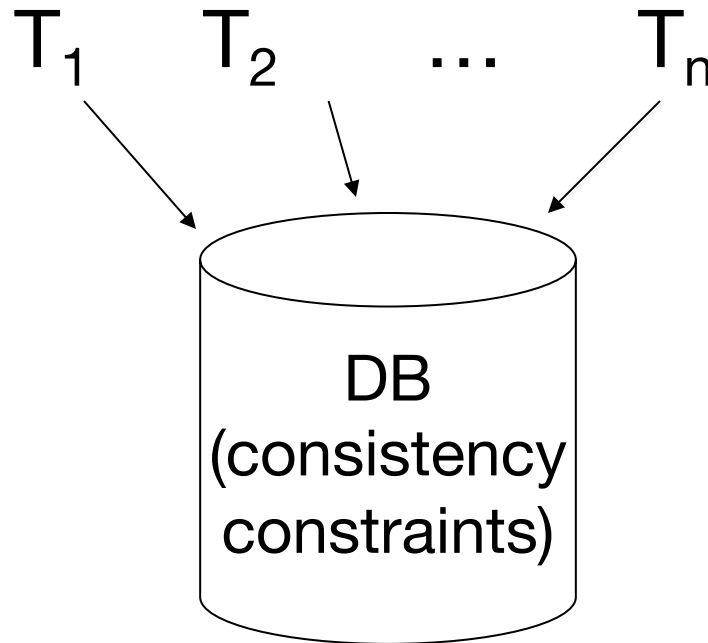


Concurrency Control

Instructor: Matei Zaharia

cs245.stanford.edu

The Problem



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

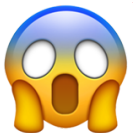
Example

Constraint: all interns have equal salaries

T_1 : add \$1000 to each intern's salary

T_2 : double each intern's salary

Salaries: ~~2000~~ ~~2000~~ ~~2000~~ ~~2000~~ ~~2000~~
~~3000~~ ~~3000~~ ~~3000~~ ~~4000~~ ~~4000~~
6000 6000 6000 5000 5000



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 the same as some serial schedule)

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

Fundamental Tradeoff

Stronger isolation
level

Weaker isolation
level



Easier to reason about
(can't see others' changes)

See others' changes,
but more concurrency

Interesting Fact

SQL standard defines serializability as “same as a serial schedule”, but then also lists 3 types of “anomalies” to define levels:

Isolation Level	Dirty Reads	Unrepeatable Reads	Phantom Reads
Read uncommitted	Y	Y	Y
Read committed	N	Y	Y
Repeatable read	N	N	Y
Serializable	N	N	N

Interesting Fact

There are isolation levels other than serializability that meet the last definition!

» I.e. don't exhibit those 3 anomalies

Virtually no commercial DBs do serializability by default, and some can't do it at all

Time to call the lawyers?

In This Course

We'll first cover how to provide serializability, then discuss other levels

- » Many ideas apply to other isolation levels

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

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

T_1 : Read(A)	T_2 : Read(A)
$A \leftarrow A+100$	$A \leftarrow A \times 2$
Write(A)	Write(A)
Read(B)	Read(B)
$B \leftarrow B+100$	$B \leftarrow B \times 2$
Write(B)	Write(B)

Constraint: $A=B$

Schedule A

T_1

Read(A); $A \leftarrow A+100$

Write(A);

Read(B); $B \leftarrow B+100$;

Write(B);

T_2

Read(A); $A \leftarrow A \times 2$;

Write(A);

Read(B); $B \leftarrow B \times 2$;

Write(B);

Schedule A

		A	B
T_1	T_2	25	25
Read(A); $A \leftarrow A+100$			
Write(A);		125	
Read(B); $B \leftarrow B+100$;			125
Write(B);			
	Read(A); $A \leftarrow A \times 2$;		
	Write(A);	250	
	Read(B); $B \leftarrow B \times 2$;		
	Write(B);		250
		250	250

Schedule B

T₁

T₂

Read(A); A ← A+100
Write(A);
Read(B); B ← B+100;
Write(B);

Read(A); A ← A×2;
Write(A);
Read(B); B ← B×2;
Write(B);

Schedule B

T_1	T_2	A	B
		25	25
	Read(A); $A \leftarrow A \times 2$;		
	Write(A);	50	
	Read(B); $B \leftarrow B \times 2$;		
	Write(B);		50
Read(A); $A \leftarrow A + 100$			
Write(A);			
Read(B); $B \leftarrow B + 100$;		150	
Write(B);			
			150
		150	150

Schedule C

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);

Schedule C

T1	T2	A	B
		25	25
Read(A); $A \leftarrow A+100$			
Write(A);		125	
	Read(A); $A \leftarrow A \times 2$;		
	Write(A);	250	
Read(B); $B \leftarrow B+100$;			
Write(B);			125
	Read(B); $B \leftarrow B \times 2$;		
	Write(B);		250
		250	250

Schedule D

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);

Schedule D

T1

 Read(A); $A \leftarrow A+100$
 Write(A);

T2

 Read(A); $A \leftarrow A \times 2$;
 Write(A);
 Read(B); $B \leftarrow B \times 2$;
 Write(B);

Read(B); $B \leftarrow B+100$;
 Write(B);

A	B
25	25
125	
250	
	50
	150
250	150

Schedule E

Same as Schedule D
but with new T2'

T1

Read(A); $A \leftarrow A+100$
Write(A);

Read(B); $B \leftarrow B+100$;
Write(B);

T2'

Read(A); $A \leftarrow A+50$;
Write(A);
Read(B); $B \leftarrow B+50$;
Write(B);

Schedule E

Same as Schedule D
but with new T2'

T1

Read(A); $A \leftarrow A+100$
Write(A);

Read(B); $B \leftarrow B+100$;
Write(B);

T2'

Read(A); $A \leftarrow A+50$;
Write(A);
Read(B); $B \leftarrow B+50$;
Write(B);

A	B
25	25
125	
175	
	75
	175
175	175

Our Goal

Want schedules that are “good”, regardless of

- » initial state and
- » transaction semantics

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

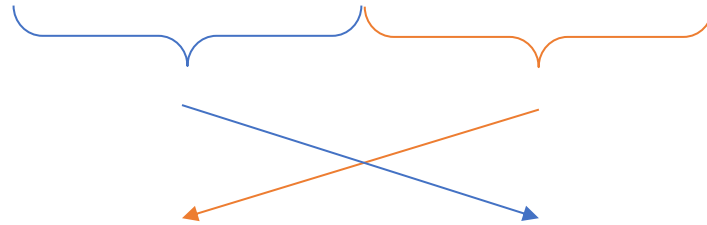
Only look at **order of read & write operations**

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:

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



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



T_1

T_2

However, for S_D :

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

The diagram shows the schedule $S_D = r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B)$. Brackets group the operations into two sets: $\{r_1(A), w_1(A)\}$ and $\{r_2(B), w_2(B)\}$. A blue arrow points from $r_1(B)$ back to $r_2(A)$, and an orange arrow points from $r_2(A)$ back to $r_1(B)$. Both arrows are crossed out with a red 'X', indicating a cycle in the dependency graph.

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

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

Conflicting actions:

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

pairs of actions that would change the result of a read or write if swapped

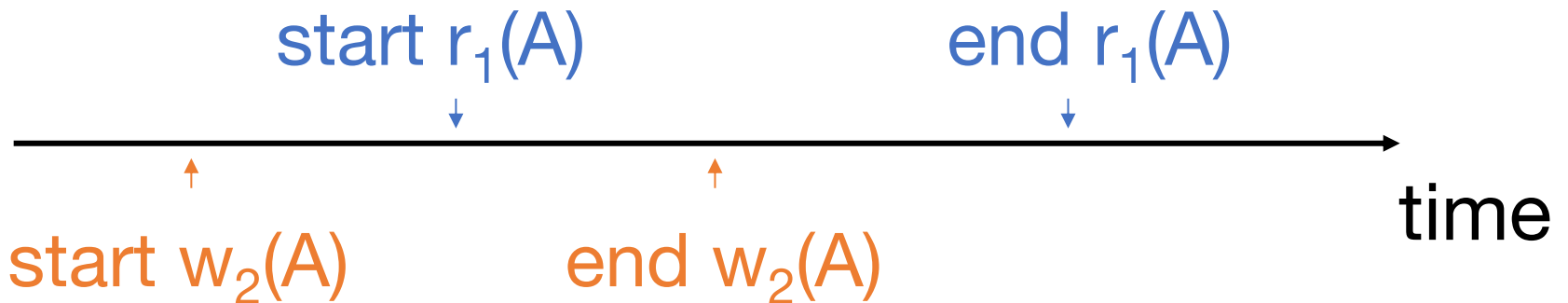
Question

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

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

Question

What about conflicting, concurrent actions on same object?



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

Definition

Schedules S_1 , S_2 are **conflict equivalent** 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 & writes
- » Conflict serializable means there exists at least one serial execution with same 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. $p_i(A)$ and $q_j(A)$ are conflicting actions)

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

$\Rightarrow S_1 = \dots p_i(A) \dots q_j(A) \dots$
 $S_2 = \dots q_j(A) \dots p_i(A) \dots$

$\left\{ \begin{array}{l} p_i, q_j \\ \text{conflict} \end{array} \right.$

$\Rightarrow S_1, S_2$ 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 $\iff S_1$ 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 $\iff S_1$ conflict serializable

(\implies) Assume $P(S_1)$ is acyclic

Transform S_1 as follows:

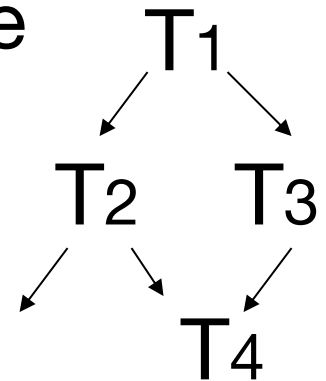
(1) Take T_1 to be transaction with no inbound edges

(2) Move all T_1 actions to the front

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


(3) we now have $S_1 = \langle T_1 \text{ actions} \rangle \langle \dots \text{ rest } \dots \rangle$

(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

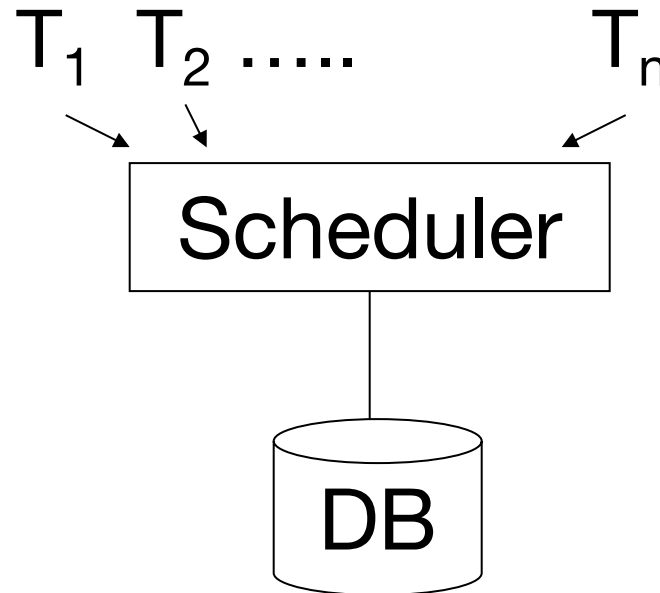
Optimistic concurrency with validation

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

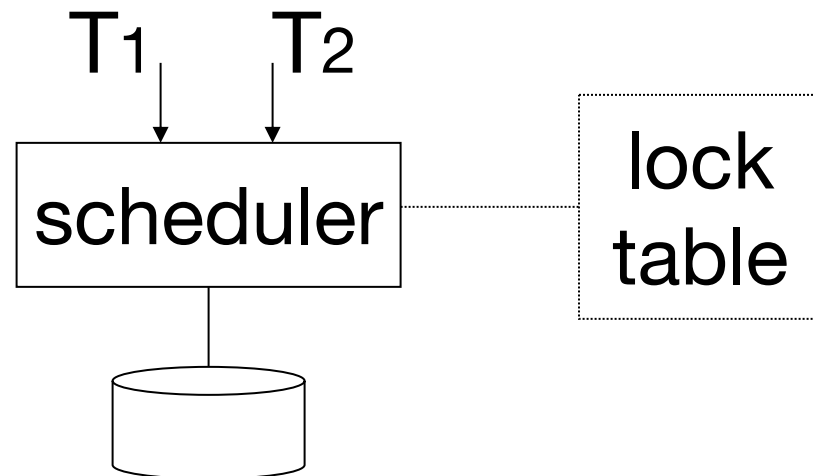


A Locking Protocol

Two new actions:

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

unlock: $u_i(A)$



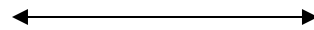
Rule #1: Well-Formed Transactions

$T_i: \dots l_i(A) \dots r_i(A) \dots u_i(A) \dots$

Transactions can only operate on locked items

Rule #2: Legal Scheduler

$S = \dots\dots\dots I_i(A) \dots\dots\dots u_i(A) \dots\dots\dots$



no $I_j(A)$

Only one transaction can lock item at a time

Exercise

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

$$S_1 = I_1(A) I_1(B) r_1(A) w_1(B) I_2(B) u_1(A) u_1(B) \\ r_2(B) w_2(B) u_2(B) I_3(B) r_3(B) u_3(B)$$

$$S_2 = I_1(A) r_1(A) w_1(B) u_1(A) u_1(B) I_2(B) r_2(B) \\ w_2(B) I_3(B) r_3(B) u_3(B)$$

$$S_3 = I_1(A) r_1(A) u_1(A) I_1(B) w_1(B) u_1(B) I_2(B) \\ r_2(B) w_2(B) u_2(B) I_3(B) r_3(B) u_3(B)$$

Exercise

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

$S_1 = I_1(A) I_1(B) r_1(A) w_1(B) I_2(B) u_1(A) u_1(B)$
 $r_2(B) w_2(B) u_2(B) I_3(B) r_3(B) u_3(B)$

$S_2 = I_1(A) r_1(A) w_1(B) u_1(A) u_1(B) I_2(B) r_2(B)$
 $w_2(B) I_3(B) r_3(B) u_3(B) \quad u_2(B) \text{ missing}$

$S_3 = I_1(A) r_1(A) u_1(A) I_1(B) w_1(B) u_1(B)$
 $I_2(B) r_2(B) w_2(B) u_2(B) I_3(B) r_3(B) u_3(B)$

Schedule F

		A	B
T1	T2	25	25
$I_1(A); \text{Read}(A)$			
$A \leftarrow A + 100; \text{Write}(A); u_1(A)$		125	
	$I_2(A); \text{Read}(A)$		
	$A \leftarrow A \times 2; \text{Write}(A); u_2(A)$	250	
	$I_2(B); \text{Read}(B)$		
	$B \leftarrow B \times 2; \text{Write}(B); u_2(B)$		50
$I_1(B); \text{Read}(B)$			
$B \leftarrow B + 100; \text{Write}(B); u_1(B)$			150
		250	150

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

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

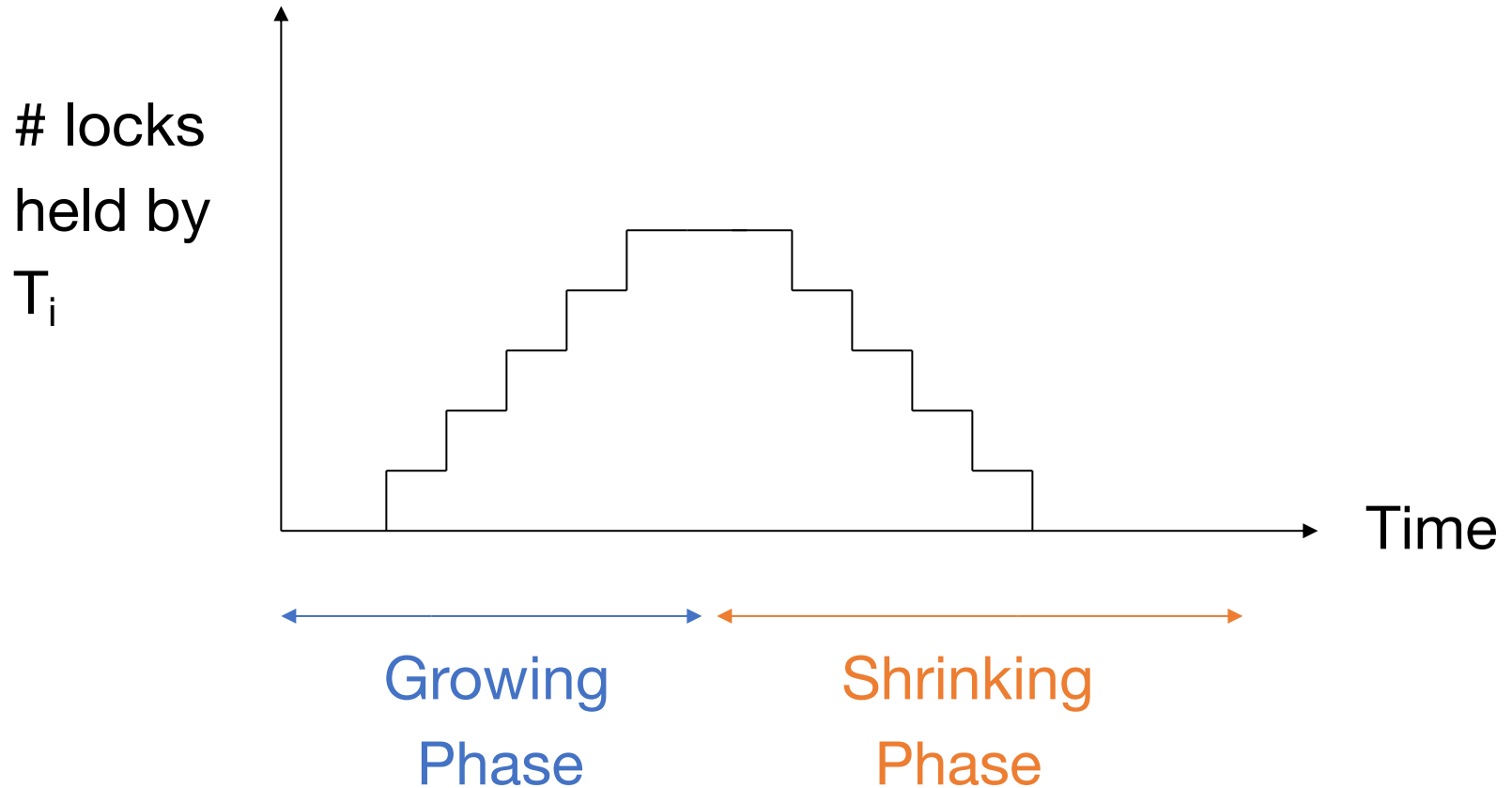


no unlocks

no locks

Transactions must first lock all items they need, then unlock them

2-Phase Locking (2PL)



Schedule G

T1	T2
$I_1(A); \text{Read}(A)$	
$A \leftarrow A + 100; \text{Write}(A)$	
$I_1(B); u_1(A)$	

Schedule G

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

Schedule G

T1	T2
$I_1(A); \text{Read}(A)$	
$A \leftarrow A + 100; \text{Write}(A)$	
$I_1(B); u_1(A)$	
	$I_2(A); \text{Read}(A)$
	$A \leftarrow A \times 2; \text{Write}(A)$
	$I_2(B) \leftarrow \text{delayed}$
$\text{Read}(B); B \leftarrow B + 100$	
$\text{Write}(B); u_1(B)$	

Schedule G

T1	T2
$I_1(A); \text{Read}(A)$	
$A \leftarrow A + 100; \text{Write}(A)$	
$I_1(B); u_1(A)$	
	$I_2(A); \text{Read}(A)$
	$A \leftarrow A \times 2; \text{Write}(A)$
	$I_2(B) \leftarrow \text{delayed}$
$\text{Read}(B); B \leftarrow B + 100$	
$\text{Write}(B); u_1(B)$	
	$I_2(B); u_2(A); \text{Read}(B)$
	$B \leftarrow B \times 2; \text{Write}(B); u_2(B)$

Schedule H (T2 Ops Reversed)

T1	T2
$I_1(A)$; Read(A)	$I_2(B)$; Read(B)
$A \leftarrow A + 100$; Write(A)	$B \leftarrow B \times 2$; Write(B)
$I_1(B)$ ← delayed (T2 holds B)	$I_2(A)$ ← delayed (T1 holds A)

Problem: Deadlock between the 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 $\langle u_i(A), u_j(A) \rangle, \langle l_i(A), r_j(A) \rangle, \dots$

Theorem

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

To help in proof:

Definition: $\text{Shrink}(T_i) = \text{SH}(T_i) =$
first unlock action of T_i

Lemma

$$T_i \rightarrow T_j \text{ in } S \Rightarrow SH(T_i) <_S SH(T_j)$$

Proof:

$T_i \rightarrow T_j$ means that

$S = \dots p_i(A) \dots q_j(A) \dots$; p, q conflict

By rules 1, 2:

$S = \dots p_i(A) \dots u_i(A) \dots l_j(A) \dots q_j(A) \dots$



By rule 3: $SH(T_i)$ $SH(T_j)$

So, $SH(T_i) <_S SH(T_j)$

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

Proof:

(1) Assume $P(S)$ has cycle

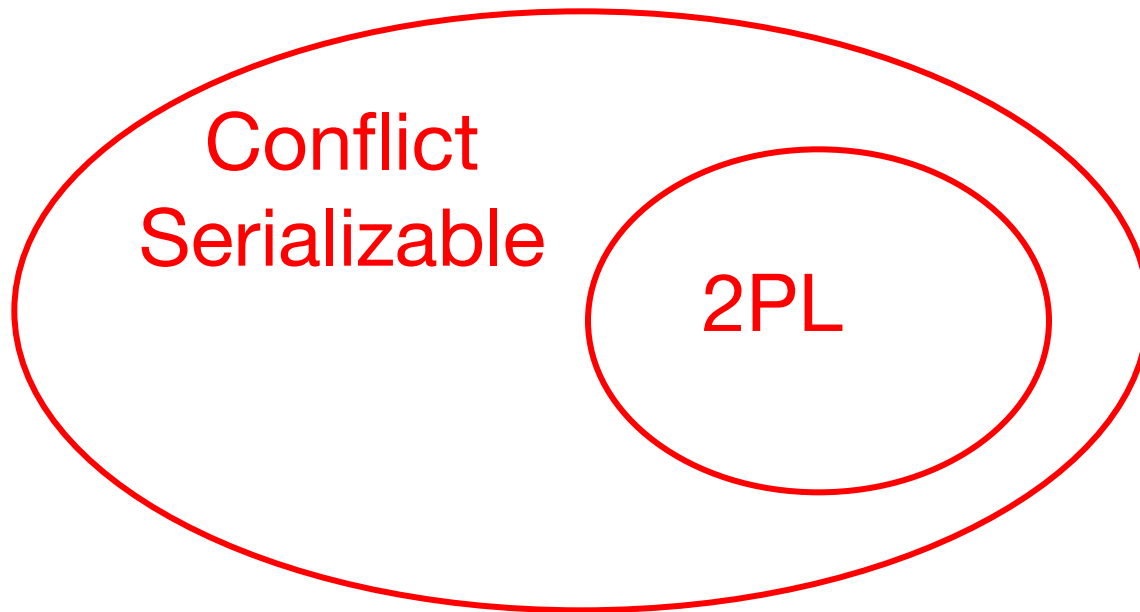
$$T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$$

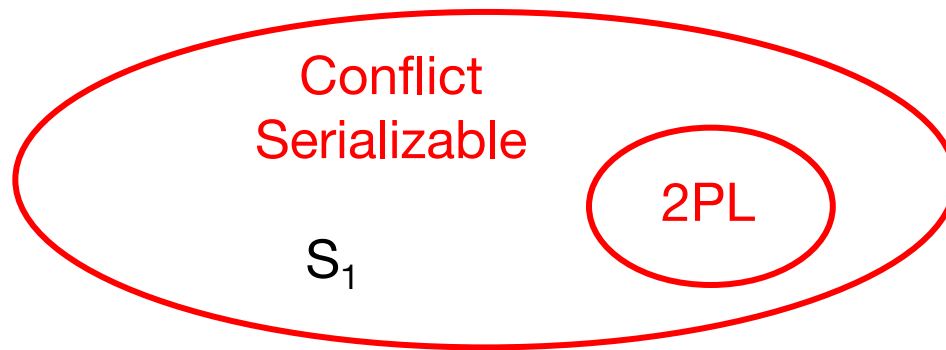
(2) By lemma: $SH(T_1) < SH(T_2) < \dots < SH(T_1)$

(3) Impossible, so $P(S)$ acyclic

(4) $\Rightarrow S$ is conflict serializable

2PL is a Subset of Conflict Serializable





S₁: w₁(X) w₃(X) w₂(Y) w₁(Y)

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

But S₁ cannot be achieved via 2PL:

- » The lock by T₁ for Y must occur after w₂(Y), so the unlock by T₁ for X must occur after this point (and before w₁(X)). Thus, w₃(X) cannot occur under 2PL where shown in S₁.

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

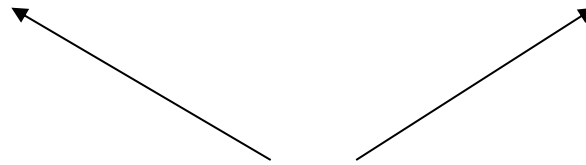
Beyond this simple 2PL protocol, many ways to improve performance & concurrency:

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

Shared Locks

So far:

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

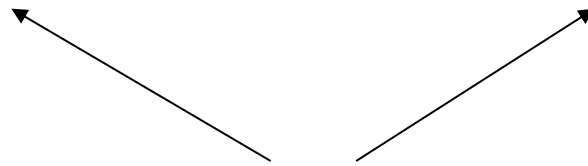


Do not conflict

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?

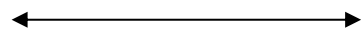
Option 2: Upgrade lock to X on write

$T_1 = \dots l-S_1(A) \dots r_1(A) \dots l-X_1(A) \dots w_1(A) \dots u_1(A) \dots$

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

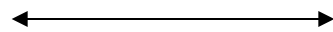
Rule 2: Legal Scheduler

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



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:

(I) If upgrade gets more locks

(e.g., $S \rightarrow \{S, X\}$) then no change!

(II) If upgrade releases read lock (e.g., $S \rightarrow X$)

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 $\text{compat}(m,n)$

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

Lock Modes Beyond S/X

Examples:

(1) increment lock

(2) update lock

Example 1: Increment Lock

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

compat

New request

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

Lock already held in

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 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

compat

New request

Lock already held in

	S	X	U
S	T	F	
X	F	F	
U			

Compatibility Matrix

compat

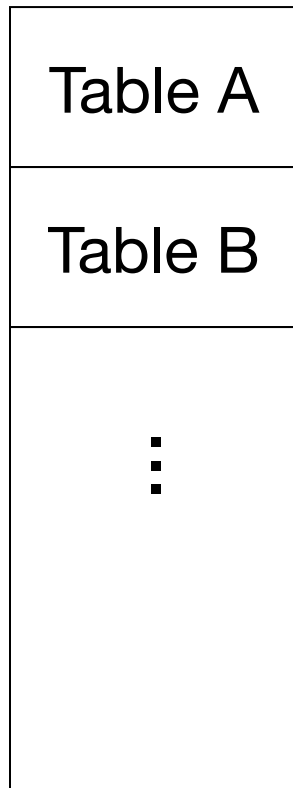
New request

Lock already held in

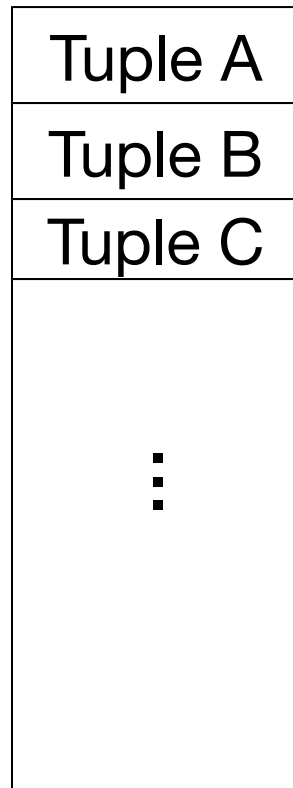
	S	X	U
S	T	F	T
X	F	F	F
U	F	F	F

Note: asymmetric table!

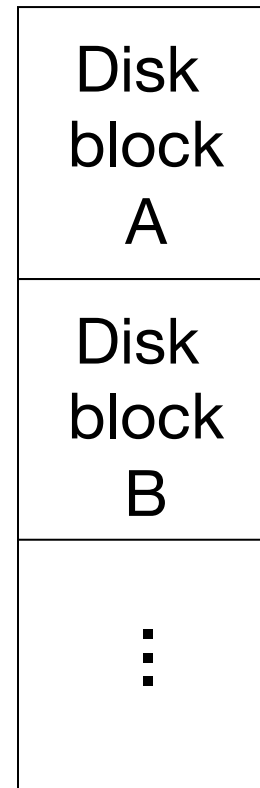
Which Objects Do We Lock?



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)

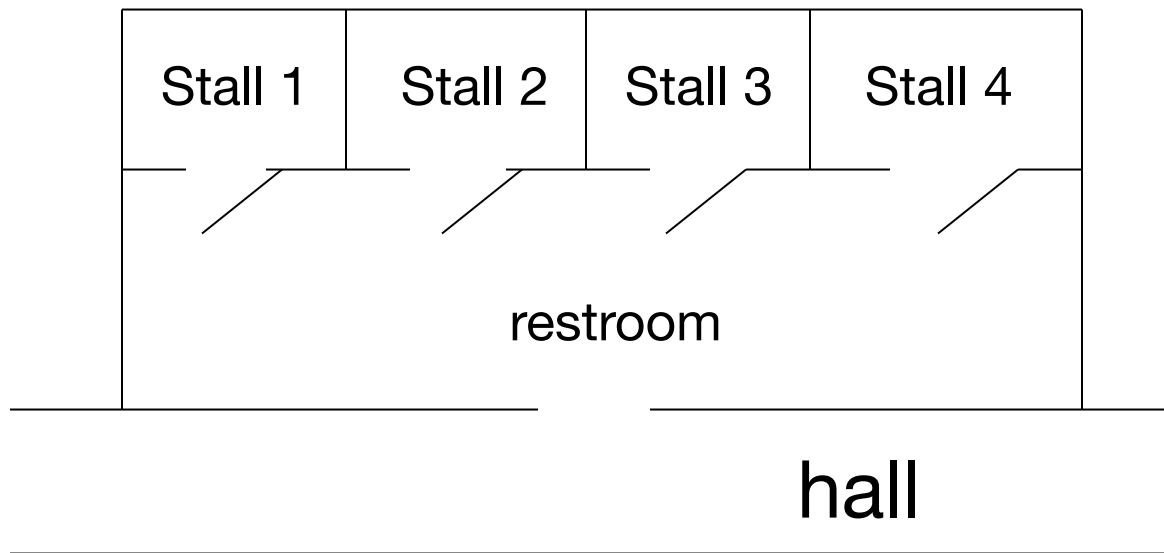
- Need few locks
- Low concurrency

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

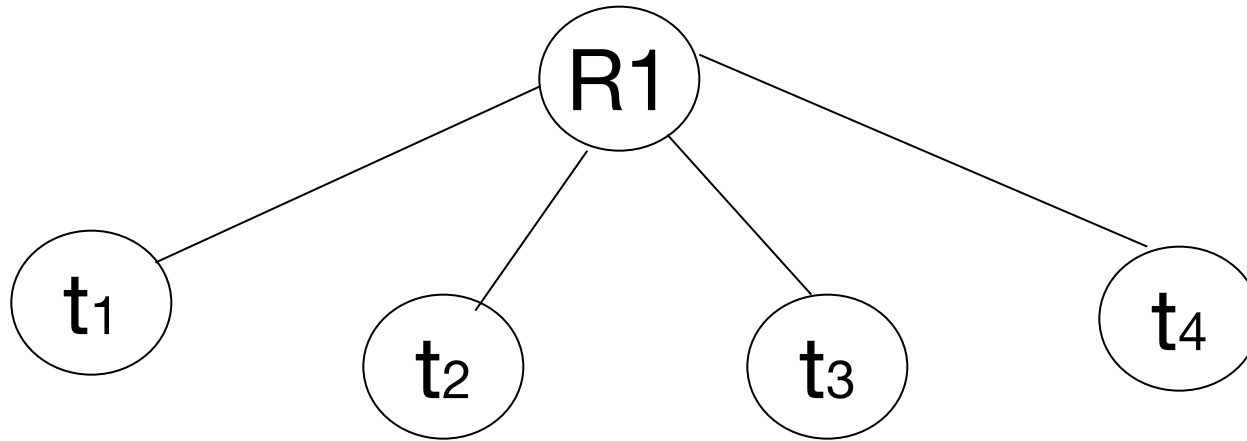
- Need more locks
- More concurrency

We Can Have It Both Ways!

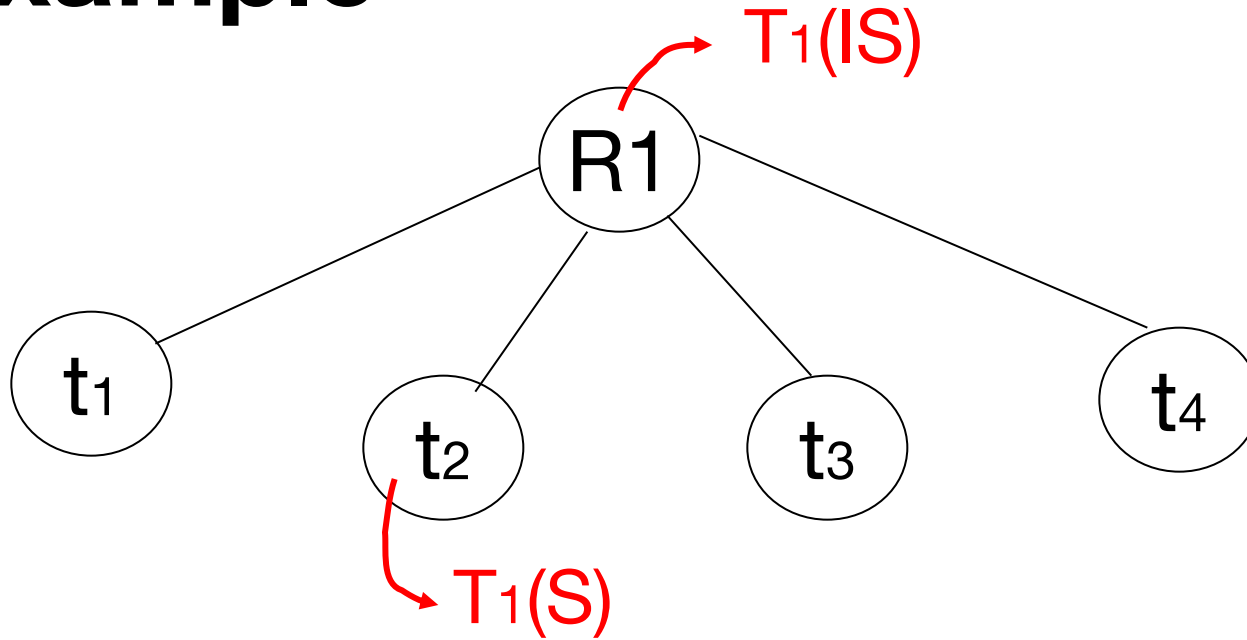
Ask any janitor to give you the solution...



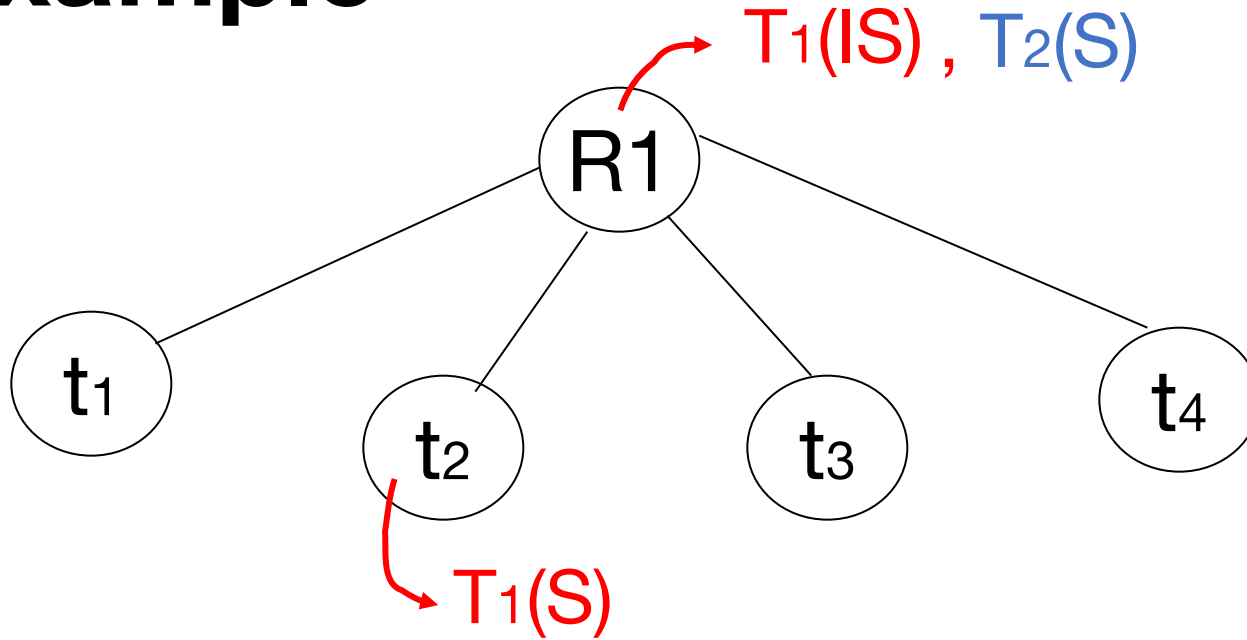
Example



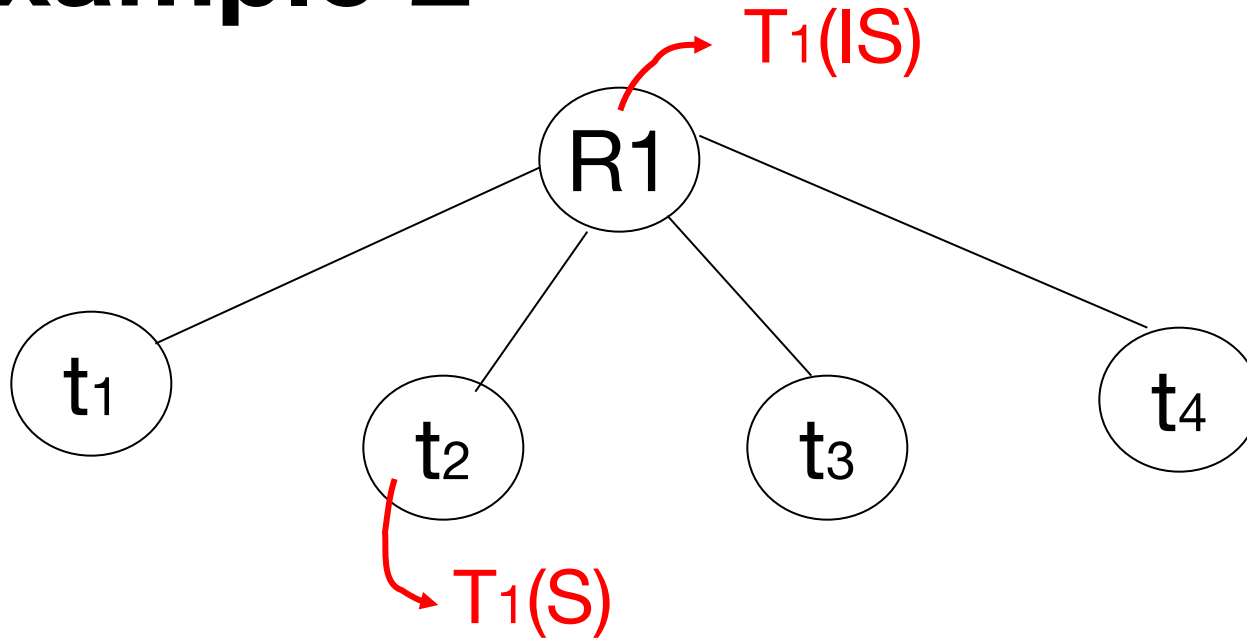
Example



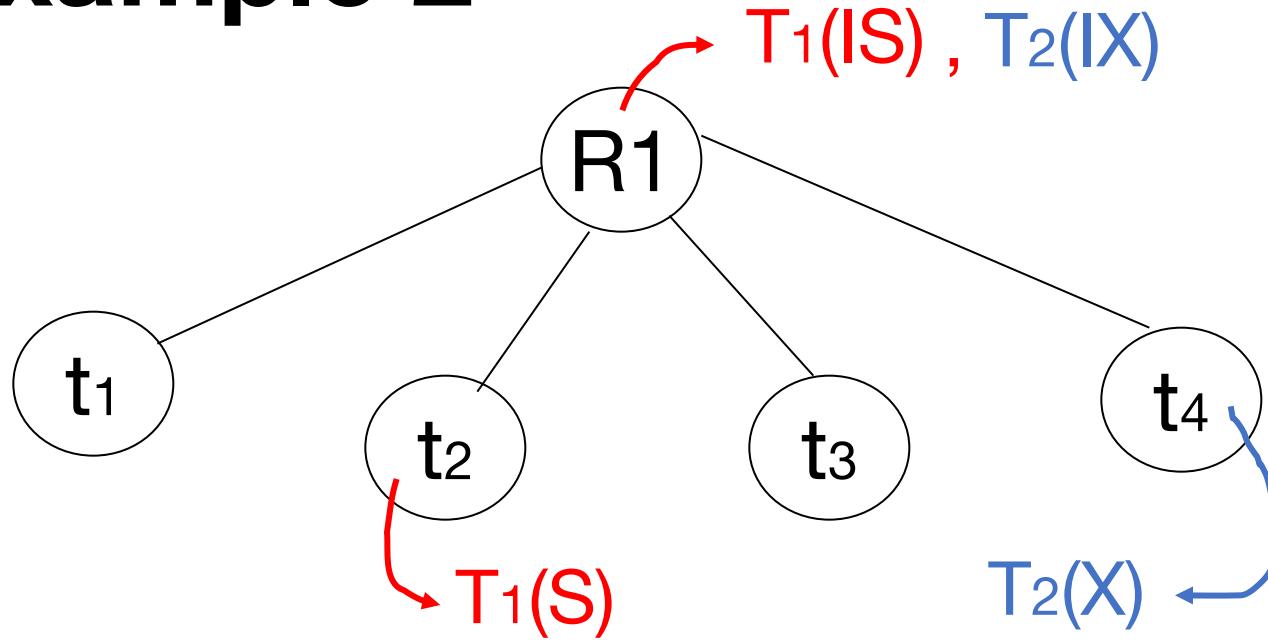
Example



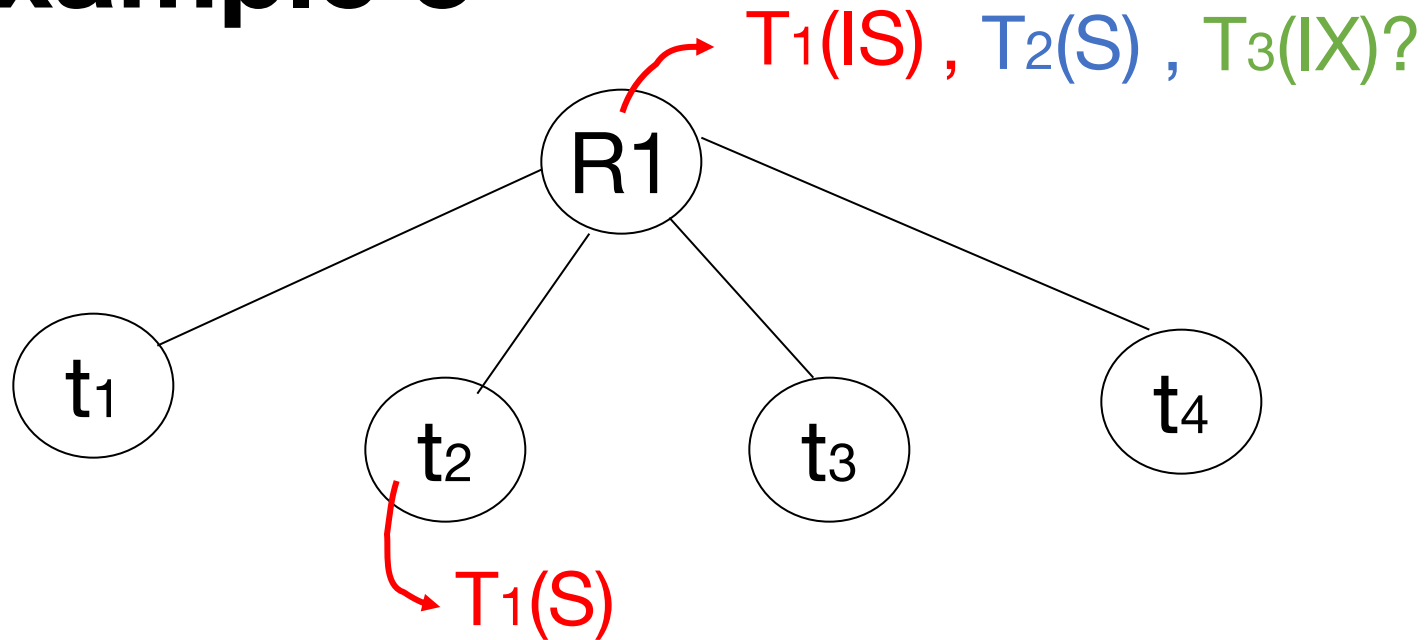
Example 2



Example 2



Example 3



Multiple Granularity Locks

compat

Requester

IS IX S SIX X

Holder

	IS	IX	S	SIX	X
IS					
IX					
S					
SIX					
X					

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