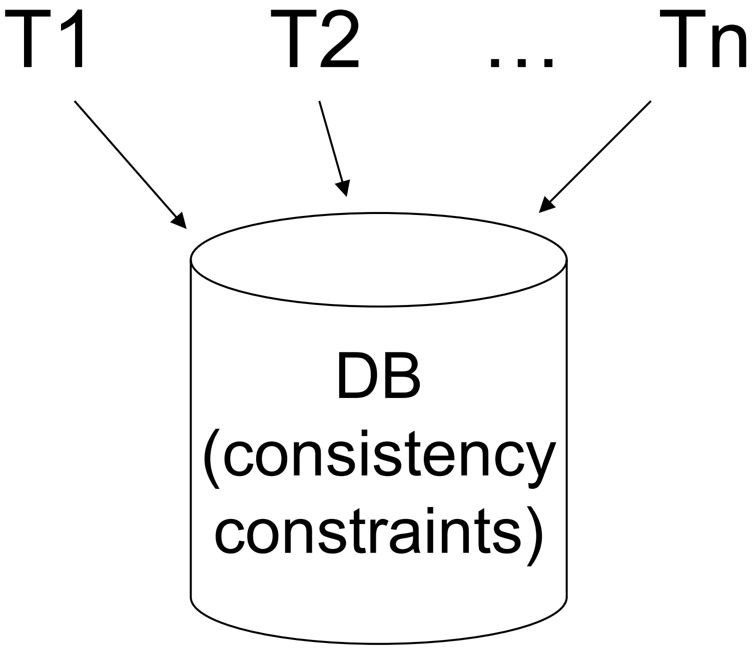
Concurrency Control

Instructor: Matei Zaharia

cs245.stanford.edu

The Problem



Different transactions may need to access data items at the same time, violating constraints

The Problem

Even if each transaction maintains constraints by itself, interleaving their actions does not

Could try to run just one transaction at a time (**serial schedule**), but this has problems

- » **Too slow!** Especially with external clients & IO

High-Level Approach

Define **isolation levels**: sets of guarantees about what transactions may experience

Strongest level: **serializability** (result is same as some serial schedule)

Many others possible: **snapshot isolation**, **read committed**, **read uncommitted**, …

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

Example

```
T1: Read(A)
    $A \leftarrow A+100$
    Write(A)
    Read(B)
    $B \leftarrow B+100$
    Write(B)
```

```
T2: Read(A)
    $A \leftarrow A\times 2$
    Write(A)
    Read(B)
    $B \leftarrow B\times 2$
    Write(B)
```

Constraint: A=B

Schedule C

T1	T2
Read(A); A ← A+100	
Write(A);	
	Read(A); A ← A×2;
	Write(A);
Read(B); B ← B+100;	
Write(B);	
	Read(B); B ← B×2;
	Write(B);

Schedule C

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A); A ← A×2;		
	Write(A);	250	
Read(B); B ← B+100;			
Write(B);			125
	Read(B); B ← B×2;		
	Write(B);		250
		250	250

Schedule D

T1	T2
Read(A); A ← A+100	
Write(A);	
	Read(A); A ← A×2;
	Write(A);
	Read(B); B ← B×2;
	Write(B);
Read(B); B ← B+100;	
Write(B);	

Schedule D

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A); A ← A×2;		
	Write(A);	250	
	Read(B); B ← B×2;		
	Write(B);		50
Read(B); B ← B+100;			
Write(B);			150
		250	150

Our Goal

Want schedules that are “good”, regardless of

» initial state and

» transaction semantics

We don’t know the logic in external client apps!

Only look at **order of read & write operations**

Example:

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

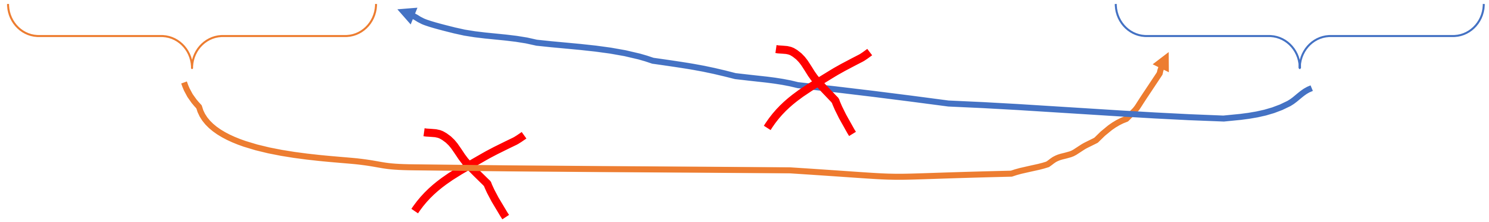
Example:

$$S_C = r1(A)w_1(A)\underbrace{r_2(A)w_2(A)}\underbrace{r_1(B)w_1(B)}r_2(B)w_2(B)$$

$$S_C' = \underbrace{r_1(A)w_1(A)r_1(B)w_1(B)}_{T_1}\underbrace{r_2(A)w_2(A)r_2(B)w_2(B)}_{T_2}$$

However, for S_D:

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



Another way to view this:

- $r_1(B)$ after $w_2(B)$ means T_1 should be after T_2 in an equivalent serial schedule ($T_2 \rightarrow T_1$)
- $r_2(A)$ after $w_1(A)$ means T_2 should be after T_1 in an equivalent serial schedule ($T_1 \rightarrow T_2$)
- Can’t have both of these!

Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking

» Shared and exclusive locks

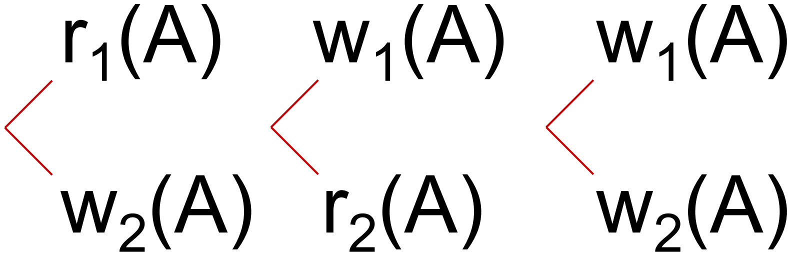
» Lock tables and multi-level locking

Optimistic concurrency with validation

Concepts

Transaction: sequence of $r_i(x)$, $w_i(x)$ actions

Conflicting actions:



$r_1(A)$ $w_1(A)$ $w_1(A)$

$w_2(A)$ $r_2(A)$ $w_2(A)$

Schedule: a chronological order in which all the transactions' actions are executed

Serial schedule: no interleaving of actions from different transactions

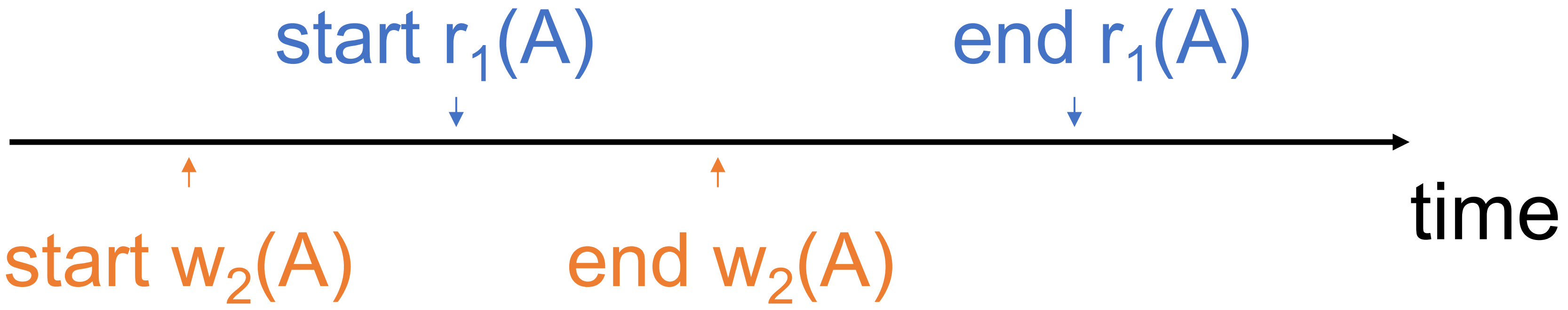
Question

Is it OK to model reads & writes as occurring at a single point in time in a schedule?

$$S = \ldots \ r_1(x) \ \ldots \ w_2(b) \ \ldots$$

Question

What about conflicting, concurrent actions on same object?



start r_1(A)

end r_1(A)

start w_2(A)

end w_2(A)

time

Assume “atomic actions” that only occur at one point in time (e.g. implement using locking)

Definition

S_1, S_2 are **conflict equivalent** schedules if S_1 can be transformed into S_2 by a series of **swaps** of non-conflicting actions

(i.e., can reorder non-conflicting operations in S_1 to obtain S_2)

Definition

A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule

Key idea:

» Conflicts “change” result of reads and writes

» Conflict serializable means there exists some equivalent serial execution that does not change the effects

How can we compute whether a schedule is conflict serializable?

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

Precedence Graph P(S)

Nodes: transactions in a schedule S

Edges: $T_i \rightarrow T_j$ whenever

» $p_i(A)$, $q_j(A)$ are actions in S

» $p_i(A) <_S q_j(A)$ (occurs earlier in schedule)

» at least one of p_i, q_j is a write (i.e. conflict)

Exercise

What is P(S) for

$S = w_3(A)\ w_2(C)\ r_1(A)\ w_1(B)\ r_1(C)\ w_2(A)\ r_4(A)\ w_4(D)$

Is S serializable?

Another Exercise

What is P(S) for

$S = w_1(A)\ r_2(A)\ r_3(A)\ w_4(A)$

Lemma

S_1, S_2 conflict equivalent $\Rightarrow$ $P(S_1)=P(S_2)$

Lemma

S_1, S_2 conflict equivalent $\Rightarrow P(S_1)=P(S_2)$

Proof:

Assume $P(S_1) \neq P(S_2)$

$\Rightarrow \exists\ T_i$: $T_i \rightarrow T_j$ in S_1 and not in S_2

$$\begin{array}{l} \Rightarrow S_1 = \ldots p_i(A) \ldots\ q_j(A) \ldots \\ \quad S_2 = \ldots q_j(A) \ldots\ p_i(A) \ldots \end{array} \qquad \left\{ \begin{array}{l} p_i,\ q_j \\ \text{conflict} \end{array} \right.$$

$\Rightarrow S_1$, S_2 not conflict equivalent

Note: $P(S_1)=P(S_2) \not\Rightarrow S_1, S_2$ conflict equivalent

Note: $P(S_1)=P(S_2) \not\Rightarrow S_1, S_2$ conflict equivalent

Counter example:

$S_1 = w_1(A)\ r_2(A)\ w_2(B)\ r_1(B)$

$S_2 = r_2(A)\ w_1(A)\ r_1(B)\ w_2(B)$

Theorem

$P(S_1)$ acyclic $\Longleftrightarrow$ S_1 conflict serializable

($\Leftarrow$) Assume S_1 is conflict serializable

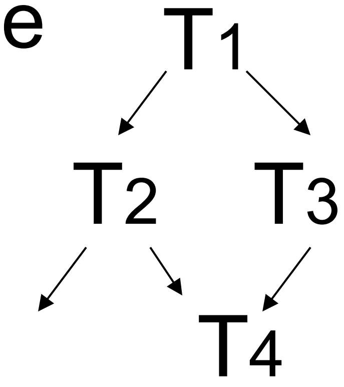
$\Rightarrow \exists\ S_s$ (serial): S_s, S_1 conflict equivalent

$\Rightarrow P(S_s) = P(S_1)$ (by previous lemma)

$\Rightarrow P(S_1)$ acyclic since $P(S_s)$ is acyclic

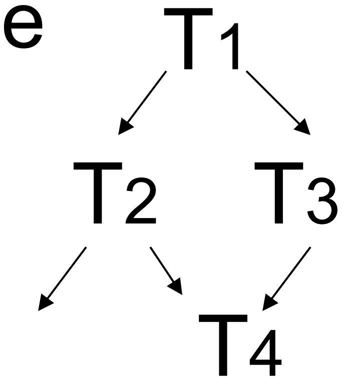
Theorem

P(S_1) acyclic $\Longleftrightarrow$ S_1 conflict serializable



Theorem

$P(S_1)$ acyclic ⟺ S_1 conflict serializable



(⇒) Assume $P(S_1)$ is acyclic

Transform S_1 as follows:

(1) Take T_1 to be transaction with no inbound edges

(2) Move all T_1 actions to the front

$$S_1 = \ldots\ldots\ q_j(A)\ldots\ldots p_1(A)\ldots\ldots$$

(3) we now have S_1 = <T_1 actions><... rest ...>

(4) repeat above steps to serialize rest!

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

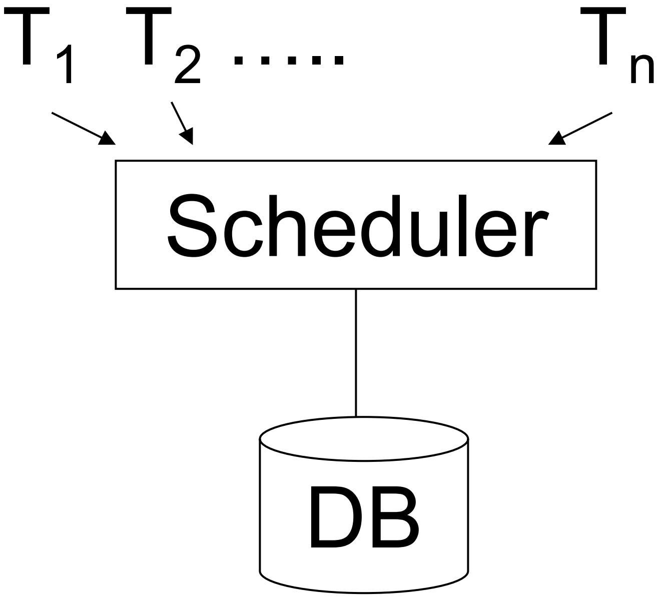
Optimistic concurrency with validation

How to Enforce Serializable Schedules?

Option 1: run system, recording P(S); at end of day, check for cycles in P(S) and declare whether execution was good

How to Enforce Serializable Schedules?

Option 2: prevent P(S) cycles from occurring



T_1 T_2 ….. T_n

Scheduler

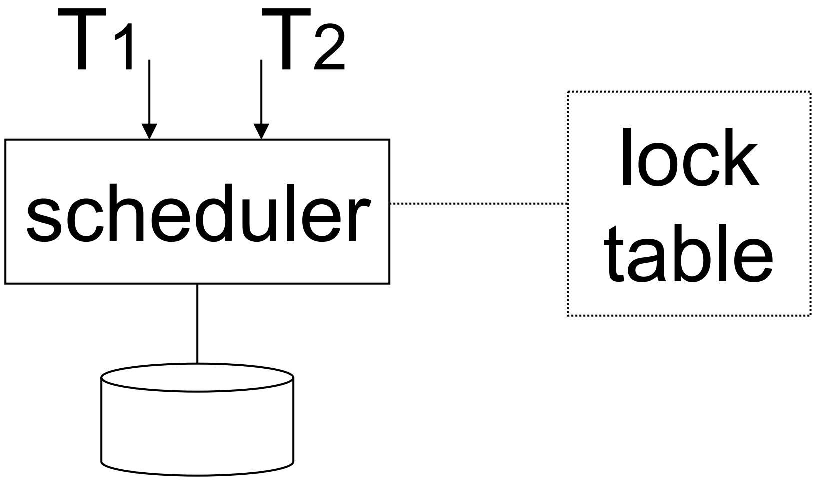
DB

A Locking Protocol

Two new actions:

lock: l_i (A) ← Transaction i locks object A

unlock: u_i (A)



Rule #1: Well-Formed Transactions

Ti: … l_i(A) … r_i(A) … u_i(A) ...

Transactions can only operate on locked items

Rule #2: Legal Scheduler

$S = \ldots\ldots\ l_i(A)\ \ldots\ldots\ldots\ldots\ u_i(A)\ \ldots\ldots\ldots$

$\longleftrightarrow$

no $l_j(A)$

Only one transaction can lock item at a time

Exercise

Which schedules are legal?
Which transactions are well-formed?

$S_1 = l_1(A)\ l_1(B)\ r_1(A)\ w_1(B)\ l_2(B)\ u_1(A)\ u_1(B)$
$r_2(B)\ w_2(B)\ u_2(B)\ l_3(B)\ r_3(B)\ u_3(B)$

$S_2 = l_1(A)\ r_1(A)\ w_1(B)\ u_1(A)\ u_1(B)\ l_2(B)\ r_2(B)$
$w_2(B)\ l_3(B)\ r_3(B)\ u_3(B)$

$S_3 = l_1(A)\ r_1(A)\ u_1(A)\ l_1(B)\ w_1(B)\ u_1(B)\ l_2(B)$
$r_2(B)\ w_2(B)\ u_2(B)\ l_3(B)\ r_3(B)\ u_3(B)$

Exercise

Which schedules are legal?
Which transactions are well-formed?

$S_1 = l_1(A)\ l_1(B)\ r_1(A)\ w_1(B)\ l_2(B)\ u_1(A)\ u_1(B)$
$\quad r_2(B)\ w_2(B)\ u_2(B)\ l_3(B)\ r_3(B)\ u_3(B)$

$S_2 = l_1(A)\ r_1(A)\ w_1(B)\ u_1(A)\ u_1(B)\ l_2(B)\ r_2(B)$
$\quad w_2(B)\ l_3(B)\ r_3(B)\ u_3(B)$ $u_2(B)$ missing

$S_3 = l_1(A)\ r_1(A)\ u_1(A)\ l_1(B)\ w_1(B)\ u_1(B)$
$\quad l_2(B)\ r_2(B)\ w_2(B)\ u_2(B)\ l_3(B)\ r_3(B)\ u_3(B)$

Schedule F

T1	T2	A	B
		25	25
l_1(A);Read(A)			
A←A+100;Write(A);u_1(A)		125	
	l_2(A);Read(A)		
	A←Ax2;Write(A);u_2(A)	250	
	l_2(B);Read(B)		
	B←Bx2;Write(B);u_2(B)		50
l_1(B);Read(B)			
B←B+100;Write(B);u_1(B)			150
		250	150

Rule #3: 2-Phase Locking (2PL)

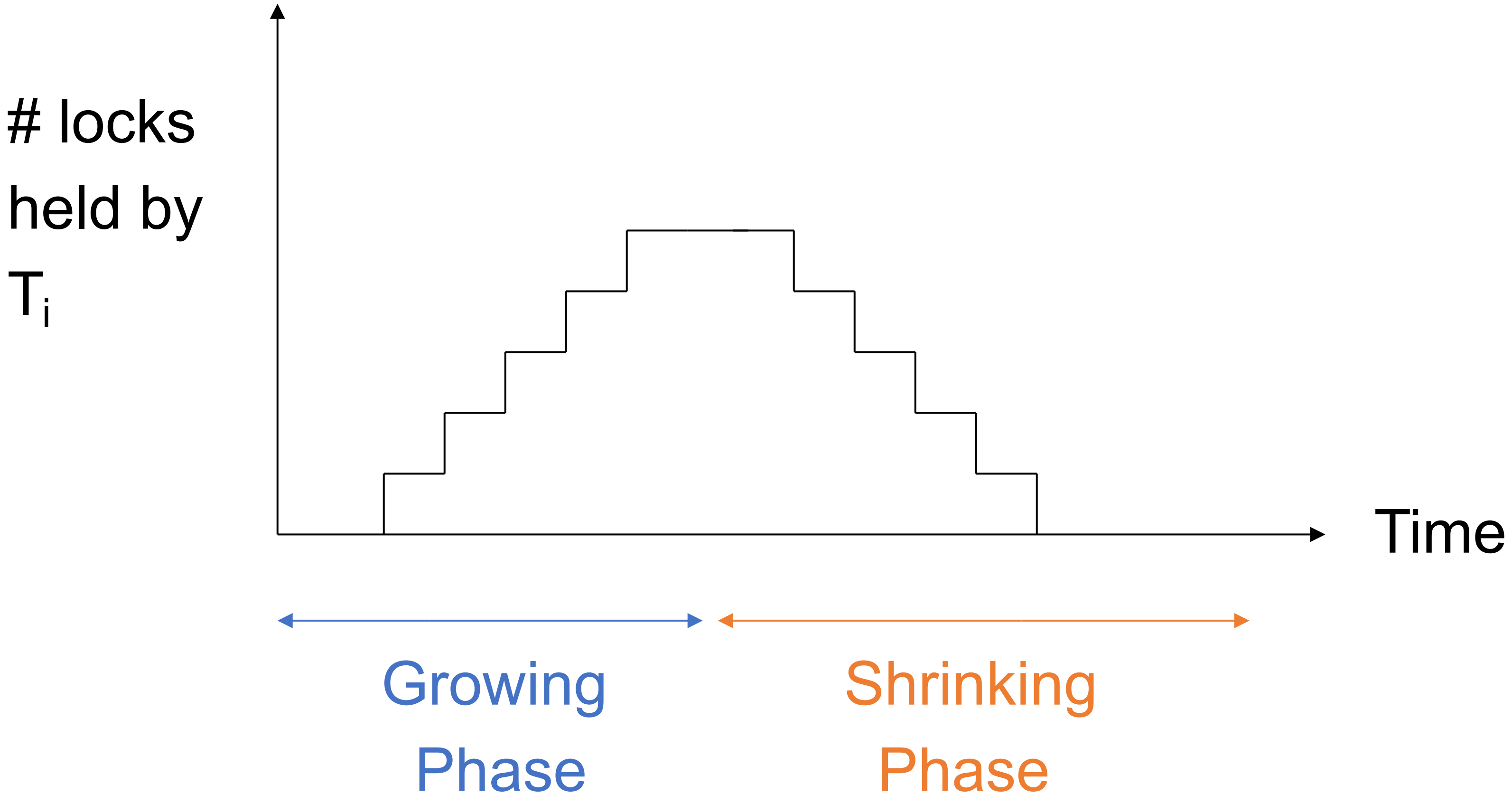
$T_i = \ldots\ldots\ldots\ l_i(A) \ldots\ldots\ldots\ldots\ u_i(A) \ldots\ldots\ldots$

no unlocks

no locks

Transactions first lock all items they need, then unlock them

2-Phase Locking (2PL)



Schedule G

T1	T2
$l_1(A)$;Read(A)	
A←A+100;Write(A)	
$l_1(B)$;$u_1(A)$	

Schedule G

T1	T2
$l_1(A)$;Read(A)	
A←A+100;Write(A)	
$l_1(B)$;$u_1(A)$	
	$l_2(A)$;Read(A)
	A←A×2;Write(A)
	$l_2(B)$ ← delayed

Schedule G

T1	T2
$l_1(A)$;Read(A)	
$A \leftarrow A+100$;Write(A)	
$l_1(B)$;$u_1(A)$	
	$l_2(A)$;Read(A)
	$A \leftarrow A \times 2$;Write(A)
	$l_2(B)$ ← delayed
Read(B);$B \leftarrow B+100$	
Write(B);$u_1(B)$	

Schedule G

T1	T2
$l_1(A)$;Read(A)	
$A \leftarrow A+100$;Write(A)	
$l_1(B)$;$u_1(A)$	
	$l_2(A)$;Read(A)
	$A \leftarrow A \times 2$;Write(A)
	$l_2(B)$ ← delayed
Read(B);$B \leftarrow B+100$	
Write(B);$u_1(B)$	
	$l_2(B)$;$u_2(A)$;Read(B)
	$B \leftarrow B \times 2$;Write(B);$u_2(B)$

Schedule H (T2 Ops Reversed)

T1	T2
l_1(A); Read(A)	l_2(B); Read(B)
A←A+100; Write(A)	B←B×2; Write(B)
l_1(B) ← delayed (T2 holds B)	l_2(A) ← delayed (T1 holds A)

Problem: Deadlock between transactions

Dealing with Deadlock

Option 1: Detect deadlocks and roll back one of the deadlocked transactions

- » The rolled back transaction no longer appears in our schedule

Option 2: Agree on an order to lock items in that prevents deadlocks

- » E.g. transactions acquire locks in key order
- » Must know which items T_i will need up front!

Is 2PL Correct?

Yes! We can prove that following rules #1,2,3 gives conflict-serializable schedules

Conflict Rules for Lock Ops

$l_i(A)$, $l_j(A)$ conflict

$l_i(A)$, $u_j(A)$ conflict

Note: no conflict $<u_i(A), u_j(A)>$, $<l_i(A), r_j(A)>$,...

Theorem

Rules #1,2,3 ⇒ conflict-serializable schedule (2PL)

To help in proof:

Definition: Shrink(Ti) = SH(Ti) =
first unlock action of Ti

Lemma

Ti → Tj in S ⇒ SH(Ti) $<_S$ SH(Tj)

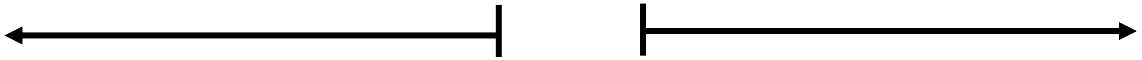
Proof:

Ti → Tj means that

S = … p_i(A) … q_j(A) …; p,q conflict

By rules 1, 2:

S = … p_i(A) … u_i(A) … l_j(A) ... q_j(A) …



By rule 3: SH(Ti) SH(Tj)

So, SH(Ti) $<_S$ SH(Tj)

Theorem: Rules #1,2,3 $\Rightarrow$ Conflict Serializable Schedule

Proof:

(1) Assume P(S) has cycle

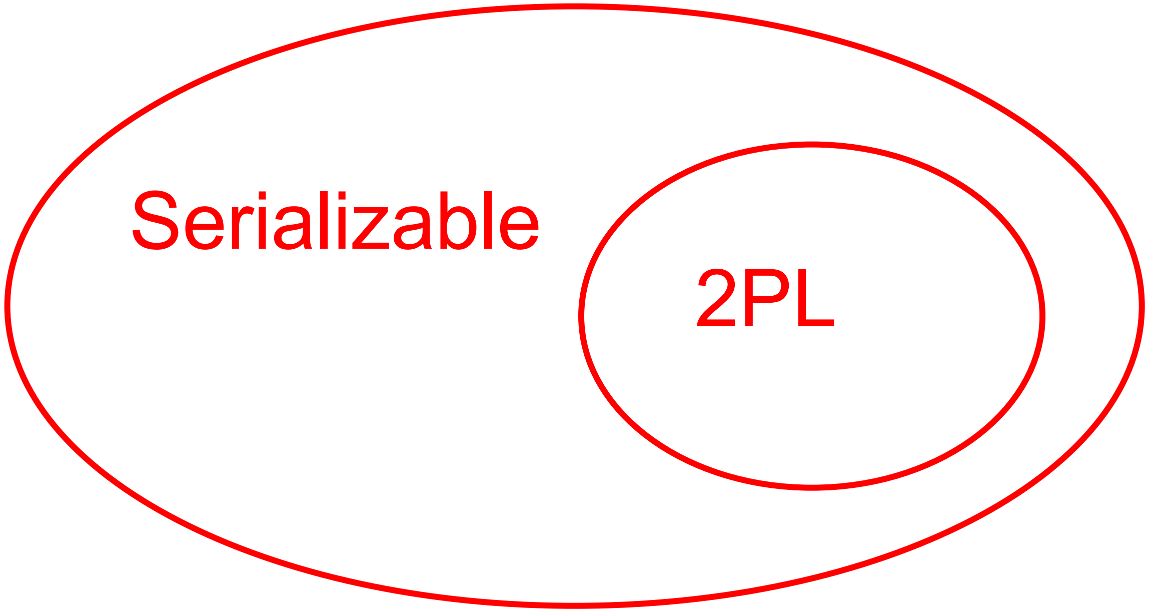
$$T1 \rightarrow T2 \rightarrow \ldots. Tn \rightarrow T1$$

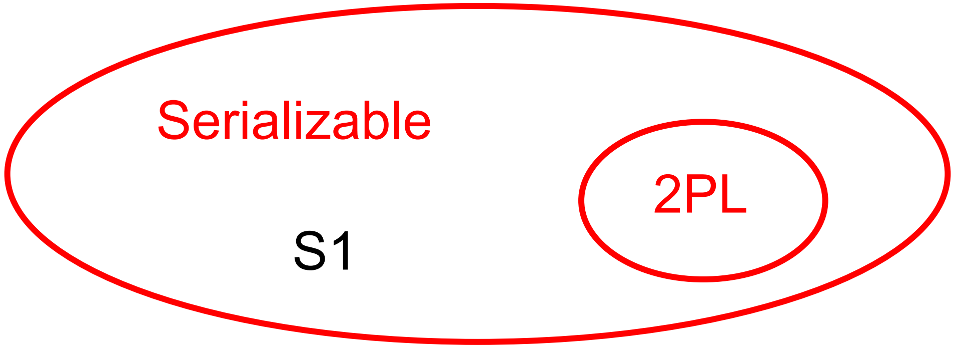
(2) By lemma: SH(T1) < SH(T2) < ... < SH(T1)

(3) Impossible, so P(S) acyclic

(4) $\Rightarrow$ S is conflict serializable

2PL Subset of Serializable





S1: $w_1(X)$ $w_3(X)$ $w_2(Y)$ $w_1(Y)$

S1 cannot be achieved via 2PL:
The lock by T1 for Y must occur after $w_2(Y)$, so the unlock by T1 for X must occur after this point (and before $w_1(X)$). Thus, $w_3(X)$ cannot occur under 2PL where shown in S1.

But S1 is serializable: equivalent to T2, T1, T3.

If You Need More Practice

Are our schedules S_C and S_D 2PL schedules?

S_C: $w_1(A)$ $w_2(A)$ $w_1(B)$ $w_2(B)$

S_D: $w_1(A)$ $w_2(A)$ $w_2(B)$ $w_1(B)$

Optimizing Performance

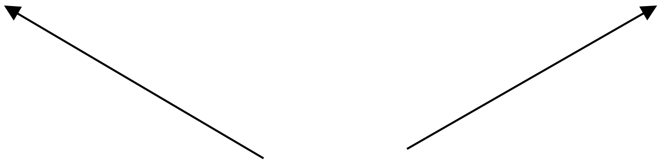
Beyond this simple 2PL protocol, it is all a matter of improving performance and allowing more concurrency….

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

Shared Locks

So far:

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

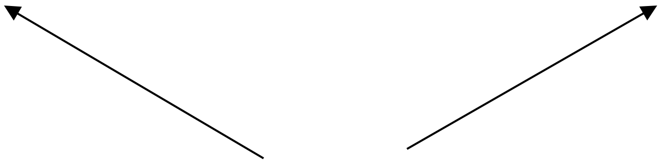


Do not conflict

Shared Locks

So far:

S = ...$l_1(A)$ $r_1(A)$ $u_1(A)$ … $l_2(A)$ $r_2(A)$ $u_2(A)$ …



Do not conflict

Instead:

S=... $ls_1(A)$ $r_1(A)$ $ls_2(A)$ $r_2(A)$ …. $us_1(A)$ $us_2(A)$

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:

ui(A): unlock whatever modes Ti has locked A

Rule 1: Well-Formed Transactions

T_i =... l-S_1(A) … r_1(A) … u_1 (A) …

T_i =... l-X_1(A) … w_1(A) … u_1 (A) …

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

T1 = ...l-X$_1$(A) … r$_1$(A) ... w$_1$(A) ... u(A) …

Rule 1: Well-Formed Transactions

What about transactions that read and write same object?

Option 2: Upgrade lock to X on write

T1 = ...$\text{l-S}_1\text{(A)}\ldots\text{r}_1\text{(A)}\ldots\text{l-X}_1\text{(A)}\ldots\text{w}_1\text{(A)}\ldots\text{u}_1\text{(A)}\ldots$

(Think of this as getting a 2nd lock, or dropping S to get X.)

Rule 2: Legal Scheduler

$S = ...\ l\text{-}S_i(A)\ \ldots \quad \ldots\ u_i(A)\ \ldots$

$\longleftrightarrow$

no $l\text{-}X_j(A)$

$S = ...\ l\text{-}X_i(A)\ \ldots \quad \ldots\ u_i(A)\ \ldots$

$\longleftrightarrow$

no $l\text{-}X_j(A)$

no $l\text{-}S_j(A)$

A Way to Summarize Rule #2

Lock mode compatibility matrix

compat =

	S	X
S	true	false
X	false	false

Rule 3: 2PL Transactions

No change except for upgrades:

(I) If upgrade gets more locks

(e.g., S → {S, X}) then no change!

(II) If upgrade releases read lock (e.g., S→X)

can be allowed 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 compat(m,n)

l-m_i(A), u-n_j(A) do not conflict if compat(m,n)

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 ← A+k; Write(A)}

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

Compatibility Matrix

compat

	S	X	I
S			
X			
I			

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

Lock already held in

compat	S	X	U
S	T	F	
X	F	F	
U			

Compatibility Matrix

New request

Lock already held in

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

Note: asymmetric table!

How Is Locking Implemented In Practice?

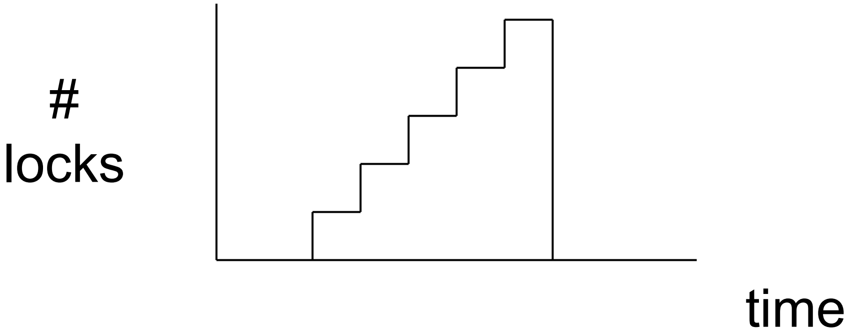
Every system is different (e.g., may not even provide conflict serializable schedules)

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



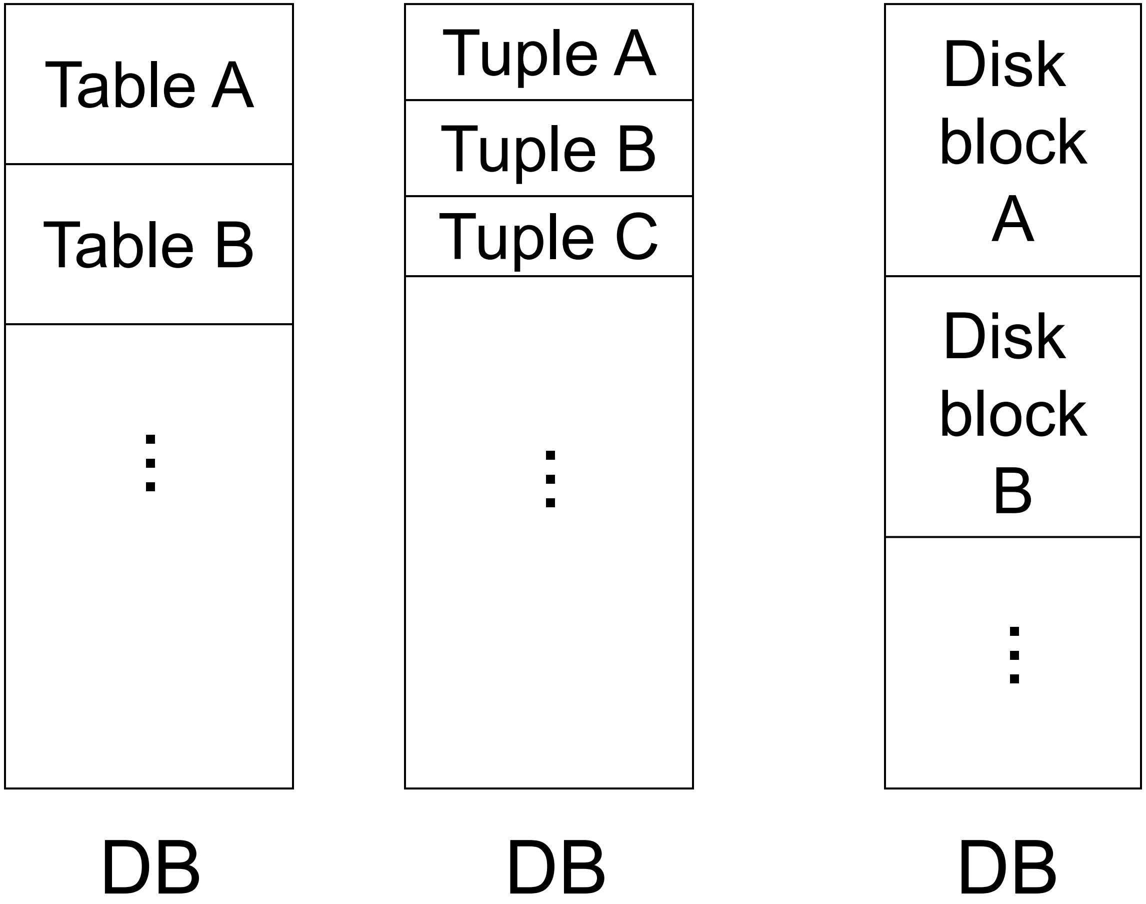
Sample Locking System

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?



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)

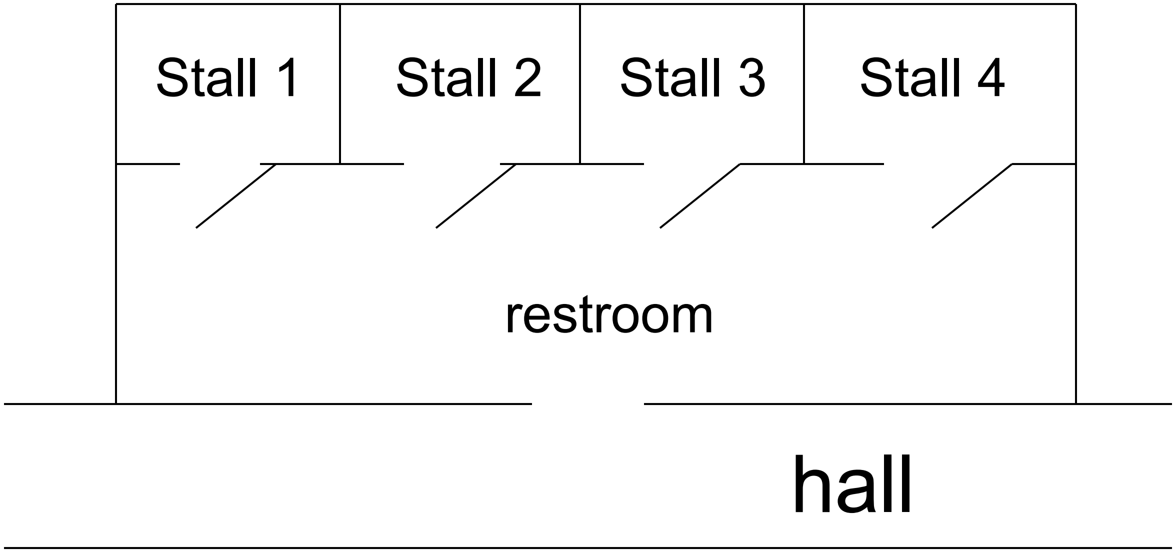
- Need few locks
- Low concurrency

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

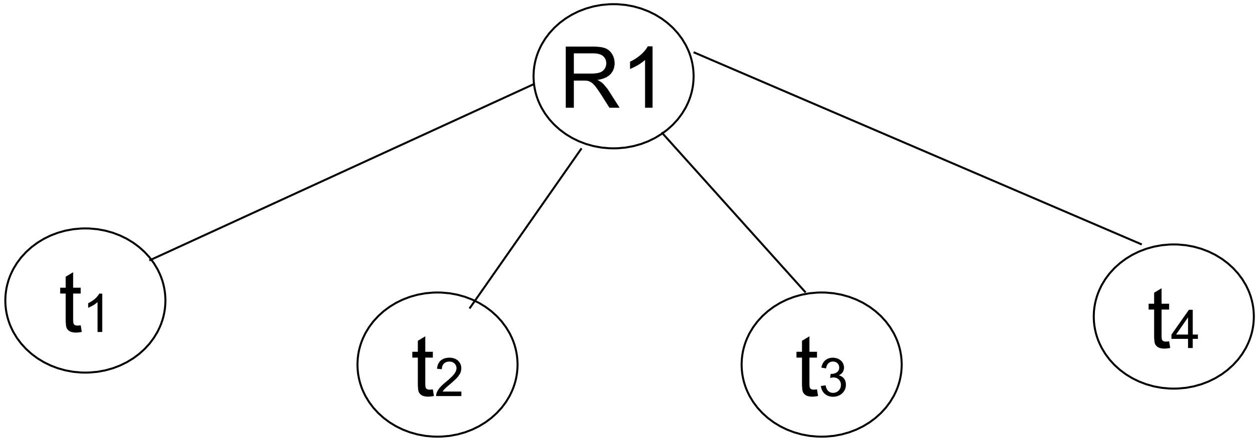
- Need more locks
- More concurrency

We Can Have It Both Ways!

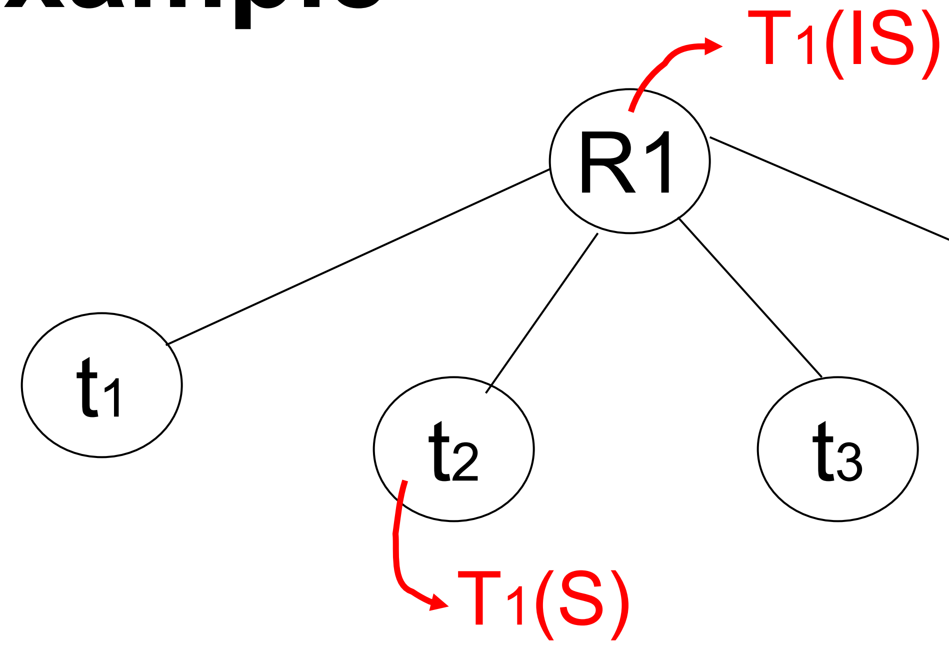
Ask any janitor to give you the solution...



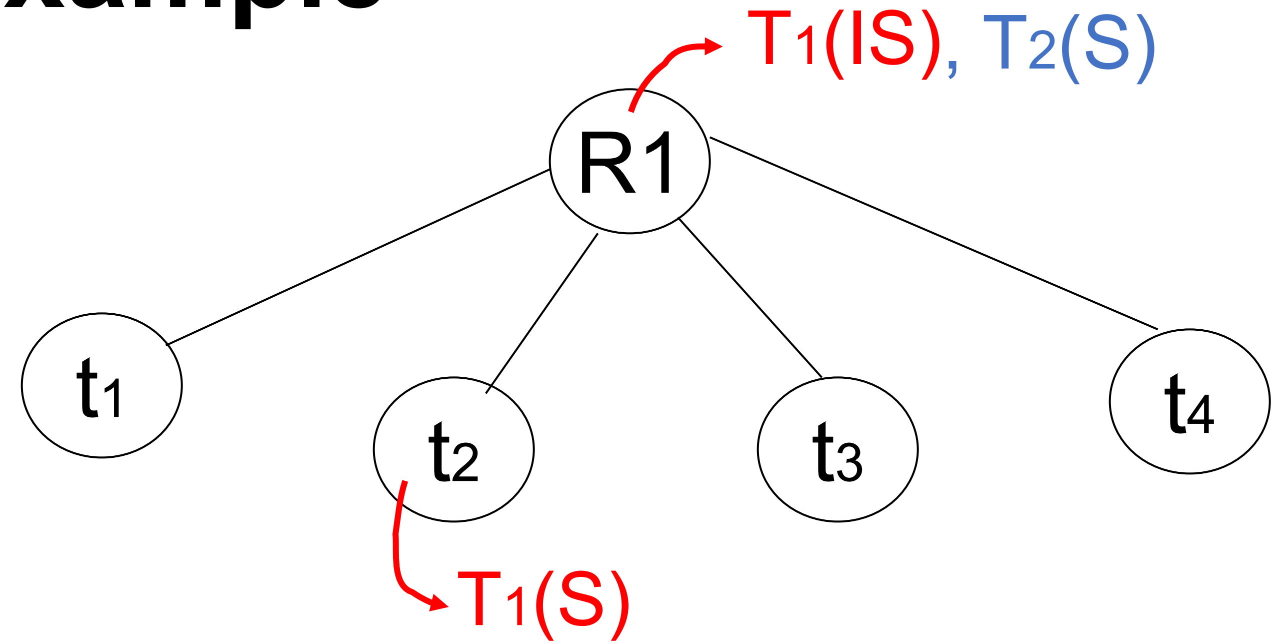
Example



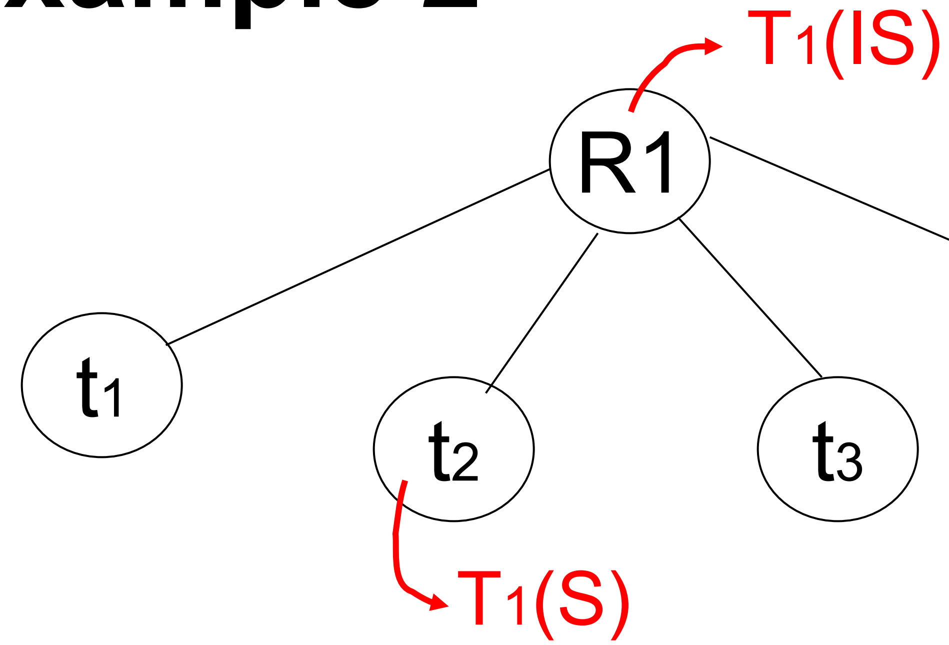
Example



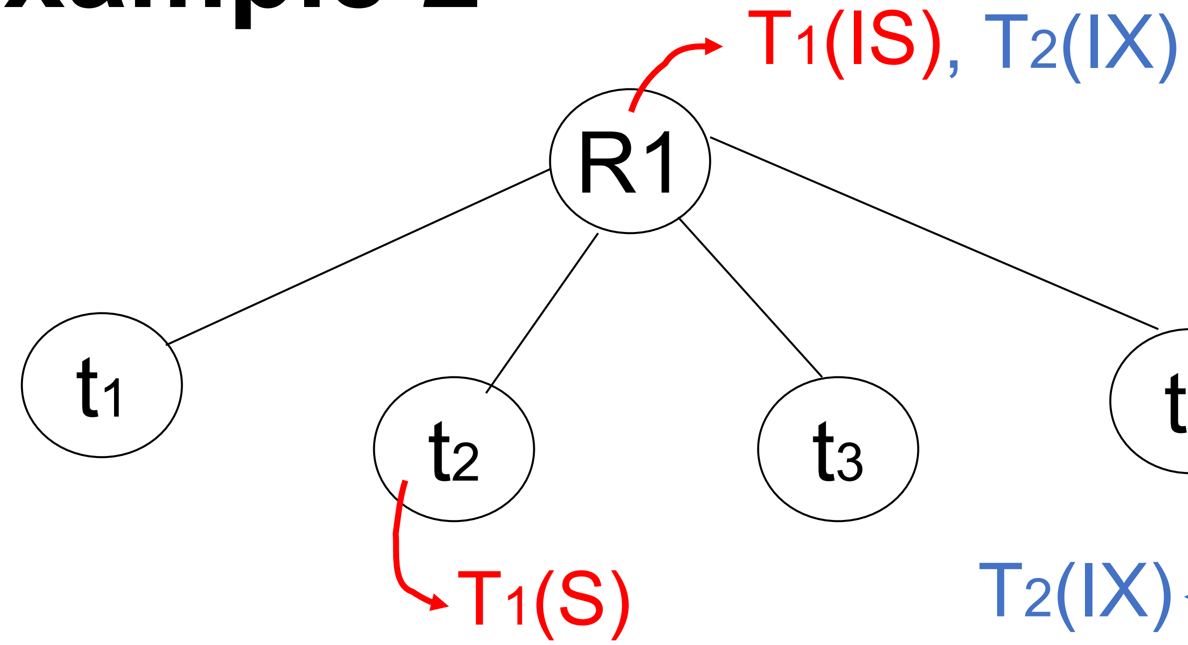
Example



Example 2



Example 2



T1(IS), T2(IX)

R1

t1

t2

t3

t4

T1(S)

T2(IX)

Multiple Granularity Locks

compat Requestor

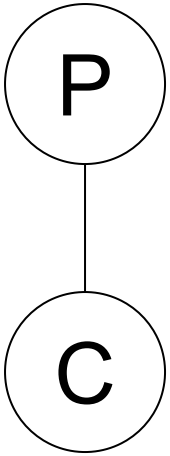
		IS	IX	S	SIX	X
	IS					
Holder	IX					
	S					
	SIX					
	X					

Multiple Granularity Locks

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



Rules

(1) Follow multiple granularity comp 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

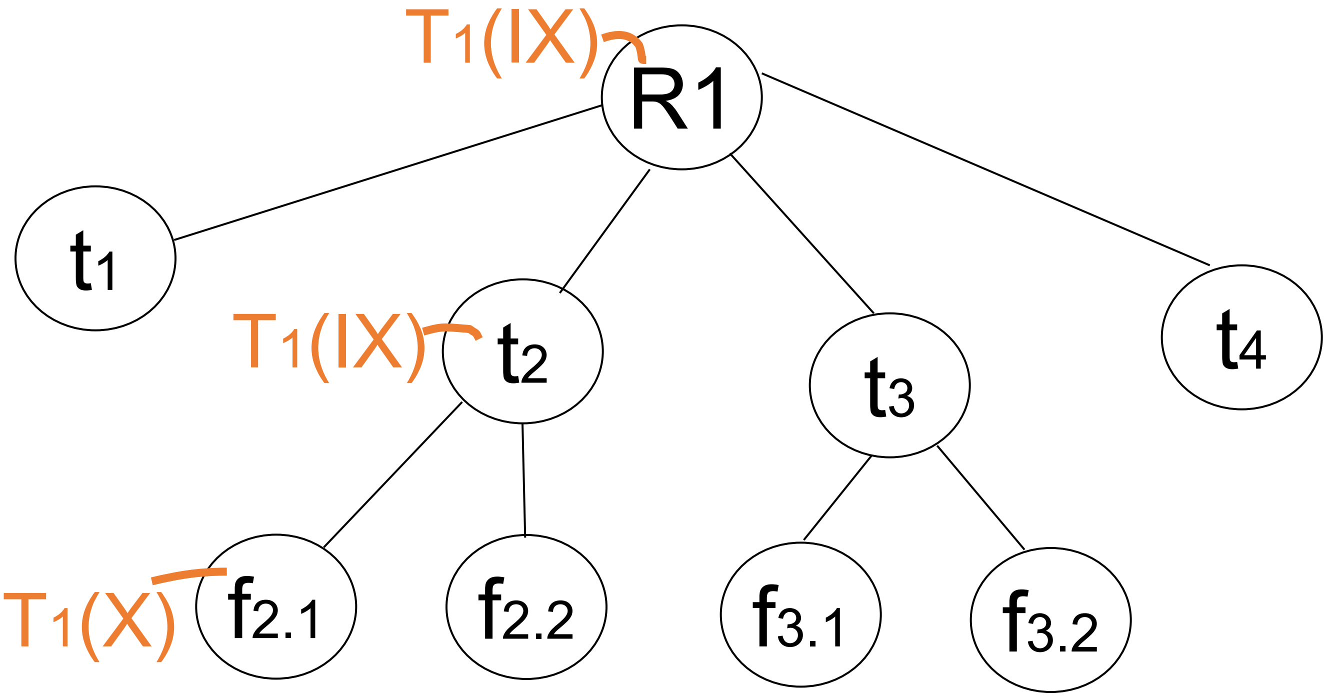
(4) Node Q can be locked by Ti in X,SIX,IX only if parent(Q) locked by Ti in IX,SIX

(5) Ti is two-phase

(6) Ti can unlock node Q only if none of Q's children are locked by Ti

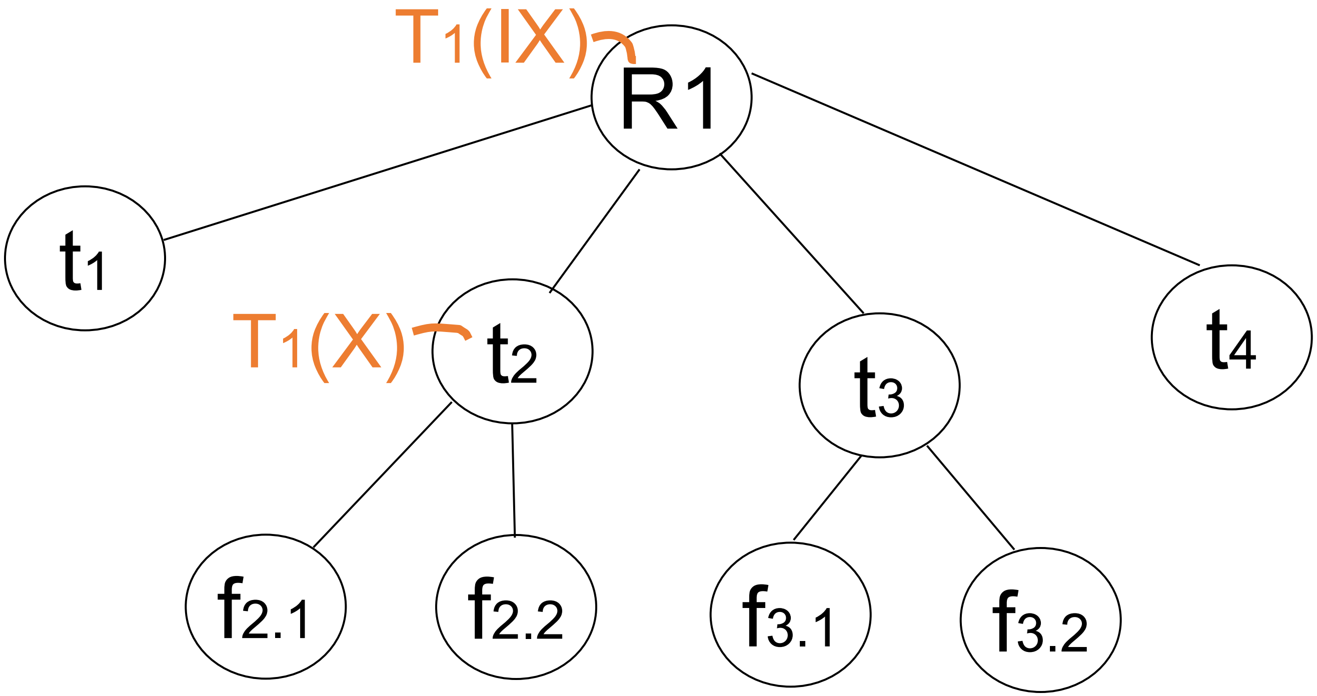
Exercise:

Can T2 access object f2.2 in X mode? What locks will T2 get?



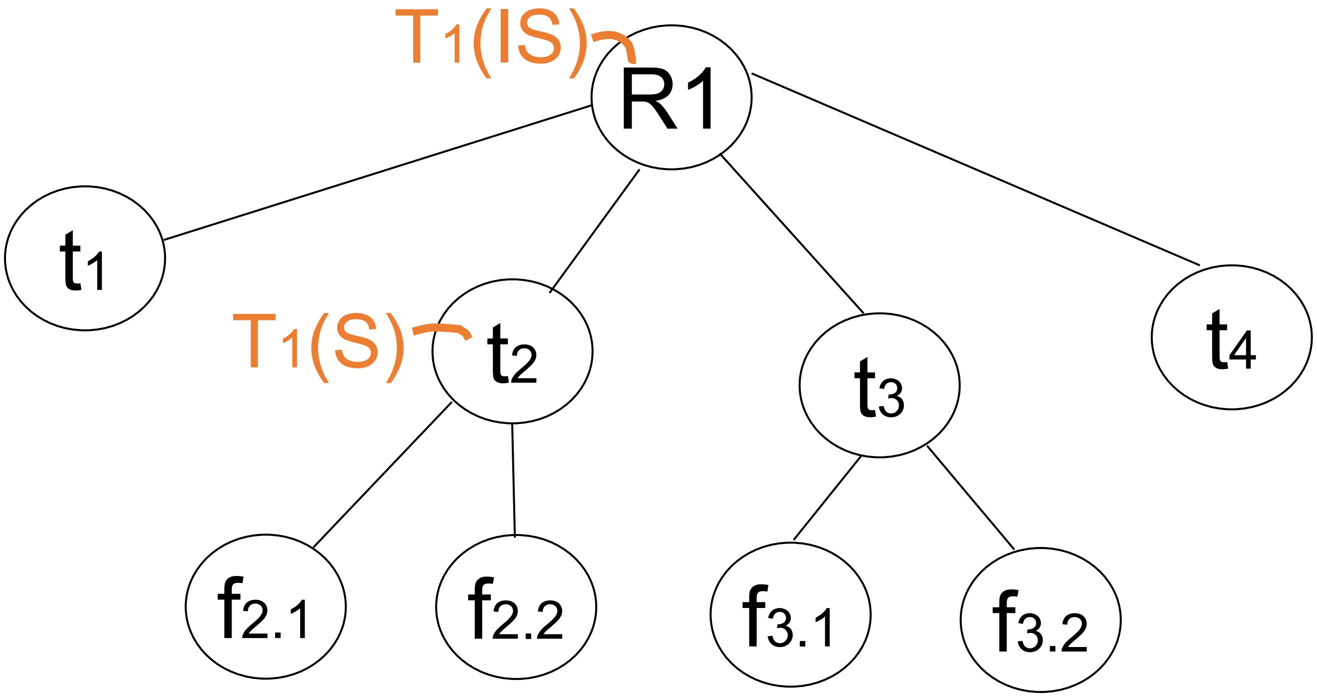
Exercise:

Can T2 access object f2.2 in X mode? What locks will T2 get?



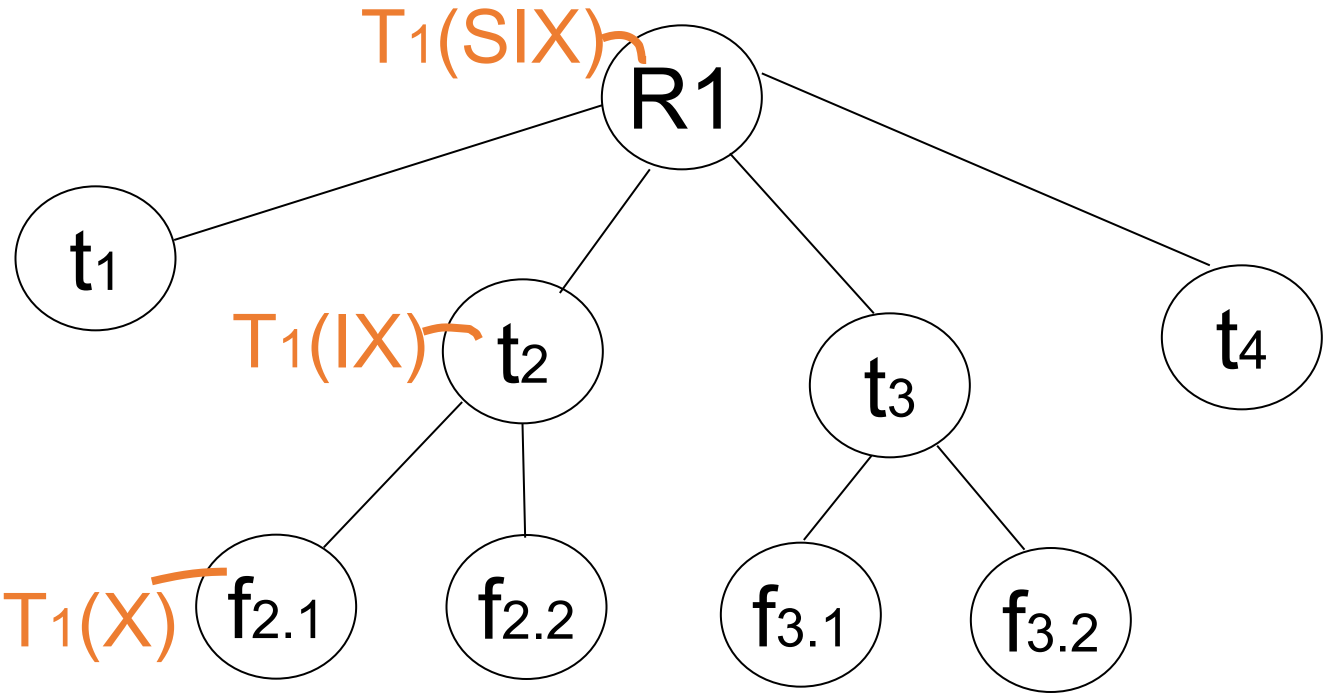
Exercise:

Can T2 access object f3.1 in X mode? What locks will T2 get?



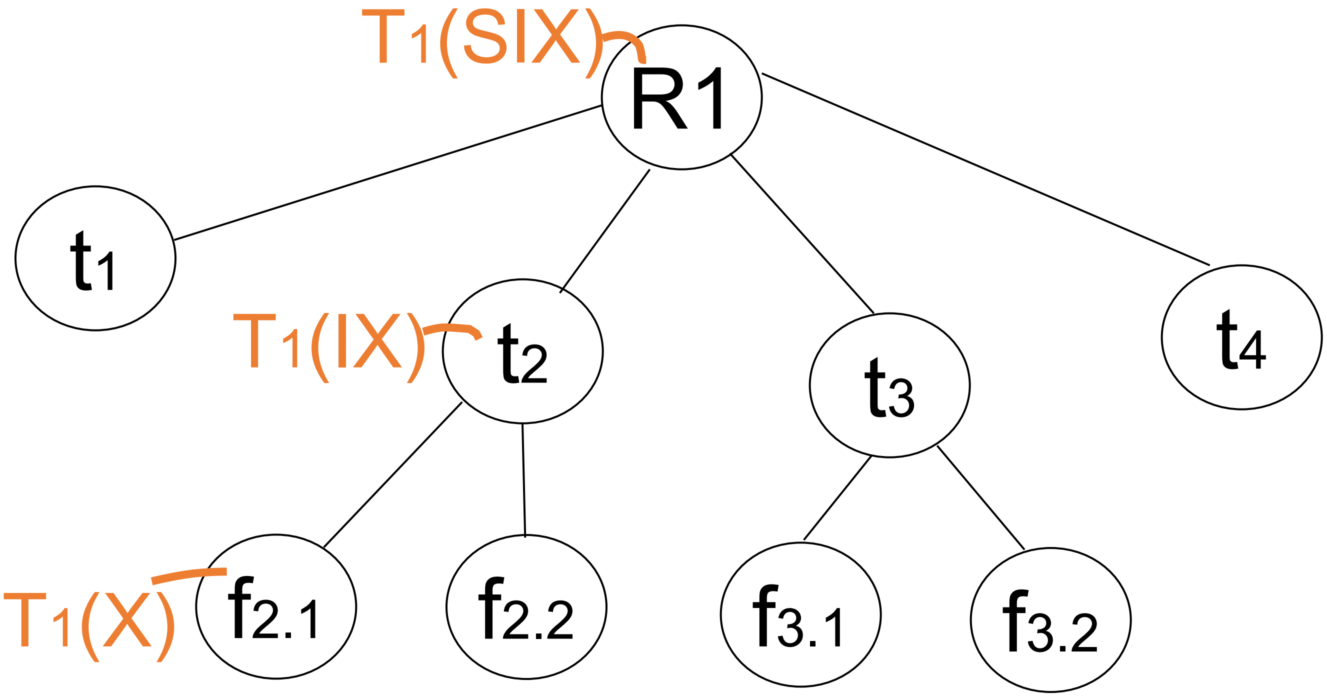
Exercise:

Can T2 access object f2.2 in S mode? What locks will T2 get?

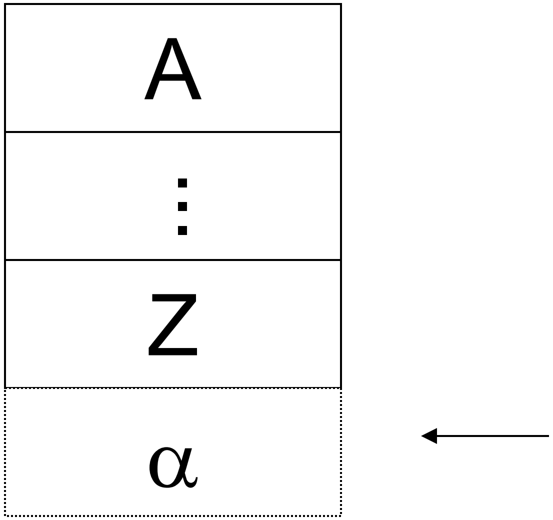


Exercise:

Can T2 access object f2.2 in X mode? What locks will T2 get?



Insert + delete operations



Insert

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	

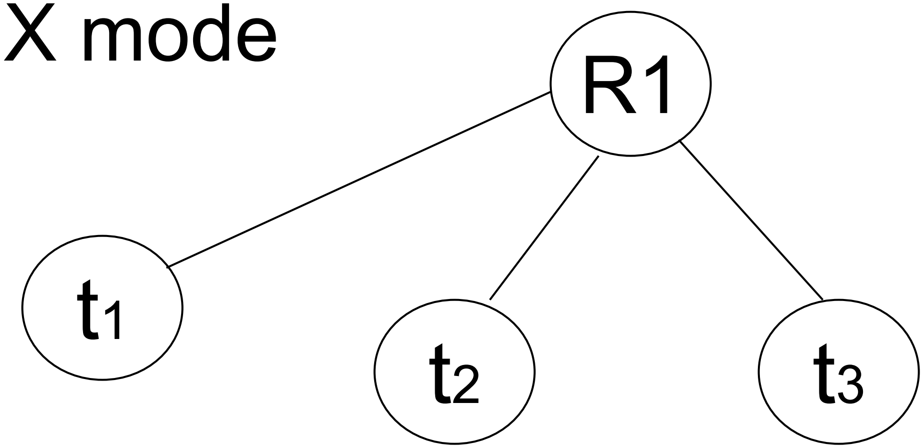
T1: Insert <12,Mary,…> into R
T2: Insert <12,Sam,…> into R

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



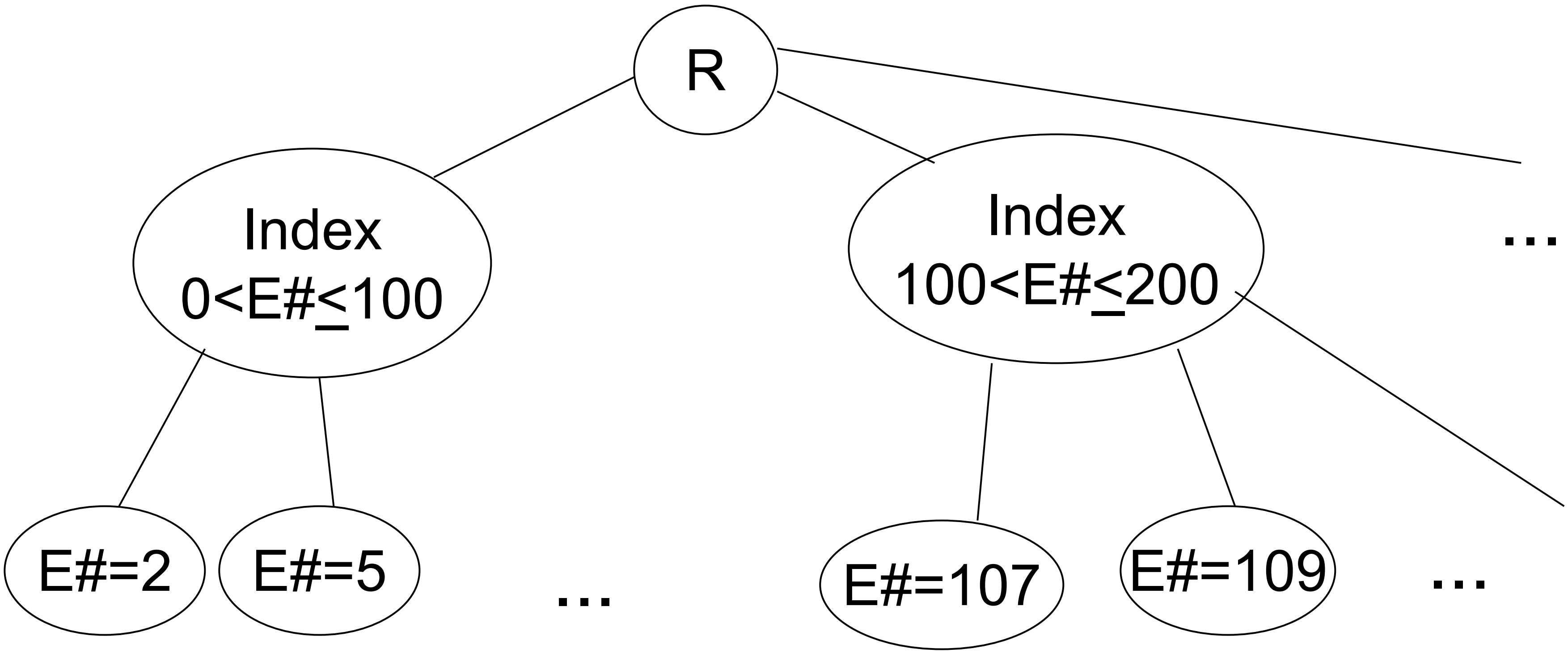
Back to example

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

Next Class

Guest talk by **Reynold Xin** from Databricks:

Delta Lake: Making Cloud Data Lakes Transactional and Scalable

The same concurrency issues we saw happen in large data lakes with billions of files… how to offer transactions there?

DELTA LAKE