CMSC 424 – Database design
Lecture 22
Concurrency/recovery

Mihai Pop

Admin

- Signup sheet for project presentations

Recap...1

- ACID properties:
  - Atomicity (recovery)
  - Consistency (transaction design, , concurrency control, recovery)
  - Isolation (concurrency control)
  - Durability (recovery)

Recap

- Concurrency Control Scheme
  - A way to guarantee serializability, recoverability etc
- Lock-based protocols
  - Use *locks* to prevent multiple transactions accessing the same data items
- 2 Phase Locking
  - Locks acquired during *growing phase,* released during *shrinking phase*
- Strict 2PL, Rigorous 2PL

More Locking Issues: Deadlocks

- No xction proceeds:

Deadlock

  - T1 waits for T2 to unlock A
  - T2 waits for T1 to unlock B

T1	T2
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Rollback transactions
Can be costly...

2PL and Deadlocks

- 2PL does not prevent deadlock
  – Strict doesn't either

T1	T2
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

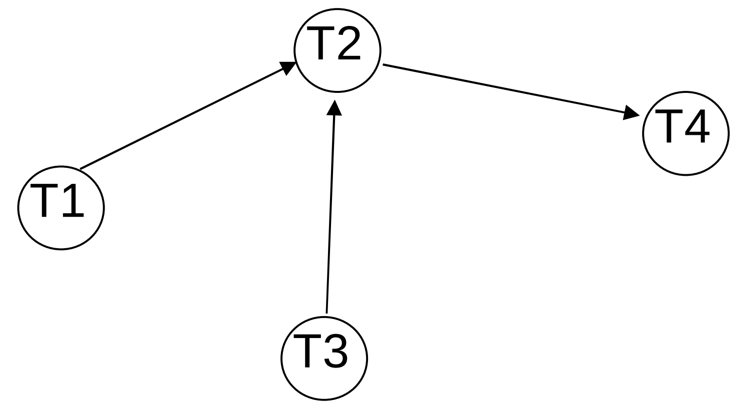
- > 2 xctions involved?
  - Rollbacks expensive

Preventing deadlocks

- Solution 1: A transaction must acquire all locks before it begins
  - Not acceptable in most cases
- Solution 2: A transaction must acquire locks in a particular order over the data items
  - Also called *graph-based protocols*
- Solution 3: Use time-stamps; say T1 is older than T2
  - *wait-die scheme:* T1 will wait for T2. T2 will not wait for T1; instead it will abort and restart
  - *wound-wait scheme:* T1 will *wound* T2 (force it to abort) if it needs a lock that T2 currently has; T2 will wait for T1.
- Solution 4: Timeout based
  - Transaction waits a certain time for a lock; aborts if it doesn't get it by then

Deadlock detection and recovery

- Instead of trying to prevent deadlocks, let them happen and deal with them if they happen
- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from Ti to Tj
    - Ti waiting for Tj



T1	T2	T3	T4
		X(Z)	
	X(V)		
			X(W)
S(V)			
	S(W)		
		S(V)	

Suppose T4 requests lock-S(Z)....

Dealing with Deadlocks

- Deadlock detected, now what ?
  - Will need to abort some transaction
  - Prefer to abort the one with the minimum work done so far
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continuing

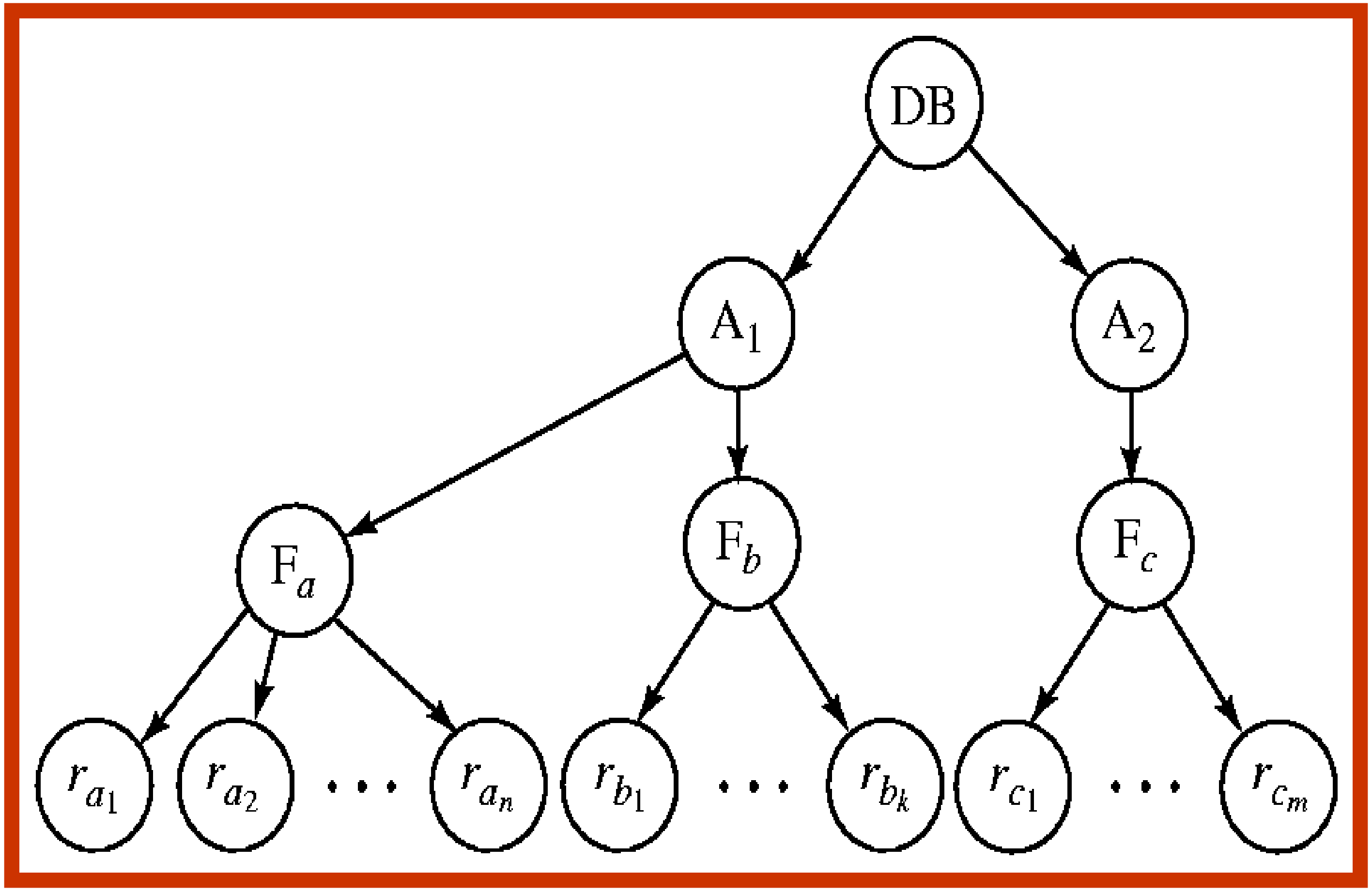
Locking granularity

- Locking granularity
  - What are we taking locks on ? Tables, tuples, attributes ?

- Coarse granularity
  - e.g. take locks on tables
  - less overhead (the number of tables is not that high)
  - very low concurrency

- Fine granularity
  - e.g. take locks on tuples
  - much higher overhead
  - much higher concurrency
  - What if I want to lock 90% of the tuples of a table ?
    - Prefer to lock the whole table in that case

Granularity Hierarchy



The highest level in the example hierarchy is the entire database.

The levels below are of type *area, file or relation* and *record* in that order.

Can lock at any level in the hierarchy

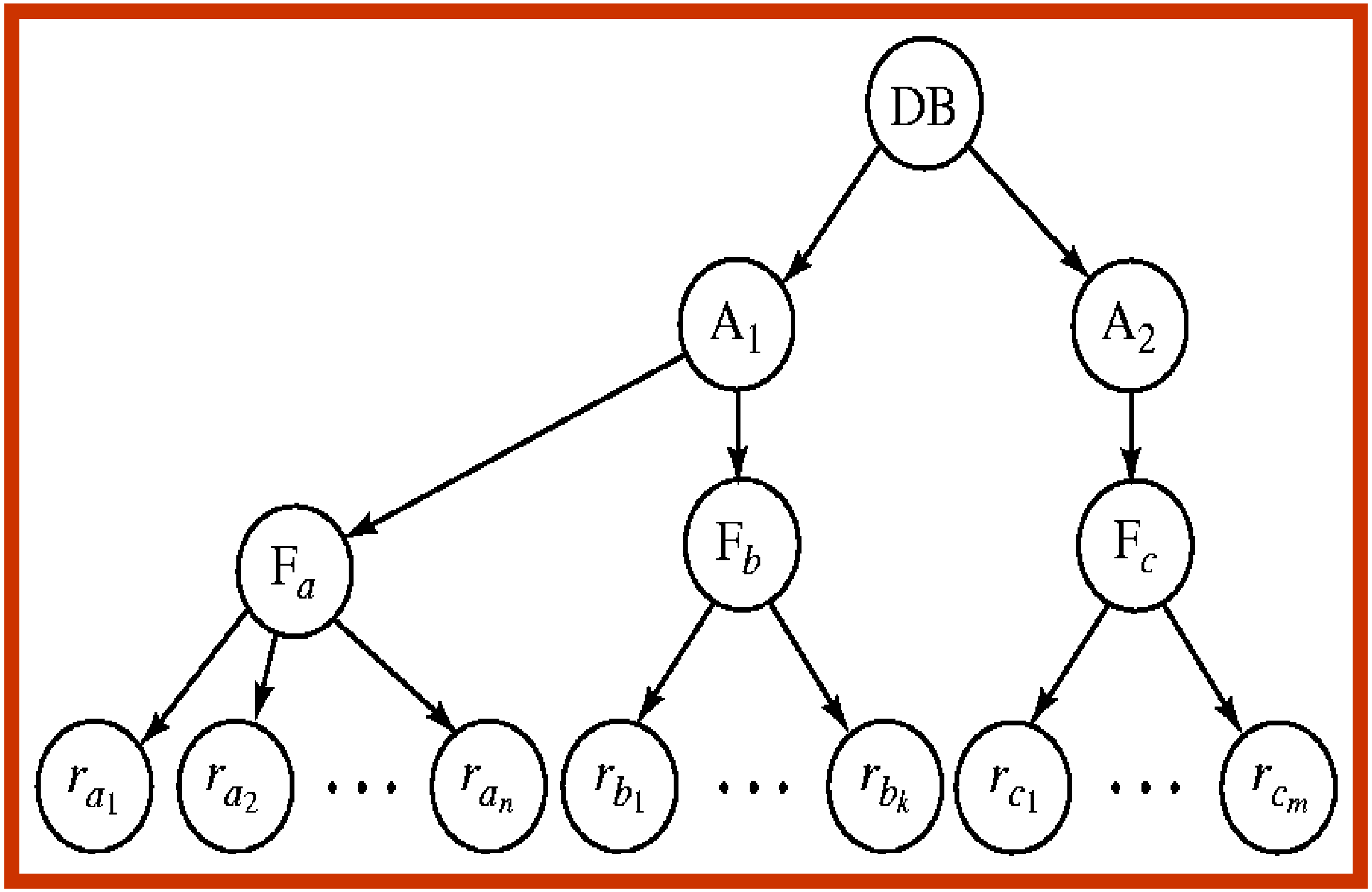
Granularity Hierarchy

- New lock mode, called *intentional* locks
  - Declare an intention to lock parts of the subtree below a node
  - IS: *intention shared*
    - The lower levels below may be locked in the shared mode
  - IX: *intention exclusive*
  - *SIX: shared and intention-exclusive*
    - The entire subtree is locked in the shared mode, but I might also want to get exclusive locks on the nodes below
- Protocol:
  - If you want to acquire a lock on a data item, all the ancestors must be locked as well, at least in the intentional mode
  - So you always start at the top *root* node

Granularity Hierarchy

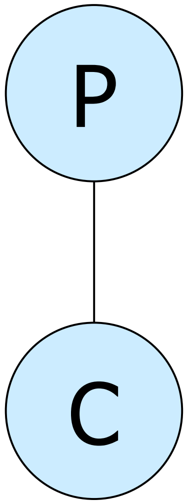
(1) Want to lock *F_a* in shared mode, *DB* and *A1* must be locked in at least IS mode (but IX, SIX, S, X are okay too)

(2) Want to lock *rc1* in exclusive mode, *DB, A2,Fc* must be locked in at least IX mode (SIX, X are okay too)



Granularity Hierarchy

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



Compatibility Matrix with Intention Lock Modes

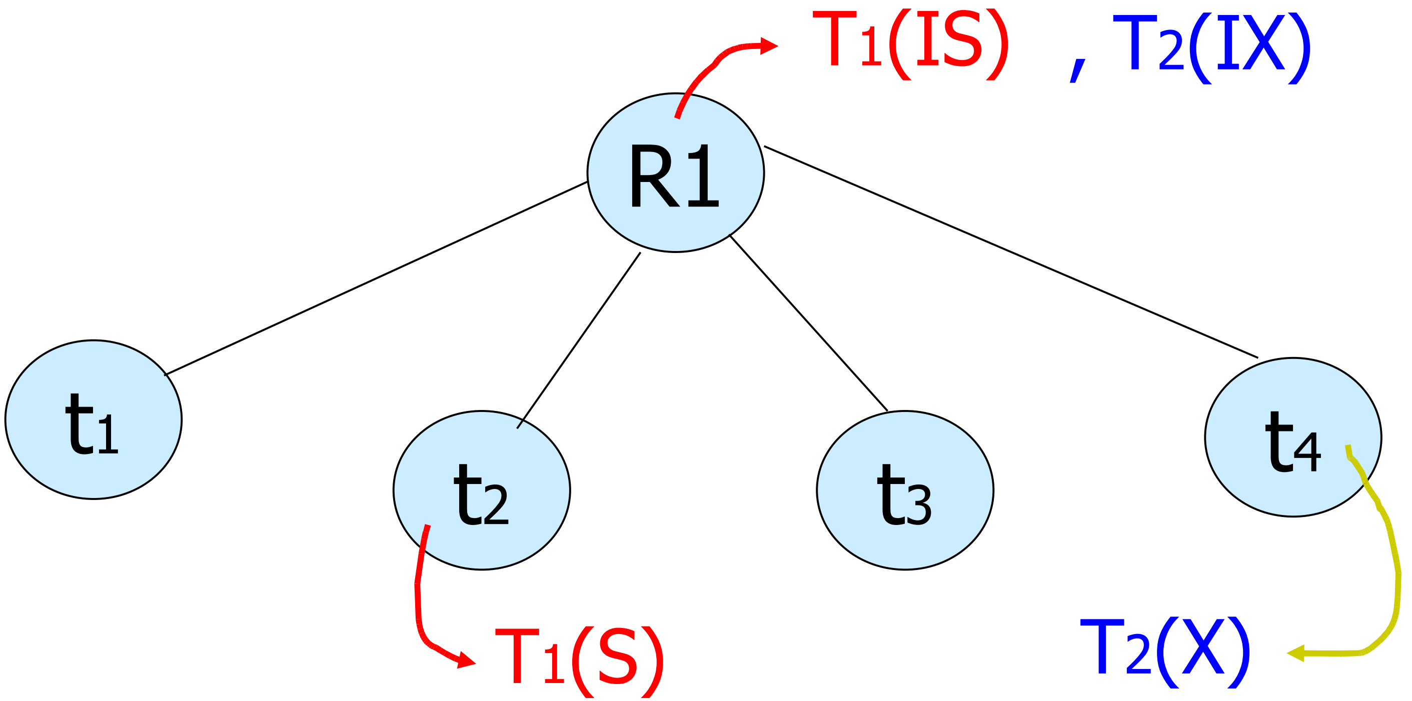
- The compatibility matrix (which locks can be present simultaneously on the same data item) for all lock modes is:

requestor

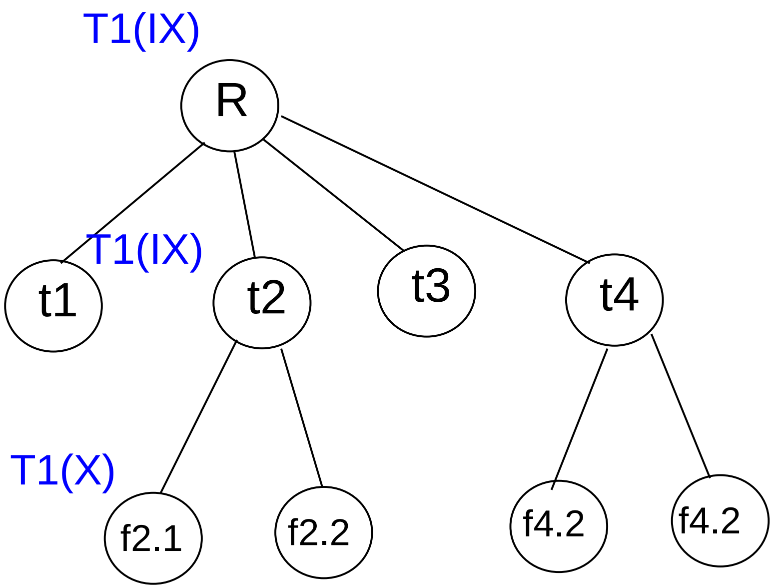
holder

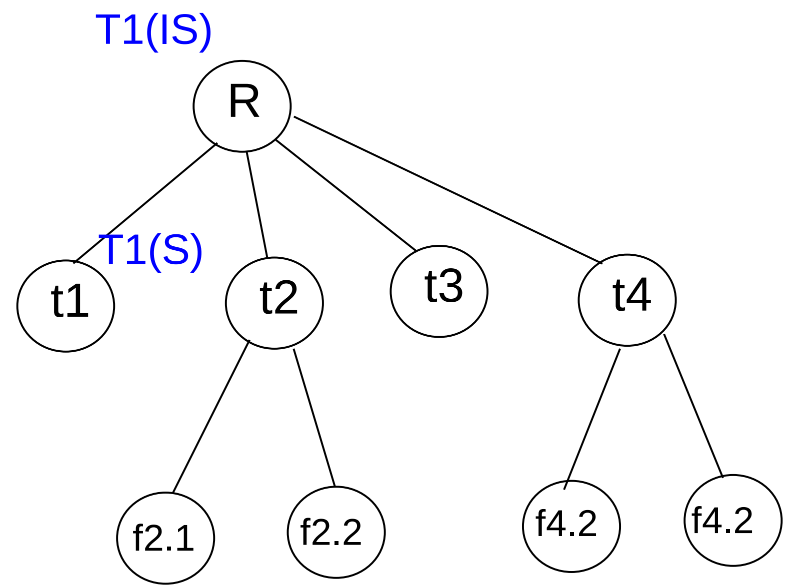
	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

Example



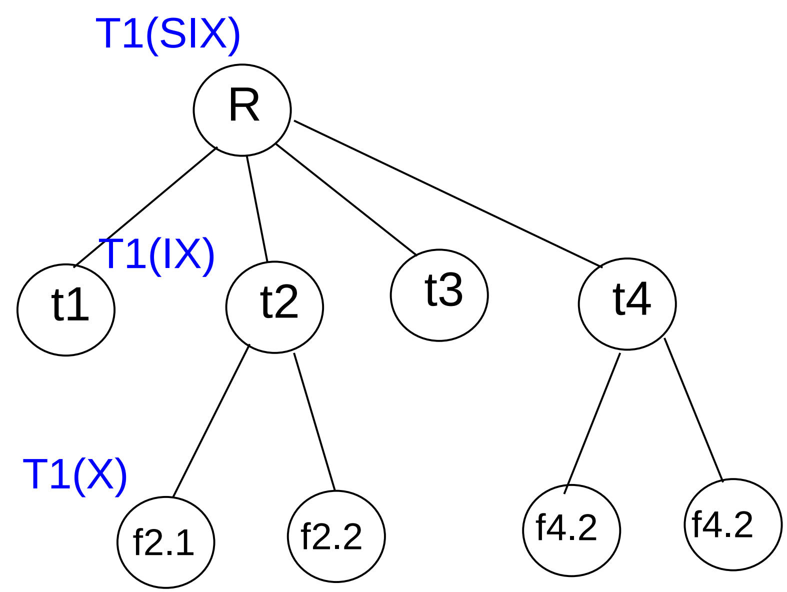
Examples





Can T2 access object f2.2 in X mode?

What locks will T2 get?



Examples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can

  use lock escalation to decide which.

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

Recap, Next….

- Deadlocks
  - Detection, prevention, recovery

- Locking granularity
  - Arranged in a hierarchy
  - Intentional locks

- Next…
  - Brief discussion of some other concurrency schemes

Other CC Schemes

- Time-stamp based
  - Transactions are issued time-stamps when they enter the system
  - The time-stamps determine the *serializability* order
  - So if T1 entered before T2, then T1 should be before T2 in the serializability order
  - Say *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read that data item or written it, then the write is *rejected* and T1 is aborted
  - Aborted transaction are restarted with a new timestamp
    - Possibility of *starvation*

Other CC Schemes

- Time-stamp based
  - As discussed here, has too many problems
    - Starvation
    - Non-recoverable
    - Cascading rollbacks required
  - Most can be solved fairly easily
    - Read up
  - Remember: We can always put more and more restrictions on what the transactions can do to ensure these things
    - The goal is to find the minimal set of restrictions to as to not hinder concurrency

Other CC Schemes

- Optimistic concurrency control
  - Also called validation-based

  - Intuition
    - Let the transactions execute as they wish
    - At the very end when they are about to commit, check if there might be any problems/conflicts etc
      - If no, let it commit
      - If yes, abort and restart

  - Optimistic: The hope is that there won't be too many problems/aborts

- Rarely used any more

The "Phantom" problem

- An interesting problem that comes up for dynamic databases
- Schema: *accounts(branchname, acct_no, balance, …)*
- Transaction 1: Find the maximum *balance* in each *branch*
- Transaction 2: Insert *<"branch1", acctX, $10000000>,* and delete *<"branch2", acctY, $100000000>.*
  - Both maximum entries in the corresponding branches
- Execution sequence:
  - T1 locks all tuples corresponding to "branch1", finds the maximum balance and releases the locks
  - T2 does its two insert/deletes
  - T1 locks all tuples corresponding to "branch2", finds the maximum balance and releases the locks
- Not serializable