

# CS 245: Database System Principles

## Notes 09: Concurrency Control

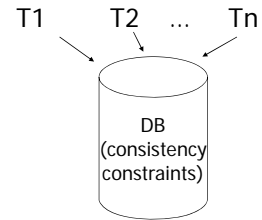
Hector Garcia-Molina

CS 245

Notes 09

1

## Chapter 18 [18] Concurrency Control



CS 245

Notes 09

2

### Example:

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

Constraint: A=B

CS 245

Notes 09

3

### Schedule A

<p>T1</p> <hr/> <p>Read(A); A ← A+100 Write(A); Read(B); B ← B+100; Write(B);</p>	<p>T2</p> <hr/> <p>Read(A); A ← A×2; Write(A); Read(B); B ← B×2; Write(B);</p>
---	--

CS 245

Notes 09

4

### Schedule A

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

CS 245

Notes 09

5

### Schedule B

<p>T1</p> <hr/> <p>Read(A); A ← A+100 Write(A); Read(B); B ← B+100; Write(B);</p>	<p>T2</p> <hr/> <p>Read(A); A ← A×2; Write(A); Read(B); B ← B×2; Write(B);</p>
---	--

CS 245

Notes 09

6

### Schedule B

		A	B
T1	T2	25	25
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);	50	50
		150	50
		150	150

CS 245
Notes 09
7

### Schedule C

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);		

CS 245
Notes 09
8

### Schedule C

		A	B
T1	T2	25	25
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);	125	125
		250	250
		250	250

CS 245
Notes 09
9

### Schedule D

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);		

CS 245
Notes 09
10

### Schedule D

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

CS 245
Notes 09
11

Same as Schedule D but with new T2'

### Schedule E

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

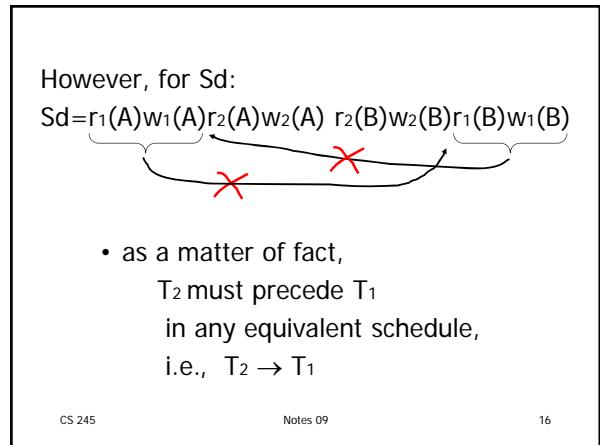
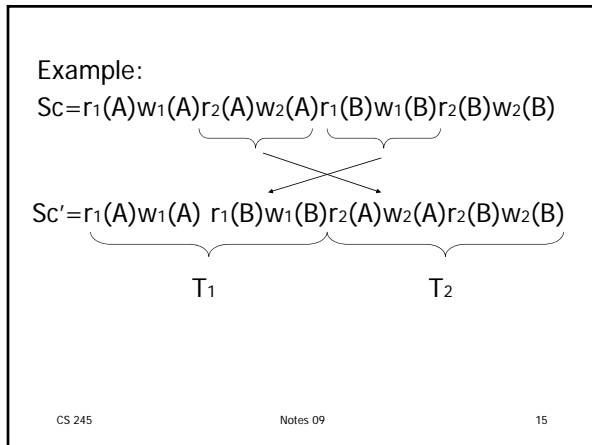
CS 245
Notes 09
12

**Schedule E** Same as Schedule D but with new T2'

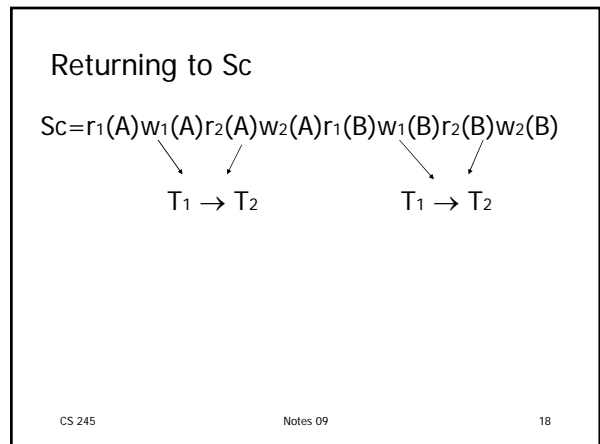
		A	B
T1	T2'	25	25
Read(A); A ← A+100		125	
Write(A);	Read(A); A ← A×1;	125	
	Write(A);		25
	Read(B); B ← B×1;		125
	Write(B);	125	125
Read(B); B ← B+100;			
Write(B);			

CS 245 Notes 09 13

- Want schedules that are “good”, regardless of
    - initial state and
    - transaction semantics
  - Only look at order of read and writes
- Example:  
 $Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$
- CS 245 Notes 09 14



- $T_2 \rightarrow T_1$
  - Also,  $T_1 \rightarrow T_2$
- 
- ⇒ Sd cannot be rearranged into a serial schedule
  - ⇒ Sd is not “equivalent” to any serial schedule
  - ⇒ Sd is “bad”
- CS 245 Notes 09 17



### Returning to Sc

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

$T_1 \rightarrow T_2$                        $T_1 \rightarrow T_2$

• no cycles  $\Rightarrow$  Sc is "equivalent" to a serial schedule (in this case  $T_1, T_2$ )

CS 245                      Notes 09                      19

### Concepts

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

*Schedule:* represents chronological order in which actions are executed  
*Serial schedule:* no interleaving of actions or transactions

CS 245                      Notes 09                      20

### What about concurrent actions?

Ti issues read(x,t)    System issues input(x)    Input(X) completes    t ← x

time

CS 245                      Notes 09                      21

### What about concurrent actions?

T1 issues read(x,t)    System issues input(x)    Input(X) completes    t ← x  
 T2 issues write(B,S)    System issues input(B)    input(B) completes    B ← S  
 System issues output(B)    output(B) completes

time

CS 245                      Notes 09                      22

So net effect is either

- $S = \dots r_1(x) \dots w_2(b) \dots$  or
- $S = \dots w_2(B) \dots r_1(x) \dots$

CS 245                      Notes 09                      23

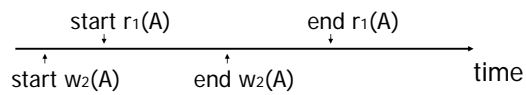
### What about conflicting, concurrent actions on same object?

start  $r_1(A)$                       end  $r_1(A)$   
 start  $w_2(A)$                       end  $w_2(A)$

time

CS 245                      Notes 09                      24

What about conflicting, concurrent actions on same object?



- Assume equivalent to either  $r_1(A) w_2(A)$   
or  $w_2(A) r_1(A)$
- $\Rightarrow$  low level synchronization mechanism
- Assumption called "atomic actions"

CS 245

Notes 09

25

### 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.

CS 245

Notes 09

26

### Definition

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

CS 245

Notes 09

27

### Precedence graph $P(S)$ ( $S$ is schedule)

Nodes: transactions in  $S$

Arcs:  $T_i \rightarrow T_j$  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

CS 245

Notes 09

28

### 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?

CS 245

Notes 09

29

### Another Exercise:

- What is  $P(S)$  for  
 $S = w_1(A) r_2(A) r_3(A) w_4(A)$  ?

CS 245

Notes 09

30

### Lemma

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

CS 245

Notes 09

31

### 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

CS 245

Notes 09

32

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

CS 245

Notes 09

33

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

#### Counter example:

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

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

CS 245

Notes 09

34

### Theorem

$P(S_1)$  acyclic  $\iff S_1$  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

CS 245

Notes 09

35

### Theorem

$P(S_1)$  acyclic  $\iff S_1$  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

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

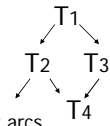
(3) we now have  $S_1 = \langle T_1 \text{ actions} \rangle \dots \text{rest} \dots$

(4) repeat above steps to serialize rest!

CS 245

Notes 09

36



## 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

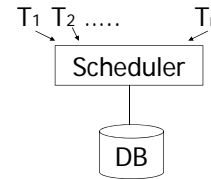
CS 245

Notes 09

37

## How to enforce serializable schedules?

*Option 2:* prevent P(S) cycles from  
occurring



CS 245

Notes 09

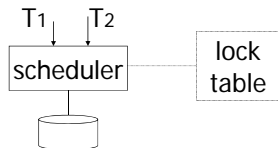
38

## A locking protocol

Two new actions:

lock (exclusive):  $l_i(A)$

unlock:  $u_i(A)$



CS 245

Notes 09

39

## Rule #1: Well-formed transactions

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

CS 245

Notes 09

40

## Rule #2 Legal scheduler

$S = \dots l_i(A) \dots u_i(A) \dots$   
 $\longleftrightarrow$   
 no  $l_j(A)$

CS 245

Notes 09

41

## Exercise:

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

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

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

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

CS 245

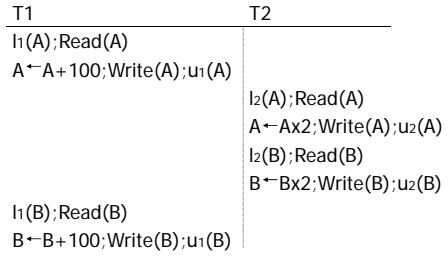
Notes 09

42

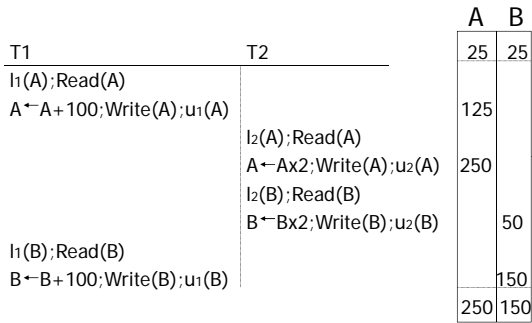
Exercise:

- What schedules are legal?  
What transactions are well-formed?
- S1 = l<sub>1</sub>(A)l<sub>1</sub>(B)r<sub>1</sub>(A)w<sub>1</sub>(B)l<sub>2</sub>(B)u<sub>1</sub>(A)u<sub>1</sub>(B)  
r<sub>2</sub>(B)w<sub>2</sub>(B)u<sub>2</sub>(B)l<sub>3</sub>(B)r<sub>3</sub>(B)u<sub>3</sub>(B)
- S2 = l<sub>1</sub>(A)r<sub>1</sub>(A)w<sub>1</sub>(B)u<sub>1</sub>(A)u<sub>1</sub>(B)  
l<sub>2</sub>(B)r<sub>2</sub>(B)w<sub>2</sub>(B)l<sub>3</sub>(B)r<sub>3</sub>(B)u<sub>3</sub>(B)
- S3 = l<sub>1</sub>(A)r<sub>1</sub>(A)u<sub>1</sub>(A)l<sub>1</sub>(B)w<sub>1</sub>(B)u<sub>1</sub>(B)  
l<sub>2</sub>(B)r<sub>2</sub>(B)w<sub>2</sub>(B)u<sub>2</sub>(B)l<sub>3</sub>(B)r<sub>3</sub>(B)u<sub>3</sub>(B)

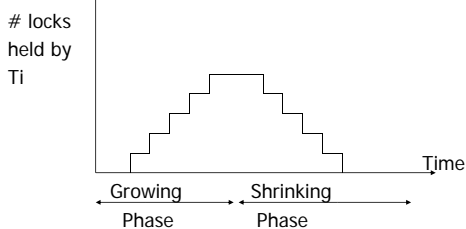
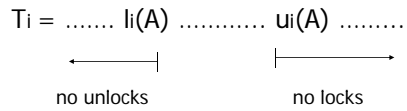
Schedule F



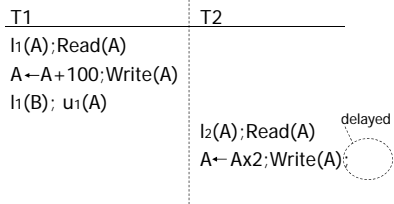
Schedule F



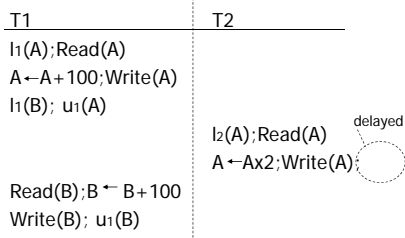
Rule #3 Two phase locking (2PL)  
for transactions



Schedule G



### Schedule G

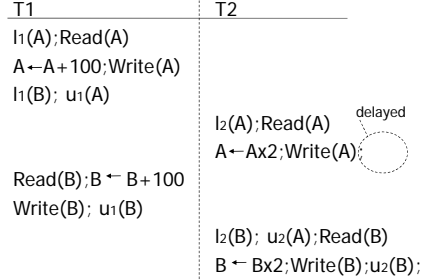


CS 245

Notes 09

49

### Schedule G

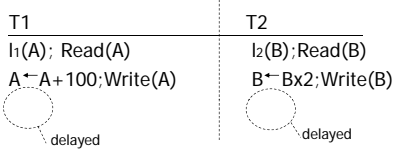


CS 245

Notes 09

50

### Schedule H (T2 reversed)



CS 245

Notes 09

51

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

E.g., Schedule H = This space intentionally left blank!

CS 245

Notes 09

52

### Next step:

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

CS 245

Notes 09

53

### Conflict rules for $l_i(A), u_i(A)$ :

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

Note: no conflict  $\langle u_i(A), u_j(A) \rangle, \langle l_i(A), r_j(A) \rangle, \dots$

CS 245

Notes 09

54

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

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

To help in proof:

Definition Shrink( $T_i$ ) = SH( $T_i$ ) =  
 first unlock action of  $T_i$

Lemma  
 $T_i \rightarrow T_j$  in  $S \Rightarrow SH(T_i) <_S SH(T_j)$

Lemma  
 $T_i \rightarrow T_j$  in  $S \Rightarrow SH(T_i) <_S SH(T_j)$

Proof of lemma:

$T_i \rightarrow T_j$  means that

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

By rules 1,2:

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

Lemma  
 $T_i \rightarrow T_j$  in  $S \Rightarrow SH(T_i) <_S SH(T_j)$

Proof of lemma:

$T_i \rightarrow T_j$  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  
 (2PL) serializable  
 schedule

Proof:

(1) Assume P(S) has cycle

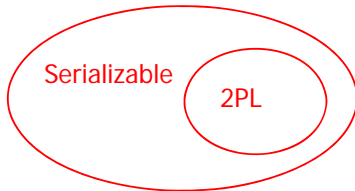
$T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$

(2) By lemma: SH( $T_1$ )  $<$  SH( $T_2$ )  $<$  ...  $<$  SH( $T_1$ )

(3) Impossible, so P(S) acyclic

(4)  $\Rightarrow$  S is conflict serializable

## 2PL subset of Serializable



CS 245

Notes 09

61

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

CS 245

Notes 09

62

- 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

CS 245

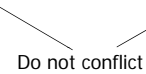
Notes 09

63

## Shared locks

So far:

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



CS 245

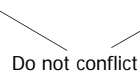
Notes 09

64

## Shared locks

So far:

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



Instead:

$S = \dots l_{s_1}(A) \ r_1(A) \ l_{s_2}(A) \ r_2(A) \ \dots \ u_{s_1}(A) \ u_{s_2}(A)$

CS 245

Notes 09

65

## Lock actions

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

$u-t_i(A)$ : unlock t mode (t is S or X)

Shorthand:

$u_i(A)$ : unlock whatever modes

T<sub>i</sub> has locked A

CS 245

Notes 09

66

Rule #1 Well formed transactions

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

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

- What about transactions that read and write same object?

Option 1: Request exclusive lock

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

- What about transactions that read and write same object?

Option 2: Upgrade

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

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

Think of  
 - Get 2nd lock on A, or  
 - Drop S, get X lock

Rule #2 Legal scheduler

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

no  $I-X_j(A)$

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

no  $I-X_j(A)$   
 no  $I-S_j(A)$

A way to summarize Rule #2

Compatibility matrix

Comp

	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.,  $S \rightarrow \{S, X\}$ ) then no change!
- (II) If upgrade releases read (shared) lock (e.g.,  $S \rightarrow X$ ) - can be allowed in growing phase

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  $comp(t,r)$

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

CS 245

Notes 09

73

## Lock types beyond S/X

Examples:

(1) increment lock

(2) update lock

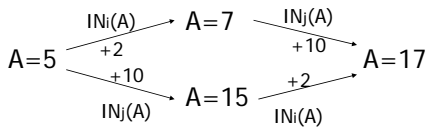
CS 245

Notes 09

74

### Example (1): increment lock

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



CS 245

Notes 09

75

Comp

	S	X	I
S			
X			
I			

CS 245

Notes 09

76

Comp

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

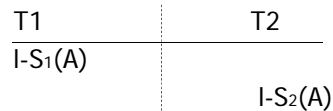
CS 245

Notes 09

77

## Update locks

A common deadlock problem with upgrades:



--- Deadlock ---

CS 245

Notes 09

78

Solution

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

New request

		S	X	U
Comp	Lock already held in	S		
		X		
		U		

New request

		S	X	U
Comp	Lock already held in	S	T	F
		X	F	F
		U	TorF	F

-> symmetric table?

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

$$S_1 = \dots I-S_1(A) \dots I-S_2(A) \dots I-U_3(A) \dots \left\{ \begin{array}{l} I-S_4(A) \dots? \\ I-U_4(A) \dots? \end{array} \right.$$

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

$$S_1 = \dots I-S_1(A) \dots I-S_2(A) \dots I-U_3(A) \dots \left\{ \begin{array}{l} I-S_4(A) \dots? \\ I-U_4(A) \dots? \end{array} \right.$$

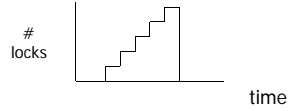
- To grant a lock in mode t, mode t must be compatible with all currently held locks on object

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 ...

Sample Locking System:

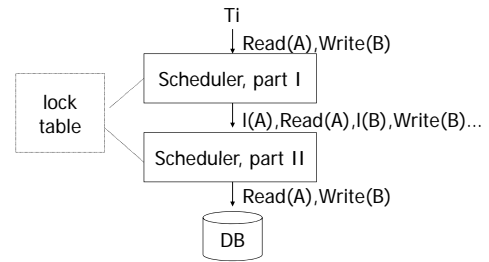
- (1) Don't trust transactions to request/release locks
- (2) Hold all locks until transaction commits



CS 245

Notes 09

85

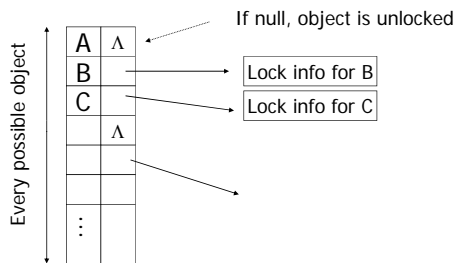


CS 245

Notes 09

86

Lock table Conceptually

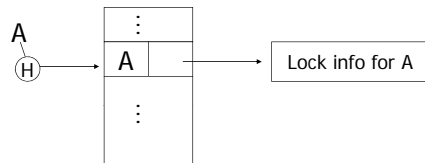


CS 245

Notes 09

87

But use hash table:



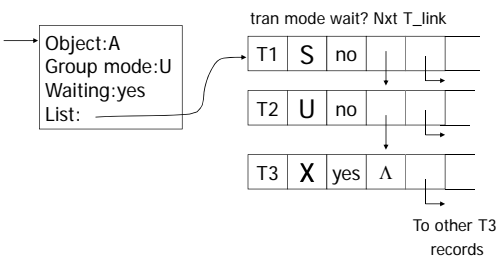
If object not found in hash table, it is unlocked

CS 245

Notes 09

88

Lock info for A - example

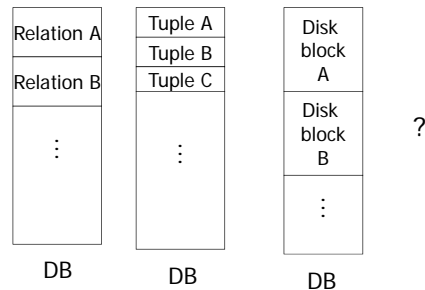


CS 245

Notes 09

89

What are the objects we lock?



CS 245

Notes 09

90

- Locking works in any case, but should we choose small or large objects?

CS 245

Notes 09

91

- 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

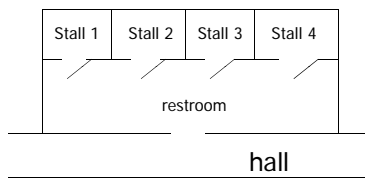
CS 245

Notes 09

92

We can have it both ways!!

Ask any janitor to give you the solution...

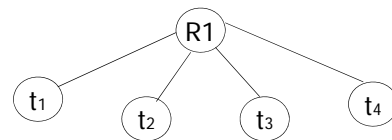


CS 245

Notes 09

93

Example

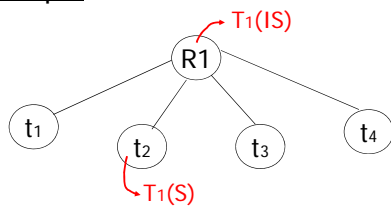


CS 245

Notes 09

94

Example

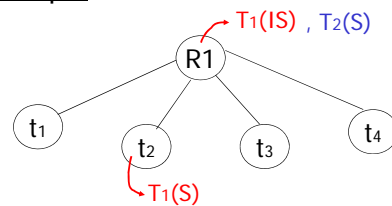


CS 245

Notes 09

95

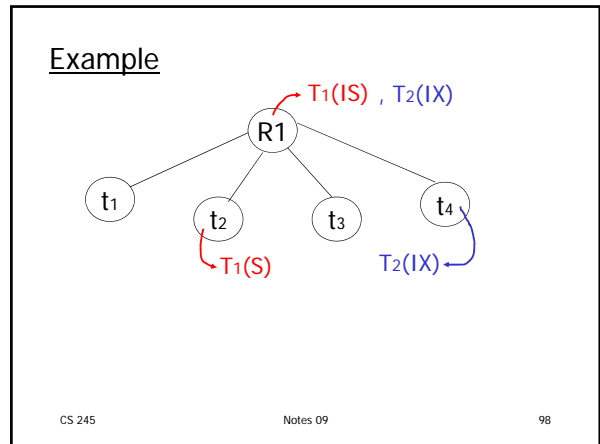
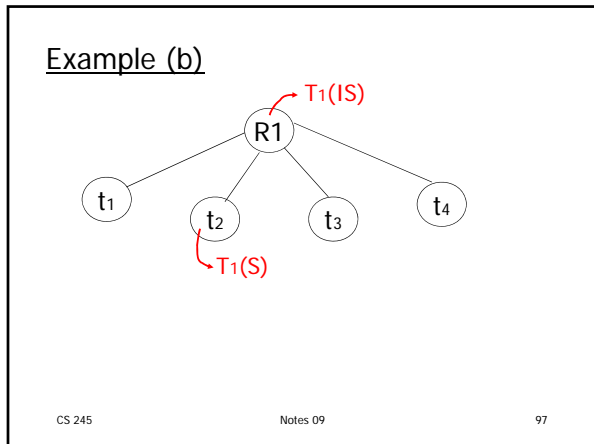
Example



CS 245

Notes 09

96



### Multiple granularity

Comp Requestor

		IS	IX	S	SIX	X
Holder	IS					
	IX					
	S					
	SIX					
	X					

CS 245 Notes 09 99

### Multiple granularity

Comp Requestor

		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

CS 245 Notes 09 100

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

CS 245 Notes 09 101

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

CS 245 Notes 09 102

### 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

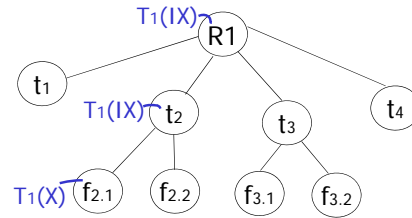
CS 245

Notes 09

103

### Exercise:

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



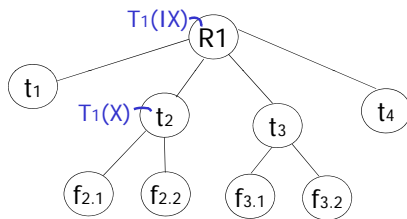
CS 245

Notes 09

104

### Exercise:

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



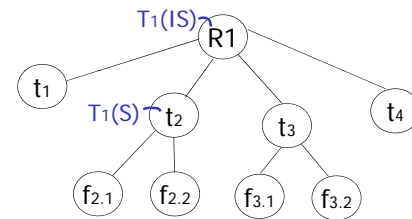
CS 245

Notes 09

105

### Exercise:

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



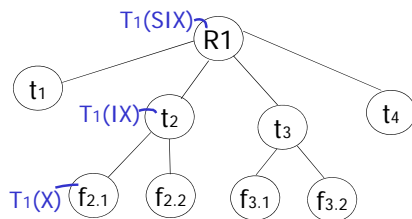
CS 245

Notes 09

106

### Exercise:

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



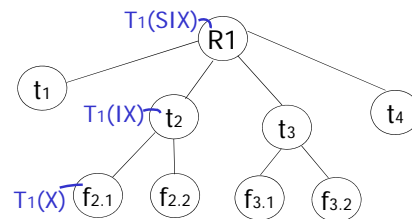
CS 245

Notes 09

107

### Exercise:

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

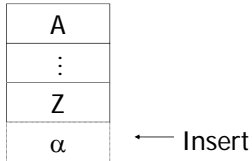


CS 245

Notes 09

108

## Insert + delete operations



CS 245

Notes 09

109

## 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

CS 245

Notes 09

110

## Still have a problem: **Phantoms**

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

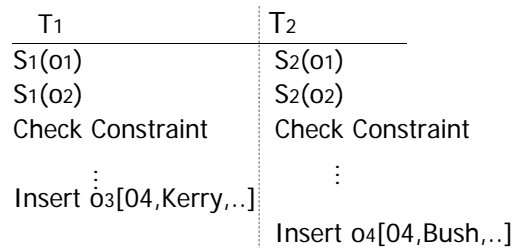
R	E#	Name	....
o1	55	Smith	
o2	75	Jones	

CS 245

Notes 09

111

- $T_1$ : Insert <04,Kerry,...> into R  
 $T_2$ : Insert <04,Bush,...> into R



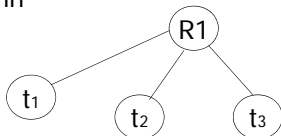
CS 245

Notes 09

112

## Solution

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

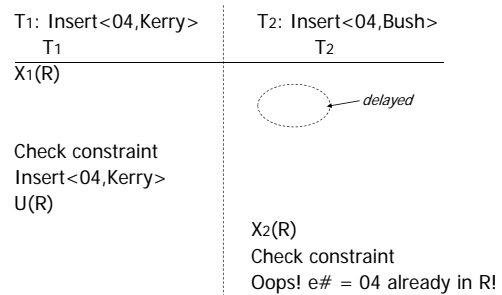


CS 245

Notes 09

113

## Back to example



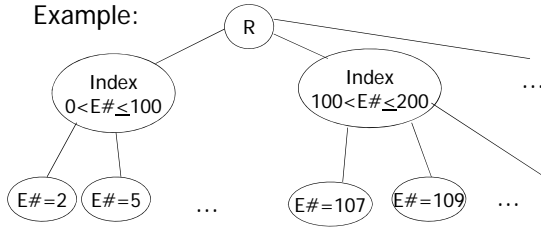
CS 245

Notes 09

114

Instead of using R, can use index on R:

Example:



CS 245

Notes 09

115

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

CS 245

Notes 09

116

Next:

- Tree-based concurrency control
- Validation concurrency control

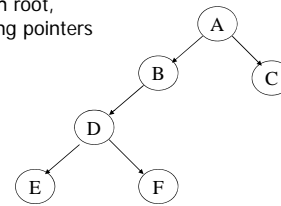
CS 245

Notes 09

117

Example

- all objects accessed through root, following pointers



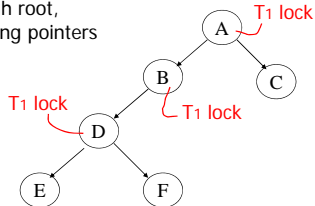
CS 245

Notes 09

118

Example

- all objects accessed through root, following pointers



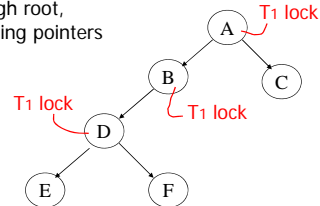
CS 245

Notes 09

119

Example

- all objects accessed through root, following pointers



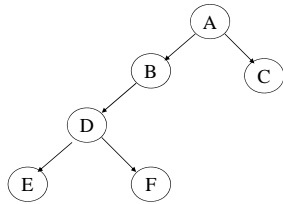
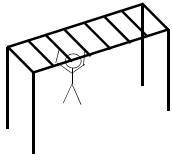
- can we release A lock if we no longer need A??

CS 245

Notes 09

120

Idea: traverse like "Monkey Bars"

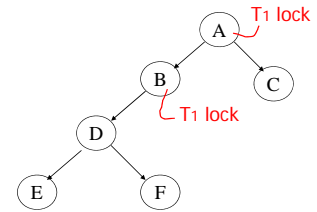


CS 245

Notes 09

121

Idea: traverse like "Monkey Bars"

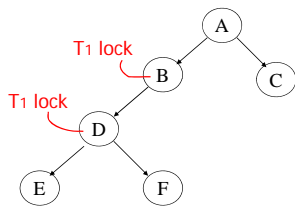
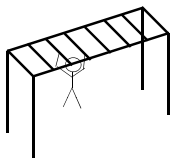


CS 245

Notes 09

122

Idea: traverse like "Monkey Bars"



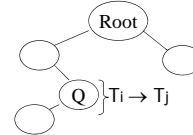
CS 245

Notes 09

123

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

CS 245

Notes 09

124

Rules: tree protocol (exclusive locks)

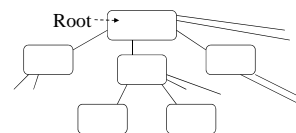
- (1) First lock by  $T_i$  may be on any item
- (2) After that, item  $Q$  can be locked by  $T_i$  only if  $\text{parent}(Q)$  locked by  $T_i$
- (3) Items may be unlocked at any time
- (4) After  $T_i$  unlocks  $Q$ , it cannot relock  $Q$

CS 245

Notes 09

125

- 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

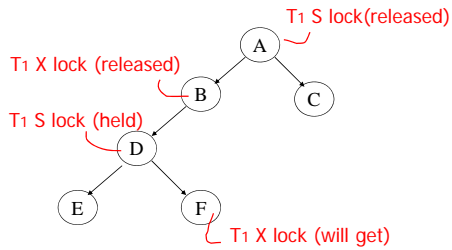
CS 245

Notes 09

126

## Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



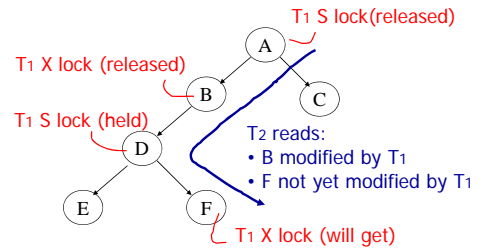
CS 245

Notes 09

127

## Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



CS 245

Notes 09

128

## 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

CS 245

Notes 09

129

## 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

CS 245

Notes 09

130

## Key idea

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

CS 245

Notes 09

131

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)

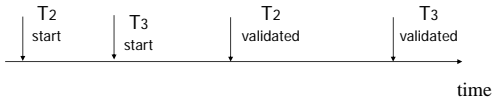
CS 245

Notes 09

132

Example of what validation must prevent:

$RS(T_2) = \{B\}$        $RS(T_3) = \{A, B\} \neq \phi$   
 $WS(T_2) = \{B, D\}$        $WS(T_3) = \{C\}$



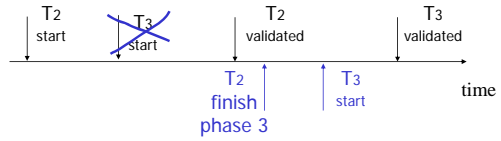
CS 245

Notes 09

133

Example of what validation must prevent:

$RS(T_2) = \{B\}$        $RS(T_3) = \{A, B\} \neq \phi$   
 $WS(T_2) = \{B, D\}$        $WS(T_3) = \{C\}$



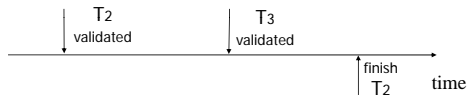
CS 245

Notes 09

134

Another thing validation must prevent:

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



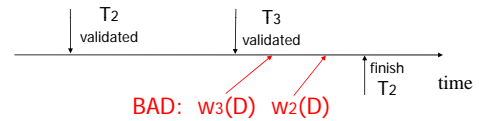
CS 245

Notes 09

135

Another thing validation must prevent:

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



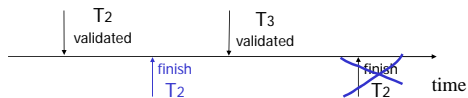
CS 245

Notes 09

136

Another thing validation must prevent:

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



CS 245

Notes 09

137

Validation rules for Tj:

- (1) When Tj starts phase 1:  
ignore(Tj) ← FIN
- (2) at Tj Validation:  
if check (Tj) then  
[ VAL ← VAL U {Tj};  
do write phase;  
FIN ← FIN U {Tj} ]

CS 245

Notes 09

138

Check (Tj):

```

For Ti ∈ VAL - IGNORE (Tj) DO
  IF [ WS(Ti) ∩ RS(Tj) ≠ ∅ OR
    Ti ∉ FIN ] THEN RETURN false;
RETURN true;

```

CS 245

Notes 09

139

Check (Tj):

```

For Ti ∈ VAL - IGNORE (Tj) DO
  IF [ WS(Ti) ∩ RS(Tj) ≠ ∅ OR
    Ti ∉ FIN ] THEN RETURN false;
RETURN true;

```

Is this check too restrictive ?

CS 245

Notes 09

140

### Improving Check(Tj)

```

For Ti ∈ VAL - IGNORE (Tj) DO
  IF [ WS(Ti) ∩ RS(Tj) ≠ ∅ OR
    (Ti ∉ FIN AND WS(Ti) ∩ WS(Tj) ≠ ∅) ]
    THEN RETURN false;
RETURN true;

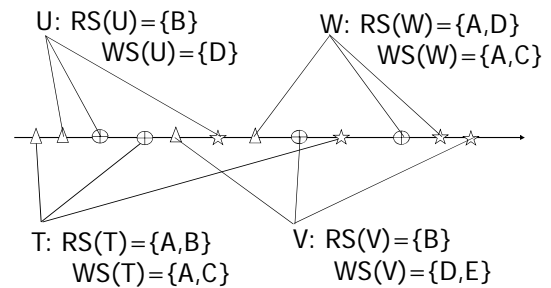
```

CS 245

Notes 09

141

### Exercise:

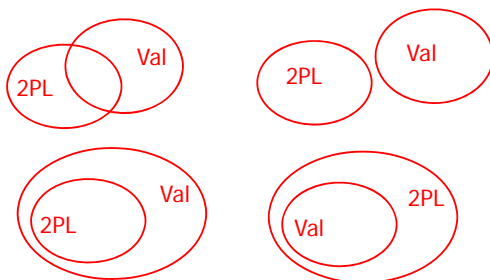


CS 245

Notes 09

142

### Is Validation = 2PL?



CS 245

Notes 09

143

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

- S2 can be achieved with 2PL: I2(y) w2(y) I1(x) w1(x) u1(x) I2(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.

CS 245

Notes 09

144

## 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)

CS 245

Notes 09

145

- 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  $\notin$  FIN AND  $WS(T1) \cap WS(T2) \neq \emptyset$
  - contradiction!

CS 245

Notes 09

146

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

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

CS 245

Notes 09

147

## Summary

Have studied C.C. mechanisms used in practice

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

CS 245

Notes 09

148