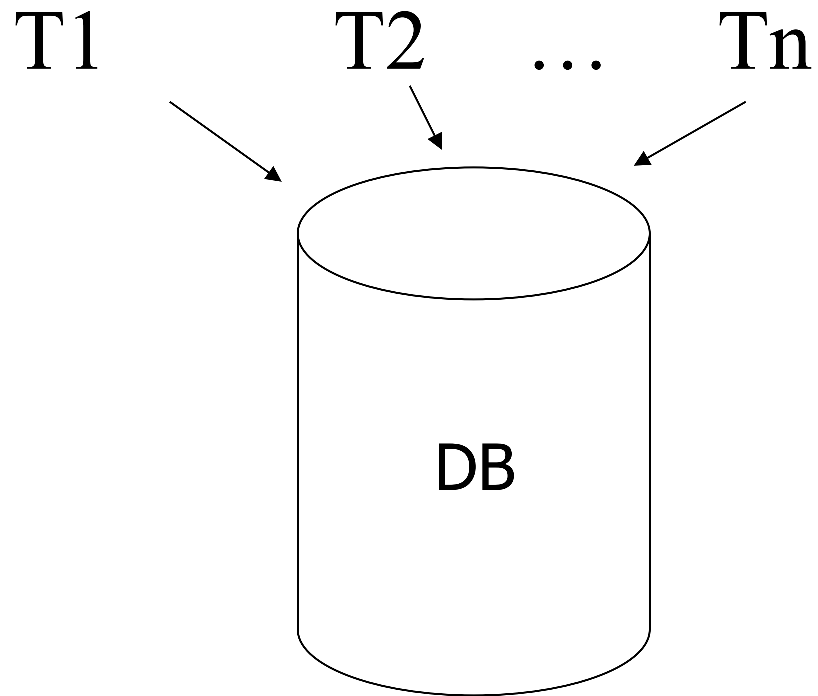
Concurrency Control

Concurrency Control



Overall Goal/Outline

- Problems that can arise in “interleaving” different transactions.

  Interleaving = Schedule.
- Define what are GOOD schedules.
- Develop a mechanism to ensure only good schedules happen.
- Optimizations.

Example:

T1:	T2:
Read(A)	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)

Schedule A

"Serial" Schedule

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

Schedule B

Another "Serial" Schedule

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

Schedule C

"Good" Schedule, because it has the same effect on DB as a serial schedule

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

"Bad" Schedule, because it has a different effect on DB

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

Schedule E

Same as Schedule D but with new T2'

Looks "good" now.

T1	T2'	A	B
		25	25
Read(A); A ← A+100 Write(A);		125	
	Read(A);A ← A×1; Write(A);	125	
	Read(B);B ← B×1; Write(B);		25
Read(B); B ← B+100; Write(B);			125
		125	125

What is a Good Schedule?

- Informally, what is "good" schedule?
  - Be "equivalent" to an "isolated" (serial) execution
- Also, we want the "good" definition to be independent of
  - initial state and
  - transaction semantics

  to simply the scheduler's job.

➔ Define good schedule only in terms of the order of reads and writes [Thus, we will *conservatively* reject schedule E (last slide), even though it seems good.]

Example of a Good Schedule

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

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

Sc is good, because it is "equivalent" (shown through swap-transformation) to a "serial" schedule Sc' .

Concepts

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

Conflicting actions: $r_1(A)$ / $w_2(A)$ $w_2(A)$ / $r_1(A)$ $w_1(A)$ / $w_2(A)$

Schedule: represents chronological order in which actions are executed

Serial schedule: no interleaving of actions or transactions

Next: Equivalent and Conflict-Equivalent.

Equivalent, Serializable

- Equivalent: Two schedules S1 and S2 are equivalent if they have the same effect on the database.

  $S1 = w_1(B)\ w_2(B)\ w_3(B)$

  $S2 = w_2(B)\ w_1(B)\ w_3(B)$

- Serializable schedule: Equivalent to a serial schedule. Above, S2 is serializable.

Conflict Equivalent/Serializable

- Conflict-Equivalent:

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

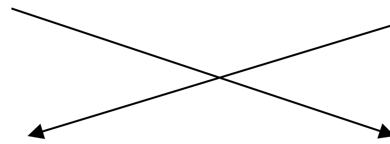
  Conflict equivalent $\Rightarrow$ equivalent

  (not vice-versa; counter example later)

- Conflict-Serializable schedule: Conflict-equivalent to a serial schedule.

Conflict-Equivalent Example

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



$Sc' = r_1(A)w_1(A)\ r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$

T_1 T_2

Thus, Sc and Sc' are conflict-equivalent, and

Sc is conflict-serializable.

Conflict-Equivalent vs. Equivalent

- Conflict equivalent $\Rightarrow$ equivalent

  (not vice-versa)

  S1 = $w_1(B)\ w_2(B)\ w_3(B)$

  S2 = $w_2(B)\ w_1(B)\ w_3(B)$

  S1 and S2 are equivalent, but not CE.

- Thus, conflict-serializable $\Rightarrow$ serializable.

What next?

- Formal definitions of “good” (i.e., conflict-serializable) schedules.
- How does the system test for conflict-serializability?
  1. Consider all possible sequences of swaps, and see we get a serial schedule. Unrealistic.
  2. Draw a graph with edges corresponding to action-pairs that can’t be swapped?

Test for Conflict-Equivalence (CE)

- Two schedules are not CE, if transformation through swaps is NOT possible.
  - Maybe, we can build a graph, with edges for actions that cannot be swapped?
- Define a "Precedence Graph" P(S) for a schedule S. Two theorems:
  1. If S1 and S2 are CE, then P(S1) = P(S2).
  2. P(S1) is acyclic iff S1 is conflict-serializable.

Precedence graph P(S) (S is schedule)

Nodes: Transactions in S

Arcs: Ti → Tj whenever

- $p_i(A)$, $q_j(A)$ are actions in S
- $p_i(A) <_S q_j(A)$
- at least one of p_i, q_j is a write

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 conflict-serializable?

Lemma

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

Proof:

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

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

$\Rightarrow \quad S_1 = \ldots p_i(A) \ldots q_j(A) \ldots$

$\quad\quad S_2 = \ldots q_j(A) \ldots p_i(A) \ldots$

p_i, q_j conflict

$\Rightarrow \quad S_1, S_2$ are not conflict equivalent

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

Counter example:

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

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

Theorem

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

($\Leftarrow$) Assume S_1 is conflict serializable

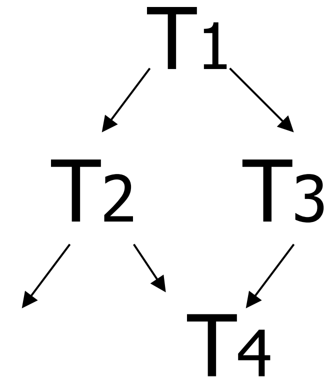
$\Rightarrow$ $\exists\, S_s$: S_s, S_1 conflict equivalent

$\Rightarrow$ $P(S_s) = P(S_1)$

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

Theorem

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



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

Transform S_1 as follows:

(1) Take T_1 to be transaction with no incident arcs

(2) Move all T_1 actions to the front

S_1 = ……. $q_j(A)$…….$p_1(A)$…..

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

(4) repeat above steps to serialize rest!

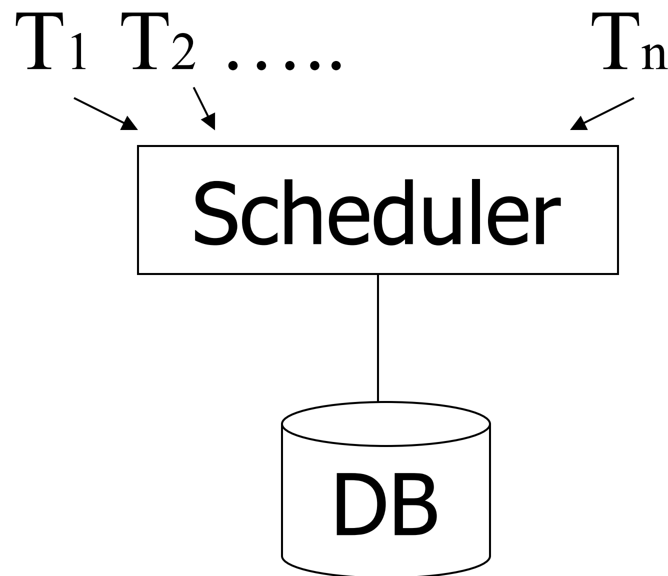
How to enforce serializable schedules?

Option 1:

run system, recording P(S); at end of day, check for P(S) cycles and declare if execution was good

How to enforce serializable schedules?

Option 2: prevent P(S) cycles from occurring



Recap/Outline

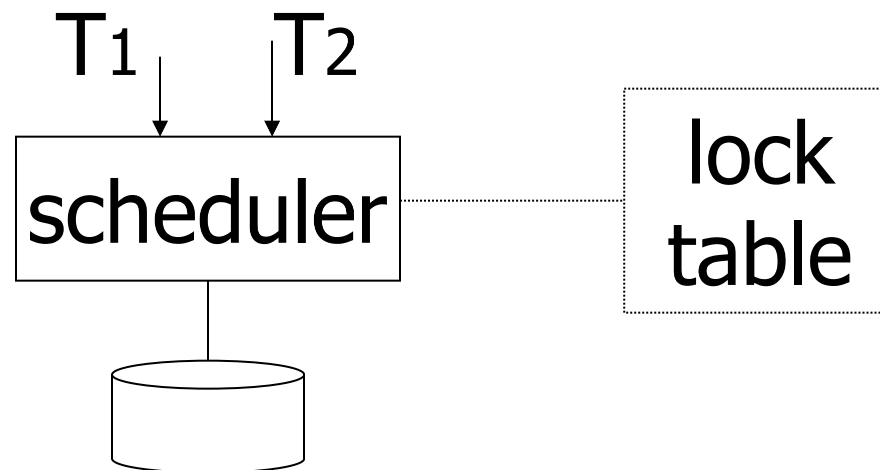
- "Conflict-Serializability"
  - Motivation; Precedence Graph test.
- Locking
  - Read/Write locks. Two-Phase Locking (2PL).
  - 2PL => Conflict Serializability
  - Shared/Upgrade, Update, Increment.
  - Simple Locking System, and Lock Table
- Lock Hierarchies (Warning Protocol).

A locking protocol

Two new actions:

lock: l_i (A)

unlock: u_i (A)



Rule #1: Well-formed transactions

T_i: … $l_i(A)$ … $p_i(A)$ … $u_i(A)$ ...

Rule #2: Legal scheduler

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

no $l_j(A)$

Exercise:

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

S1 = $l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
$r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

S2 = $l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$
$l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

S3 = $l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
$l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Exercise:

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

S1 = $l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
$r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

S2 = $l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$
$l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

S3 = $l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
$l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Schedule F

T1	T2
$l_1(A)$;Read(A)	
A ← A+100;Write(A);$u_1(A)$	
	$l_2(A)$;Read(A)
	A ← Ax2;Write(A);$u_2(A)$
	$l_2(B)$;Read(B)
	B ← Bx2;Write(B);$u_2(B)$
$l_1(B)$;Read(B)	
B ← B+100;Write(B);$u_1(B)$	

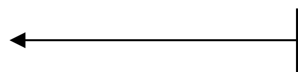
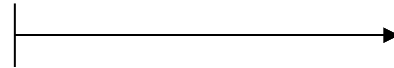
Schedule F

Rules #1 and #2 not sufficient to guarantee conflict-serializability.

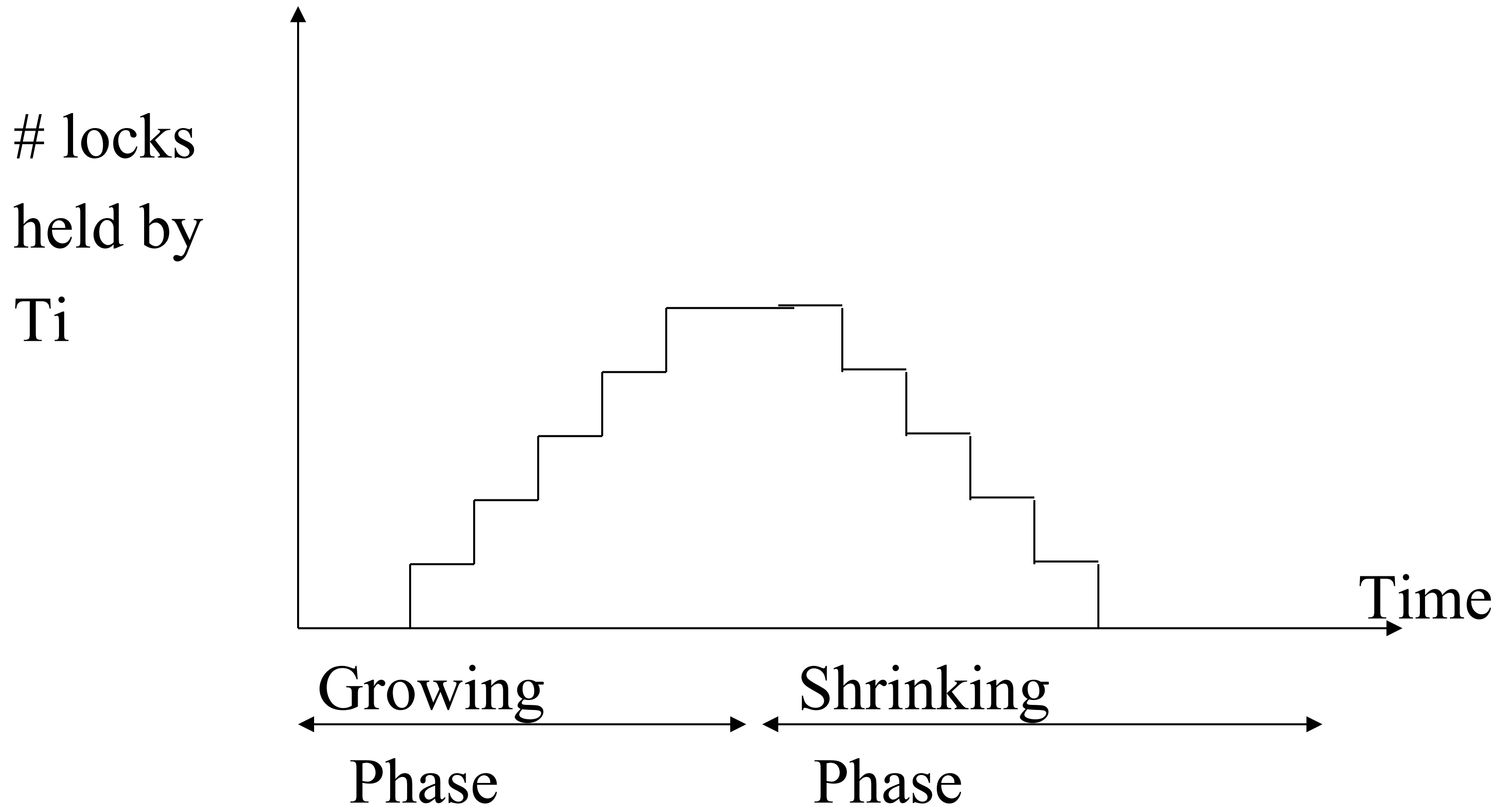
T1	T2	A	B
		25	25
$l_1(A)$;Read(A)			
A←A+100;Write(A);$u_1(A)$		125	
	$l_2(A)$;Read(A)		
	A←Ax2;Write(A);$u_2(A)$	250	
	$l_2(B)$;Read(B)		
	B← Bx2;Write(B);$u_2(B)$		50
$l_1(B)$;Read(B)			
B← B+100;Write(B);$u_1(B)$			150
		250	150

Rule #3 Two phase locking (2PL)

T_i = ……. $l_i(A)$ ………… $u_i(A)$ ………

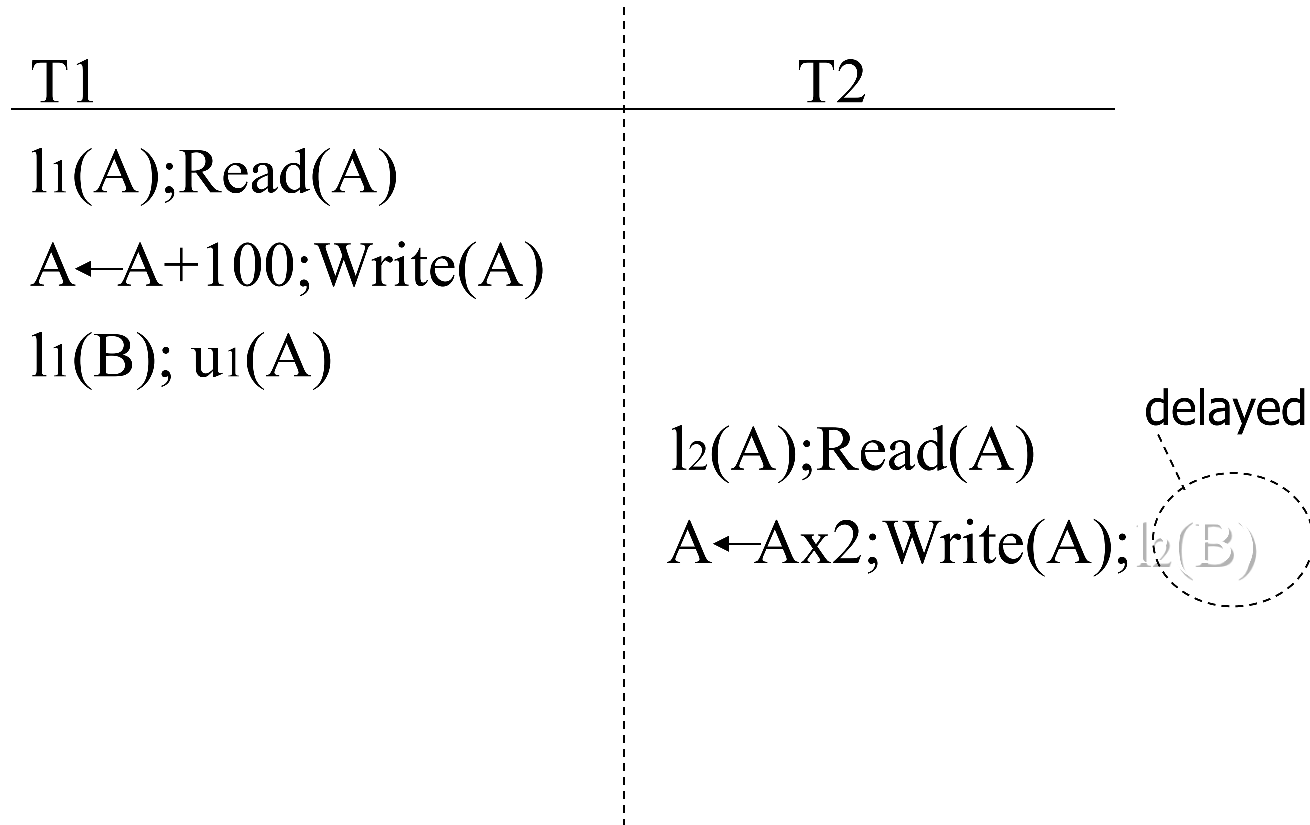
no unlocks   no locks



locks
held by
Ti
Time
Growing
Phase
Shrinking
Phase

Schedule G

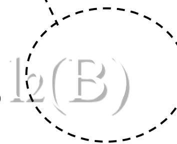
Enforcing 2PL generates a conflict-serializable schedule.



Schedule G

Enforcing 2PL generates a conflict-serializable schedule.

T1	T2
$l_1(A)$;Read(A)	
$A \leftarrow A+100$;Write(A)	
$l_1(B)$; $u_1(A)$	
	$l_2(A)$;Read(A)
	$A \leftarrow Ax2$;Write(A); $l_2(B)$
Read(B);$B \leftarrow B+100$	
Write(B); $u_1(B)$	



delayed

Schedule G

Enforcing 2PL generates a conflict-serializable schedule.

T1	T2
$l_1(A)$;Read(A)	
$A \leftarrow A+100$;Write(A)	
$l_1(B)$; $u_1(A)$	
	$l_2(A)$;Read(A)
	$A \leftarrow Ax2$;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)
	$B \leftarrow Bx2$;Write(B);$u_2(B)$;

Himanshu Gupta

Recap/Outline

- "Conflict-Serializability"
  - Motivation; Precedence Graph test.
- Locking
  - Read/Write locks. Two-Phase Locking (2PL).
  - 2PL => Conflict Serializability
  - Shared/Upgrade, Increment, Update.
  - Simple Locking System, and Lock Table
- Lock Hierarchies (Warning Protocol).

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

To help in proof, define:

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

Lemma

$Ti \rightarrow Tj$ in S $\Rightarrow$ SH(Ti) $<_S$ SH(Tj)

Proof of lemma:

$Ti \rightarrow Tj$ 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 $\Longrightarrow$ conflict-serializable schedule

Proof:

(1) Assume P(S) has cycle

$$T_1 \rightarrow T_2 \rightarrow \ldots T_n \rightarrow T_1$$

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

(3) Impossible, so P(S) acyclic

(4) $\Rightarrow$ S is conflict serializable

- Beyond this (simple 2PL), 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

Recap/Outline

- “Conflict-Serializability”
  - Motivation; Precedence Graph test.
- Locking
  - Read/Write locks. Two-Phase Locking (2PL).
  - 2PL => Conflict Serializability
  - Shared/Upgrade, Increment, Update.
  - Simple Locking System, and Lock Table
- Lock Hierarchies (Warning Protocol).

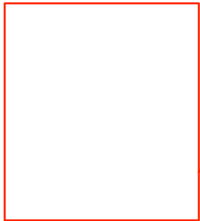
Motivation for Shared Locks

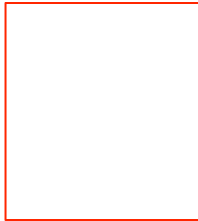
- Give a conflict-serializable schedule that can not be generated by a 2PL protocol.

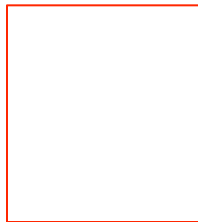
- Thus, 2PL is too restrictive.
- Fix: Introduce "shared" locks.

Shared locks

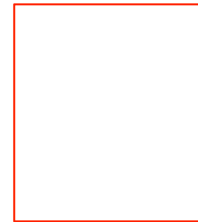
Read blocks from different transactions

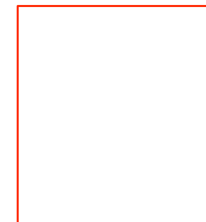


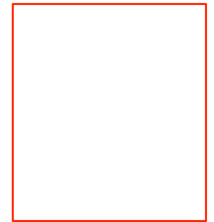




Execute them concurrently







Shared locks (contd)

So far:

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

Instead:

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

Lock actions

$l\text{-}t_i(A)$: lock A in t mode (t is S or X)

$u\text{-}t_i(A)$: unlock t mode (t is S or X)

Shorthand:

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

Rule #1 Well formed transactions

T_i =... l-S/X$_1$(A) … r$_1$(A) … u$_1$ (A) …

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

- What about transactions that read and write same object?
- Two options:
  1. Request exclusive lock (conservative):

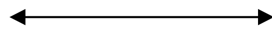
     T_i = ...l-X_1(A) … r_1(A) ... w_1(A) ... u(A)
  2. Upgrade (more efficient)

     Ti = ...l-S1(A) … r1(A) ... l-X1(A) …w1(A) ... u(A)

One can imagine a "downgrade" operation too – but, we don't consider it for simplicity.

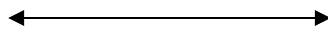
Rule #2 Legal scheduler

S = ...l-S_i(A) … … u_i(A) …



no l-X_j(A)

S = ... l-X_i(A) … … u_i(A) …



no l-X_j(A)
no l-S_j(A)

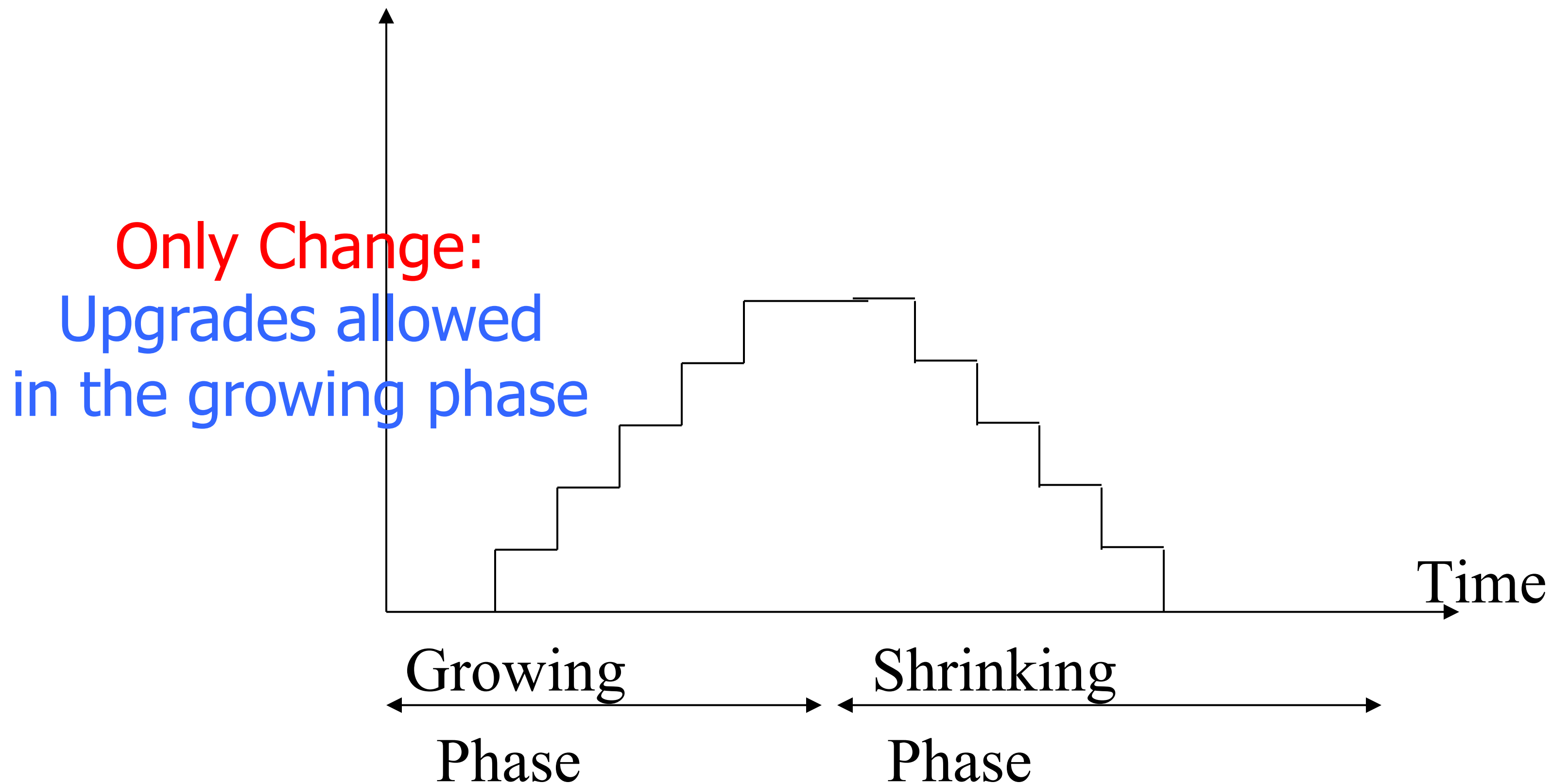
A way to summarize Rule #2

Compatibility matrix

Comp

	S	X
S	true	false
X	false	false

Rule # 3 (2PL) for Shared/Exclusive Locks



Theorem: Rules 1,2,3 for S/X locks $\Rightarrow$ Conf. serializable schedules

Proof: similar to X locks case

Lock types beyond S/X

Examples:

(1) increment lock

(2) update lock

Example (1): increment lock

- Atomic increment action: $IN_i(A)$

  {Read(A); A ← A+k; Write(A)}
- $IN_i(A)$, $IN_j(A)$ are "commutative":
  - Since {INi(A), INj(A)}and {INi(A), INj(A)}will have the same effect on database.
- Thus, we can define "increment" locks which are allowed to be "concurrent."

Comp

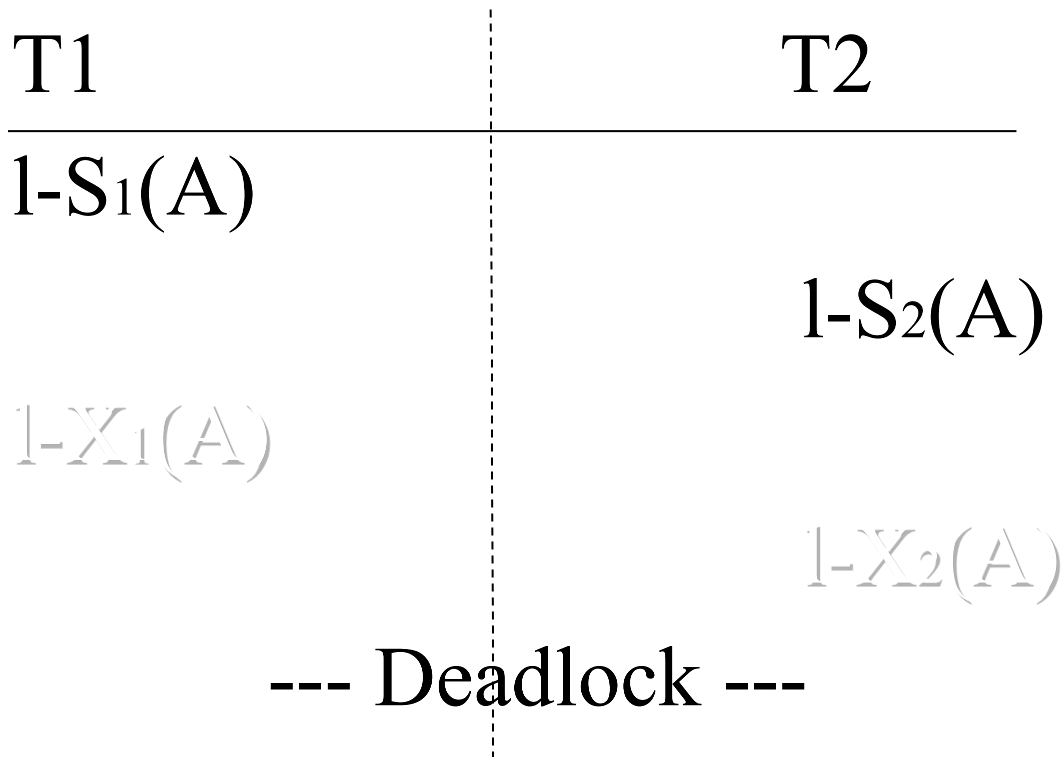
	S	X	I
S			
X			
I			

Comp

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

Example (2): Update locks

A common deadlock problem with upgrades:



Solution

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

Comp

New request

Lock already held in

	S	X	U
S			
X			
U			

New request

Comp

Lock already held in

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

Also, S → X upgrade not allowed.

Recap/Outline

- "Conflict-Serializability"
  - Motivation; Precedence Graph test.
- Locking
  - Read/Write locks. Two-Phase Locking (2PL).
  - 2PL => Conflict Serializability
  - Shared/Upgrade, Increment, Update.
  - Simple Locking System, and Lock Table
- Lock Hierarchies (Warning Protocol).

How does locking work in practice?

- Every system is different

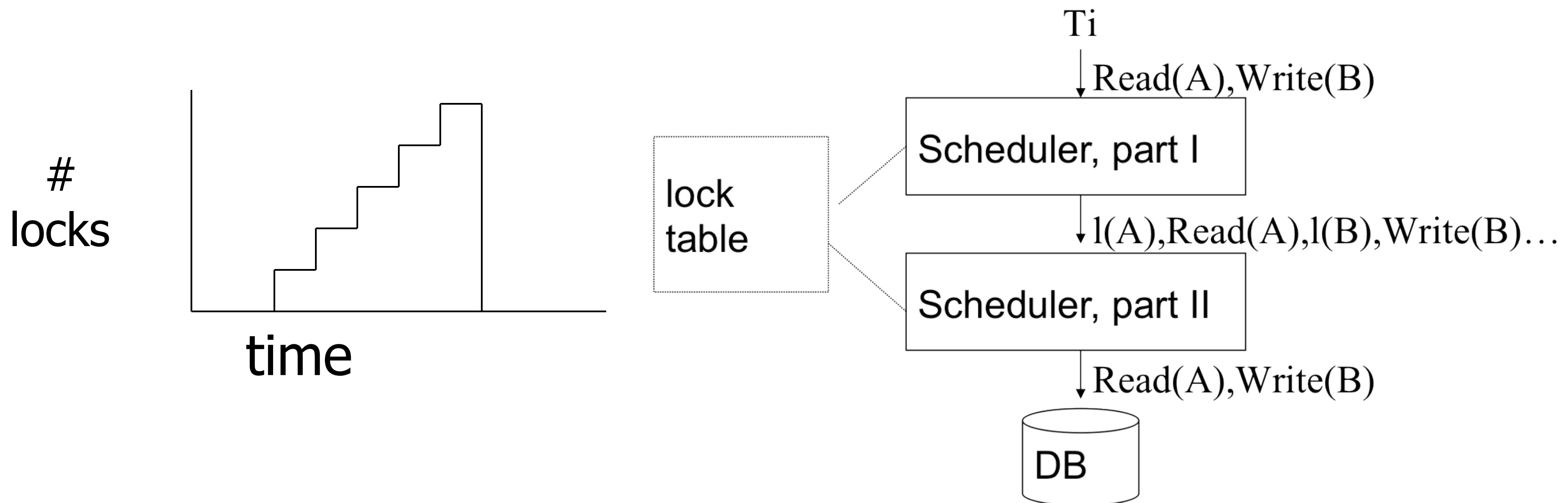
  (E.g., may not even provide
  CONFLICT-SERIALIZABLE schedules)

- But here is one (simplified) way ...

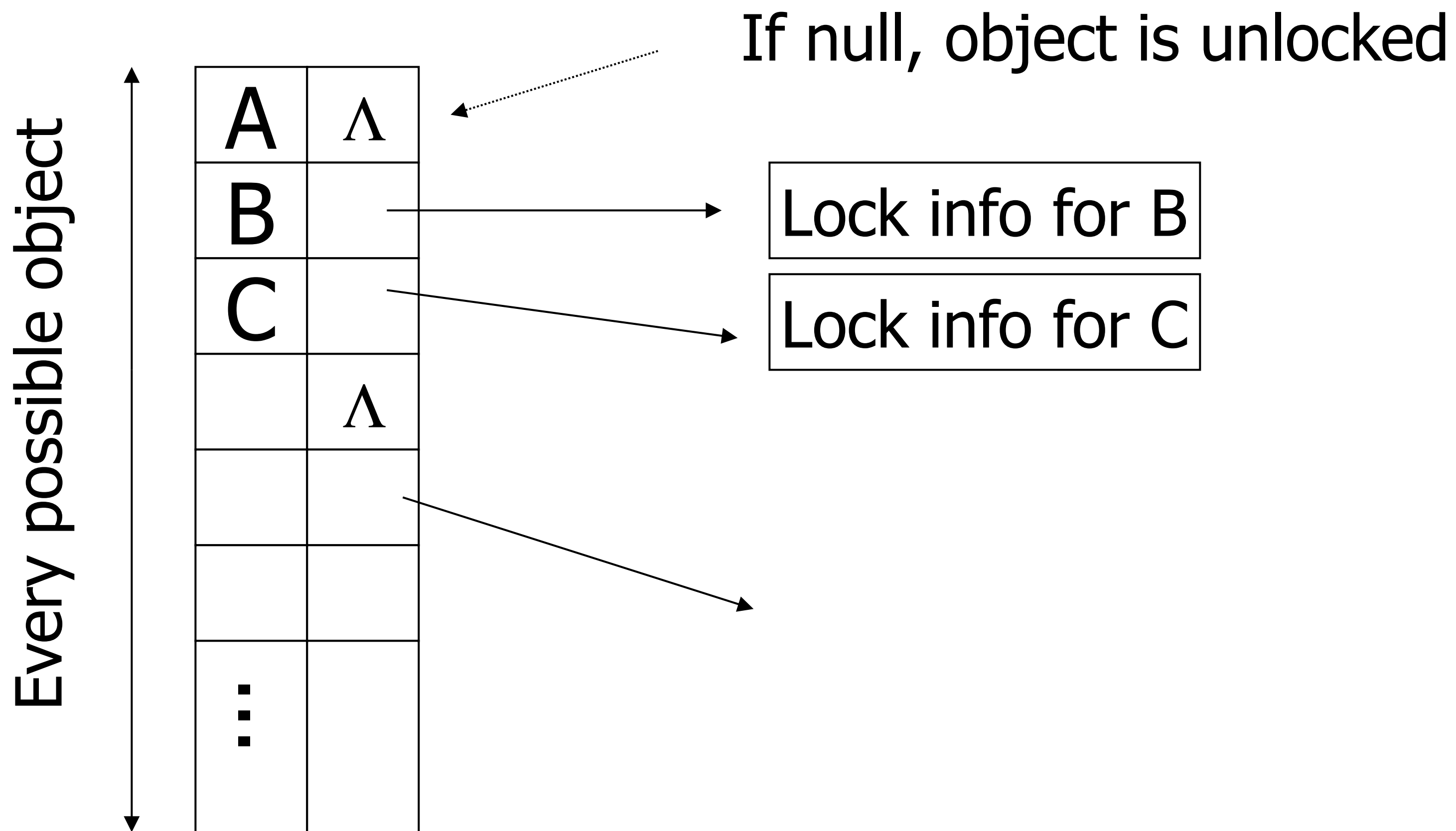
Himanshu Gupta

Sample Locking System:

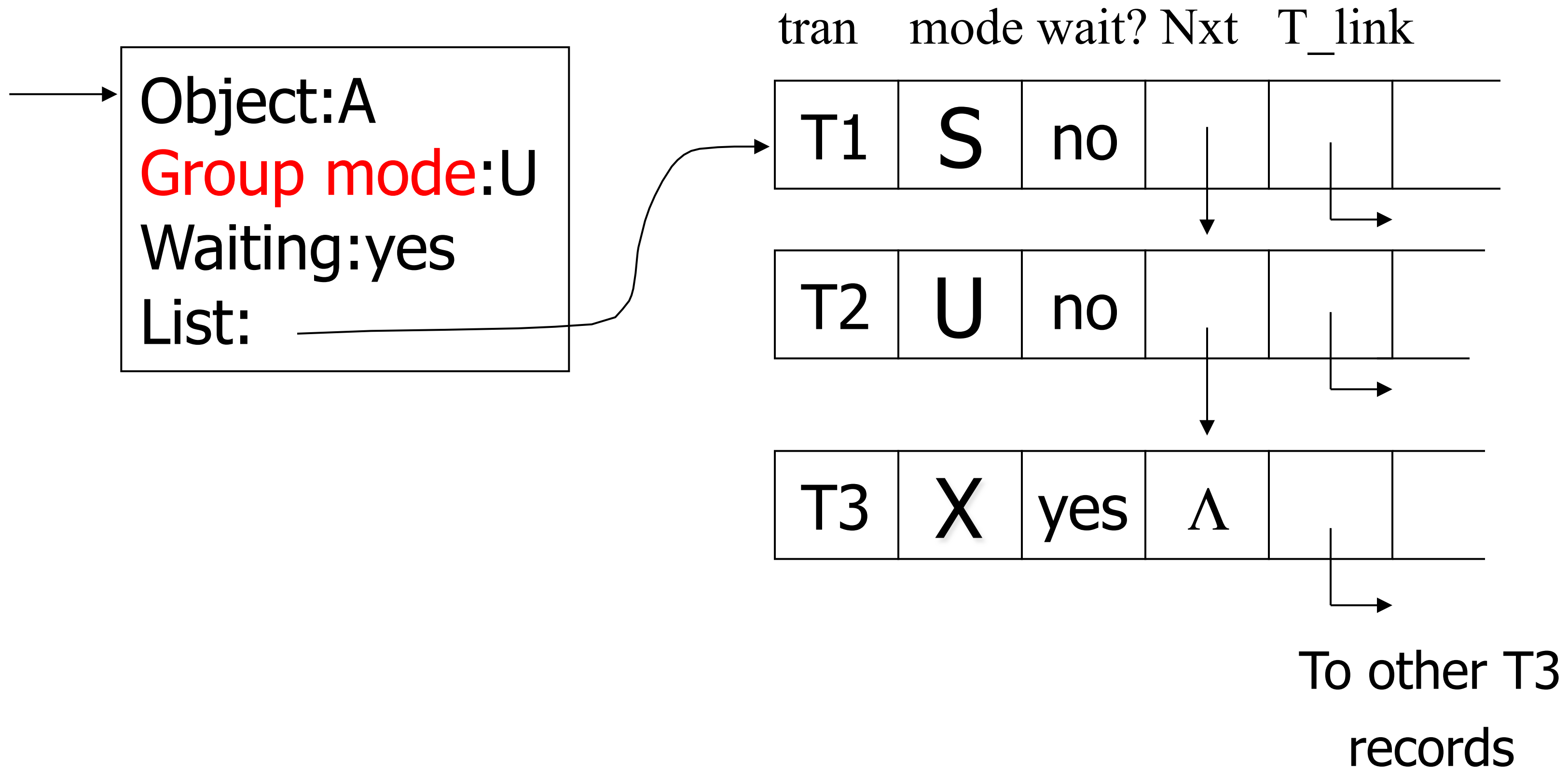
1. Don't trust transactions to request/release locks
2. Grant locks when needed, and hold all locks until transaction commits



Lock table:



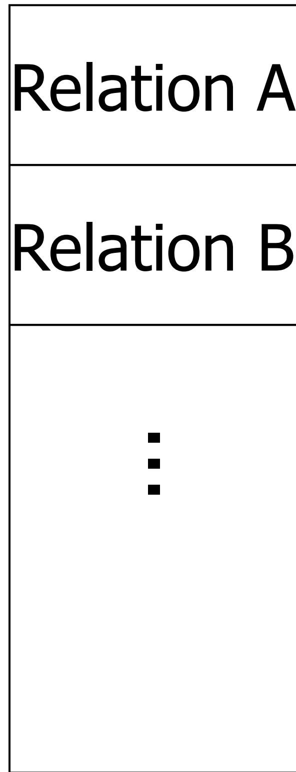
Lock info for A – example

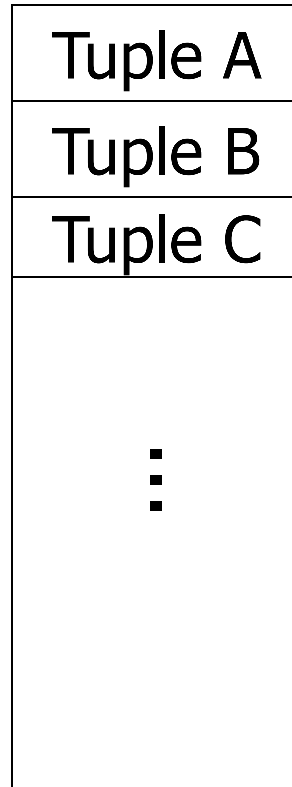


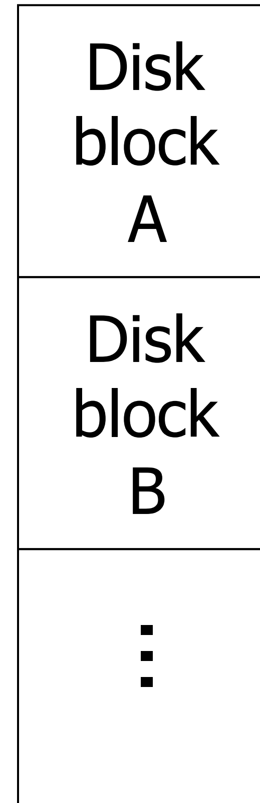
Recap/Outline

- "Conflict-Serializability"
  - Motivation; Precedence Graph test.
- Locking
  - Read/Write locks. Two-Phase Locking (2PL).
  - 2PL => Conflict Serializability
  - Shared/Upgrade, Increment, Update.
  - Simple Locking System, and Lock Table
- Lock Hierarchies (Warning Protocol).

What are the objects 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

Having it both ways

- If I lock a relation and you lock one of its tuples, can we indulge in unserializable behavior?
- Solution: Think of DB elements as a hierarchy composed of relations, composed of blocks, composed of tuples.

Warning Protocol

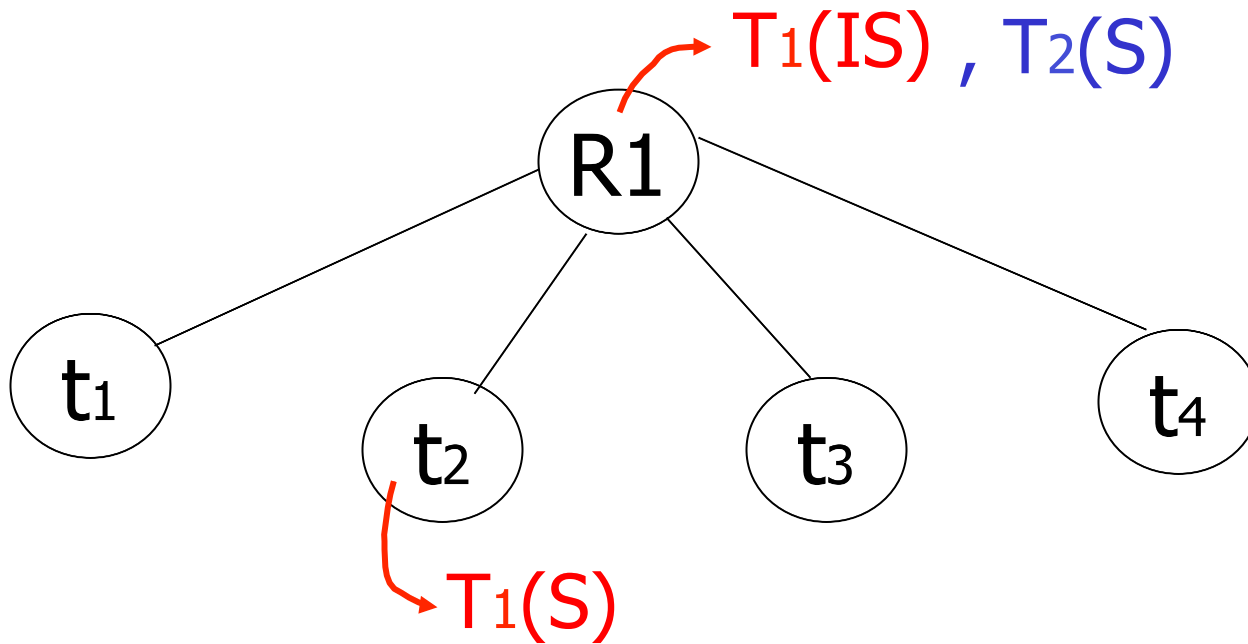
- Start at root.
- If you want to read/write an element, put a lock on it.
- If you want to read/write a sub-element, put a warning ("intention," denoted I) on it, and proceed to the appropriate child or children.
- Locking rule: Intentions don't conflict.

Warning Protocol: Example

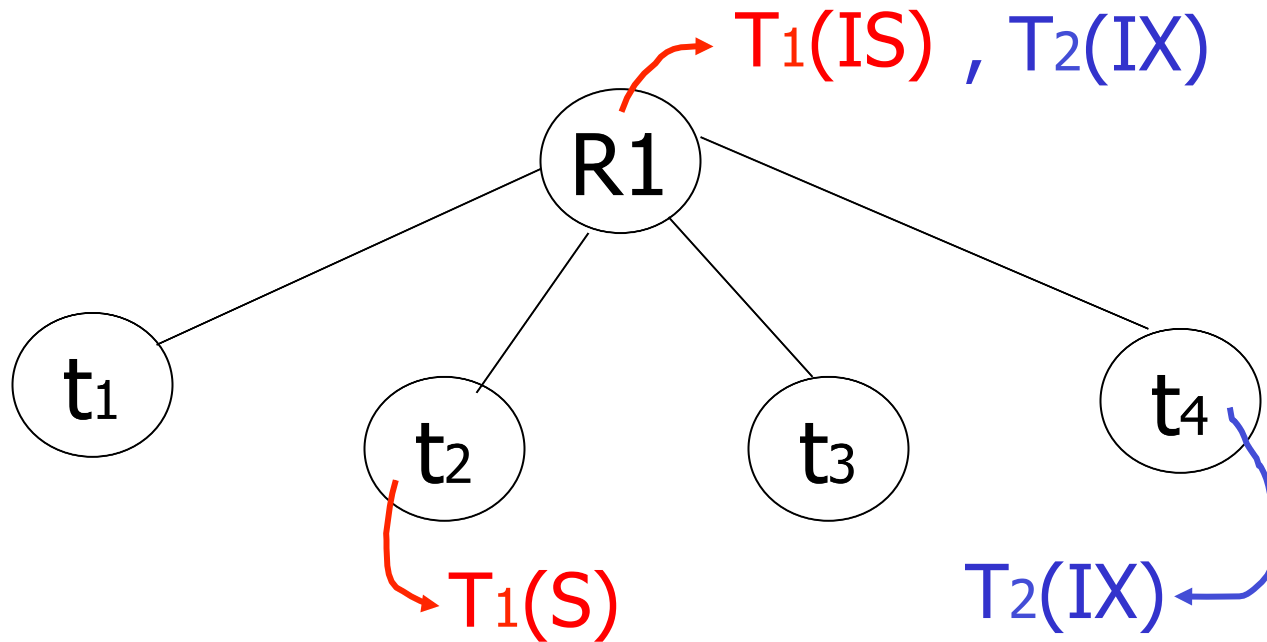
I want to lock tuple t1 of relation R; you want to lock tuple t2

- We both put a warning on R.
- We each put a warning on the block containing our desired tuples (may or may not be the same block).
- We each lock our respective tuples.

Example 2



Example 3

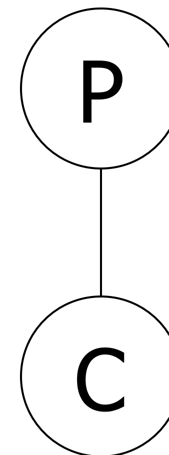


Multiple Granularity Comp. Matrix

Requestor

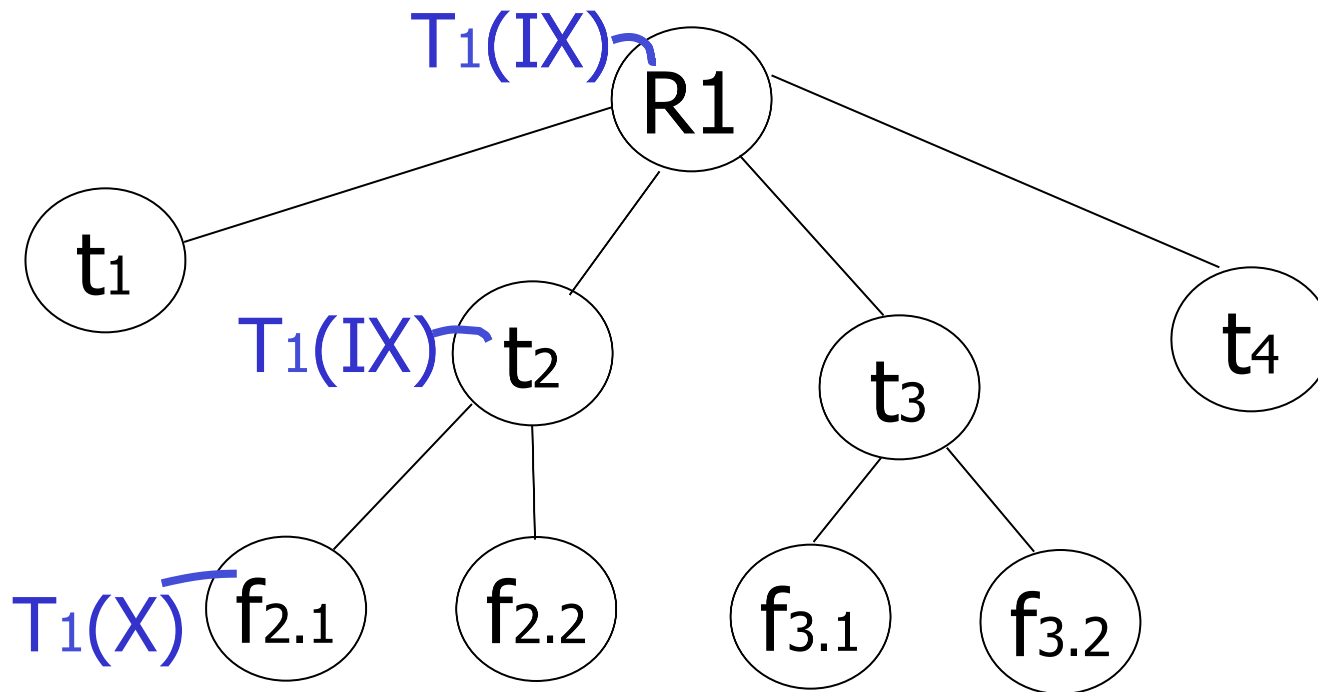
Holder	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

Parent locked in	Child can be locked in
IS	
IX	
S	
SIX	
X	



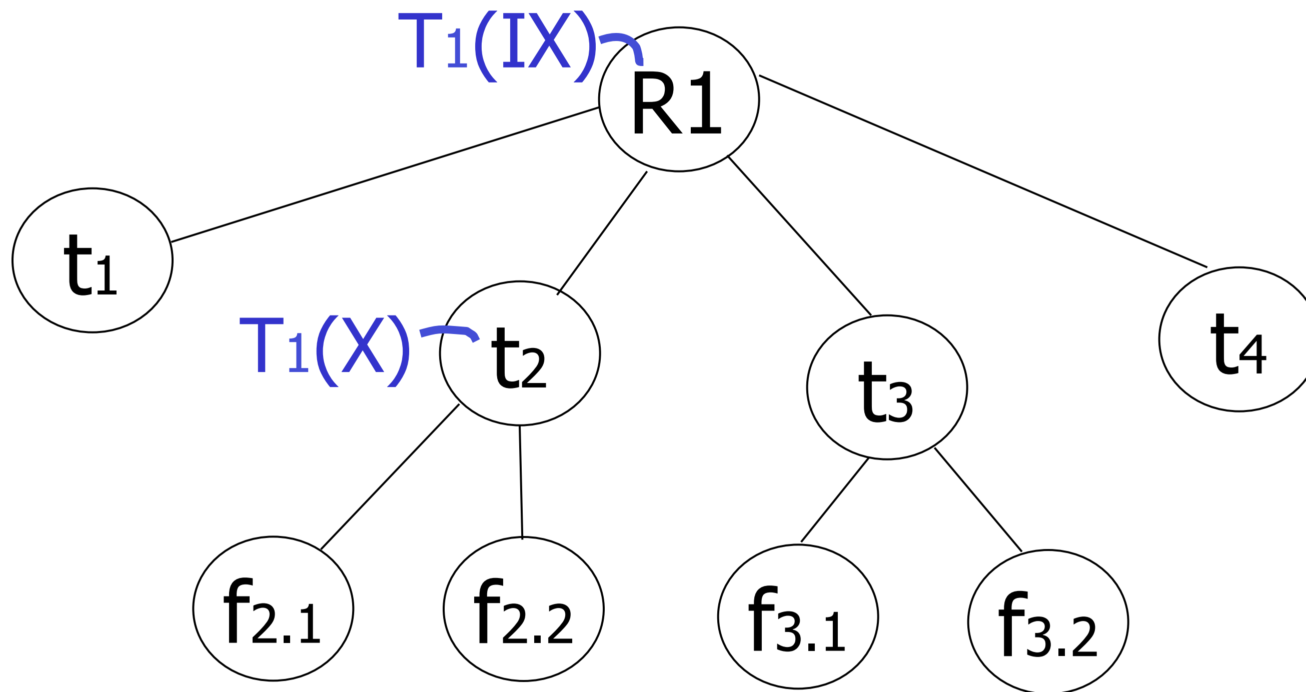
Exercise:

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



Exercise:

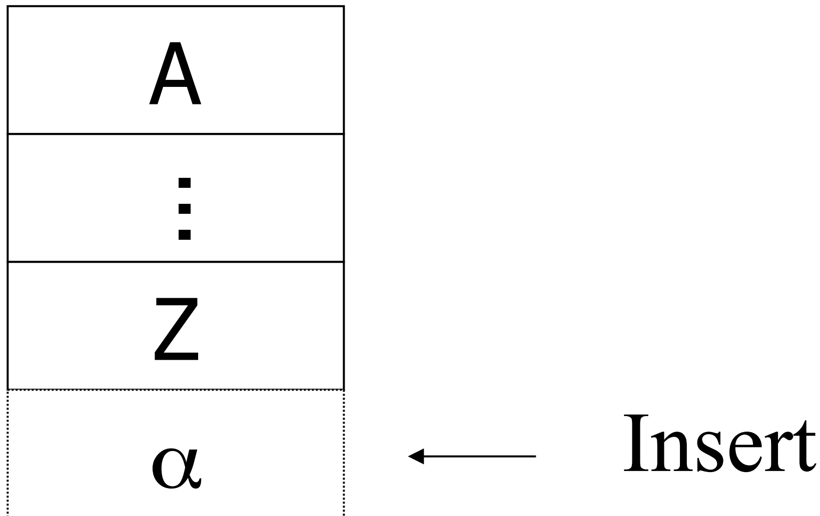
- Can T_2 access object $f_{2.2}$ in X mode? What locks will T_2 get?



Recap/Outline

- "Conflict-Serializability"
  - Motivation; Precedence Graph test.
- Locking
  - Rules. 2 Phase Locking.
  - 2PL => Conflict Serializability
  - Shared/Upgrade, Update, Increment.
  - Simple Locking System, and Lock Table
- Warning Protocol, Phantoms.

Insert + Delete operations



- What shall we lock, when the element doesn't even exist?
- Solution: Lock its parent.

Summary

- "Conflict-Serializability"
  - Motivation; Precedence Graph test.
- Locking
  - Rules. 2 Phase Locking.
  - 2PL => Conflict Serializability
  - Shared/Upgrade, Update, Increment.
  - Locking System: Group Modes, Lock Table, Schedulers
- Warning Protocol, Phantoms.