CS 245: Database System Principles

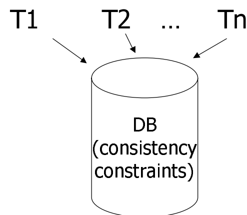
Notes 09: Concurrency Control

Hector Garcia-Molina
(Some modifications by Chris Olston)

ACID Properties

- Atomicity
  - Actions are never left partially executed
- Consistency
  - Actions leave the DB in a consistent state
- Isolation
  - Actions are not affected by other concurrent actions
- Durability
  - Effects of completed actions are resilient against system failures

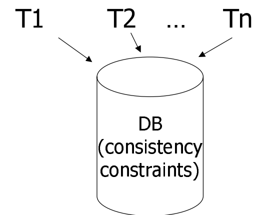
Chapter 9
Concurrency Control



A
C
Isolation
D

Desired Effect:
Isolation

Transactions not affected by other concurrent transactions



Example:

T1:	T2:
Read(A)	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

Schedule A

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

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

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

Schedule D

T1	T2	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

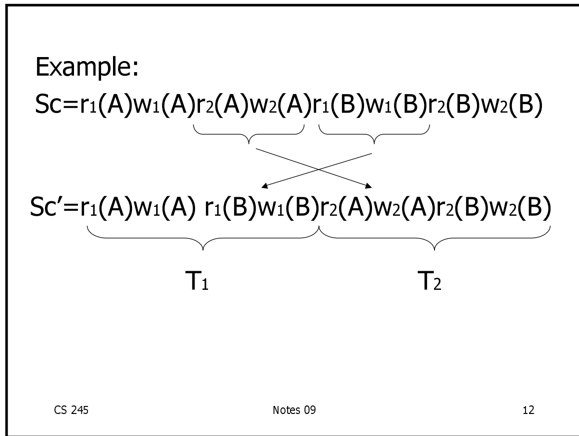
Same as Schedule D but with new T2'

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

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

Example:

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



However, for Sd:

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

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

- $T_2 \rightarrow T_1$
- Also, $T_1 \rightarrow T_2$

T_1 T_2

⇨ Sd cannot be rearranged into a serial schedule

⇨ Sd is not "equivalent" to any serial schedule

⇨ Sd is "bad"

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 ⇒ Sc is "equivalent" to a serial schedule (in this case T_1,T_2)

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

What about concurrent actions?

T1 issues read(x,t)

System issues input(x)

Input(x) completes

$t \leftarrow x$

time

T2 issues write(y,u)

System issues input(y)

input(y) completes

$y \leftarrow u$

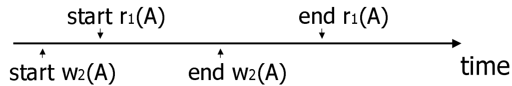
System issues output(y)

output(y) completes

So net effect is either

- $S=\ldots r_1(x)\ldots w_2(y)\ldots$ or
- $S=\ldots w_2(y)\ldots r_1(x)\ldots$

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"

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.

Definition

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

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

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?

Exercise 2:

- What is P(S) for
  $S = r_1(A)\ w_1(B)\ r_1(C)\ w_2(C)\ w_2(A)\ w_3(A)\ r_4(A)\ w_4(D)$
- S serial $\Rightarrow$ P(S) acyclic

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

$\Rightarrow S_1 = \ldots p_i(A) \ldots q_j(A) \ldots$
$\quad S_2 = \ldots q_j(A) \ldots p_i(A) \ldots$
$\Big\}$ p_i, q_j conflict

$\Rightarrow S_1$, S_2 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

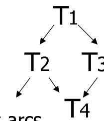
$\Rightarrow \exists S_s$: S_s, S_1 conflict equivalent

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

$\Rightarrow P(S_1)$ 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

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

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

(4) repeat above steps to serialize rest!

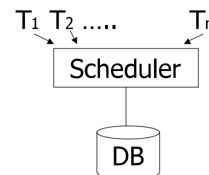
How to enforce serializable schedules?

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

T_1 T_2 ..... T_n

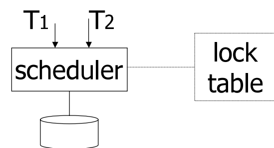


A locking protocol

Two new actions:

lock (exclusive): 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 = …….. l_i(A) ……….. u_i(A) ……...

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

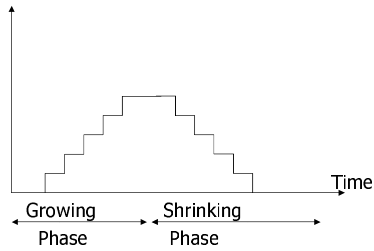
T1	T2	A	B
		25	25
l_1(A);Read(A)			
A ← A+100;Write(A);u_1(A)		125	
	l_2(A);Read(A)		
	A ← Ax2;Write(A);u_2(A)	250	
	l_2(B);Read(B)		
	B ← Bx2;Write(B);u_2(B)		50
l_1(B);Read(B)			
B ← B+100;Write(B);u_1(B)			150
		250	150

Rule #3 Two phase locking (2PL)

for transactions

T_i = ……. l_i(A) ………... u_i(A) ………

← no unlocks   no locks →



Schedule G

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

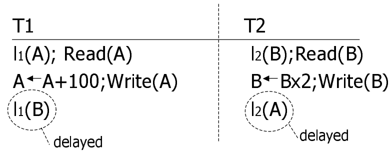
Schedule G

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

Schedule G

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

Schedule H (T2 reversed)



- 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!

Next step:

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

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 < $u_i(A)$, $u_j(A)$>, < $l_i(A)$, $r_j(A)$>,...

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

To help in proof:

Definition 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
(2PL) serializable schedule

Proof:

(1) Assume P(S) has cycle

$T_1 \rightarrow T_2 \rightarrow .... T_n \rightarrow T_1$

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

(3) Impossible, so P(S) acyclic

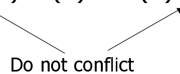
(4) $\Rightarrow$ S is conflict serializable

- 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:

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



Do not conflict

Instead:

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

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

$T_i = ...\ l\text{-}S_1(A)\ ...\ r_1(A)\ ...\ u_1(A)\ ...$

$T_i = ...\ l\text{-}X_1(A)\ ...\ w_1(A)\ ...\ u_1(A)\ ...$

- What about transactions that read and write same object?

Option 1: Request exclusive lock

$T_i = ...l\text{-}X_1(A)\ ...\ r_1(A)\ ...\ w_1(A)\ ...\ u(A)\ ...$

- 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=... l-S_1(A) … r_1(A) ... l-X_1(A) …w_1(A) ...u(A)…

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

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 transactions

No change except for upgrades:
(I) If upgrade gets more locks
(e.g., S → {S, X}) then no change!
(II) If upgrade releases read (shared) lock (e.g., S → X)
- can be allowed in growing phase

Theorem Rules 1,2,3 ⇒ 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)

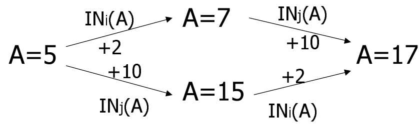
Lock types beyond S/X

Examples:
(1) increment lock
(2) update lock

Example (1): increment lock

- Atomic increment action: $IN_i(A)$

  {Read(A); A ← A+k; Write(A)}
- $IN_i(A)$, $IN_j(A)$ do not conflict!



Comp

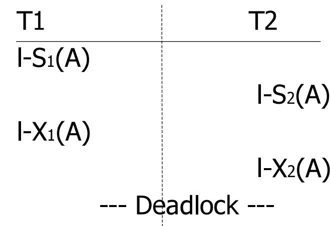
	S	X	I
S			
X			
I			

Comp

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

Update locks

A common deadlock problem with upgrades:



T1	T2
$l\text{-}S_1(A)$	
	$l\text{-}S_2(A)$
$l\text{-}X_1(A)$	
	$l\text{-}X_2(A)$

--- Deadlock ---

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			

Comp

New request

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

-> symmetric table?

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

S_1=...l-S_1(A)...l-S_2(A)...l-U_3(A)... { l-S_4(A)...? / l-U_4(A)...? }

- 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?

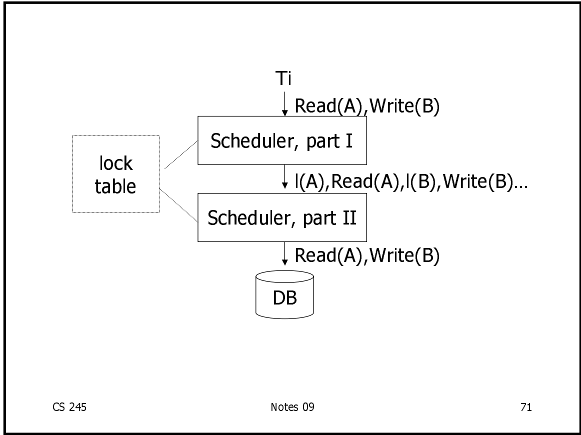
- Every system is different
  (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)
- But here is one (simplified) way ...

Sample Locking System:

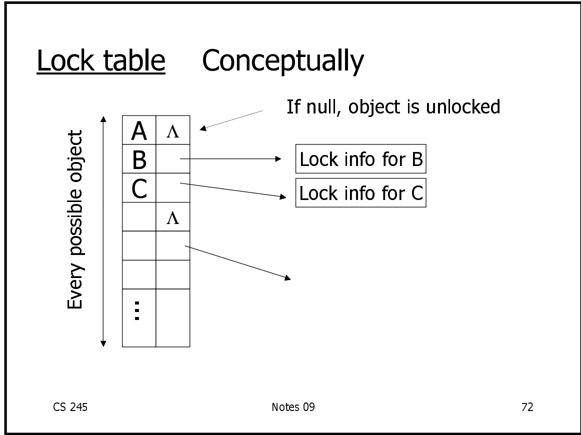
(1) Don't trust transactions to request/release locks
(2) Hold all locks until transaction commits

locks

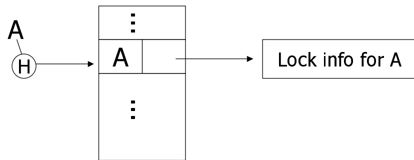
time



Lock table Conceptually



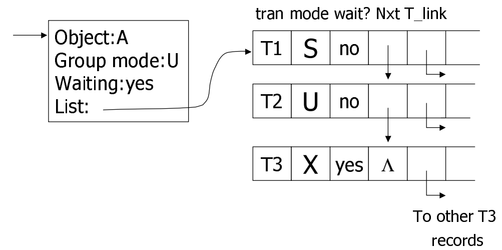
But use hash table:



If object not found in hash table, it is unlocked

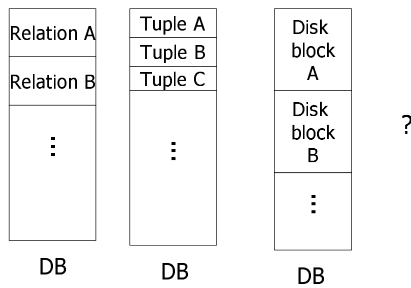
CS 245 Notes 09 73

Lock info for A - example



CS 245 Notes 09 74

What are the objects we lock?

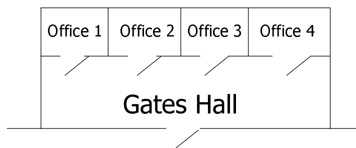


CS 245 Notes 09 75

- 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

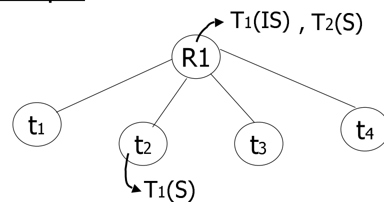
CS 245 Notes 09 76

We can have it both ways!!



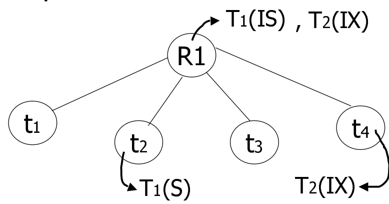
CS 245 Notes 09 77

Example



CS 245 Notes 09 78

Example



Multiple granularity

Comp — Requestor (columns), Holder (rows)

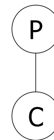
Holder \ Requestor	IS	IX	S	SIX	X
IS					
IX					
S					
SIX					
X					

Multiple granularity

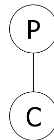
Comp — Requestor (columns), Holder (rows)

Holder \ Requestor	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	



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

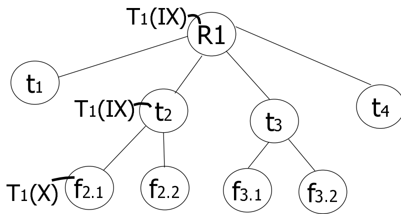


Rules

(1) Follow multiple granularity comp function
(2) Lock root of tree first, any mode
(3) Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
(4) Node Q can be locked by Ti in X,SIX,IX only if parent(Q) locked by Ti in IX,SIX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q's children are locked by Ti

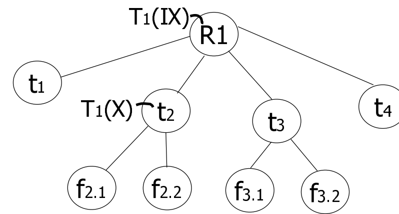
Exercise:

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



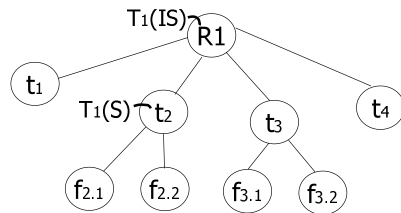
Exercise:

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



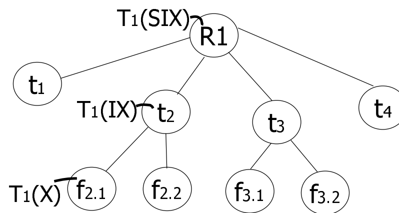
Exercise:

- Can T_2 access object $f_{3.1}$ in X mode? What locks will T_2 get?



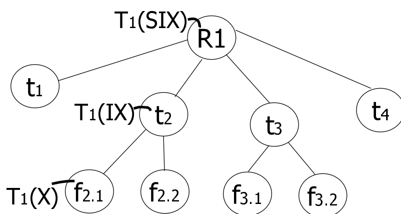
Exercise:

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

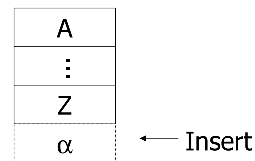


Exercise:

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



Insert + delete operations



Modifications to locking rules:

(1) Get exclusive lock on A before deleting A

(2) At insert A operation by Ti, Ti is given exclusive lock on A

Still have a problem: **Phantoms**

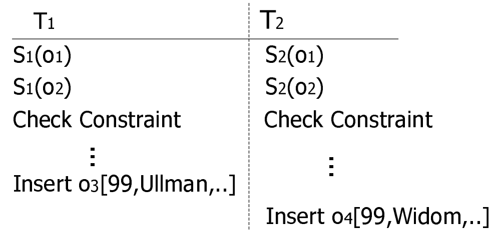
Example: relation R (E#,name,…)
constraint: E# is key
use tuple locking

R

	E#	Name	….
o1	55	Smith	
o2	75	Jones	

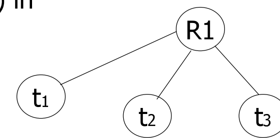
T_1: Insert <99,Ullman,…> into R

T_2: Insert <99,Widom,…> into R

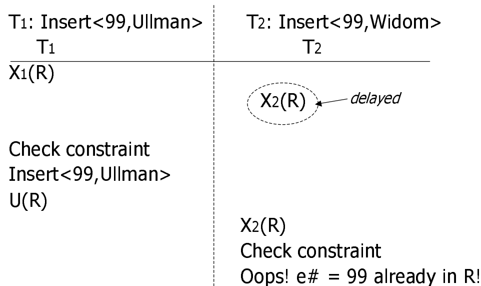


Solution

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

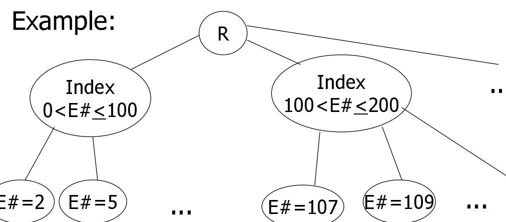


Back to example



Instead of using R, can use index on R:

Example:

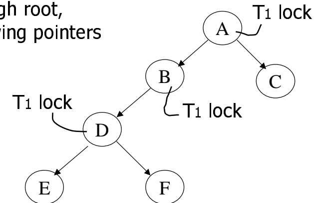


Next:

- Tree-based concurrency control
- Validation concurrency control

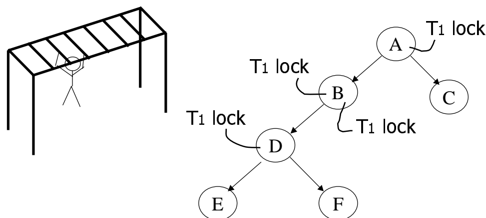
Example

- all objects accessed through root, following pointers



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

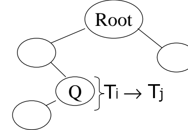
Idea: traverse like "Monkey Bars"



- Idea: don't let another transaction "slip past"

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

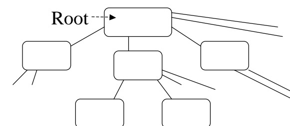


- Actually works if we don't always start at root

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

Validation

Transactions have 3 phases:

(1) Read
- all DB values read
- writes to temporary storage
- no locking

(2) Validate
- check if schedule so far is serializable

(3) Write
- if validate ok, write to DB

Key idea

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

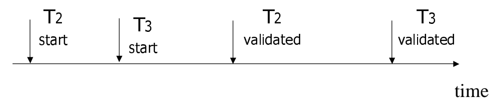
To implement validation, system keeps two sets:

- 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:

$RS(T_2)=\{B\}$ $\cap$ $RS(T_3)=\{A,B\} \neq \phi$

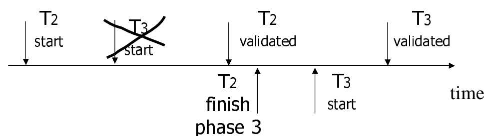
$WS(T_2)=\{B,D\}$ $WS(T_3)=\{C\}$



Example of what validation must ~~prevent~~ allow:

$RS(T_2)=\{B\}$ $\cap$ $RS(T_3)=\{A,B\} \neq \phi$

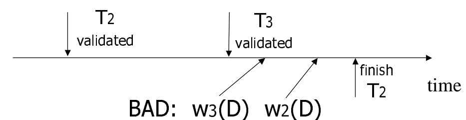
$WS(T_2)=\{B,D\}$ $WS(T_3)=\{C\}$



Another thing validation must prevent:

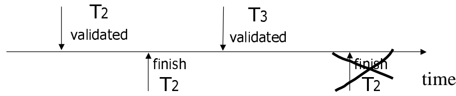
$RS(T_2)=\{A\}$ $RS(T_3)=\{A,B\}$

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



Another thing validation must ~~prevent~~ allow:

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



Validation rules for T_j:

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

Check (T_j):

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

Is this check too restrictive ?

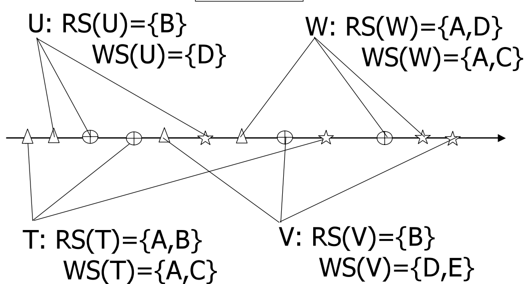
Improving Check(T_j)

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

Exercise:

U: RS(U)={B}
WS(U)={D}

W: RS(W)={A,D}
WS(W)={A,C}



T: RS(T)={A,B}
WS(T)={A,C}

V: RS(V)={B}
WS(V)={D,E}

Validation (also called optimistic concurrency control) is useful in some cases:
- Conflicts rare
- System resources plentiful
- Have real time constraints

Summary

Have studied C.C. mechanisms used in practice

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

CS 245 Notes 09 115