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

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

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

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.

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*.

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 *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* 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.

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.

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

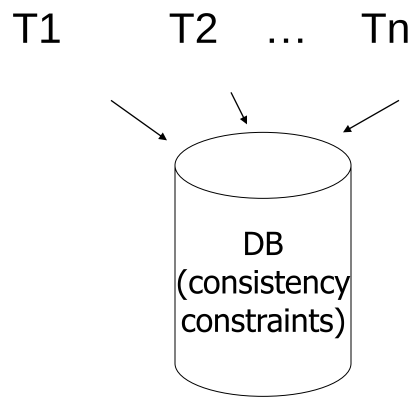
CS 54100 Concurrency Control

Chris Clifton

2 April, 2012

Indiana Center for Database Systems ™

Chapters 16-17
Concurrency Control



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

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

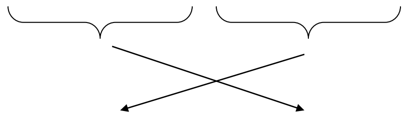
- Want schedules that are "good", regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

Example:

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

Example:

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

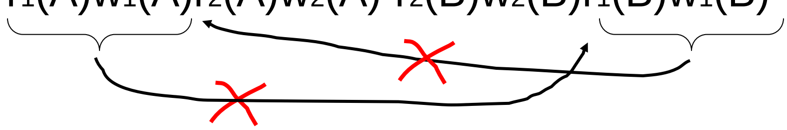


$Sc' = r_1(A)w_1(A)\ r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$

T_1 T_2

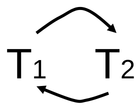
However, for Sd:

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



- 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, $T_1 \rightarrow T_2$



⇨ 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

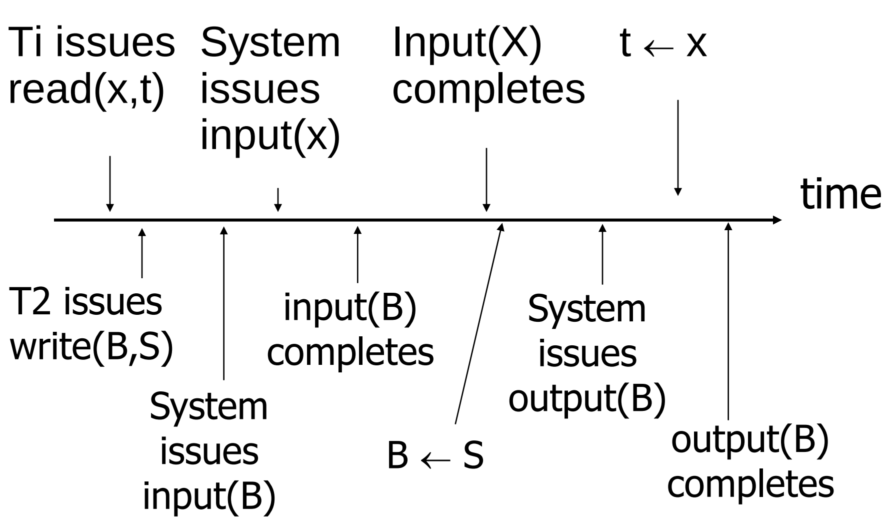
Conflicting actions: $r_1(A)$ – $w_2(A)$; $w_2(A)$ – $r_1(A)$; $w_1(A)$ – $w_2(A)$

Schedule: represents chronological order in which actions are executed

Serial schedule: no interleaving of actions or transactions

4/2/2012 Chris Clifton - CS541 29

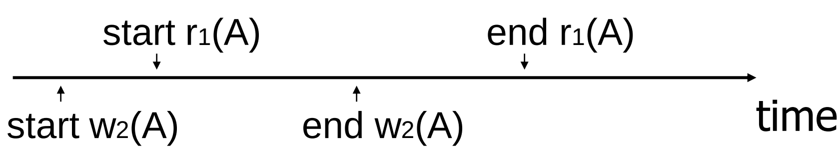
What about concurrent actions?



So net effect is either

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

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: Ti → Tj whenever

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

Exercise:

- What is P(S) for
  S = $w_3(A)$ $w_2(C)$ $r_1(A)$ $w_1(B)$ $r_1(C)$ $w_2(A)$ $r_4(A)$ $w_4(D)$
- Is S serializable?

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

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

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

$\}$ 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

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

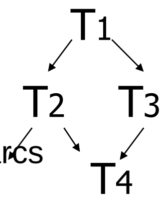
$\Rightarrow P(S_1)$ acyclic since $P(S_s)$ is acyclic

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!

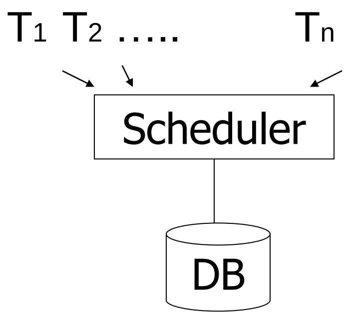
How to enforce serializable schedules?

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

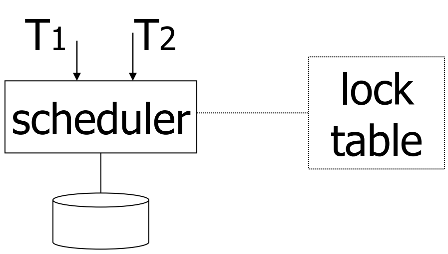


A locking protocol

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)	

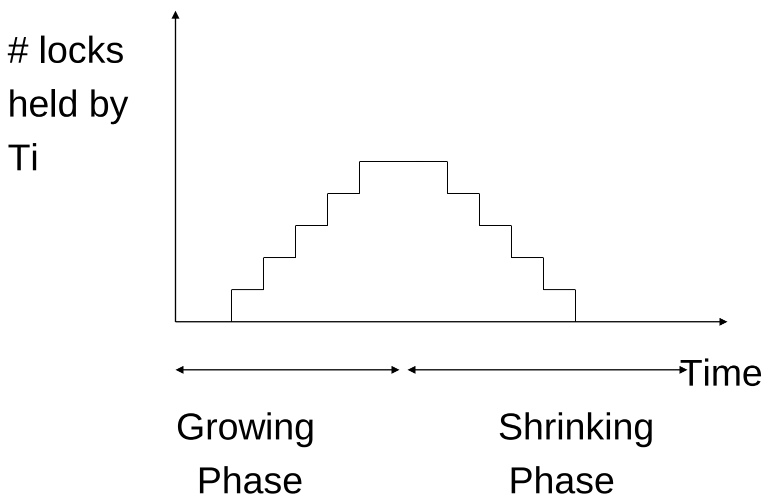
Schedule F

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)

T1	T2
l_1(A); Read(A)	l_2(B);Read(B)
A ← A+100;Write(A)	B ← Bx2;Write(B)
l_1(B) (delayed)	l_2(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 $< u_i(A), u_j(A)>$, $< l_i(A), r_j(A)>$,...

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

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

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

By rules 1,2:

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

By rule 3: SH(Ti) SH(Tj)

So, $SH(Ti) <_S SH(Tj)$

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

Proof:

(1) Assume P(S) has cycle

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

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

(3) Impossible, so P(S) acyclic

(4) $\Rightarrow$ S is conflict serializable

- Beyond this simple 2PL 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-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 $\rightarrow$ {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:

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

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

Lock types beyond S/X

Examples:

(1) increment lock

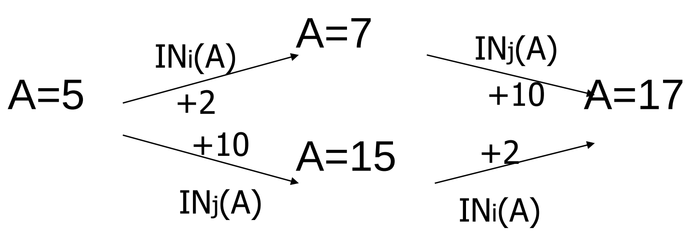
(2) update lock

Example (1): increment lock

- Atomic increment action: $IN_i(A)$

  {Read(A); A $\leftarrow$ 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

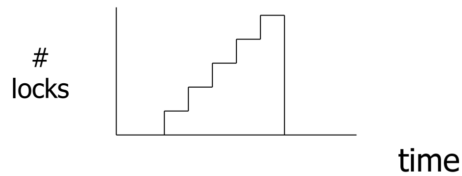
  (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)

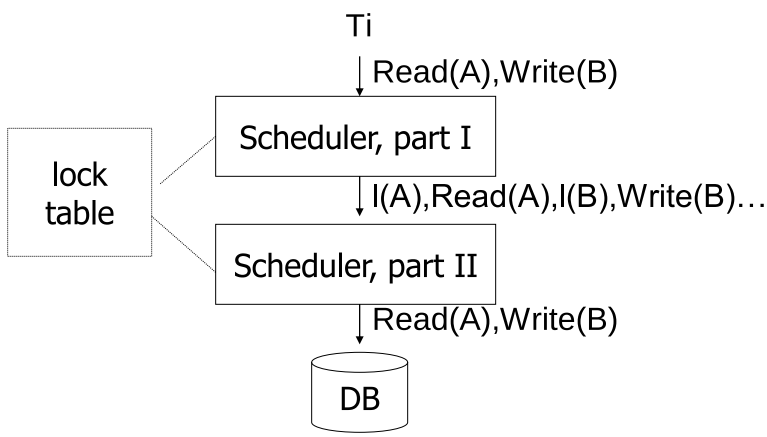
- But here is one (simplified) way ...

Sample Locking System:

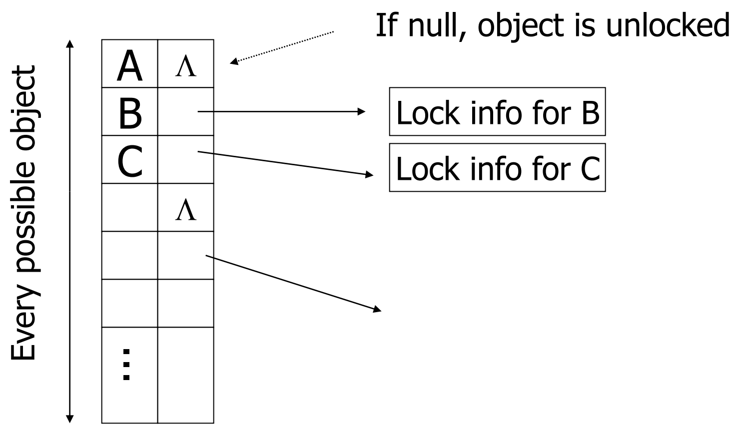
(1) Don't trust transactions to request/release locks

(2) Hold all locks until transaction commits



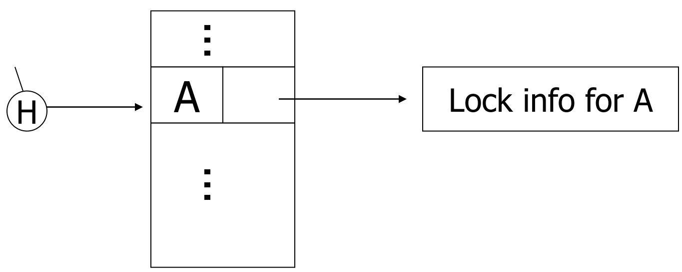


Lock table Conceptually



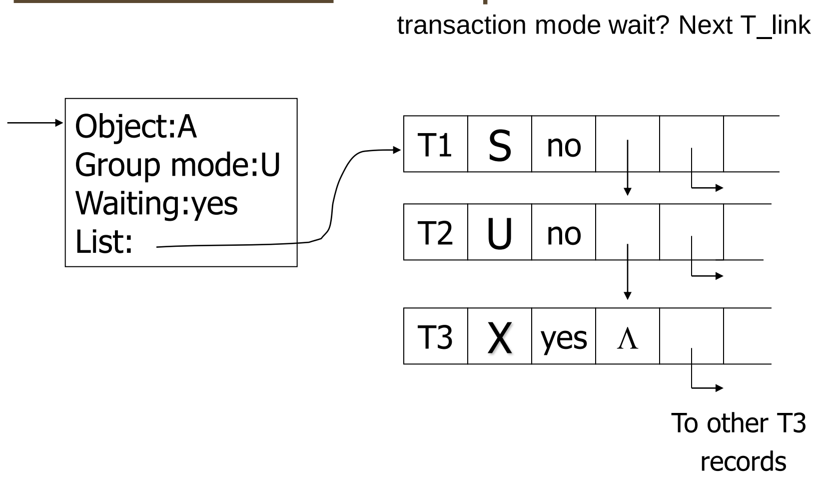
But use hash table:

A

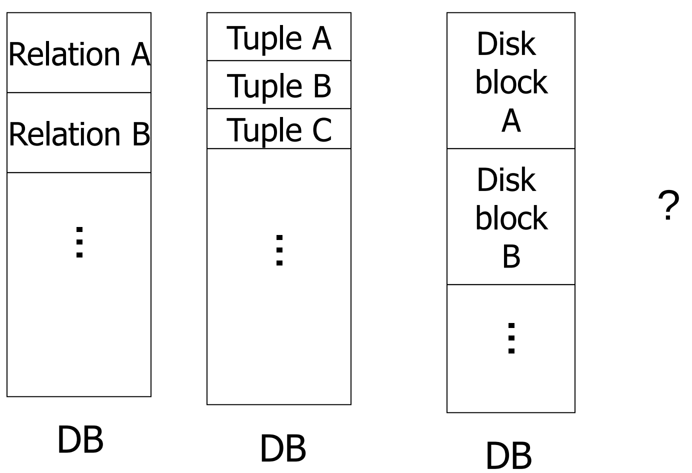


If object not found in hash table, it is unlocked

Lock info for A - example



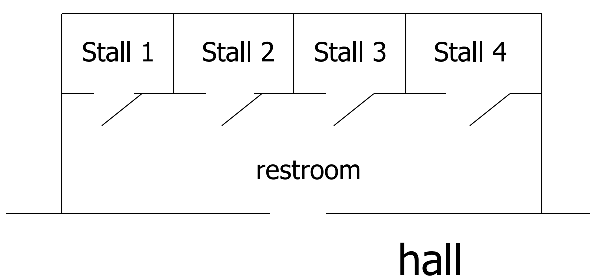
What are the objects we lock?



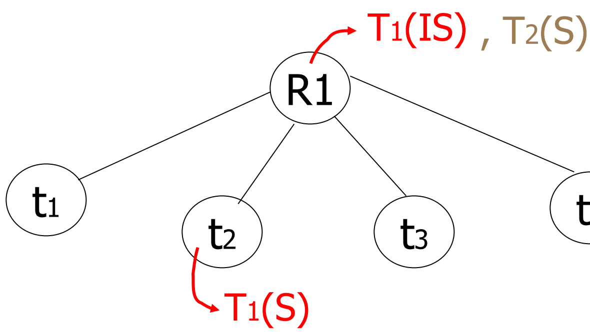
- Locking works in any case, but should we choose small or large objects?
- If we lock large objects (e.g., Relations)
  - Need few locks
  - Low concurrency
- If we lock small objects (e.g., tuples,fields)
  - Need more locks
  - More concurrency

We can have it both ways!!

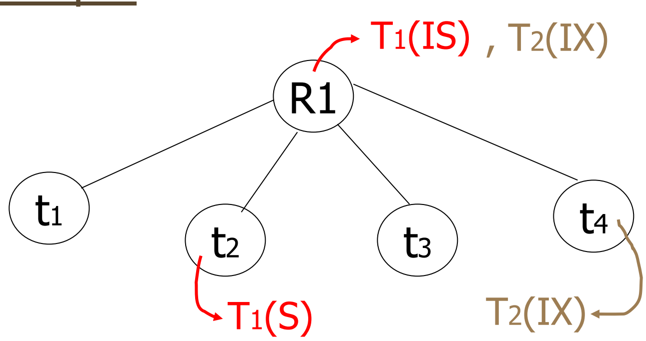
Ask any janitor to give you the solution...



Example



Example



Multiple granularity

Comp

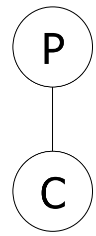
Holder \ Requestor	IS	IX	S	SIX	X
IS					
IX					
S					
SIX					
X					

Multiple granularity

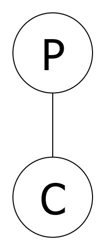
Comp

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



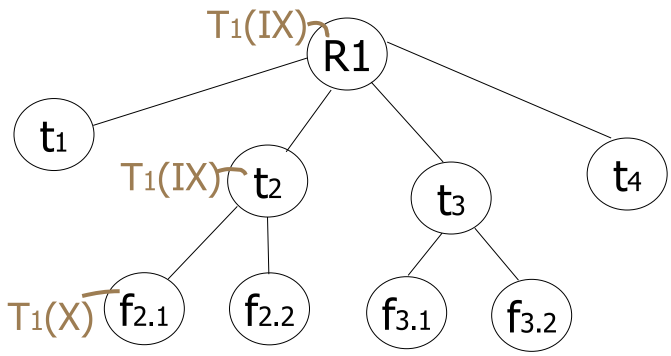
Rules

(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

- End 11/4

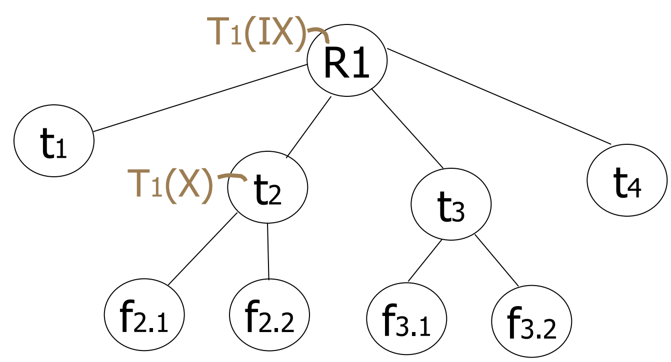
Exercise:

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



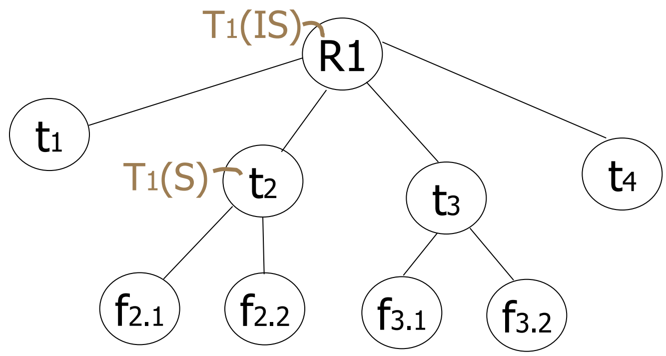
Exercise:

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



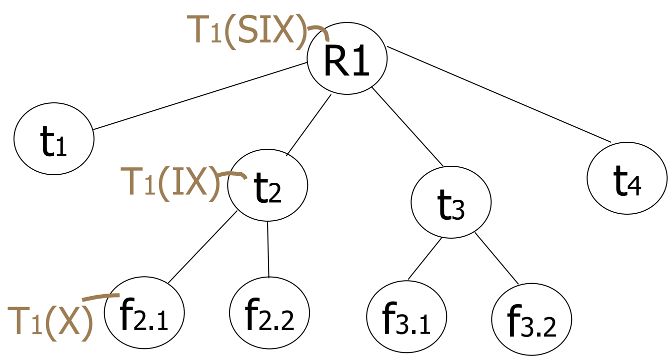
Exercise:

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



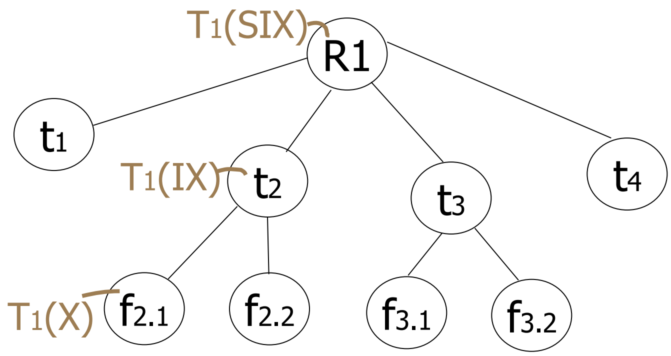
Exercise:

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

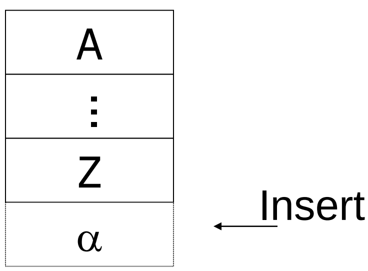


Exercise:

- 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

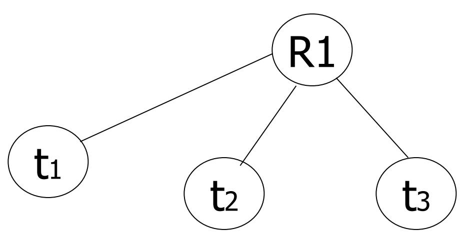
	E#	Name	….
o1	55	Smith	
o2	75	Jones	

T_1: Insert <99,Gore,…> into R
T_2: 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

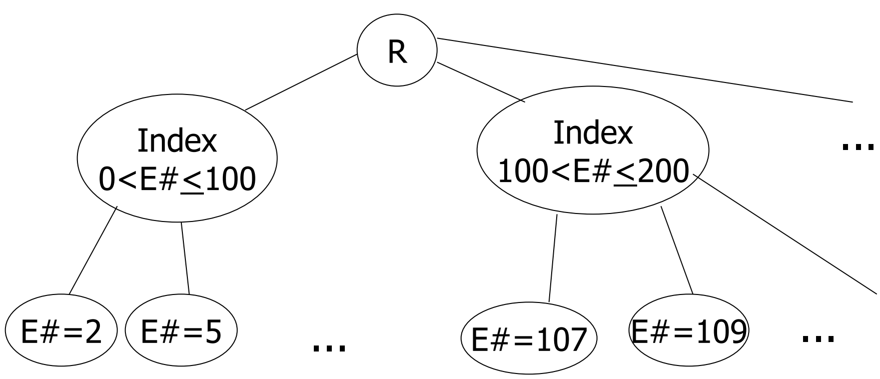


Back to example

T_1: Insert<99,Gore>	T_2: Insert<99,Bush>
T_1	T_2
$X_1(R)$	
	$X_2(R)$ ← *delayed*
Check constraint Insert<99,Gore> U(R)	
	$X_2(R)$ 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

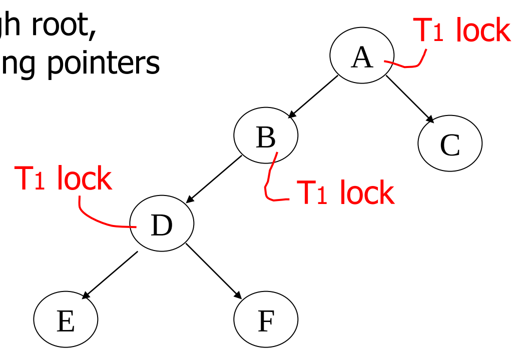
Indiana Center for Database Systems™

Next:

- Tree-based concurrency control
- Validation concurrency control

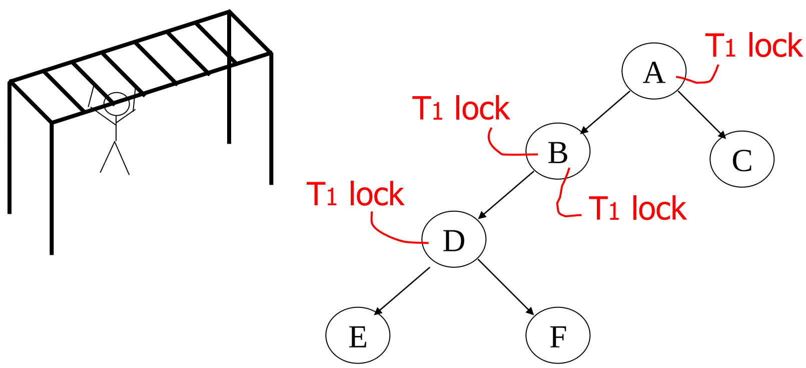
Example

- all objects accessed through root, following pointers



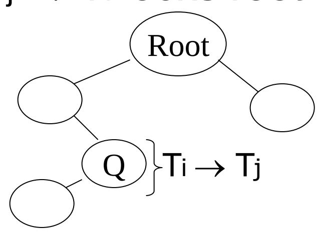
☛ can we release A lock if we no longer need A??

Idea: traverse like "Monkey Bars"



Why does this work?

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



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

Rules: tree protocol (exclusive locks)

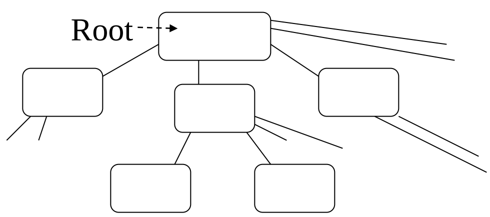
(1) First lock by T_i may be on any item

(2) After that, item Q can be locked by T_i only if parent(Q) locked by T_i

(3) Items may be unlocked at any time

(4) After T_i unlocks Q, it cannot relock Q

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



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

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

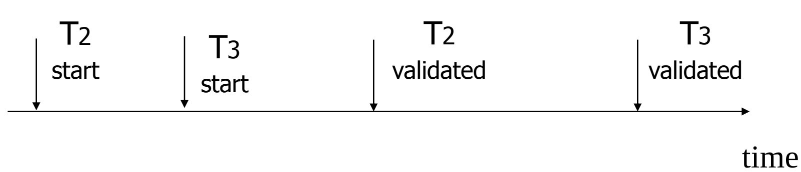
To implement validation, system keeps two sets:

- 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:

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

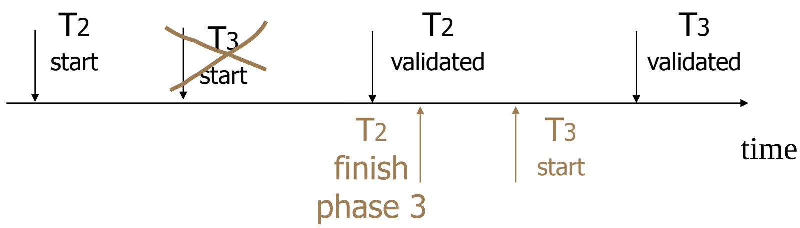
$WS(T_2)=\{B,D\}$ $\cap$ $WS(T_3)=\{C\}$ $\neq \phi$



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

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

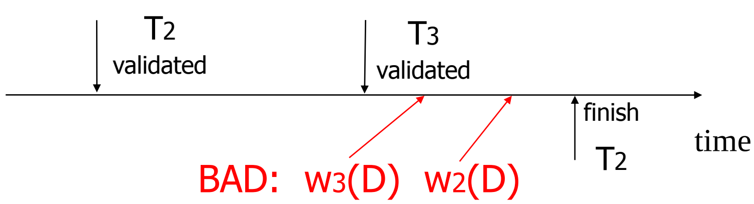
$WS(T_2)=\{B,D\} \cap WS(T_3)=\{C\} \neq \phi$



Another thing validation must prevent:

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

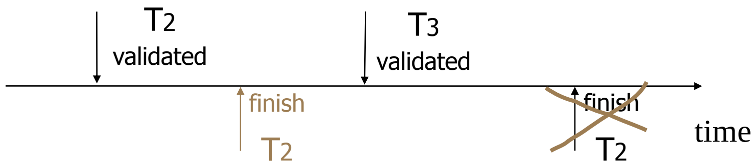
$WS(T_2)=\{D,E\}$ $WS(T_3)=\{C,D\}$



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 Tj:

(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 U $\{T_j\}$;

do write phase;

FIN $\leftarrow$ 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

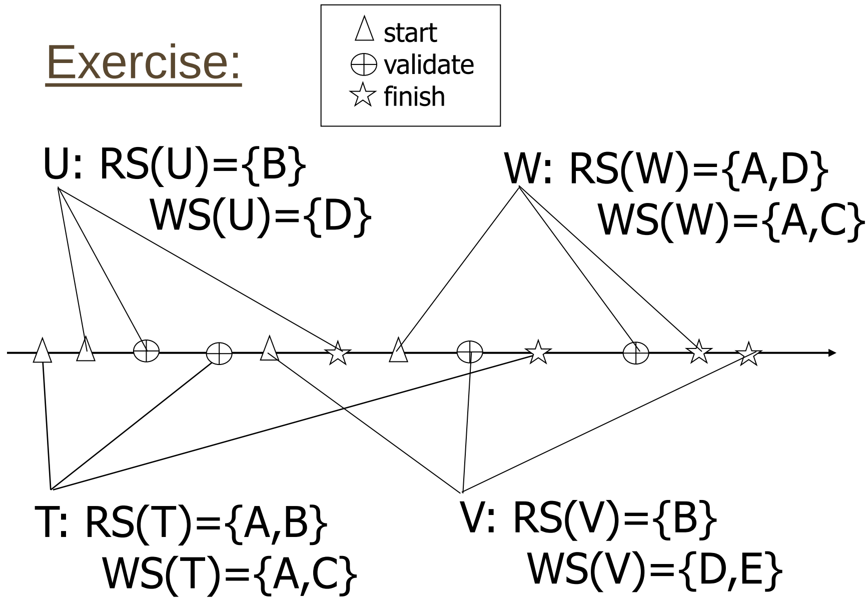
IF [WS(T_i) $\cap$ RS(T_j) $\neq \emptyset$ OR

($T_i \notin$ FIN AND WS(T_i) $\cap$ WS(T_j) $\neq \emptyset$)]

THEN RETURN false;

RETURN true;

Exercise:



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