



PURDUE
UNIVERSITY

CS 54100
Transactions

Chris Clifton
2 April, 2012



Indiana
Center for
Database
Systems



Goal: Integrity Across Sequence of Operations

- Update should complete entirely
 - update stipend set stipend = stipend*1.03;
 - What if it gets halfway and the machine crashes?
- What about multiple operations?
 - Withdraw x from Account1
 - ~~Deposit x into Account2~~
- Simultaneous operations?
 - Print paychecks while stipend being updated

4/2/2012 Chris Clifton - CS541 2



Solution: *Transaction*

- Sequence of operations grouped into a transaction
 - Externally viewed as *Atomic*: All happens at once
 - DBMS manages so even the programmer gets this view

4/2/2012

Chris Clifton - CS541

3



ACID properties

Transactions have:

- Atomicity
 - All or nothing
- Consistency
 - Changes to values maintain integrity
- Isolation
 - Transaction occurs as if nothing else happening
- Durability
 - Once completed, changes are permanent

4/2/2012

Chris Clifton - CS541

4

Transactions



- ❖ Concurrent execution of user programs is essential for good DBMS performance.
 - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- ❖ A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- ❖ A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes.

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

5

Concurrency in a DBMS



- ❖ Users submit transactions, and can think of each transaction as executing by itself.
 - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
 - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
 - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
 - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- ❖ Issues: Effect of *interleaving* transactions, and *crashes*.

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

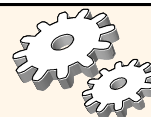
6

Atomicity of Transactions



- ❖ A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- ❖ A very important property guaranteed by the DBMS for all transactions is that they are atomic. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
 - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.

Example



- ❖ Consider two transactions (*Xacts*):

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- ❖ Intuitively, the first transaction is transferring \$100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- ❖ There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (Contd.)



- ❖ Consider a possible interleaving (*schedule*):

T1:	A=A+100,	B=B-100
T2:	A=1.06*A,	B=1.06*B

- ❖ This is OK. But what about:

T1:	A=A+100,	B=B-100
T2:	A=1.06*A, B=1.06*B	

- ❖ The DBMS's view of the second schedule:

T1:	R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)	

Scheduling Transactions



- ❖ *Serial schedule*: Schedule that does not interleave the actions of different transactions.
 - ❖ *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
 - ❖ *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.
- (Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with Interleaved Execution

- ❖ Reading Uncommitted Data (WR Conflicts, “dirty reads”):

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), C	

- ❖ Unrepeatable Reads (RW Conflicts):

T1:	R(A),	R(A), W(A), C
T2:	R(A), W(A), C	

Anomalies (Continued)

- ❖ Overwriting Uncommitted Data (WW Conflicts):

T1:	W(A),	W(B), C
T2:	W(A), W(B), C	

Aborting a Transaction



- ❖ If a transaction T_i is aborted, all its actions have to be undone. Not only that, if T_j reads an object last written by T_i , T_j must be aborted as well!
- ❖ Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
 - If T_i writes an object, T_j can read this only after T_i commits.
- ❖ In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

13

The Log



- ❖ The following actions are recorded in the log:
 - *Ti writes an object*: the old value and the new value.
 - Log record must go to disk *before* the changed page!
 - *Ti commits/aborts*: a log record indicating this action.
- ❖ Log records are chained together by Xact id, so it's easy to undo a specific Xact.
- ❖ Log is often *duplexed* and *archived* on stable storage.
- ❖ All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

14

Recovering From a Crash



- ❖ There are 3 phases in the *Aries* recovery algorithm:
 - **Analysis:** Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
 - **Redo:** Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
 - **Undo:** The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke

15

PURDUE
UNIVERSITY

CS 54100
Concurrency Control

Chris Clifton
2 April, 2012

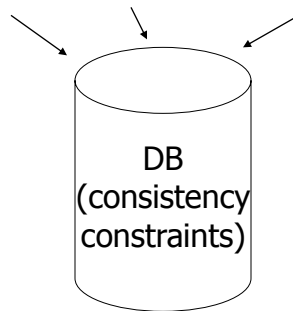




Chapters 16-17

Concurrency Control

T1 T2 ... Tn



4/2/2012

Chris Clifton - CS541

17



Example:

T1: Read(A)	T2: Read(A)
$A \leftarrow A+100$	$A \leftarrow A \times 2$
Write(A)	Write(A)
Read(B)	Read(B)
$B \leftarrow B+100$	$B \leftarrow B \times 2$
Write(B)	Write(B)

Constraint: $A=B$

4/2/2012

Chris Clifton - CS541

18




Schedule A

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




Schedule B

T1	T2	A	B
		25	25
	Read(A); A \leftarrow A \times 2;		
	Write(A);	50	
	Read(B); B \leftarrow B \times 2;		
	Write(B);		50
Read(A); A \leftarrow A+100			
Write(A);		150	
Read(B); B \leftarrow B+100;			
Write(B);			150
		150	150




Schedule C

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



Schedule D


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



Schedule E

Same as Schedule D
but with new T2'

			A	B
T1	T2'		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:

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

4/2/2012 Chris Clifton - CS541 24

Example:

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

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

T_1

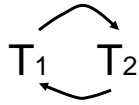
T_2

However, for S_d :

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

- 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, $T_1 \rightarrow T_2$



S_d cannot be rearranged
into a serial schedule



S_d is not “equivalent” to
any serial schedule



S_d is “bad”

Returning to S_c

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



$T_1 \rightarrow T_2$



$T_1 \rightarrow T_2$



- no cycles $\Rightarrow S_c$ 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: $r_1(A)$ $w_2(A)$ $w_1(A)$
 $w_2(A)$ $r_1(A)$ $w_2(A)$

Schedule: represents chronological order in which actions are executed

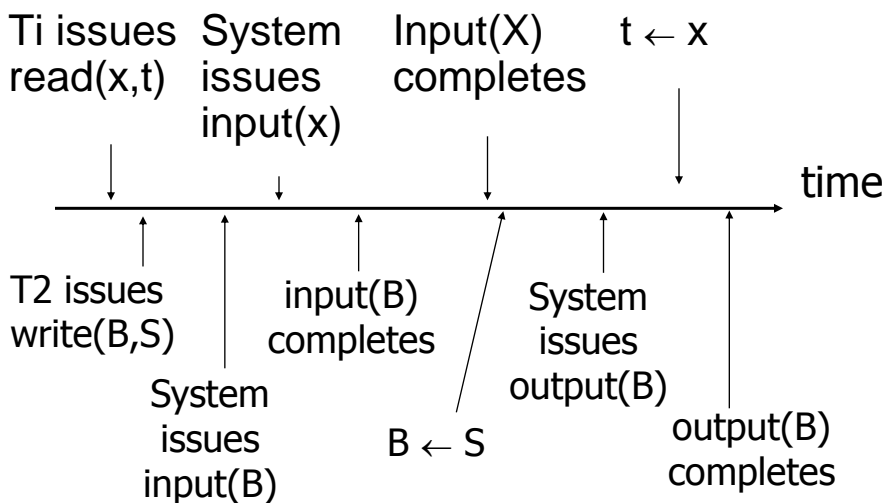
Serial schedule: no interleaving of actions or transactions

4/2/2012

Chris Clifton - CS541

29

What about concurrent actions?





So net effect is either

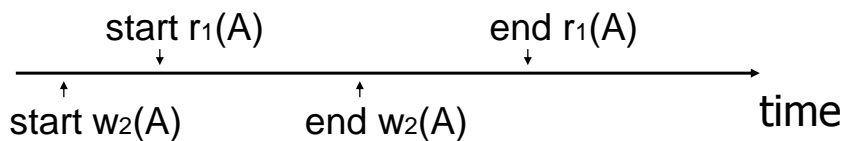
- $S = \dots r_1(x) \dots w_2(b) \dots$ or
- $S = \dots w_2(B) \dots r_1(x) \dots$

4/2/2012

Chris Clifton - CS541

31

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.

4/2/2012

Chris Clifton - CS541

33



Definition

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

4/2/2012

Chris Clifton - CS541

34



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

4/2/2012

Chris Clifton - CS541

35




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?

4/2/2012

Chris Clifton - CS541


36



PURDUE
UNIVERSITY

CS 54100
Transactions

Chris Clifton
4 April, 2012



Indiana
Center for
Database
Systems



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

$$\begin{array}{l} \Rightarrow S_1 = \dots p_i(A) \dots q_j(A) \dots \\ \quad S_2 = \dots q_j(A) \dots p_i(A) \dots \end{array} \quad \left\{ \begin{array}{l} p_i, q_j \\ \text{conflict} \end{array} \right.$$

$\Rightarrow S_1, S_2$ not conflict equivalent

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

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 $\iff S_1$ 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

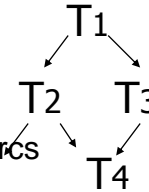
Theorem

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

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

Transform S_1 as follows:

- (1) Take T_1 to be transaction with no incident arcs
- (2) Move all T_1 actions to the front



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

- (3) we now have $S_1 = \leftarrow T_1 \text{ actions} \rightarrow \dots \text{rest} \dots$
- (4) repeat above steps to serialize rest!



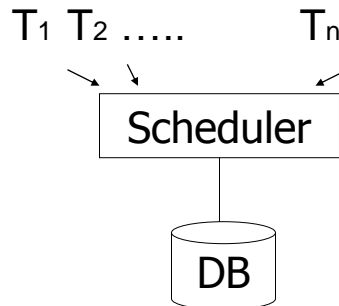
How to enforce serializable schedules?

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



4/2/2012

Chris Clifton - CS541

43

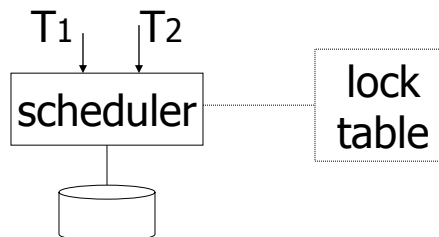


A locking protocol

Two new actions:

lock (exclusive): $li(A)$

unlock: $ui(A)$



4/2/2012

Chris Clifton - CS541

44



Rule #1: Well-formed transactions

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

4/2/2012

Chris Clifton - CS541

45



Rule #2 Legal scheduler

S = l_i(A) u_i(A)

↔
no l_j(A)

4/2/2012

Chris Clifton - CS541

46



Exercise:

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

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

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

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

4/2/2012

Chris Clifton - CS541

47

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

l₁(A);Read(A)A ← A+100;Write(A);u₁(A)l₁(B);Read(B)B ← B+100;Write(B);u₁(B)

T2

l₂(A);Read(A)A ← Ax2;Write(A);u₂(A)l₂(B);Read(B)B ← Bx2;Write(B);u₂(B)

4/2/2012

Chris Clifton - CS541

49



Schedule F

T1

l₁(A);Read(A)A ← A+100;Write(A);u₁(A)l₁(B);Read(B)B ← B+100;Write(B);u₁(B)

T2

l₂(A);Read(A)A ← Ax2;Write(A);u₂(A)l₂(B);Read(B)B ← Bx2;Write(B);u₂(B)

A	B
25	25
125	
250	
	50
	150
250	150



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

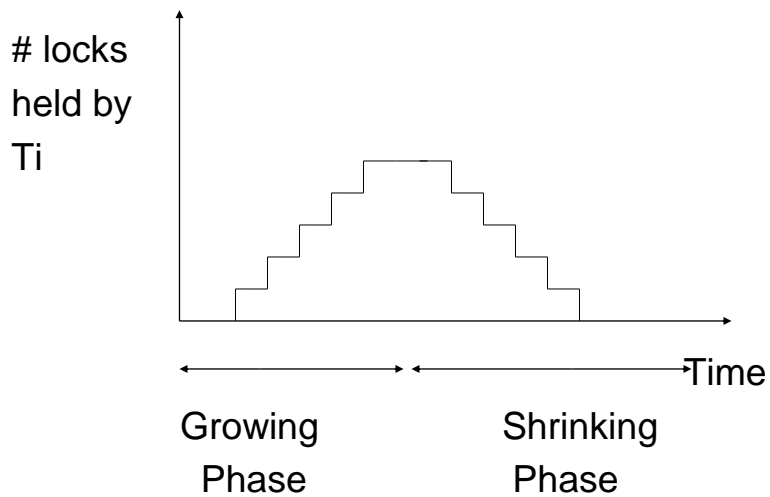
$T_i = \dots li(A) \dots ui(A) \dots$

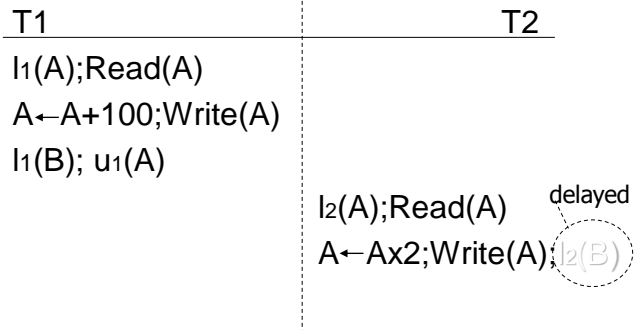
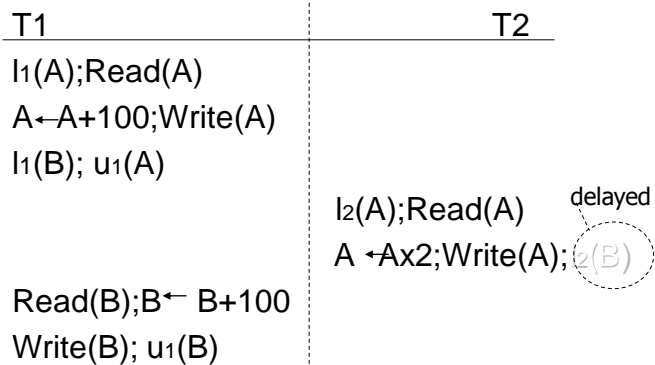


4/2/2012

Chris Clifton - CS541

51



Schedule GSchedule G

Schedule G

T1

l₁(A); Read(A)

A ← A+100; Write(A)

l₁(B); u₁(A)

Read(B); B ← B+100

Write(B); u₁(B)

T2

l₂(A); Read(A) delayedA ← Ax2; Write(A); l₂(B)l₂(B); u₂(A); Read(B)B ← Bx2; Write(B); u₂(B);Schedule H (T₂ reversed)

T1

l₁(A); Read(A)

A ← A+100; Write(A)

l₁(B)

delayed

T2


l₂(B); Read(B)

B ← Bx2; Write(B)

l₂(A)

delayed

- 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 $\langle u_i(A), u_j(A) \rangle$, $\langle l_i(A), r_j(A) \rangle$, ...

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

To help in proof:

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

Lemma

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

$$T_i \rightarrow T_j \text{ means that}$$

$$S = \dots p_i(A) \dots q_j(A) \dots; \quad p, q \text{ conflict}$$

$$\text{By rules 1,2:}$$

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

$$\begin{array}{ccc} \longleftarrow & | & | \longrightarrow \\ \text{By rule 3: } & SH(T_i) & SH(T_j) \end{array}$$

$$\text{So, } SH(T_i) <_S SH(T_j)$$

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: $SH(T_1) < SH(T_2) < \dots < SH(T_1)$

(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:

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

Do not conflict

Instead:

$S = \dots l_{s1}(A) \ r_1(A) \ l_{s2}(A) \ r_2(A) \ \dots \ u_{s1}(A) \ u_{s2}(A)$

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_i has locked A

Rule #1 Well formed transactions

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

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

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

$$\longleftrightarrow$$

no $I-X_j(A)$

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

$$\longleftrightarrow$$

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 $\text{comp}(t,r)$

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

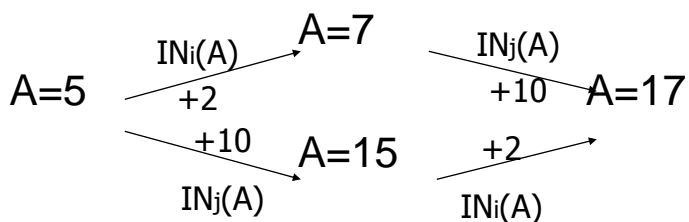
Lock types beyond S/X

Examples:

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

Example (1): increment lock

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



Comp

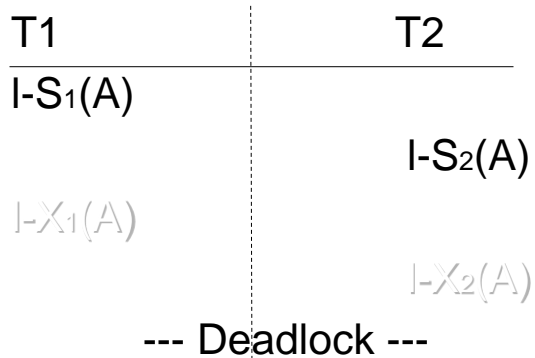
	S	X	I
S			
X			
I			

Comp

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

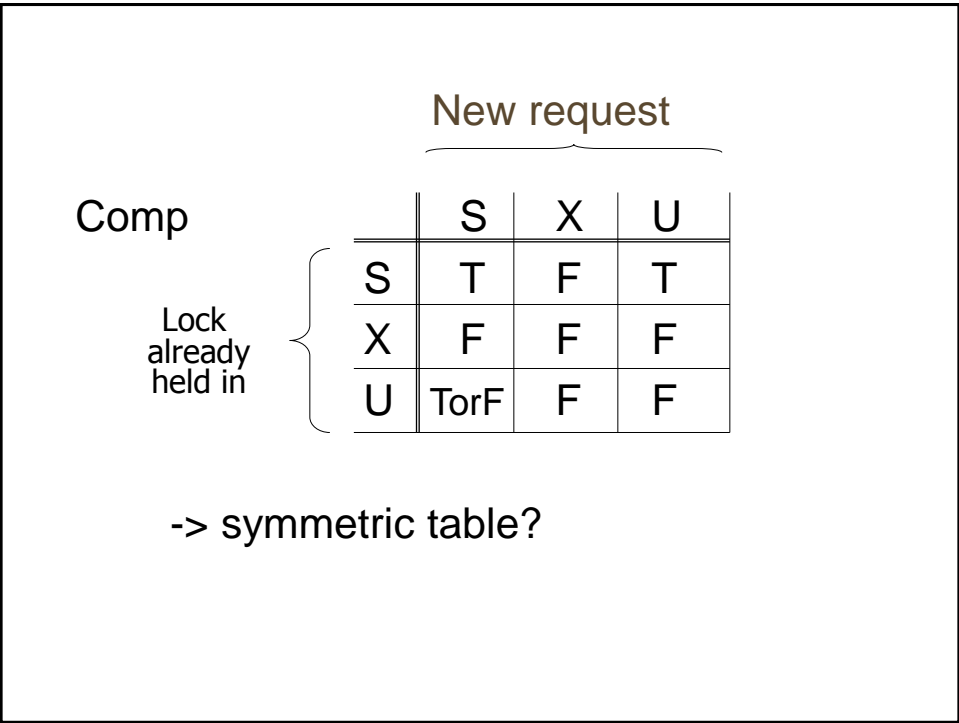
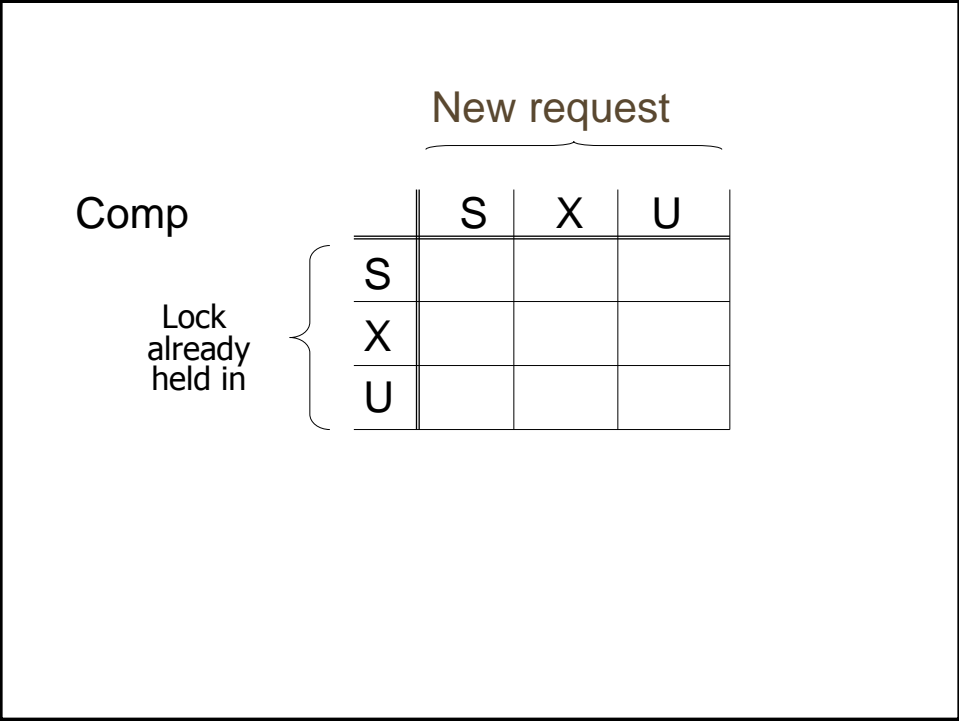
Update locks

A common deadlock problem with upgrades:



Solution

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



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

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

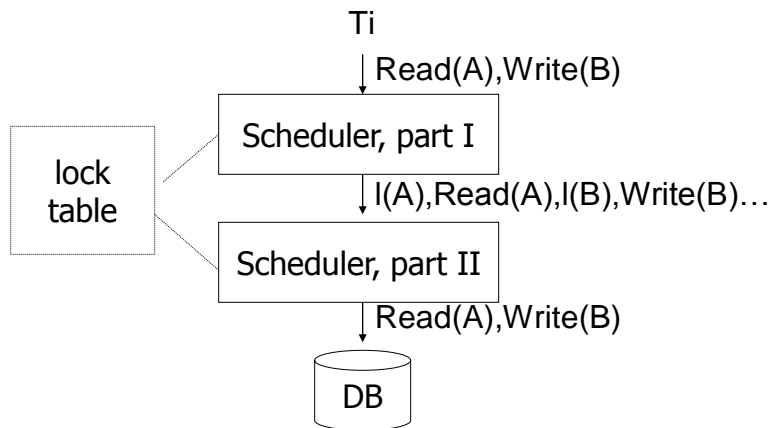
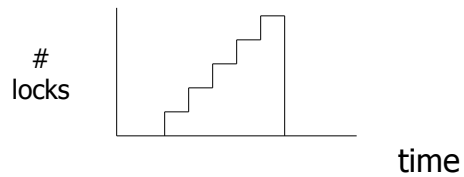
- To grant a lock in mode t, mode t must be compatible with all currently held locks on object

How does locking work in practice?

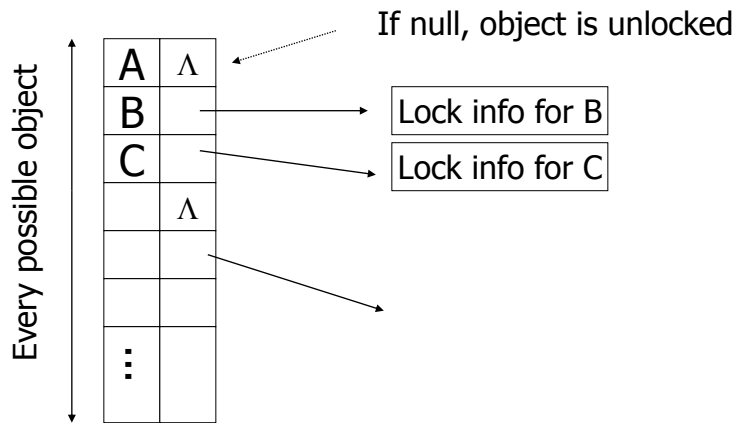
- Every system is different
(E.g., may not even provide CONFLICT-SERIALIZABLE schedules)
- But here is one (simplified) way ...

Sample Locking System:

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

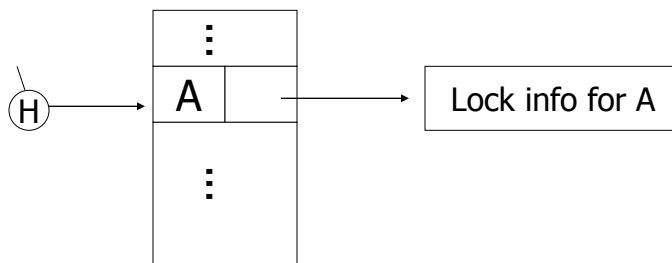


Lock table Conceptually



But use hash table:

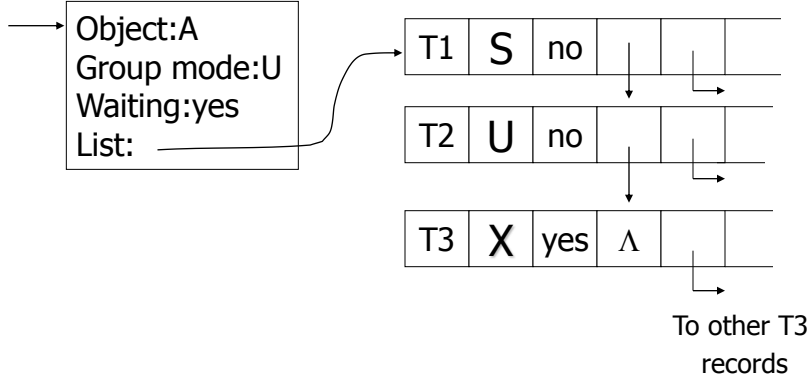
A



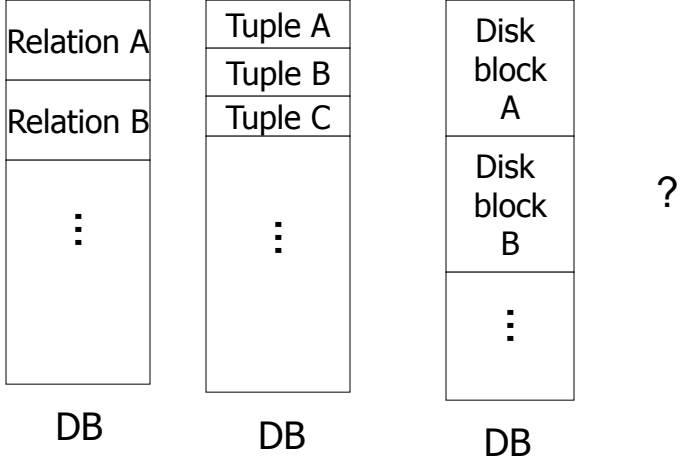
If object not found in hash table, it is unlocked

Lock info for A - example

transaction mode wait? Next T_link



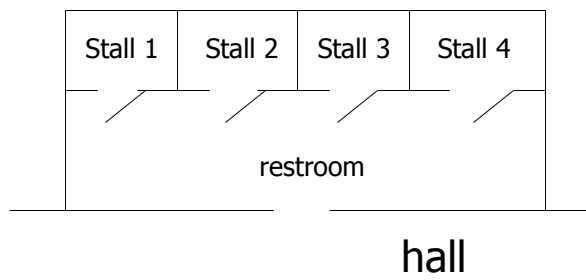
What are the objects we lock?

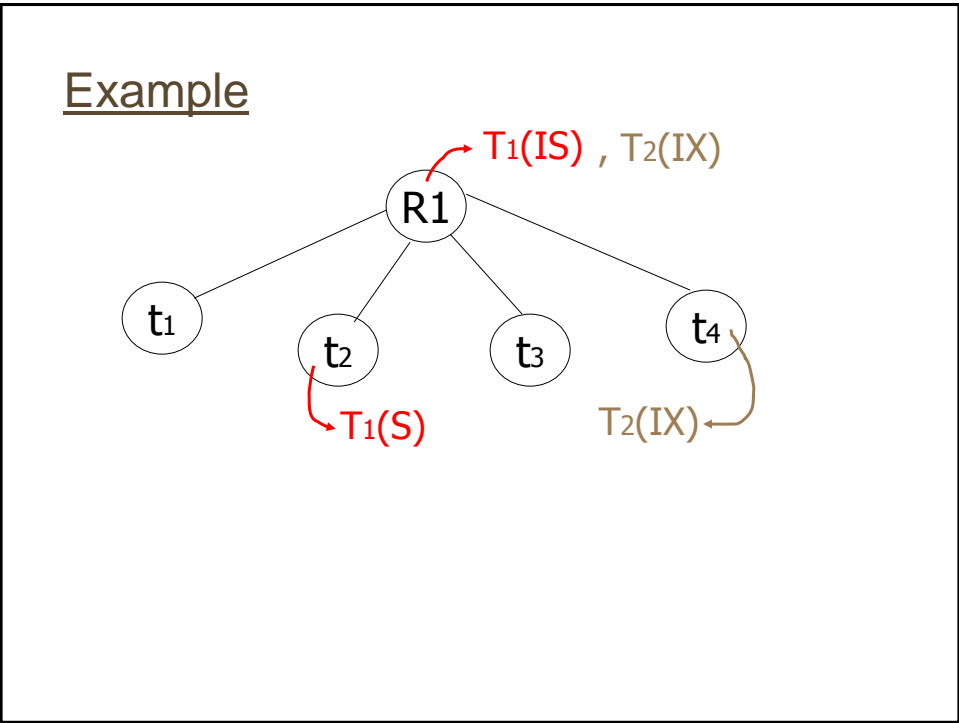
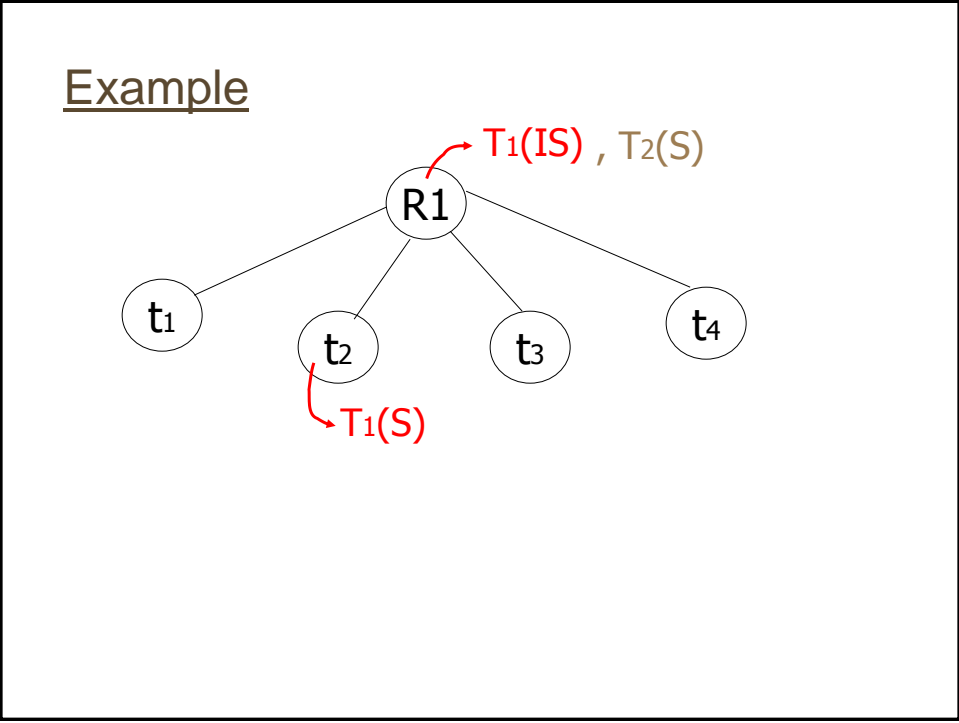


- Locking works in any case, but should we choose small or large objects?
- If we lock large objects (e.g., Relations)
 - Need few locks
 - Low concurrency
- If we lock small objects (e.g., tuples, fields)
 - Need more locks
 - More concurrency

We can have it both ways!!

Ask any janitor to give you the solution...





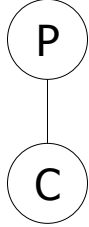
Multiple granularity

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

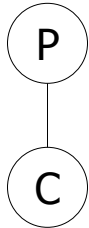
Multiple granularity

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

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



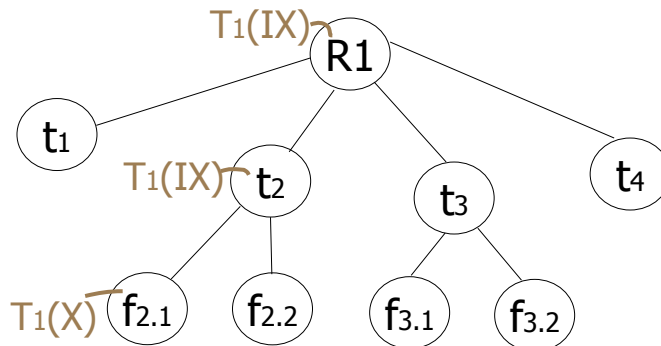
Rules

- (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

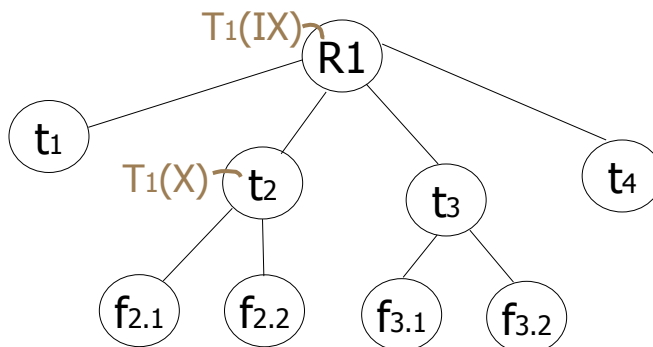
- End 11/4

Exercise:

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

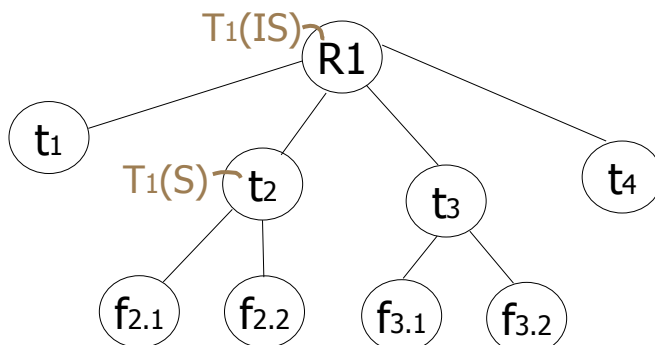
Exercise:

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

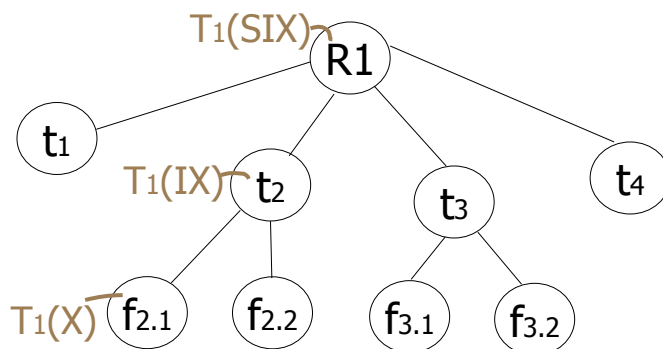


Exercise:

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

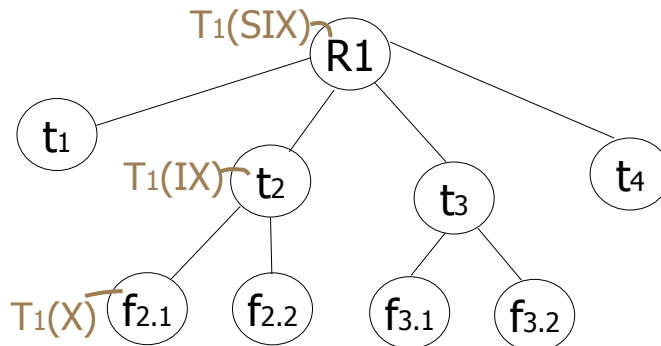
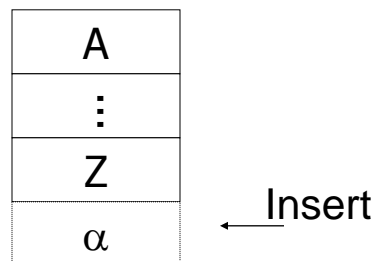
Exercise:

- 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?

Insert + delete operations

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

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	

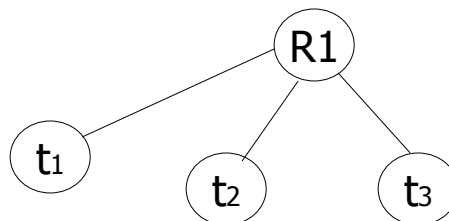
T₁: Insert <99,Gore,...> into R

T₂: Insert <99,Bush,...> into R

T ₁	T ₂
S ₁ (o ₁)	S ₂ (o ₁)
S ₁ (o ₂)	S ₂ (o ₂)
Check Constraint	Check Constraint
⋮	⋮
Insert o ₃ [99,Gore,...]	Insert o ₄ [99,Bush,...]

Solution

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



Back to example

T1: Insert<99,Gore>

T1

X1(R)

Check constraint

Insert<99,Gore>

U(R)

T2: Insert<99,Bush>

T2

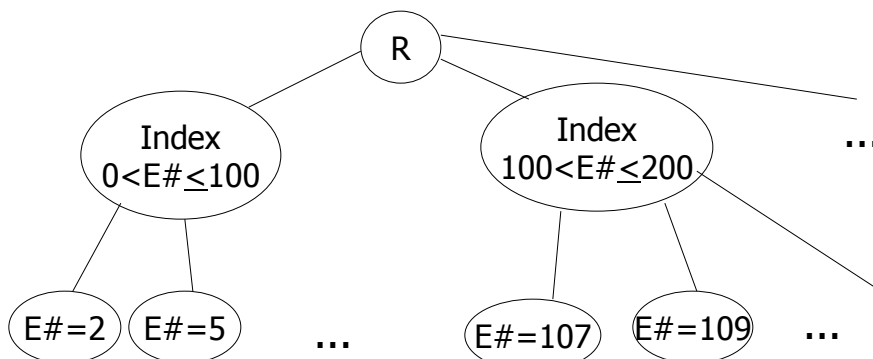
X2(R) ← *delayed*

X2(R)

Check constraint

Oops! e# = 99 already in R!

Instead of using R, can use index on R:
Example:



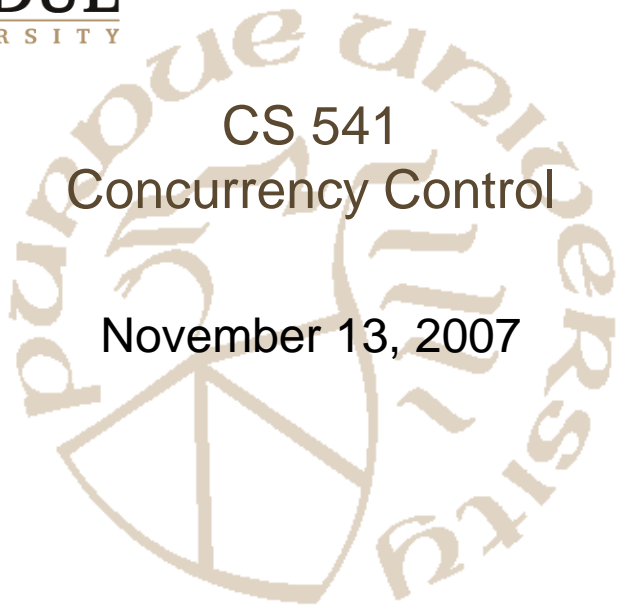
- This approach can be generalized to multiple indexes...

PURDUE
UNIVERSITY

CS 541
Concurrency Control
November 13, 2007

Fall 2007 Chris Clifton - CS541

Indiana
Center for
Database
Systems
112
TM

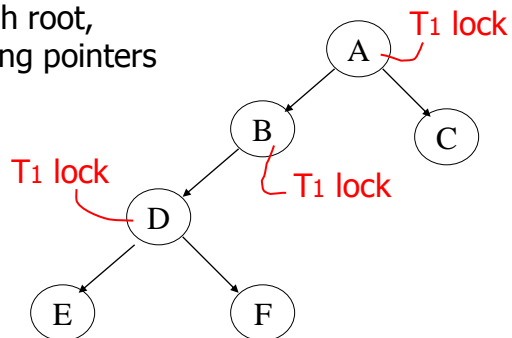
A large, faint watermark of the Purdue University seal is centered in the background of the slide. The seal features a shield with a book and a lamp, surrounded by the text "Purdue University".

Next:

- Tree-based concurrency control
- Validation concurrency control

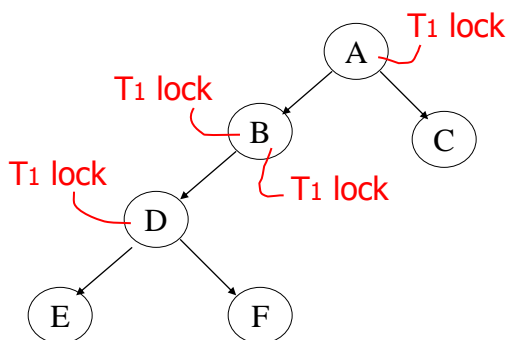
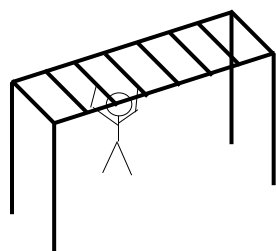
Example

- all objects accessed through root, following pointers



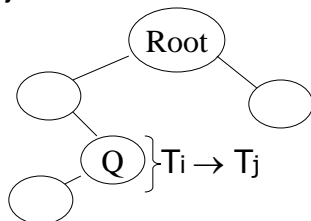
➤ can we release A lock if we no longer need A??

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

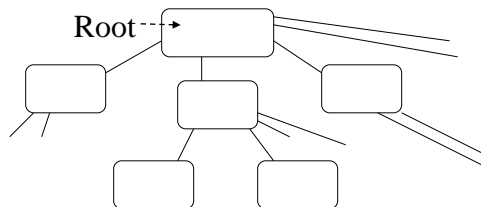


- Actually works if we don't always start at root

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 $\text{parent}(Q)$ locked by T_i
- (3) Items may be unlocked at any time
- (4) After T_i unlocks Q , it cannot relock Q

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

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)

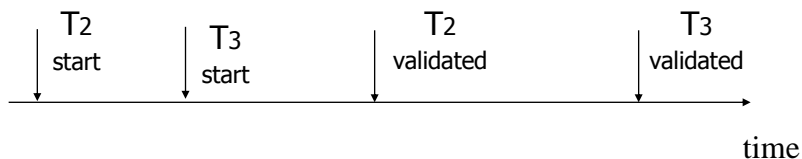
Example of what validation must prevent:

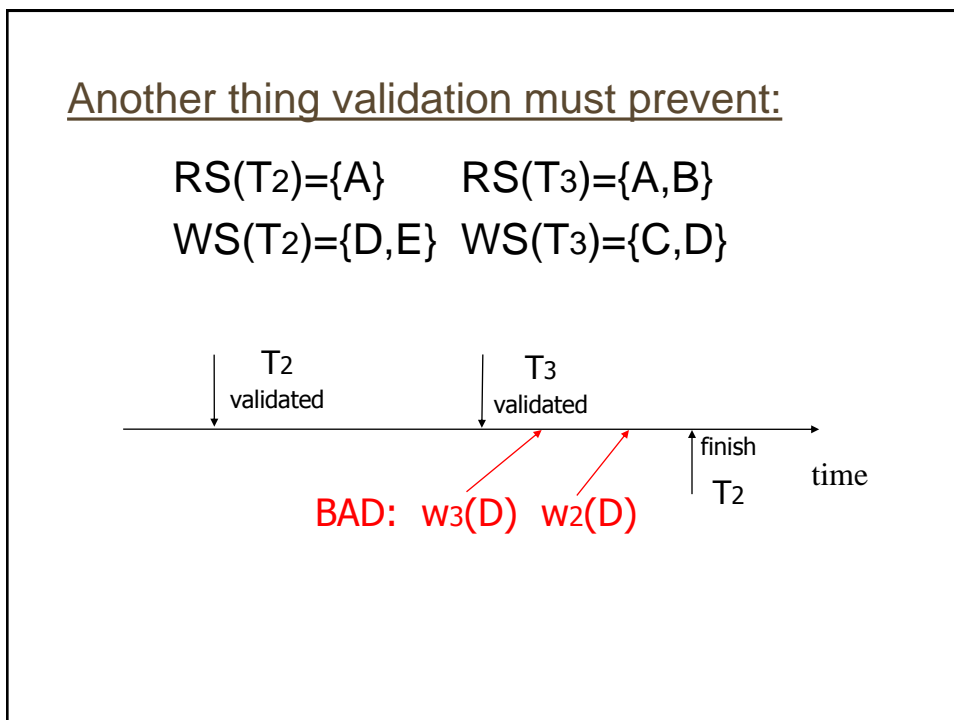
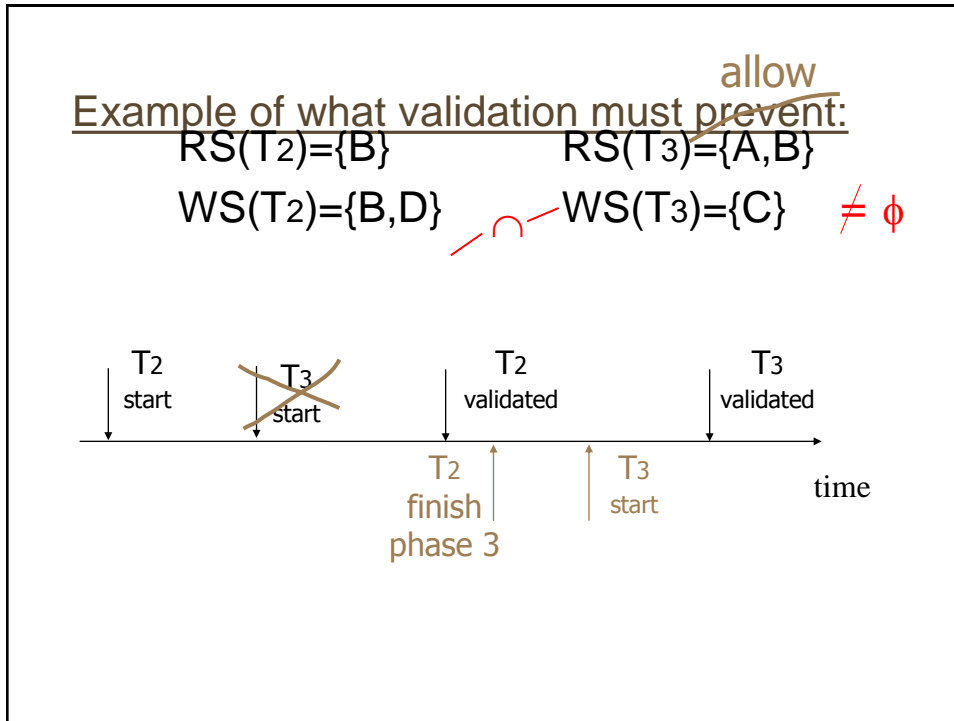
$RS(T_2) = \{B\}$

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

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

$WS(T_3) = \{C\} \neq \phi$

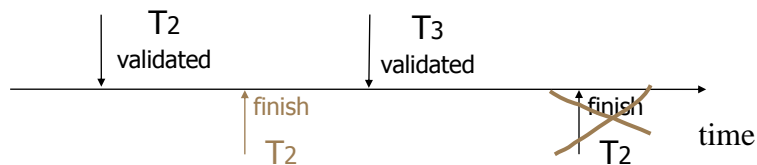




Another thing validation must ~~prevent~~ ^{allow}:

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

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



Validation rules for T_j :

(1) When T_j starts phase 1:

$ignore(T_j) \leftarrow FIN$

(2) at T_j Validation:

if check (T_j) then

[$VAL \leftarrow VAL \cup \{T_j\}$;

do write phase;

$FIN \leftarrow FIN \cup \{T_j\}$]

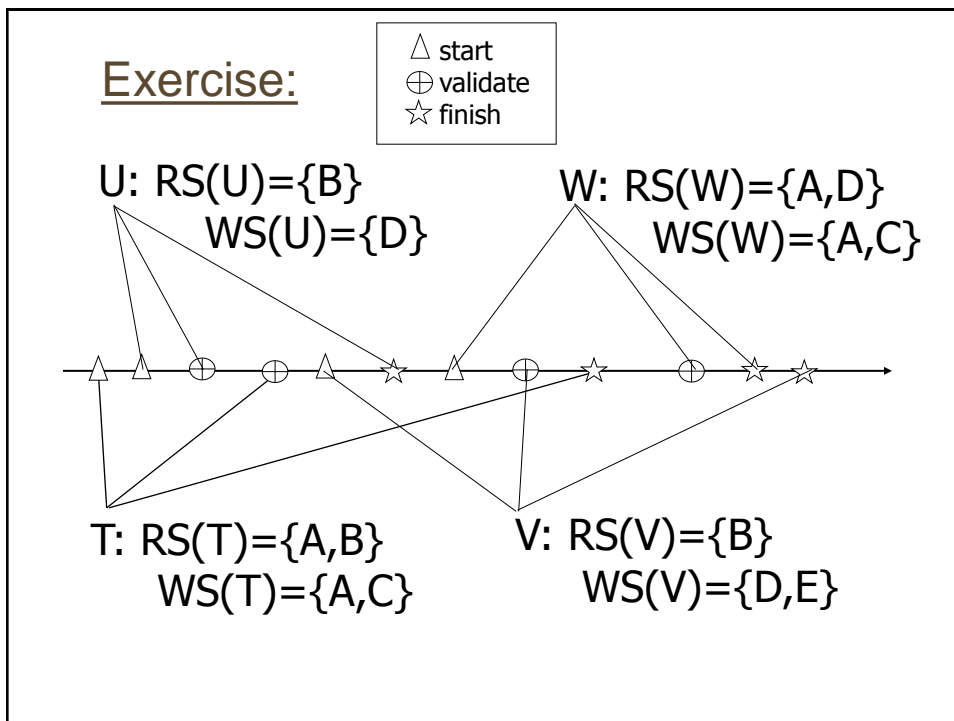
Check (T_j):

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

Is this check too restrictive ?

Improving Check(T_j)

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



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