CS-300: Data-Intensive Systems

Concurrency Control
(Chapter 17 and 18)

Prof. Anastasia Ailamaki, Prof. Sanidhya Kashyap

EPFL

Concurrency control

- Serializability

  Readings: Chapter from 17.1 to 17.6
- Two phase locking
- Lock management and deadlocks
- Locking granularity
- Phantoms and predicate locking

Transactions & Schedules: Definitions

- A program may carry out many operations on the data retrieved from the database
- The DBMS is only concerned about what data is read/written from/to the database
- *Database*
  - a fixed set of named data objects *(A, B, C, …)*
- *Transaction*
  - a sequence of actions *(read(A), write(B),commit, abort …)*
- *Schedule*
  - an interleaving of actions from various transactions

Formal properties of schedules

- *Serial schedule:* Schedule that does not interleave the actions of different transactions

T_1: $R_1(X)$
X=X-20
$W_1(X)$

T_2: $R_2(X)$
X=X+10
$W_2(X)$

	T_1, T_2	
t0:	$R_1(X)$	
t1:	X=X-20	
t2:	$W_1(X)$	
t3:		$R_2(X)$
t4:		X=X+10
t5:		$W_2(X)$

	T_2, T_1	
t0:		$R_2(X)$
t1:		X=X+10
t2:		$W_2(X)$
t3:	$R_1(X)$	
t4:	X=X-20	
t5:	$W_1(X)$	

Formal properties of schedules

- *Equivalent schedules*: For any database state, the effect 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.

Conflicting operations

- We need a formal notion of equivalence that can be implemented efficiently
  - Base it on the notion of "conflicting" operations

- Definition: Two operations conflict if:
  - They are done by different transactions,
  - And they are done on the same object,
  - And at least one of them is a write

Examples:

$R_1(A), W_2(A)$

$W_1(A), R_2(A)$

$W_1(A), W_2(A)$

$R_1(B), W_2(B)$

$W_1(B), R_2(B)$

$W_1(B), W_2(B)$

Conflict serializable schedules

- <u>Definition</u>: Two schedules are conflict equivalent iff:
  - They involve the same actions of the same transactions,
  - And every pair of conflicting actions is ordered the same way

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

$S_1 \equiv S_2$

$S_1 \equiv S_3$??

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$

S_2:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
$R_1(B)$	
	$W_2(A)$
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$

S_3:

T_1	T_2
$R_1(A)$	
	$R_2(A)$
$W_1(A)$	
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
$R_1(B)$	
$W_1(B)$	

Conflict serializable schedules

- <u>Definition</u>: Schedule S is conflict serializable if:
  - S is conflict equivalent to some serial schedule

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$

S_2:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
$R_1(B)$	
$W_1(B)$	
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$

S_3:

T_1	T_2
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
$R_1(A)$	
$W_1(A)$	
$R_1(B)$	
$W_1(B)$	

Conflict serializability: Definition

- A schedule S is conflict serializable if:
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- *Example:*

R(A)	W(A)			R(B)	W(B)		
		R(A)	W(A)			R(B)	W(B)

≡

R(A)	W(A)	R(B)	W(B)				
				R(A)	W(A)	R(B)	W(B)

Conflict serializability (cont.)

- Here's another example:

R(A) W(A)

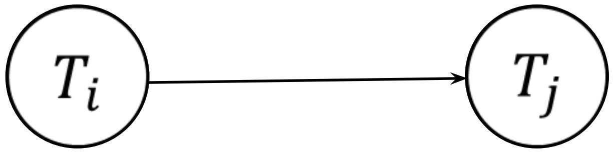
R(A) W(A)

- Conflict serializable or not?

NOT!

Testing for conflict serializability

- *Precedence graph*:
  - One node per transaction
  - Edge from T_i to T_j if:
    - An operation O_i of T_i conflicts with an operation O_j of T_j and
    - Oi appears earlier in the schedule than O_j



- Theorem: Schedule is conflict serializable if and only if its precedence graph is acyclic

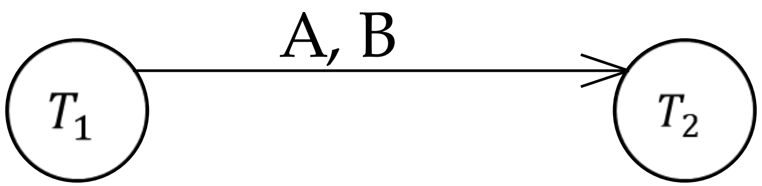
Precedence graph

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$



Examples:

$R_1(A)$, $W_2(A)$

$W_1(A)$, $R_2(A)$

$W_1(A)$, $W_2(A)$

$R_1(B)$, $W_2(B)$

$W_1(B)$, $R_2(B)$

$W_1(B)$, $W_2(B)$

Precedence graph

Examples:

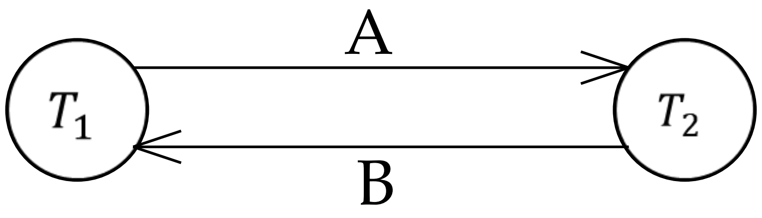
$R_1(A)$, $W_2(A)$
$W_1(A)$, $R_2(A)$
$W_1(A)$, $W_2(A)$
$R_1(B)$, $W_2(B)$
$W_1(B)$, $R_2(B)$
$W_1(B)$, $W_2(B)$

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
$R_1(B)$	
$W_1(B)$	

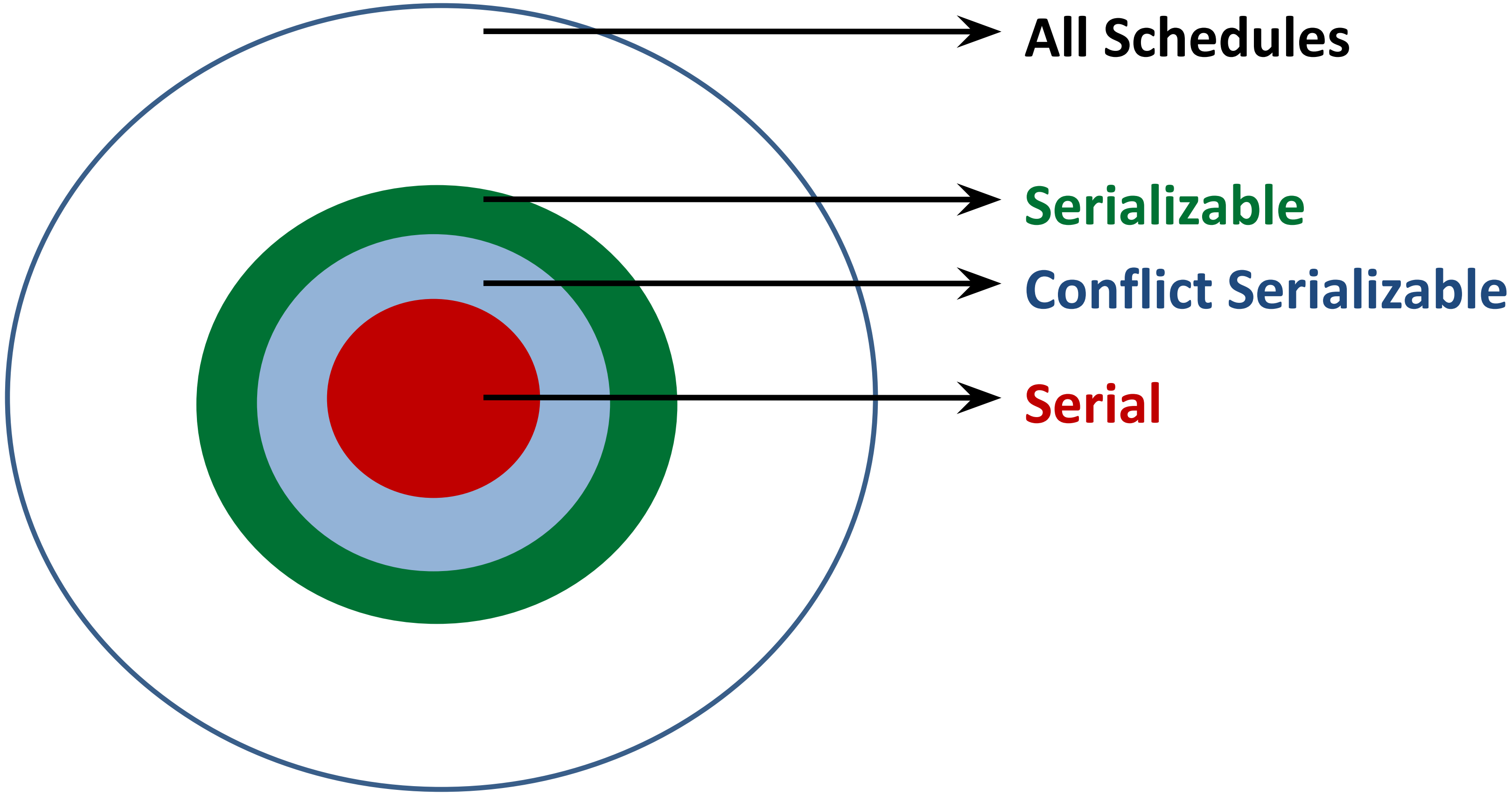


NOT conflict serializable:

- The cycle in the graph reveals the problem.
- The output of T_1 depends on T_2, and vice-versa

Conflict serializable schedules

- Note, some “serializable” schedules are NOT conflict serializable
  - A price we pay to achieve efficient enforcement



More equivalences of schedules

- View Equivalence
- Result Equivalence
- Most commonly used is Conflict Equivalence

Concurrency control

- Serializability
- Two phase locking   Readings: Chapter 18.1
- Lock management and deadlocks
- Locking granularity
- Phantoms and predicate locking

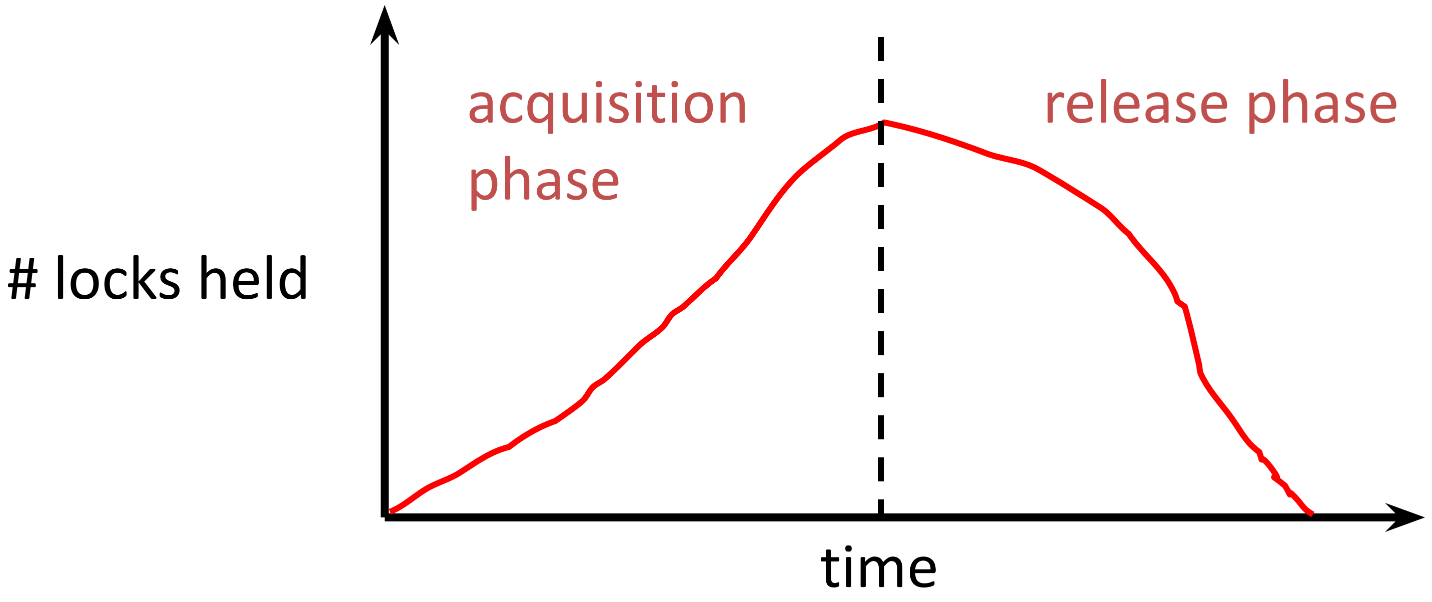
Two-Phase Locking (2PL)

- Locking protocol
  - Each transaction must obtain an S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing
  - A transaction cannot request additional locks once it releases any locks
  - Thus, there is a “growing phase” followed by a “shrinking phase”

- Lock compatibility matrix:

	S	X
S	true	false
X	false	false

Two-Phase Locking (2PL)



2PL on its own is sufficient to guarantee conflict serializability (i.e., schedules whose precedence graph is acyclic),

BUT it is subject to Cascading Aborts!

Strict 2PL

- Problem: Cascading Aborts
- Example: Rollback of T1 requires rollback of T2!

T1: $R_1(A), W_1(A), R_1(B), W_1(B),$ Abort

T2: $R_2(A), W_2(A)$

- To avoid Cascading Aborts, use Strict 2PL
- Strict Two-Phase Locking (Strict 2PL) Protocol:
  - Same as 2PL, except:
  - All locks held by a transaction are released only when the transaction completes

Non-2PL, A= 100, B=200, output =?

Lock_X(A)		
Read(A)	Lock_S(A)	
A: = A-50		
Write(A)		➡ A=50
Unlock(A)		
	Read(A)	
	Unlock(A)	
	Lock_S(B)	
Lock_X(B)		
	Read(B)	
	Unlock(B)	
	PRINT(A+B)	➡ **250**
Read(B)		
B := B +50		
Write(B)		➡ B=250
Unlock(B)		

2PL, A= 100, B=200, output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	Unlock(A)
	Read(B)
	Unlock(B)
ABORT	PRINT(A+B)

→ A=50

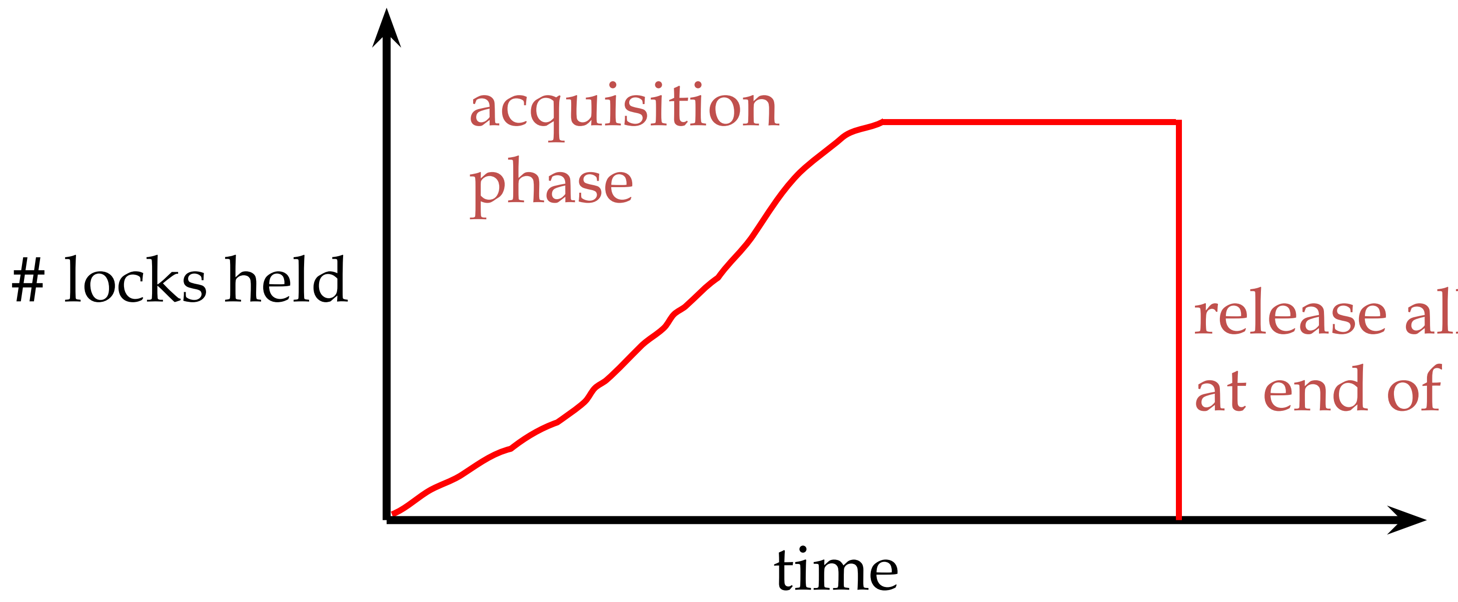
→ B=250

→ **300**

Strict 2PL, A= 100, B=200, output =?

Lock_X(A)		
Read(A)	Lock_S(A)	
A: = A-50		
Write(A)		A=50
Lock_X(B)		
Read(B)		
B := B +50		B=250
Write(B)		
Unlock(A)		
Unlock(B)		
	Read(A)	
	Lock_S(B)	
	Read(B)	
	PRINT(A+B)	**300**
	Unlock(A)	
	Unlock(B)	

Strict 2PL (cont.)



- Allows only conflict serializable schedules, but it is actually stronger than needed for that purpose
- In effect, “shrinking phase” is delayed until
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the transaction (locks can be released after rollback)

Two phase locking: Summary

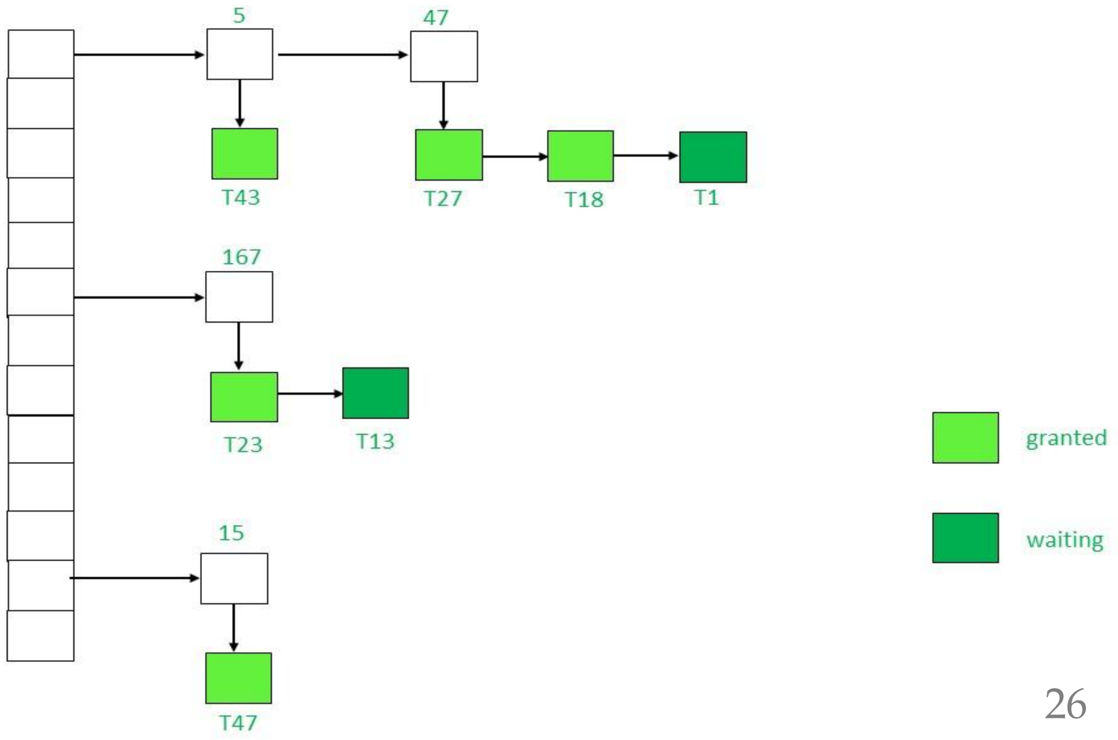
- Locks implement the notions of conflict directly
- 2PL has:
  - Growing phase where locks are acquired and no lock is released
  - Shrinking phase where locks are released and no lock is acquired
- Strict 2PL requires all locks to be released at once, when transaction ends

Concurrency control

- Serializability
- Two phase locking
- Lock management and deadlocks Readings: Chapters 18.2
- Locking granularity
- Phantoms and predicate locking

Lock management

- Lock and unlock requests handled by the Lock Manager
- Lock Manager contains an entry for each currently held lock
- Lock table entry:
  - Pointer to list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests



Lock management (cont.)

- Basic operation: when lock request arrives see if any other transaction holds a conflicting lock
  - If not, create an entry and grant the lock
  - Else, put the requestor on the wait queue
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
- Two-phase locking is simple enough, right?

Which locks are granted?

Lock_X(A)	
	Lock_S(B)
	Read(B)
	Lock_S(A)
Read(A)	
A: = A-50	
Write(A)	
Lock_X(B)	

Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

- Many systems just ‘punt’ and use Timeouts
  - What are the dangers with this approach?

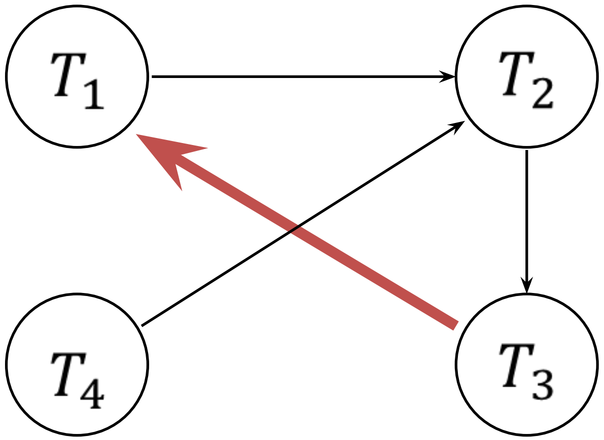
Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - Edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Periodically check for cycles in waits-for graph

Deadlock Detection (Continued)

Example:

T1: S(A), S(D), S(B)
T2: X(B) X(C)
T3: S(D), S(C), X(A)
T4: X(B)



Deadlock Prevention

- Assign priorities based on timestamp
  - $TS_i > TS_j$ <= priority(i) > priority(j)
- Say T_i wants a lock that T_j holds, two policies are possible:

Wait-Die: If T_i has higher priority than T_j, T_i **waits**;
else T_i **aborts**

Wound-Wait: If T_i has higher priority than T_j, T_j **aborts**;
else T_i **waits**

These schemes guarantee deadlock freedom. Why?

***Important detail*: If txn restarts, it must be assigned the original timestamp. Why?**

Deadlocks: summary

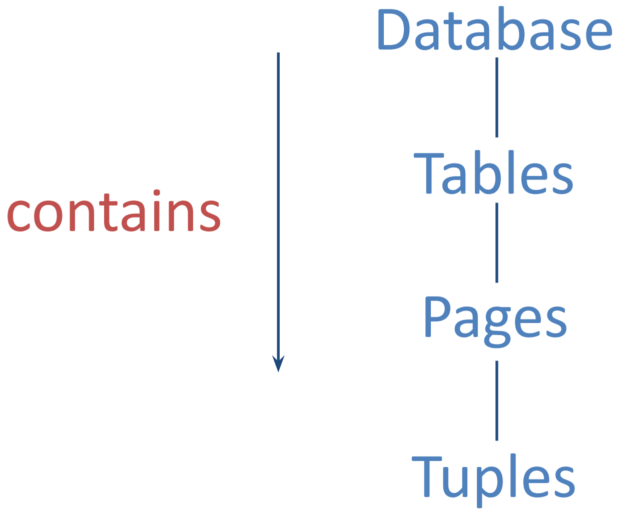
- The Lock Manager keeps track of the locks issued
- Deadlock is a cycle of transactions waiting for locks to be released to each other
- Deadlocks may arise and can be:
  - Prevented, e.g., using timestamps
  - Detected, e.g., using waits-for graphs

Concurrency control

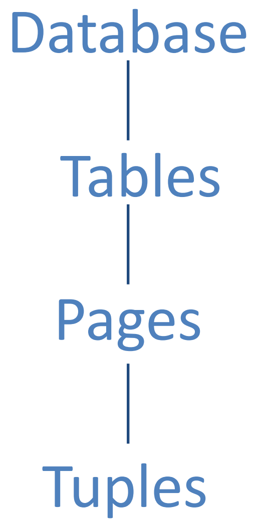
- Serializability
- Two phase locking
- Lock management and deadlocks
- Locking granularity Readings: Chapter 18.3

Multiple-granularity locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables)
- Shouldn't have to make same decision for all transactions!
- Data "containers" are nested:



Solution: New lock modes, protocol

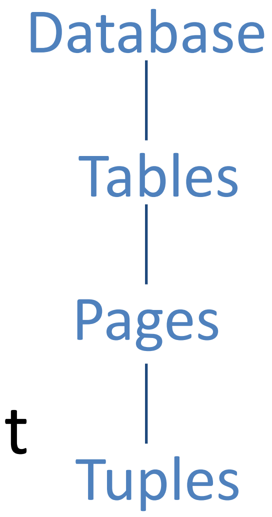


- Allow transaction to lock at each level, but with a special protocol using new "intention" locks:
- Still need S and X locks, but before locking an item, transaction must have proper intension locks on all its ancestors in the granularity hierarchy

  - IS – Intent to get S lock(s) at finer granularity
  - IX – Intent to get X lock(s) at finer granularity
  - SIX mode: Like S & IX at the same time. Why is it useful?

Goal: more concurrent transactions

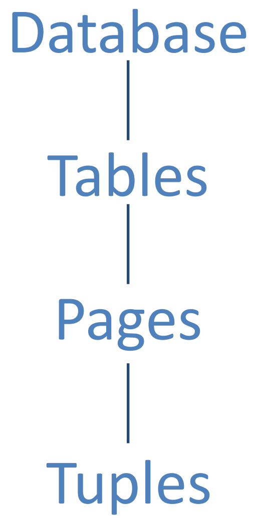
Multiple-granularity lock protocol



- Each transaction starts from the root of the hierarchy
- To get S or IS lock on a node, must hold IS or IX on parent node
  - What if transaction holds SIX on parent? S on parent?
- To get X or IX or SIX on a node, must hold IX or SIX on parent node
- Must release locks in bottom-up order

Protocol is equivalent to directly setting locks at the leaf levels of the hierarchy.

Lock compatibility matrix



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

- IS – Intent to get S lock(s) at finer granularity
- IX – Intent to get X lock(s) at finer granularity
- SIX mode: S & IX at the same time

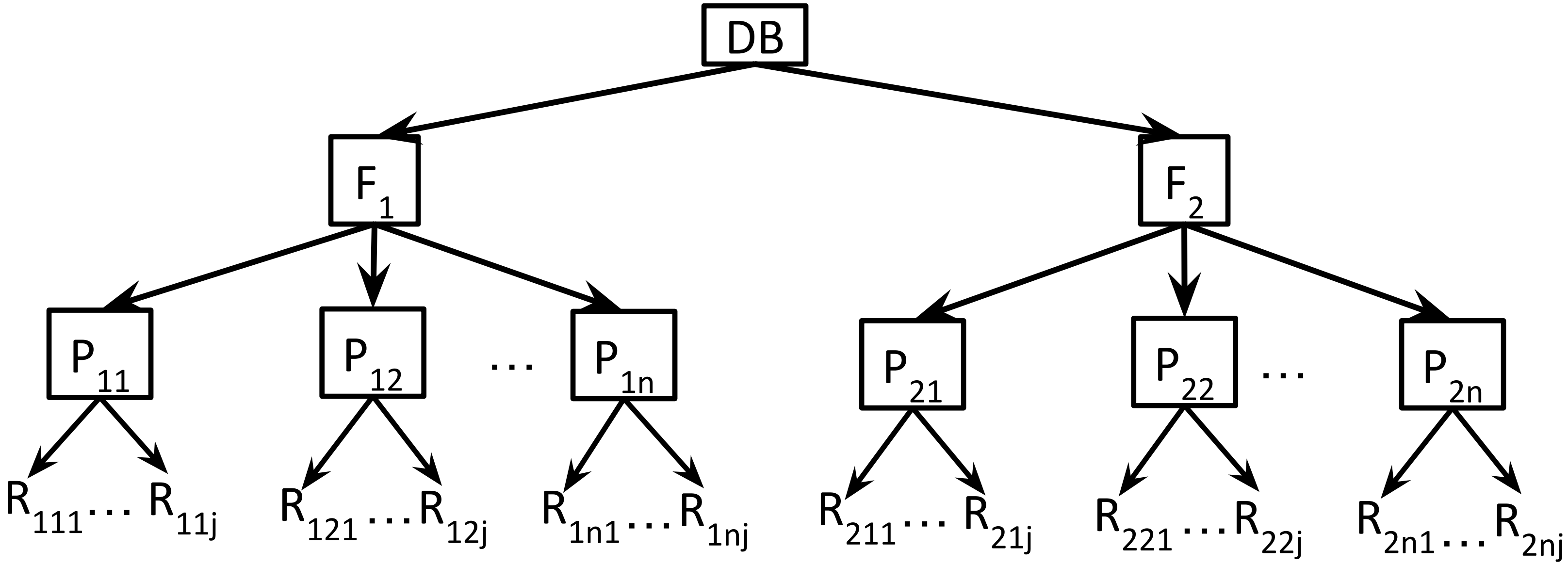
Example – 2 level of hierarchy

Tables
|
Tuples

- T_1 scans R, and updates a few tuples:
  - T_1 gets an SIX lock on R, then X lock on tuples that are updated
- T_2 uses an index to read only part of R:
  - T_2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R
- T_3 reads all of R:
  - T_3 gets an S lock on R
  - OR, T_3 could behave like T_2
  - We can use lock escalation to decide
  - Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired

	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

Example (for review)



- T_1 :update records R_{111} and R_{211}
- T_2 :update all records in page P_{12}
- T_3 :read record R_{11j} and read entire file F_2

How do these concurrent transactions acquire locks?

Example (cont.)

T1	T2	T3
IX(DB)		
IX(F1)		
	IX(DB)	
		IS(DB)
		IS(F1)
		IS(P11)
IX(P11)		
X(R11)		
	IX(F1)	
	X(P12)	
		S(R11j)
IX(F2)		
IX(P21)		
X(R211)		
UNLOCK(R211)		
UNLOCK(P21)		
UNLOCK(F2)		
		S(F2)

T1	T2	T3
	UNLOCK(P12)	
	UNLOCK(F1)	
	UNLOCK(DB)	
UNLOCK(R111)		
UNLOCK(P11)		
UNLOCK(F1)		
UNLOCK(DB)		
		UNLOCK(R11j)
		UNLOCK(P11)
		UNLOCK(F1)
		UNLOCK(F2)
		UNLOCK(DB)

Multiple-granularity locking: Summary

- Allows flexibility for each transaction to choose locking granularity independently
- Introduces hierarchy of objects
- Introduces intention locks