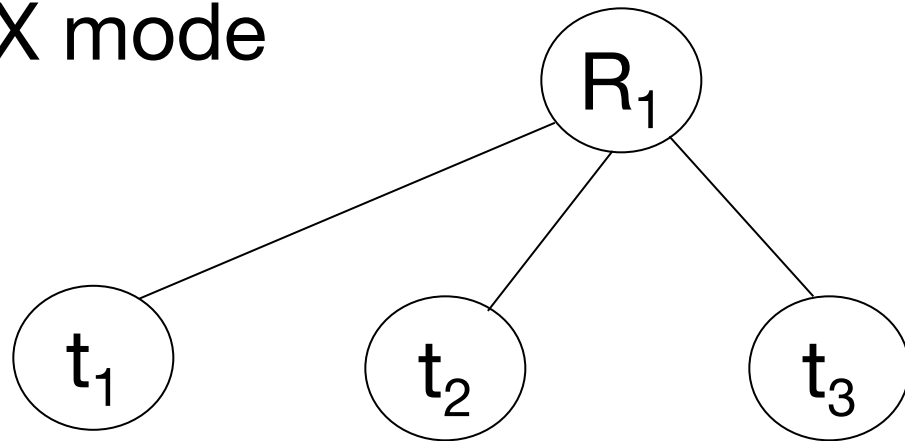
T1	T2
I-S <sub>1</sub> (o <sub>1</sub> )	I-S <sub>2</sub> (o <sub>1</sub> )
I-S <sub>1</sub> (o <sub>2</sub> )	I-S <sub>2</sub> (o <sub>2</sub> )
Check Constraint	Check Constraint
⋮	⋮
Insert o <sub>3</sub> [12, Mary, ...]	Insert o <sub>4</sub> [12, Sam, ...]



# Solution

Use multiple granularity tree

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



# Back to Example

$T_1$ : Insert<12,Mary>

$T_1$

$I-X_1(R)$

Check constraint

Insert<12,Mary>

$U_1(R)$

$T_2$ : Insert<12,Sam>

$T_2$

$I-X_2(R)$  ← delayed

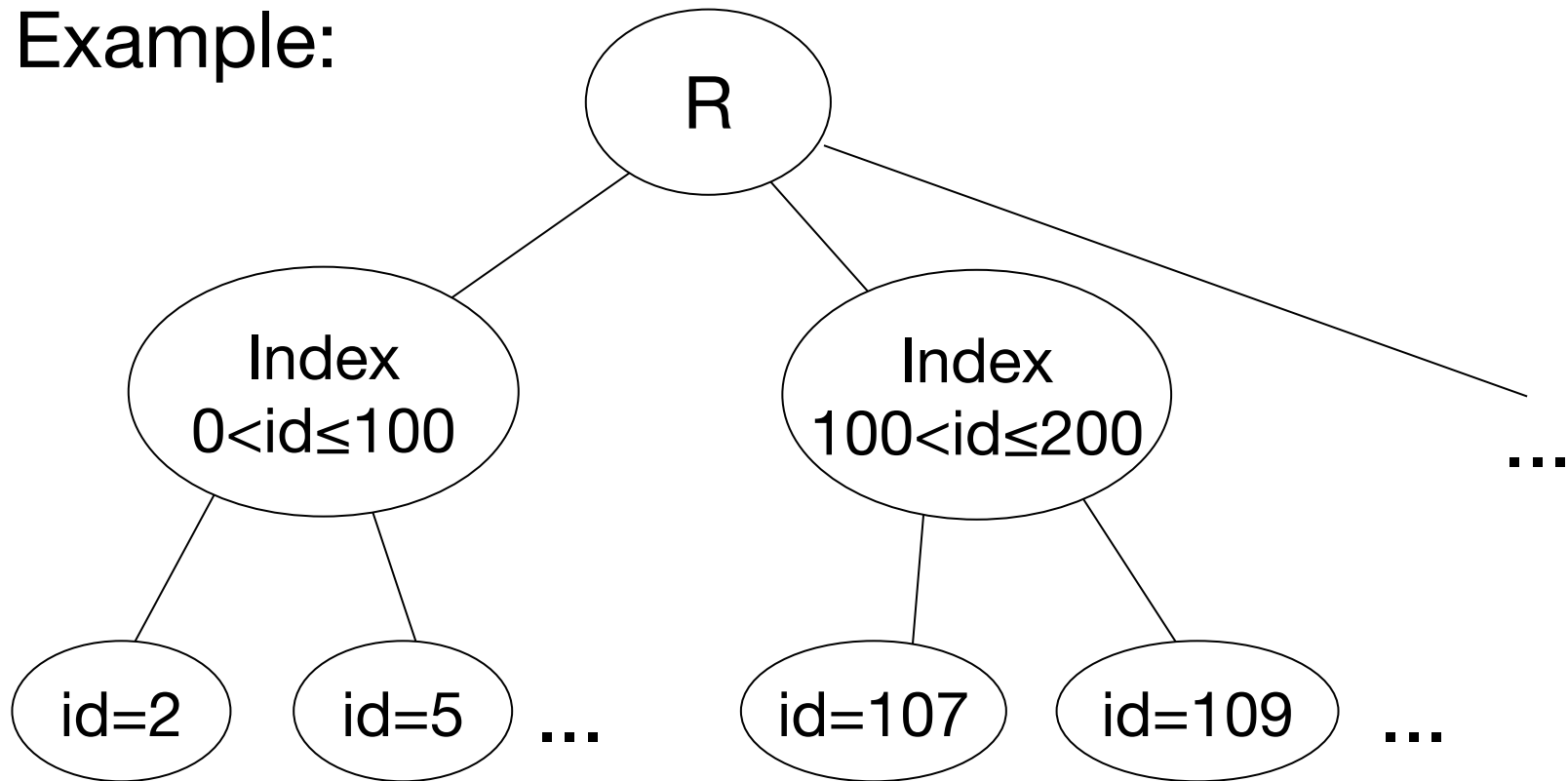
$I-X_2(R)$

Check constraint

Oops! id=12 already in R!

# Instead of Locking All of R, Can Lock Ranges of Keys

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

Beyond serializability

# Validation Overview

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

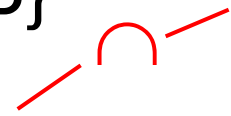
# Implementing Validation

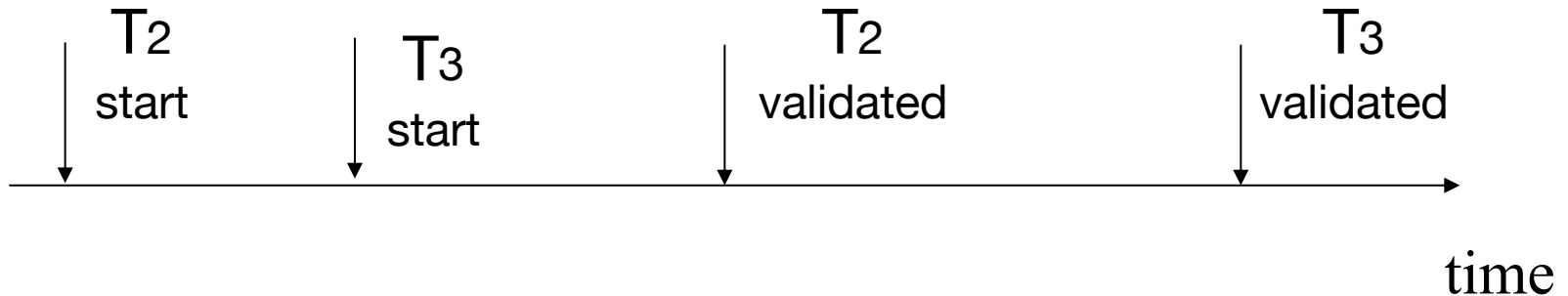
System keeps track of two sets:

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

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

# Example That Validation Must Prevent:

$$\begin{array}{ll} \text{RS}(T_2)=\{B\} & \text{RS}(T_3)=\{A,B\} \neq \emptyset \\ \text{WS}(T_2)=\{B,D\} & \text{WS}(T_3)=\{C\} \end{array}$$


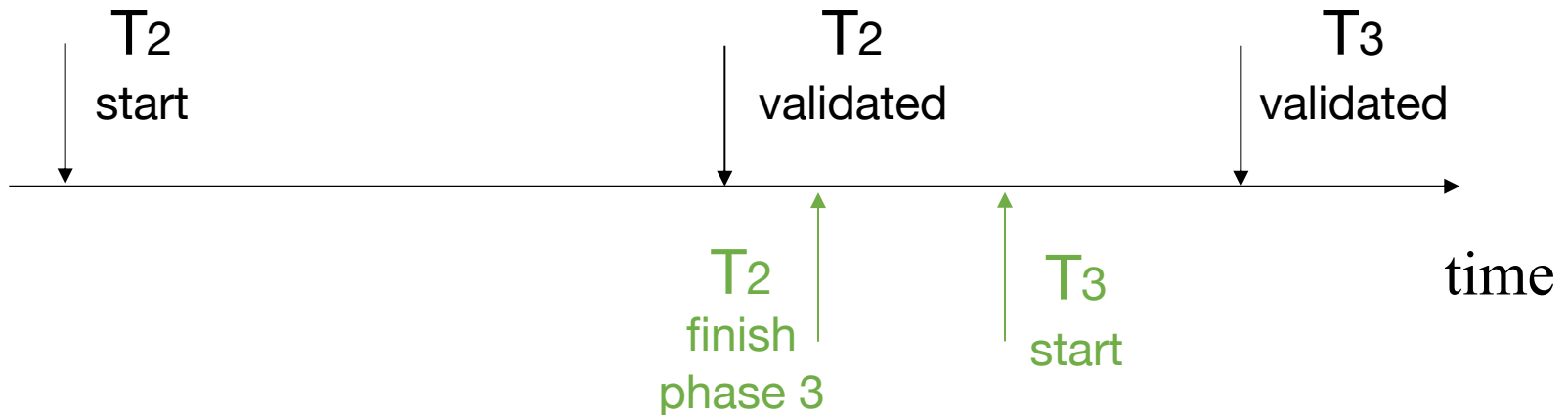




# Example That Validation Must Allow:

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

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



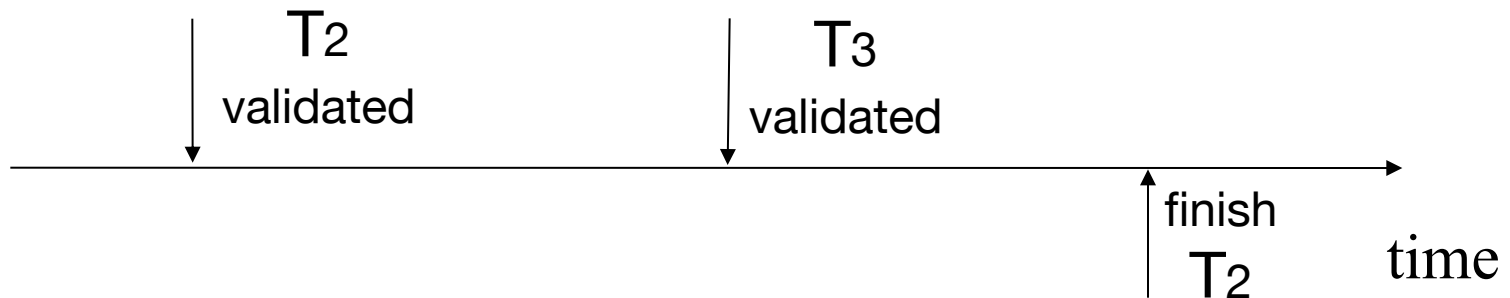
# Another Thing Validation Must Prevent:

$$RS(T_2) = \{A\}$$

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

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

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



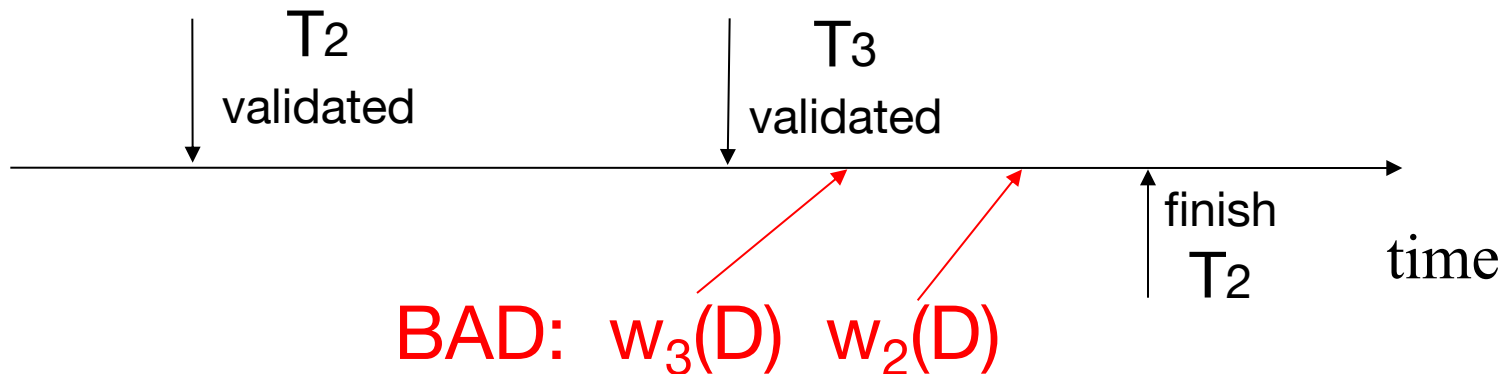
# Another Thing Validation Must Prevent:

$$RS(T_2) = \{A\}$$

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

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

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



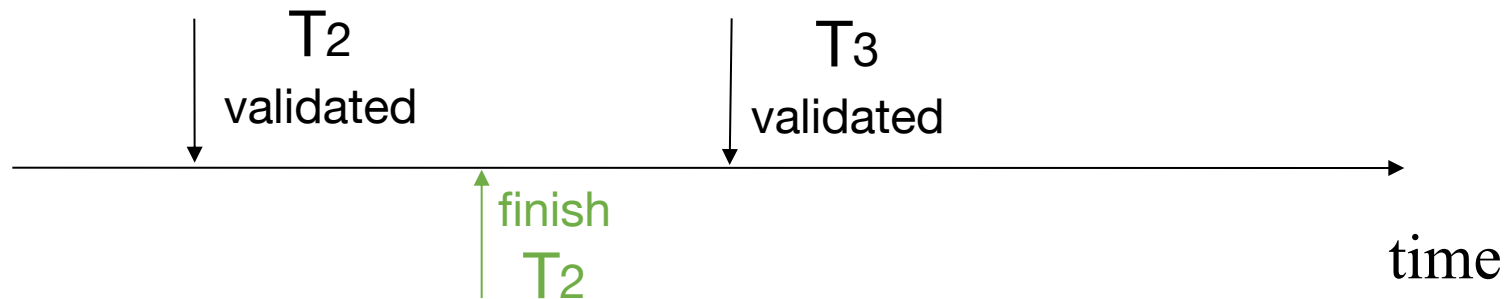
# Another Thing Validation Must Allow:

$$RS(T_2)=\{A\}$$

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

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

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



# Validation Rules for $T_j$ :

when  $T_j$  starts phase 1:

$\text{ignore}(T_j) \leftarrow \text{FIN}$

at  $T_j$  Validation:

if  $\text{Check}(T_j)$  then

$\text{VAL} \leftarrow \text{VAL} \cup \{T_j\}$

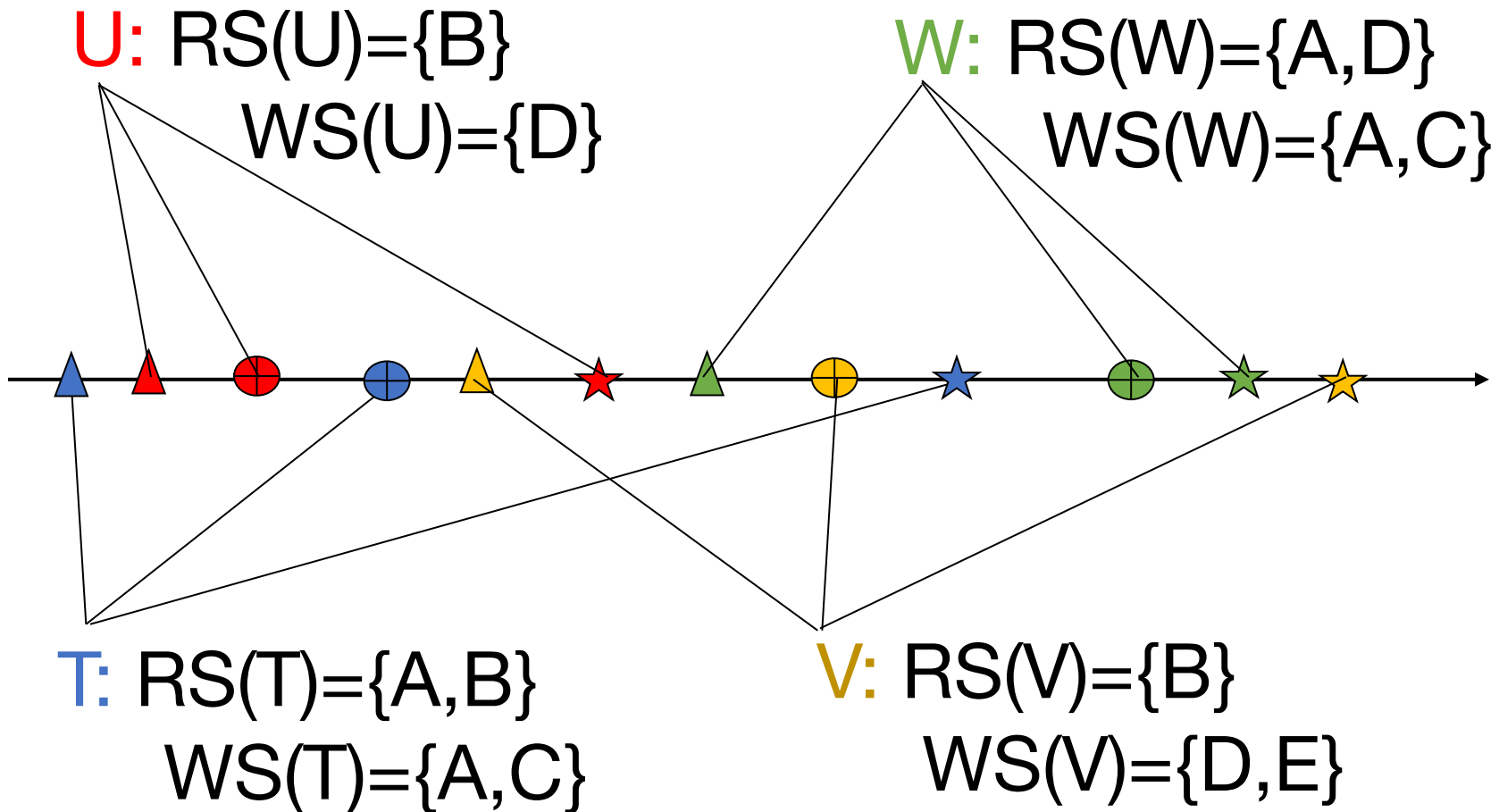
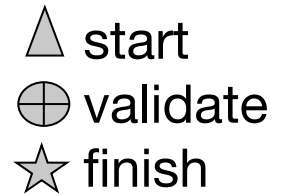
do write phase

$\text{FIN} \leftarrow \text{FIN} \cup \{T_j\}$

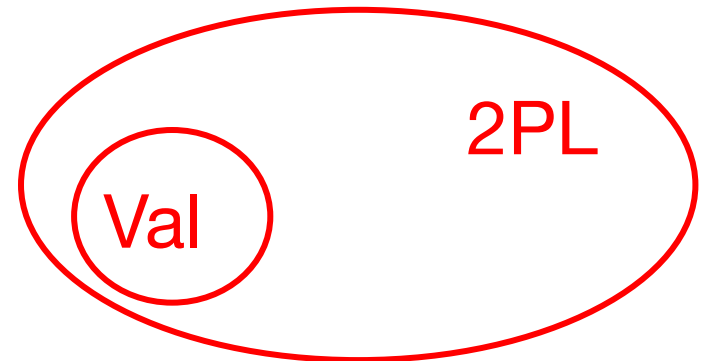
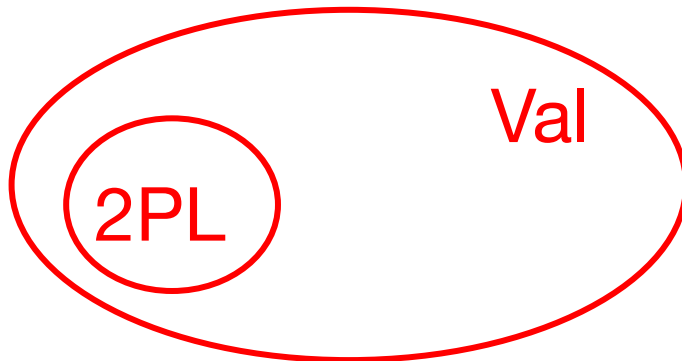
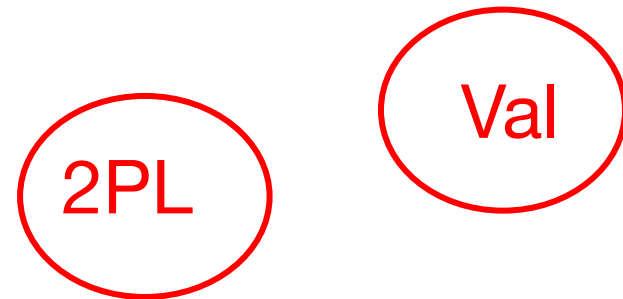
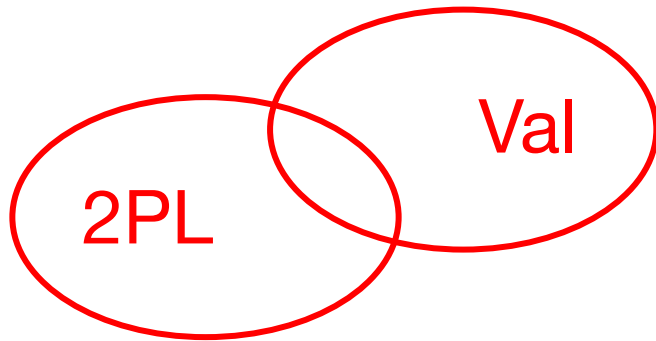
# Check( $T_j$ )

```
for  $T_i \in \text{VAL} - \text{ignore}(T_j)$  do  
    if ( $\text{WS}(T_i) \cap \text{RS}(T_j) \neq \emptyset$  or  
        ( $T_i \notin \text{FIN}$  and  $\text{WS}(T_i) \cap \text{WS}(T_j) \neq \emptyset$ ))  
        then return false  
return true
```

# Exercise



# 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) ...

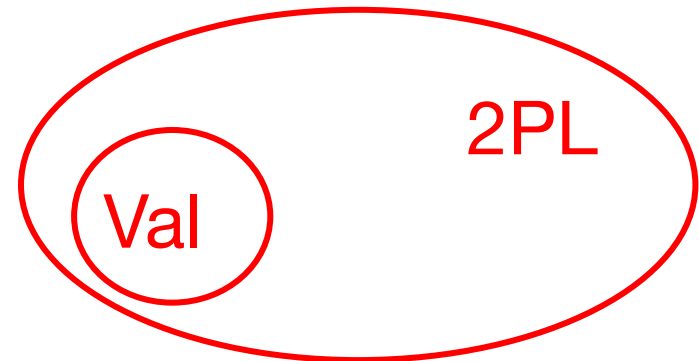
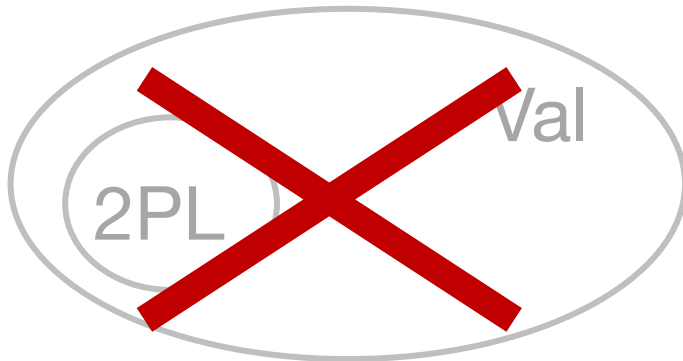
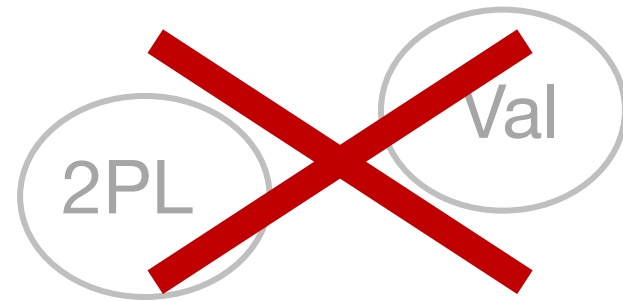
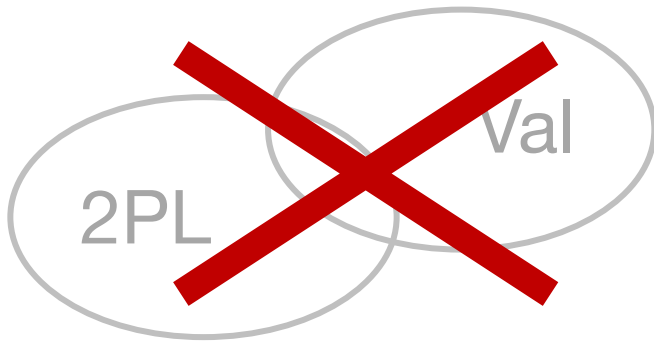
- » At val1: T2 not in Ignore(T1); T2 in VAL
- » T1 does not validate:  $WS(T2) \cap RS(T1) \neq \emptyset$
- » contradiction!

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:  
     $T2 \notin \text{FIN} \text{ AND } WS(T1) \cap WS(T2) \neq \emptyset$
- » 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

# 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

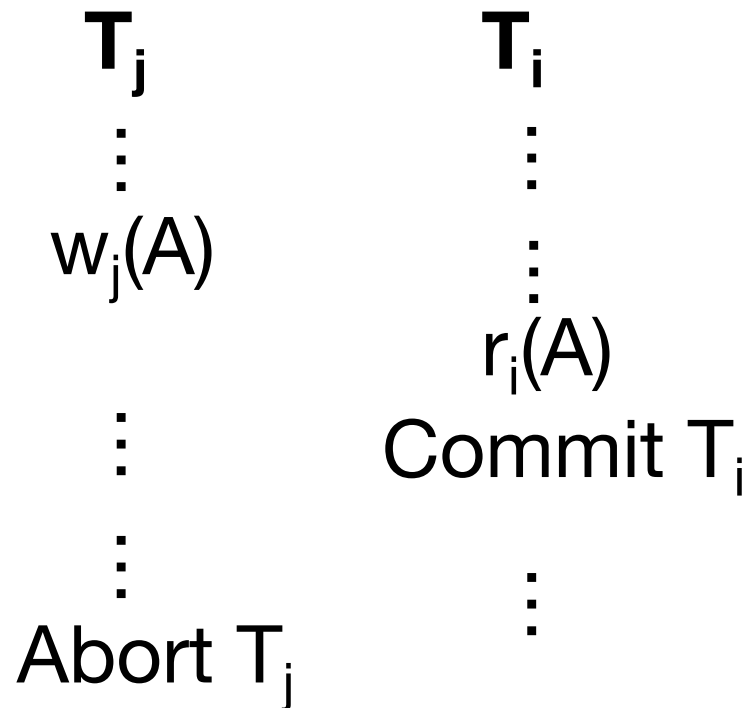
Optimistic concurrency with validation

Concurrency control + recovery

Beyond serializability

# Concurrency Control & Recovery

Example:



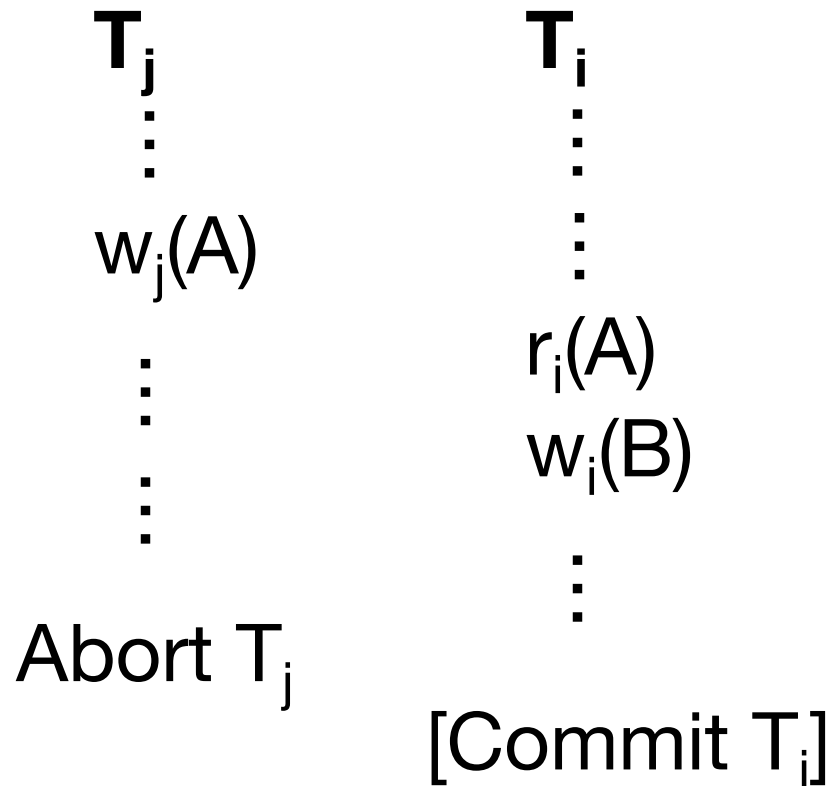
Non-persistent commit (bad!)

avoided by  
recoverable  
schedules



# Concurrency Control & Recovery

Example:



Cascading rollback (bad!)

avoided by  
avoids-cascading  
-rollback (ACR)  
schedules

# Core Problem

Schedule is conflict serializable

$$T_j \longrightarrow T_i$$

But not recoverable

# To Resolve This

Need to mark the “final” decision for each transaction in our schedules:

- » **Commit decision:** system guarantees transaction will or has completed
- » **Abort decision:** system guarantees transaction will or has been rolled back

# Model This as 2 New Actions:

$c_i$  = transaction  $T_i$  commits

$a_i$  = transaction  $T_i$  aborts

# Back to Example

$T_j$

$\vdots$

$w_j(A)$

$\vdots$

$T_i$

$\vdots$

$r_i(A)$

$\vdots$

$C_i \leftarrow$  can we commit here?

# Definition

$T_i$  reads from  $T_j$  in  $S$  ( $T_j \Rightarrow_S T_i$ ) if:

1.  $w_j(A) <_S r_i(A)$
2.  $a_j \not<_S r(A)$  ( $\not<_S$ : does not precede)
3. If  $w_j(A) <_S w_k(A) <_S r_i(A)$  then  $a_k <_S r_i(A)$

# Definition

Schedule  $S$  is **recoverable** if

whenever  $T_j \Rightarrow_S T_i$  and  $j \neq i$  and  $c_i \in S$

then  $c_j <_S c_i$

# Notes

In all transactions, reads and writes must precede commits or aborts

$\Leftrightarrow$  If  $c_i \in T_i$ , then  $r_i(A) < a_i$ ,  $w_i(A) < a_i$

$\Leftrightarrow$  If  $a_i \in T_i$ , then  $r_i(A) < a_i$ ,  $w_i(A) < a_i$

Also, just one of  $c_i$ ,  $a_i$  per transaction



# How to Achieve Recoverable Schedules?

# With 2PL, Hold Write Locks Until Commit (“Strict 2PL”)

$T_j$	$T_i$
$W_j(A)$	$\vdots$
$\vdots$	$\vdots$
$C_j$	$\vdots$
$u_j(A)$	$\vdots$
$\vdots$	$r_i(A)$

# With Validation, No Change!

Each transaction's validation point is its commit point, and only write after

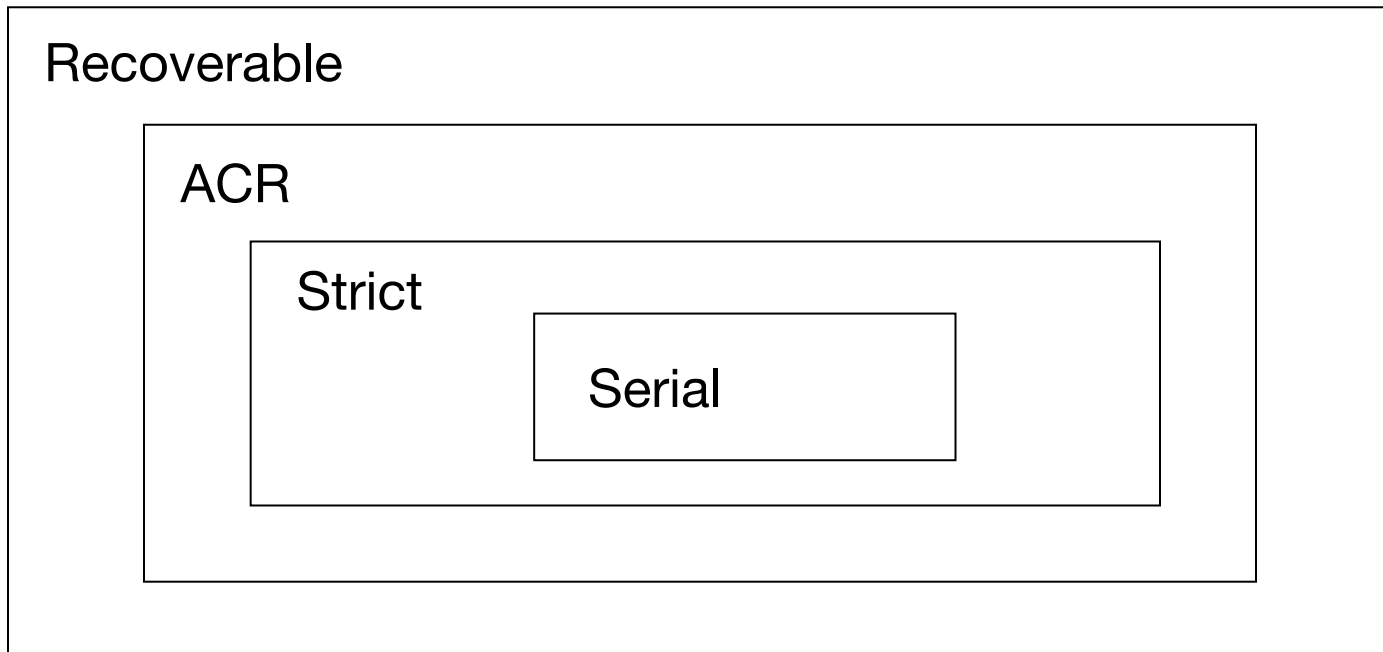
# Definitions

S is **recoverable** if each transaction commits only after all transactions from which it read have committed

S **avoids cascading rollback** if each transaction may read only those values written by committed transactions

S is **strict** if each transaction may read and write only items previously written by committed transactions ( $\equiv$  strict 2PL)

# Relationship of Recoverable, ACR & Strict Schedules



# Examples

Recoverable:

$w_1(A) \ w_1(B) \ w_2(A) \ r_2(B) \ c_1 \ c_2$

Avoids Cascading Rollback:

$w_1(A) \ w_1(B) \ w_2(A) \ c_1 \ r_2(B) \ c_2$

Strict:

$w_1(A) \ w_1(B) \ c_1 \ w_2(A) \ r_2(B) \ c_2$

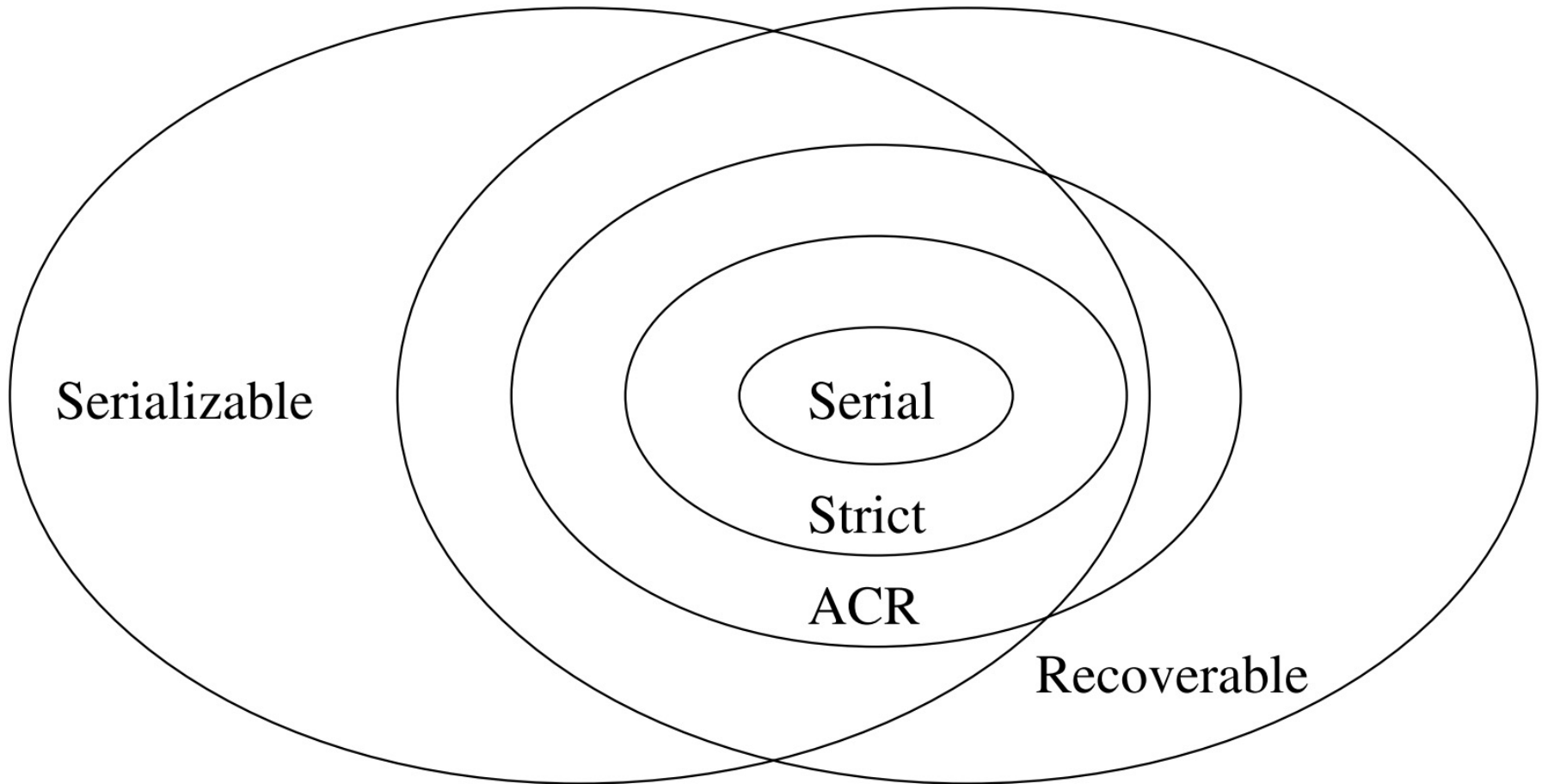
# Recoverability & Serializability

Every strict schedule is serializable

**Proof:** equivalent to serial schedule based on the order of commit points

» Only read/write from previously committed transactions

# Recoverability & Serializability





# 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

Beyond serializability

# Weaker Isolation Levels

**Dirty reads:** Let transactions read values written by other uncommitted transactions

- » Equivalent to having long-duration write locks, but no read locks

**Read committed:** Can only read values from committed transactions, but they may change

- » Equivalent to having long-duration write locks (X) and short-duration read locks (S)

# Weaker Isolation Levels

**Repeatable reads:** Can only read values from committed transactions, and each value will be the same if read again

- » Equivalent to having long-duration read & write locks (X/S) but not table locks for insert

Remaining problem: phantoms!

# Weaker Isolation Levels

**Snapshot isolation:** Each transaction sees a consistent snapshot of the whole DB (as if we saved all committed values when it began)

» Often implemented with multi-version concurrency control (MVCC)

Still has some anomalies! Example?

# Weaker Isolation Levels

**Snapshot isolation:** Each transaction sees a consistent snapshot of the whole DB (as if we saved all committed values when it began)

- » Often implemented with multi-version concurrency control (MVCC)

Write skew anomaly: txns write different values

- » Constraint:  $A+B \geq 0$
- »  $T_1$ : read A, B; if  $A+B \geq 1$ , subtract 1 from A
- »  $T_2$ : read A, B; if  $A+B \geq 1$ , subtract 1 from B
- » **Problem: what if we started with  $A=1, B=0$ ?**