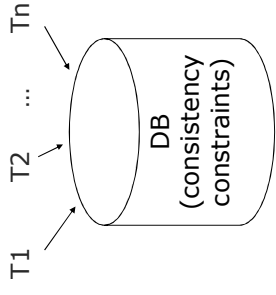


# Chapter 18: Concurrency Control



## CSE 562 Database Systems

### Concurrency Control

Some slides are based or modified from originals by  
 Database Systems: The Complete Book,  
 Pearson, Prentice Hall 2nd Edition,  
 ©2006 Garcia-Molina, Ullman, and Widom

*cse@buffalo*

### 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 A

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

UB CSE 562 Spring 2009

5

## Schedule C

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

UB CSE 562 Spring 2009

6

## Schedule D

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

UB CSE 562 Spring 2009

7

## Schedule E

Same as Schedule D  
but with new T2'

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

UB CSE 562 Spring 2009

8

- Want schedules that are "good", regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

**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:**

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

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

The diagram shows two sequences of operations. The first sequence is  $S_c = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$ . The second sequence is  $S_c' = r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$ . Brackets group the operations into two transactions: T1 (operations 1-4) and T2 (operations 5-8). In  $S_c'$ , the operations for T1 and T2 are swapped: T1's operations are now 1-2 and T2's operations are now 3-4. Arrows indicate the swap between  $r_2(A)w_2(A)$  and  $r_1(B)w_1(B)$ .

However, for  $S_d$ :

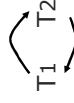
$S_d = r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B)$

The diagram shows the sequence  $S_d = r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B)$ . Brackets group the operations into two transactions: T1 (operations 1-2) and T2 (operations 3-4). A swap is indicated between  $r_2(A)w_2(A)$  and  $r_1(B)w_1(B)$  with red X marks. The resulting sequence after the swap would be  $r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$ .

- as a matter of fact, T2 must precede T1 in any equivalent schedule, i.e.,  $T_2 \rightarrow T_1$

- $T_2 \rightarrow T_1$

- Also,  $T_1 \rightarrow T_2$



- ⇨  $S_d$  cannot be rearranged into a serial schedule
- ⇨  $S_d$  is not "equivalent" to any serial schedule
- ⇨  $S_d$  is "bad"

## Returning to Sc

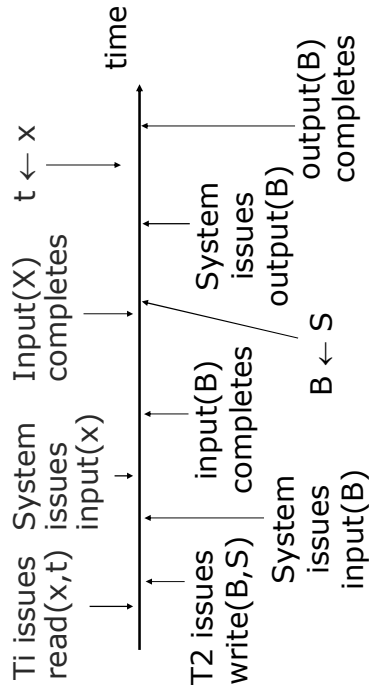
$Sc = r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)$   
 $T1 \rightarrow T2$

• no cycles  $\Rightarrow Sc$  is "equivalent" to a serial schedule (in this case  $T1, T2$ )

## Concepts

*Transaction:* sequence of  $ri(x), wi(x)$  actions  
*Conflicting actions:*  $\langle r1(A) \langle w2(A) \langle w1(A) \rangle \rangle$   
 $\langle w2(A) \langle r1(A) \langle w2(A) \rangle \rangle$   
*Schedule:* represents chronological order in which actions are executed  
*Serial schedule:* no interleaving of actions or transactions

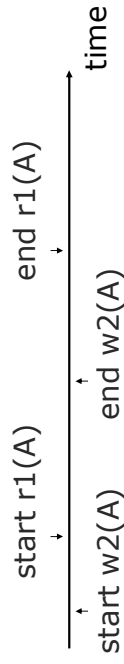
## What About Concurrent Actions?



So net effect is either
 

- $S = \dots r1(x) \dots w2(B) \dots$  or
- $S = \dots w2(B) \dots r1(x) \dots$

What about conflicting, concurrent actions on same object?



- Assume equivalent to either  $r1(A) w2(A)$  or  $w2(A) r1(A)$
- $\Rightarrow$  low level synchronization mechanism
- Assumption called "atomic actions"

UB CSE 562 Spring 2009

17

## Definition

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

UB CSE 562 Spring 2009

18

## Definition

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

UB CSE 562 Spring 2009

19

## Precedence Graph P(S) (S is schedule)

Nodes: transactions in S

Arcs:  $T_i \rightarrow T_j$  whenever

- $pi(A), qj(A)$  are actions in S
- $pi(A) <_s qj(A)$
- at least one of  $pi, qj$  is a write

UB CSE 562 Spring 2009

20

## Exercise:

- What is  $P(S)$  for  
 $S = w3(A) w2(C) r1(A) w1(B) r1(C) w2(A) r4(A) w4(D)$

- Is  $S$  serializable?

UB CSE 562 Spring 2009

21

## Another Exercise:

- What is  $P(S)$  for  
 $S = w1(A) r2(A) r3(A) w4(A)$  ?

UB CSE 562 Spring 2009

22

## Lemma

$S1, S2$  conflict equivalent  $\Rightarrow P(S1) = P(S2)$

Proof:

Assume  $P(S1) \neq P(S2)$

$\Rightarrow \exists Ti \rightarrow Tj$  in  $S1$  and not in  $S2$

$\Rightarrow S1 = \dots pi(A) \dots qj(A) \dots$   $\left\{ \begin{array}{l} pi, qj \\ \text{conflict} \end{array} \right.$

$S2 = \dots qj(A) \dots pi(A) \dots$

$\Rightarrow S1, S2$  not conflict equivalent

UB CSE 562 Spring 2009

23

Note:  $P(S1) = P(S2) \not\Rightarrow S1, S2$  conflict equivalent

Counter example:

$S1 = w1(A) r2(A) w2(B) r1(B)$

$S2 = r2(A) w1(A) r1(B) w2(B)$

UB CSE 562 Spring 2009

24

## Theorem

$P(S1)$  acyclic  $\iff$   $S1$  conflict serializable

( $\Leftarrow$ ) Assume  $S1$  is conflict serializable

$\Rightarrow \exists Ss$ :  $Ss$ ,  $S1$  conflict equivalent

$\Rightarrow P(Ss) = P(S1)$

$\Rightarrow P(S1)$  acyclic since  $P(Ss)$  is acyclic

## Theorem

$P(S1)$  acyclic  $\iff$   $S1$  conflict serializable

( $\Rightarrow$ ) Assume  $P(S1)$  is acyclic

Transform  $S1$  as follows:

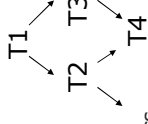
(1) Take  $T1$  to be transaction with no incident arcs

(2) Move all  $T1$  actions to the front

$S1 = \dots qj(A) \dots p1(A) \dots$

(3) we now have  $S1 = < T1 \text{ actions} > < \dots \text{rest} \dots >$

(4) repeat above steps to serialize rest!

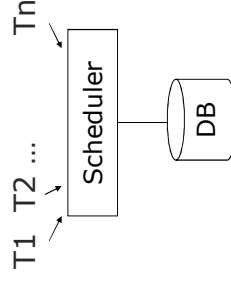


## How to Enforce Serializable Schedules?

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

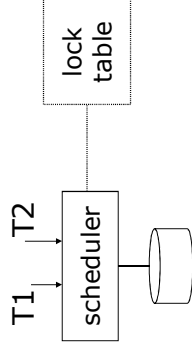
## How to Enforce Serializable Schedules?

- Option 2: prevent  $P(S)$  cycles from occurring



## A Locking Protocol

Two new actions:  
lock (exclusive): li (A)  
unlock: ui (A)



## Rule #1: Well-Formed Transactions

Ti: ... li(A) ... pi(A) ... ui(A) ...

## Rule #2: Legal Scheduler

S = ..... li(A) ..... ui(A) .....  
↔  
no lj(A)

## Exercise:

- What schedules are legal?  
What transactions are well-formed?  
S1 = l1(A)l1(B)r1(A)w1(B)l2(B)u1(A)u1(B)  
r2(B)w2(B)u2(B)l3(B)r3(B)u3(B)  
S2 = l1(A)r1(A)w1(B)u1(A)u1(B)  
l2(B)r2(B)w2(B)l3(B)r3(B)u3(B)  
S3 = l1(A)r1(A)l1(B)w1(B)u1(B)  
l2(B)r2(B)w2(B)u2(B)l3(B)r3(B)u3(B)

## Exercise:

- What schedules are legal?  
What transactions are well-formed?
- $S1 = I1(A); I1(B); r1(A); w1(B); I2(B); u1(A); u1(B); r2(B); w2(B); u2(B); I3(B); r3(B); u3(B)$   
 $S2 = I1(A); r1(A); w1(B); u1(A); u1(B); I2(B); r2(B); w2(B); I3(B); r3(B); u3(B)$   
 $S3 = I1(A); r1(A); u1(A); u1(B); I1(B); w1(B); u1(B); I2(B); r2(B); w2(B); u2(B); I3(B); r3(B); u3(B)$

## Schedule F

T1	T2
$I1(A); Read(A)$ $A \leftarrow A + 100; Write(A); u1(A)$	$I2(A); Read(A)$ $A \leftarrow Ax2; Write(A); u2(A)$ $I2(B); Read(B)$ $B \leftarrow Bx2; Write(B); u2(B)$
$I1(B); Read(B)$ $B \leftarrow B + 100; Write(B); u1(B)$	

## Schedule F

	A	B
T1	25	25
T2	125	
	250	50
	250	150
	250	150

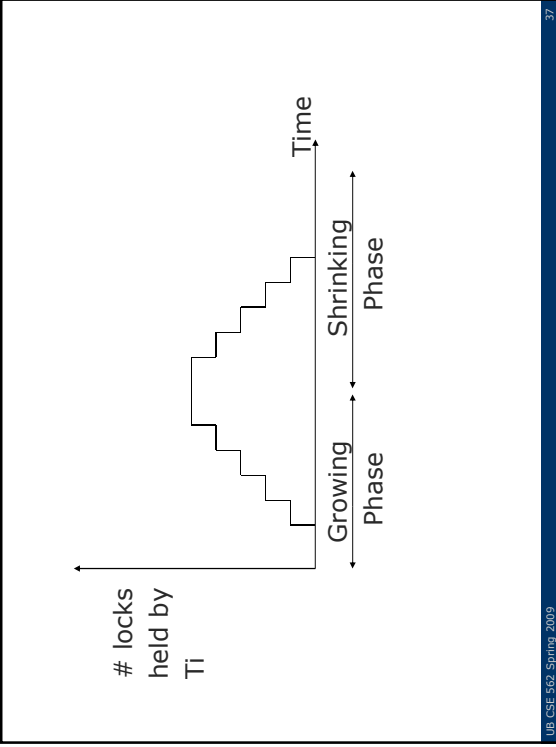
  

T1	T2
$I1(A); Read(A)$ $A \leftarrow A + 100; Write(A); u1(A)$	$I2(A); Read(A)$ $A \leftarrow Ax2; Write(A); u2(A)$ $I2(B); Read(B)$ $B \leftarrow Bx2; Write(B); u2(B)$
$I1(B); Read(B)$ $B \leftarrow B + 100; Write(B); u1(B)$	

## Rule #3: Two Phase Locking (2PL) for Transactions

$T_i = \dots li(A) \dots \dots \dots ui(A) \dots$

← no unlocks      → no locks



### Schedule G

<p>T1</p> <p>I1(A); Read(A)</p> <p>A ← A + 100; Write(A)</p> <p>I1(B); u1(A)</p>	<p>T2</p> <p>I2(A); Read(A)</p> <p>A ← Ax2; Write(A); I2(B)</p> <p style="text-align: right;">delayed</p>
--	---

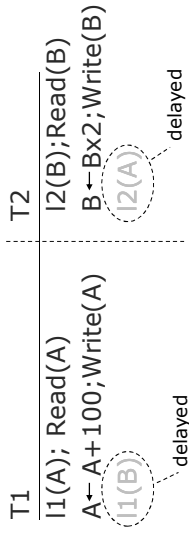
### Schedule G

<p>T1</p> <p>I1(A); Read(A)</p> <p>A ← A + 100; Write(A)</p> <p>I1(B); u1(A)</p> <p>Read(B); B ← B + 100</p> <p>Write(B); u1(B)</p>	<p>T2</p> <p>I2(A); Read(A)</p> <p>A ← Ax2; Write(A); I2(B)</p> <p style="text-align: right;">delayed</p>
---	---

### Schedule G

<p>T1</p> <p>I1(A); Read(A)</p> <p>A ← A + 100; Write(A)</p> <p>I1(B); u1(A)</p> <p>Read(B); B ← B + 100</p> <p>Write(B); u1(B)</p>	<p>T2</p> <p>I2(A); Read(A)</p> <p>A ← Ax2; Write(A); I2(B)</p> <p style="text-align: right;">delayed</p> <p>I2(B); u2(A); Read(B)</p> <p>B ← Bx2; Write(B); u2(B);</p>
---	---

## Schedule H (T2 reversed)



- Assume deadlocked transactions are rolled back
  - They have no effect
  - They do not appear in schedule

E.g., Schedule H =  $\{ \text{I1(A)}, \text{W1(A)}, \text{I2(B)}, \text{W2(B)}, \text{I1(B)}, \text{W1(B)}, \text{I2(A)}, \text{W2(A)} \}$   
 This space intentionally left blank!

## Next Step:

Show that rules #1,2,3  $\Rightarrow$  conflict-serializable schedules

Conflict rules for  $li(A), ui(A)$ :

- $li(A), lj(A)$  conflict
- $li(A), uj(A)$  conflict

Note: no conflict  $\langle ui(A), uj(A) \rangle, \langle li(A), rj(A) \rangle, \dots$

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

To help in proof:

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

Lemma

$Ti \rightarrow Tj$  in  $S \Rightarrow \text{SH}(Ti) <_s \text{SH}(Tj)$

Proof of lemma:

$Ti \rightarrow Tj$  means that

$S = \dots pi(A) \dots qj(A) \dots$ ;  $p, q$  conflict

By rules 1,2:

$S = \dots pi(A) \dots ui(A) \dots lj(A) \dots qj(A) \dots$

By rule 3:  $\text{SH}(Ti)$   $\text{SH}(Tj)$

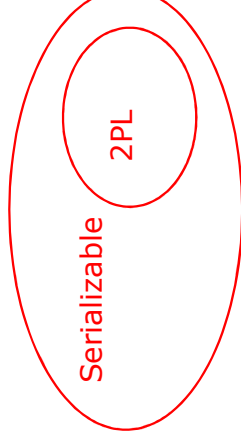
So,  $\text{SH}(Ti) <_s \text{SH}(Tj)$

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

Proof:

- (1) Assume  $P(S)$  has cycle  
 $T1 \rightarrow T2 \rightarrow \dots \rightarrow Tn \rightarrow T1$
- (2) By lemma:  $\text{SH}(T1) < \text{SH}(T2) < \dots < \text{SH}(T1)$
- (3) Impossible, so  $P(S)$  acyclic
- (4)  $\Rightarrow S$  is conflict serializable

## 2PL Subset of Serializable



## S1: w1(x) w3(x) w2(y) w1(y)

- 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 because T1 holds the x lock at that point.
- However, S1 is serializable (equivalent to T2, T1, T3).

- 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 = ...l1(A) r1(A) u1(A) ... l2(A) r2(A) u2(A) ...

Do not conflict

Instead:

S = ... ls1(A) r1(A) ls2(A) r2(A) .... us1(A) us2(A)

### Lock actions

l-ti(A): lock A in t mode (t is S or X)  
u-ti(A): unlock t mode (t is S or X)

### Shorthand:

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

## Rule #1: Well Formed Transactions

$T_i = \dots I-S_i(A) \dots r_1(A) \dots u_1(A) \dots$   
 $T_i = \dots I-X_i(A) \dots w_1(A) \dots u_1(A) \dots$

UB CSE 562 Spring 2009

53

- What about transactions that read and write same object?

Option 1: Request exclusive lock

$T_i = \dots I-X_i(A) \dots r_1(A) \dots w_1(A) \dots u(A) \dots$

UB CSE 562 Spring 2009

54

- What about transactions that read and write same object?

Option 2: Upgrade

(E.g., need to read, but don't know if will write...)

$T_i = \dots I-S_i(A) \dots r_1(A) \dots I-X_i(A) \dots w_1(A) \dots u(A) \dots$

Think of

- Get 2<sup>nd</sup> lock on A, or
- Drop S, get X lock

UB CSE 562 Spring 2009

55

## Rule #2: Legal Scheduler

$S = \dots I-S_i(A) \dots \dots u_i(A) \dots$   
no I-X<sub>j</sub>(A)

$S = \dots I-X_i(A) \dots \dots u_i(A) \dots$   
no I-X<sub>j</sub>(A)  
no I-S<sub>j</sub>(A)

UB CSE 562 Spring 2009

56

## A Way To Summarize Rule #2

Compatibility Matrix

Comp	S	X
S	true	false
X	false	false

UB CSE 562 Spring 2009

57

## 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 (shared) lock (e.g.,  $S \rightarrow X$ )
  - can be allowed in growing phase

UB CSE 562 Spring 2009

58

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

Proof: similar to X locks case

Detail:

- $l-t_i(A), l-r_j(A)$  do not conflict if  $\text{comp}(t,r)$
- $l-t_i(A), u-r_j(A)$  do not conflict if  $\text{comp}(t,r)$

UB CSE 562 Spring 2009

59

## Lock Types Beyond S/X

Examples:

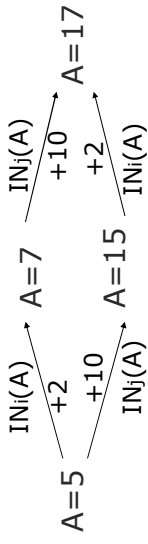
- (1) increment lock
- (2) update lock

UB CSE 562 Spring 2009

60

## Example (1): Increment Lock

- Atomic increment action:  $INI(A)$   
 $\{\text{Read}(A); A \leftarrow A+k; \text{Write}(A)\}$
- $INI(A), INj(A)$  do not conflict!

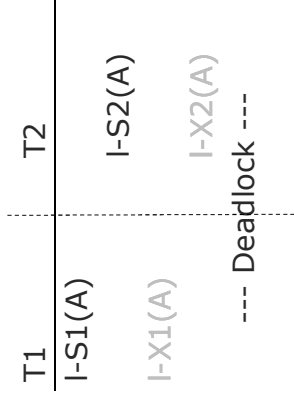


Comp

	S	X	I
S			
X			
I			

## Update Locks

A common deadlock problem with upgrades:



Comp

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

## Solution

If T1 wants to read A and knows it may later want to write A, it requests update lock (not shared)

Comp	S	S	X	U
	X			
	U			

Lock already held in

New request

New request

Comp	S	T	S	X	U
	X	F	F	F	T
	U	F	TorF	F	F

Lock already held in

-> symmetric table?

Note: object A may be locked in different modes at the same time...

$S1 = \dots | -S1(A) \dots | -S2(A) \dots | -U3(A) \dots \left\{ \begin{array}{l} | -S4(A) \dots ? \\ | -U4(A) \dots ? \end{array} \right.$

- To grant a lock in mode t, mode t must be compatible with all currently held locks on object

## How Does Locking Work in Practice?

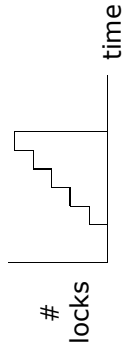
- Every system is different (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)
- But here is one (simplified) way...

UB CSE 562 Spring 2009

69

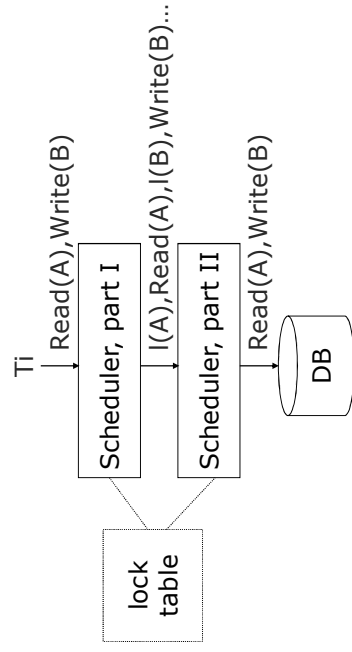
## Sample Locking System

- (1) Don't trust transactions to request/release locks
- (2) Hold all locks until transaction commits



UB CSE 562 Spring 2009

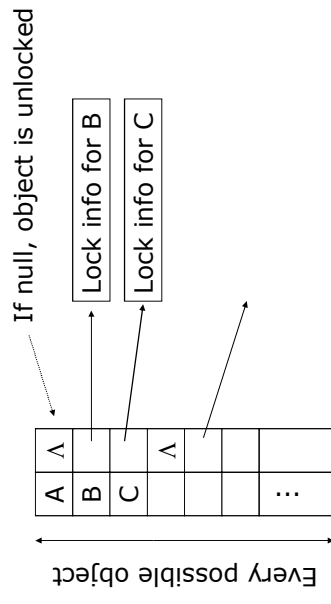
70



UB CSE 562 Spring 2009

71

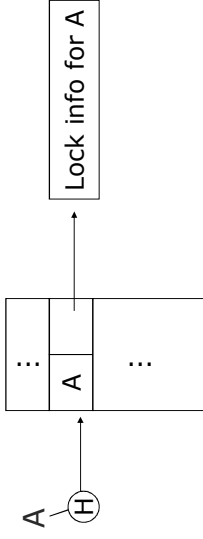
## Lock Table: Conceptually



UB CSE 562 Spring 2009

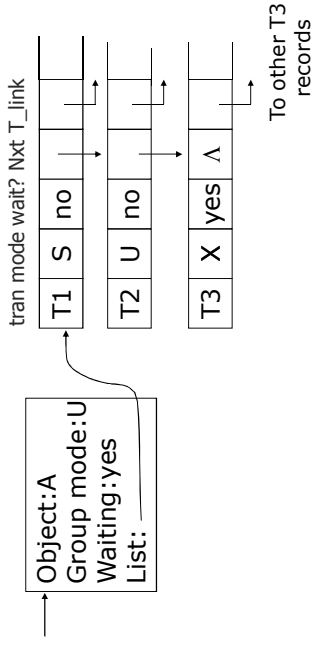
72

## But Use Hash Table:

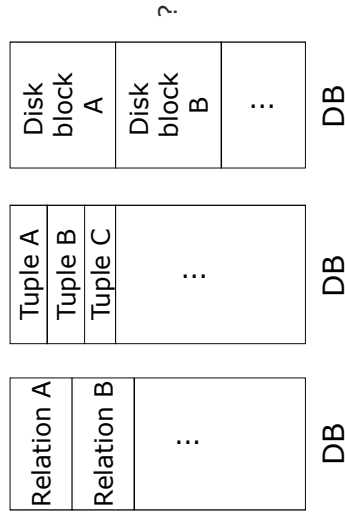


If object not found in hash table, it is unlocked

## Lock Info for A: Example



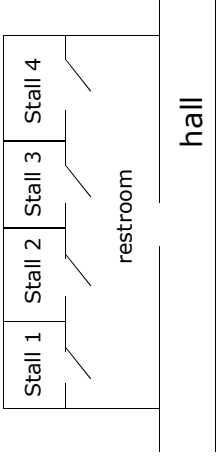
## What Are The Objects We Lock?



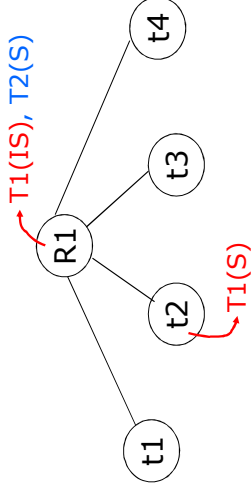
- Locking works in any case, but should we choose small or large objects?
- If we lock large objects (e.g., Relations)
  - Need few locks
  - Low concurrency
- If we lock small objects (e.g., tuples, fields)
  - Need more locks
  - More concurrency

## We Can Have It Both Ways!!

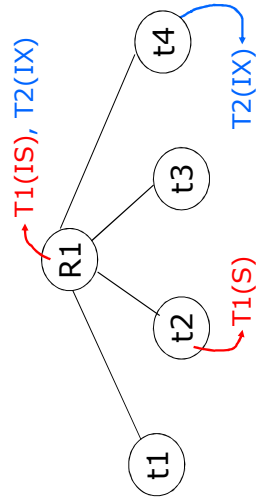
Ask any janitor to give you the solution...



## Example



## Example



## Multiple Granularity

Comp	Requestor					
	IS	IX	S	SIX	X	
Holder	IS	IX	S	SIX	X	

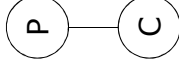
## Multiple Granularity

Comp	Requestor					
	IS	IX	S	SIX	X	
IS	T	T	T	T	T	F
IX	T	T	F	F	F	F
S	T	F	T	F	F	F
SIX	T	F	F	F	F	F
X	F	F	F	F	F	F

UB CSE 562 Spring 2009

81

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



UB CSE 562 Spring 2009

82

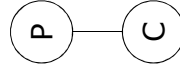
## Rules

- (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

UB CSE 562 Spring 2009

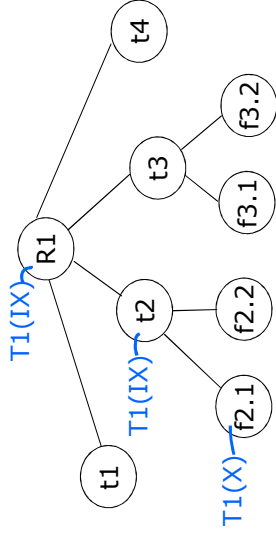
83

Parent locked in	Child can be locked in
IS	IS, S
IX	IS, S, IX, X, SIX
S	[S, IS] not necessary
SIX	X, IX, [SIX]
X	none



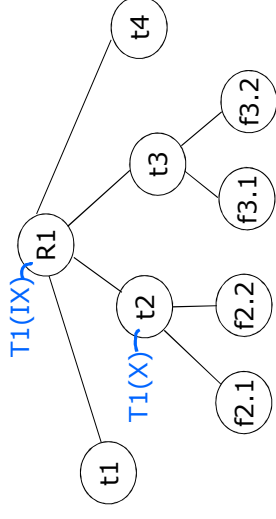
### Exercise:

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



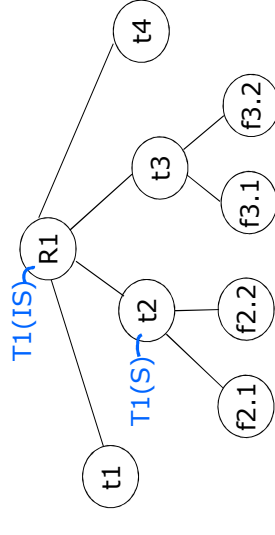
### Exercise:

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



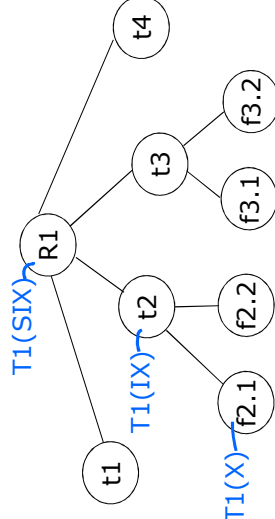
### Exercise:

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



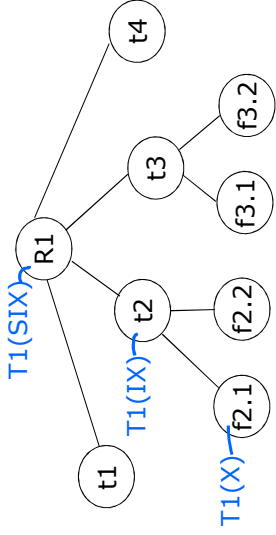
### Exercise:

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



## Exercise:

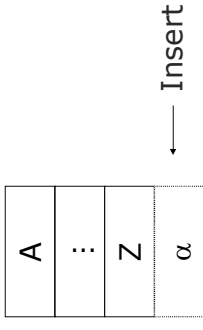
- Can T2 access object f2.2 in X mode?  
What locks will T2 get?



UB CSE 562 Spring 2009

89

## Insert + Delete Operations



UB CSE 562 Spring 2009

90

## Modifications To Locking Rules:

- (1) Get exclusive lock on A before deleting A
- (2) At insert A operation by  $T_i$ ,  $T_i$  is given exclusive lock on A

UB CSE 562 Spring 2009

91

## Still have a problem: Phantoms

Example: relation R (E#,name,...)  
constraint: E# is key  
use tuple locking

R	E#	Name	...
o1	55	Smith	
o2	75	Jones	

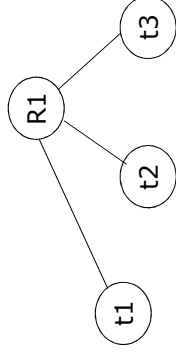
UB CSE 562 Spring 2009

92

T1: Insert <04,Kerry,...> into R	T2
T2: Insert <04,Bush,...> into R	S2(o1)
<b>T1</b>	S2(o2)
S1(o1)	Check Constraint
S1(o2)	⋮
Check Constraint	Insert o4[04,Bush,...]
⋮	
Insert o3[04,Kerry,...]	

## Solution

- Use multiple granularity tree
- Before insert of node Q, lock parent(Q) in X mode



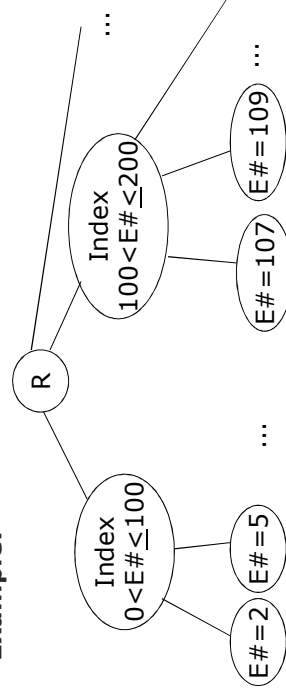
## Back To Example

T1: Insert<04,Kerry>	T2: Insert<04,Bush>
<b>T1</b>	<b>T2</b>
X1(R)	X2(R)
Check Constraint	Check Constraint
Insert<04,Kerry>	Oops! e# = 04 already in R!
U(R)	

*delayed*

## Instead of Using R, Can Use Index on R

**Example:**



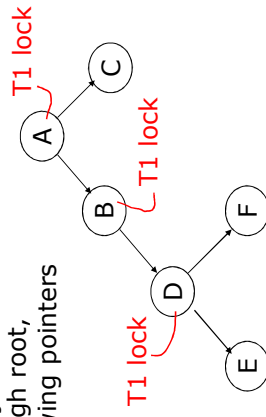
- This approach can be generalized to multiple indexes...

### Next:

- Tree-based concurrency control
- Validation concurrency control

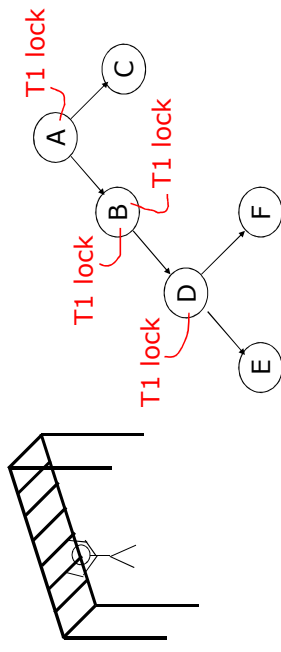
### Example

- all objects accessed through root, following pointers



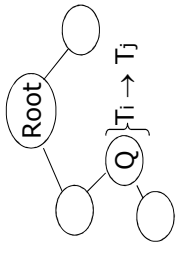
• can we release A lock if we no longer need A??

### Idea: Traverse Like "Monkey Bars"



## Why Does This Work?

- Assume all  $T_i$  start at root; exclusive lock
- $T_i \rightarrow T_j \Rightarrow T_i$  locks root before  $T_j$

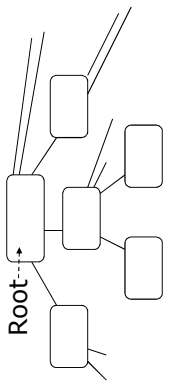


- Actually works if we don't always start at root

## Rules: Tree Protocol (exclusive locks)

- (1) First lock by  $T_i$  may be on any item
- (2) After that, item  $Q$  can be locked by  $T_i$  only if parent( $Q$ ) locked by  $T_i$
- (3) Items may be unlocked at any time
- (4) After  $T_i$  unlocks  $Q$ , it cannot relock  $Q$

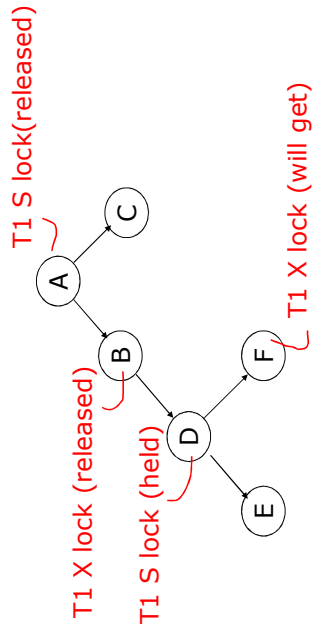
- Tree-like protocols are used typically for B-tree concurrency control



E.g., during insert, do not release parent lock, until you are certain child does not have to split

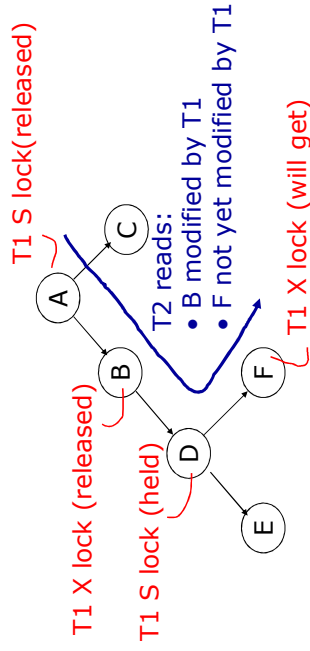
## Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



## Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



UB CSE 562 Spring 2009

105

## Tree Protocol with Shared Locks

- Need more restrictive protocol
- Will this work??
  - Once  $T_1$  locks one object in X mode, all further locks down the tree must be in X mode

UB CSE 562 Spring 2009

106

## Validation

Transactions have 3 phases:

- Read
  - all DB values read
  - writes to temporary storage
  - no locking
- Validate
  - check if schedule so far is serializable
- Write
  - if validate ok, write to DB

UB CSE 562 Spring 2009

107

## Key Idea

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

UB CSE 562 Spring 2009

108

To implement validation, system keeps

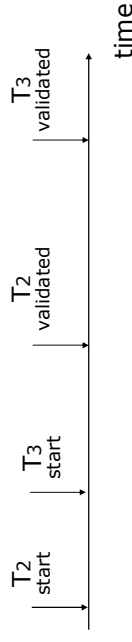
two sets:

- FIN = transactions that have finished phase 3 (and are all done)
- VAL = transactions that have successfully finished phase 2 (validation)

### Example of What Validation Must Prevent:

$$RS(T2) = \{B\} \quad \cap \quad RS(T3) = \{A, B\} \neq \emptyset$$

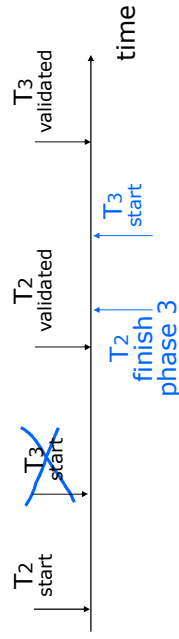
$$WS(T2) = \{B, D\} \quad \cap \quad WS(T3) = \{C\}$$



### Example of What Validation Must Prevent: Allow

$$RS(T2) = \{B\} \quad \cap \quad RS(T3) = \{A, B\} \neq \emptyset$$

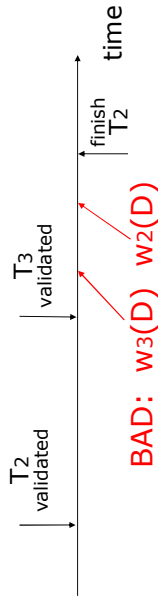
$$WS(T2) = \{B, D\} \quad \cap \quad WS(T3) = \{C\}$$



### Another Thing Validation Must Prevent:

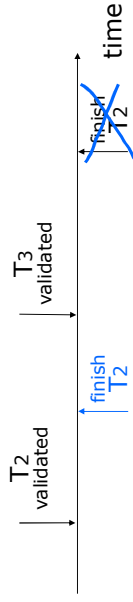
$$RS(T2) = \{A\} \quad \cap \quad RS(T3) = \{A, B\}$$

$$WS(T2) = \{D, E\} \quad \cap \quad WS(T3) = \{C, D\}$$



## Another Thing Validation Must Prevent: Allow

$RS(T2) = \{A\}$        $RS(T3) = \{A, B\}$   
 $WS(T2) = \{D, E\}$        $WS(T3) = \{C, D\}$



UB CSE 562 Spring 2009

113

## Validation Rules For Tj:

(1) when Tj starts phase 1:  
ignore(Tj)  $\leftarrow$  FIN  
(2) at Tj Validation:  
if check (Tj) then  
    [ VAL  $\leftarrow$  VAL U {Tj}];  
    do write phase;  
    FIN  $\leftarrow$  FIN U {Tj} ]

UB CSE 562 Spring 2009

114

## Improving Check(Tj)

For  $T_i \in \text{VAL} - \text{IGNORE}(T_j)$  DO  
IF [  $WS(T_i) \cap RS(T_j) \neq \emptyset$  OR  
    ( $T_i \notin \text{FIN}$  AND  $WS(T_i) \cap WS(T_j) \neq \emptyset$ )]  
    RETURN false;  
RETURN true;

UB CSE 562 Spring 2009

116

Check (Tj):

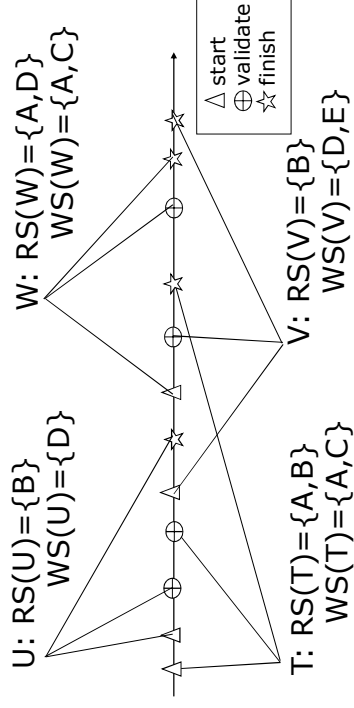
For  $T_i \in \text{VAL} - \text{IGNORE}(T_j)$  DO  
IF [  $WS(T_i) \cap RS(T_j) \neq \emptyset$  OR  $T_i \notin \text{FIN}$  ]  
    RETURN false;  
RETURN true;

Is this check too restrictive ?

UB CSE 562 Spring 2009

115

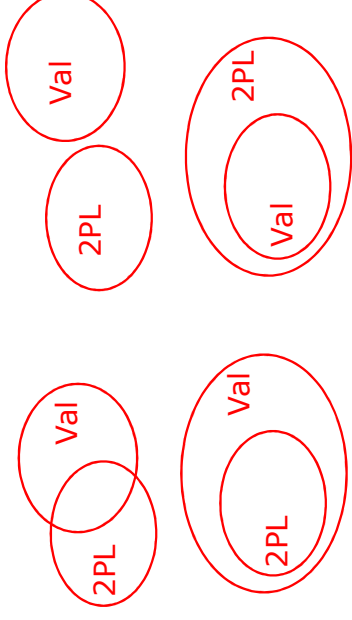
## Exercise:



UB CSE 562 Spring 2009

117

## Is Validation = 2PL?



UB CSE 562 Spring 2009

118

## S2: $w2(y)$ $w1(x)$ $w2(x)$

- S2 can be achieved with 2PL:  
 $l2(y)$   $w2(y)$   $l1(x)$   $w1(x)$   $l2(x)$   $w2(x)$   $u2(y)$   $u2(x)$
- S2 cannot be achieved by validation:  
 The validation point of T2, val2 must occur before  $w2(y)$  since transactions do not write to the database until after validation. Because of the conflict on x,  $val1 < val2$ , so we must have something like  
 $S2: val1$   $val2$   $w2(y)$   $w1(x)$   $w2(x)$   
 With the validation protocol, the writes of T2 should not start until T1 is all done with its writes, which is not the case.

UB CSE 562 Spring 2009

119

## 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 page)

UB CSE 562 Spring 2009

120

- Say S' not legal:

S': ... l1(x) w2(x) r1(x) val1 u2(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:

S': ... val1 l1(x) w2(x) w1(x) u2(x) ...  
 - Say T2 validates first (proof similar in other case)  
 - At val1: T2 not in Ignore(T1); T2 in VAL  
 - T1 does not validate:  
 $T2 \notin \text{FIN AND } WS(T1) \cap WS(T2) \neq \emptyset$   
 - contradiction!

Validation (also called optimistic concurrency control) is useful in some cases:

- Conflicts rare
- System resources plentiful
- Have real time constraints

## Summary

Have studied concurrency control mechanisms used in practice

- 2PL
- Multiple granularity
- Tree (index) protocols
- Validation