Transaction Isolation: Two-Phase Locking & Deadlocks

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Lock-Based Protocols
2. Deadlocks
3. Lock Granularities
4. Bonus: Hierarchical Locking Demo

Some material from lectures created by Professor Pavlo (CMU) (used with permission)

Context

- Previous lecture: we began a series of lectures on "how to implement transactions"
- Presented the concept of serializable schedules: schedules that are equivalent to some serial schedule
- We proceeded to define "equivalence" in terms of conflict serializability
  - *"Can the concurrent schedule be transformed into a serial schedule after swapping of non-conflicting operations?"*
- We provided a test to verify conflict serializability: *"is the schedule's dependency graph acyclic?"*

Today

- Previous lecture provided us with solid theoretical foundation for concurrency control ☺
- But: no practical guidance for how the DBMS should construct a schedule ☹
- After all: testing a schedule for serializability after it has executed (or even after it's been constructed) is a little too late!
- Our goal is to develop concurrency control protocols that will automatically guarantee serializability
  - Without needing to inspect a dependency graph

Lock-Based Protocols

Locks: The Basic Idea

- Use locks to protect database objects from concurrent access
- This is a "pessimistic" approach to concurrency control: ensure that the DBMS doesn't even create a schedule containing conflicts
- Think of a lock as a variable that's associated with a data item
- The lock describes the status of that data item with respect to *"what operations can be applied now?"*
- We'll begin by assuming that locks have binary state: either "locked" or "unlocked"
  - Locked ⇒ item cannot be accessed by another thread
  - Unlocked ⇒ item can be accessed
  - We'll change that assumption very soon ☺
- We'll begin by assuming "one lock per data item"
  - Relax that assumption later

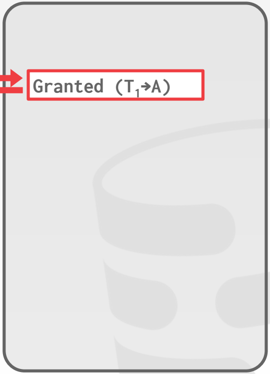
EXECUTING WITH LOCKS

TIME

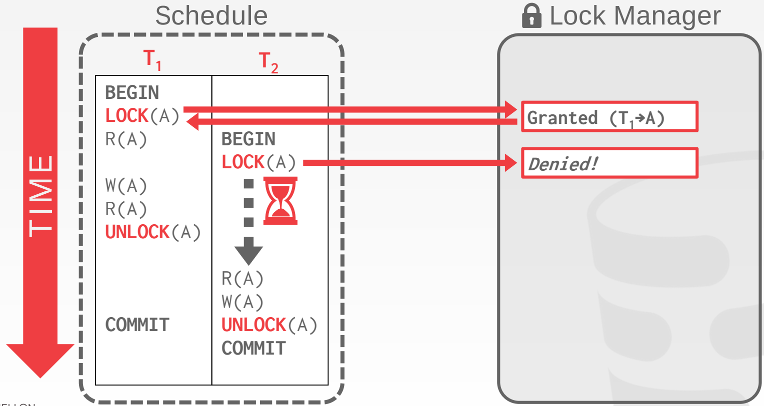
Schedule

T_1	T_2
BEGIN	
LOCK(A)	
R(A)	BEGIN
	LOCK(A)
W(A)	
R(A)	
UNLOCK(A)	
	R(A)
	W(A)
COMMIT	UNLOCK(A)
	COMMIT

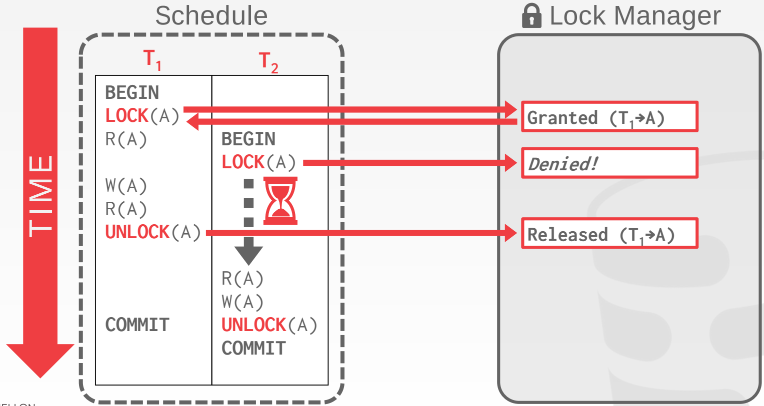
Lock Manager



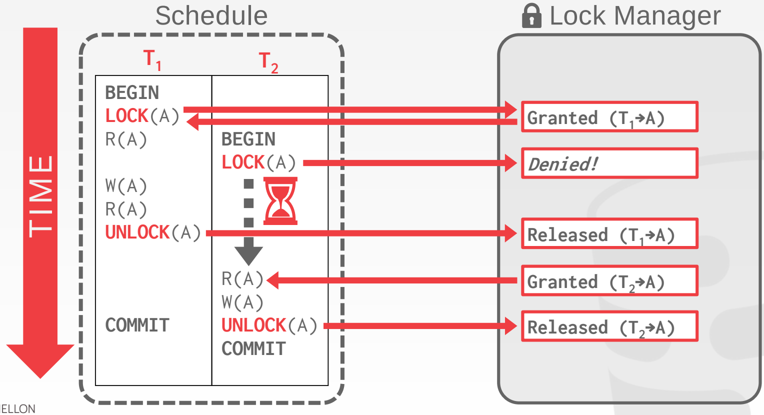
EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



Division of responsibilities:

- Transactions request locks (or “lock upgrades”)
- Lock manager responds to requests from transaction: either grants or blocks requests
- Transactions responsible for releasing locks when no longer needed
- Lock manager maintains an internal lock-table
  - Updates the lock table in response to lock request/release API invocations
- The lock-table tracks *“which transactions hold which locks?”*
- The lock table also tracks *“which transactions are waiting to acquire locks?”*
  - We refer to a tx that’s “waiting” as a “blocked transaction”

Enrich Lock Semantics With "Read Versus Write"

- We previously said that a lock has "binary state"
- In practice, that model is too restrictive for DBMS use
  - After all: why not allow multiple readers to access the same data concurrently?
- So: enrich model to allow data items to be locked in one of two modes
  - Exclusive (X) mode: data item can be both read as well as written
  - Shared (S) mode: data item can only be read
- Now the question of whether a lock should or should not be granted depends on
  - *"Is the data item already locked?"*
  - *"In what mode is the datum locked?"*

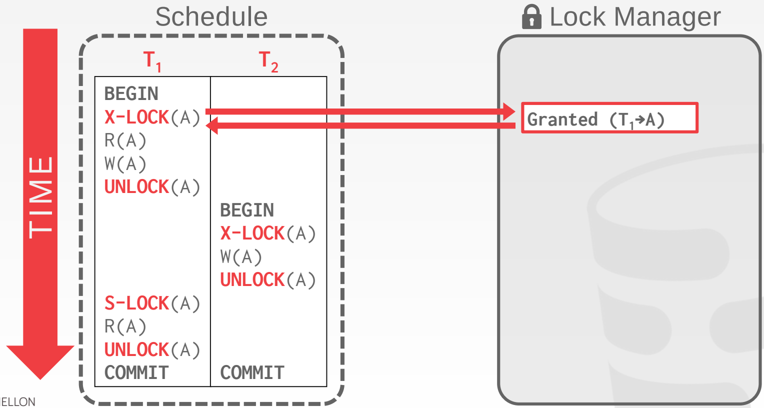
Unlike non-transactional models, programmers do not make explicit lock requests to the DBMS: these are generated automagically ☺

Lock Compatibility Matrix

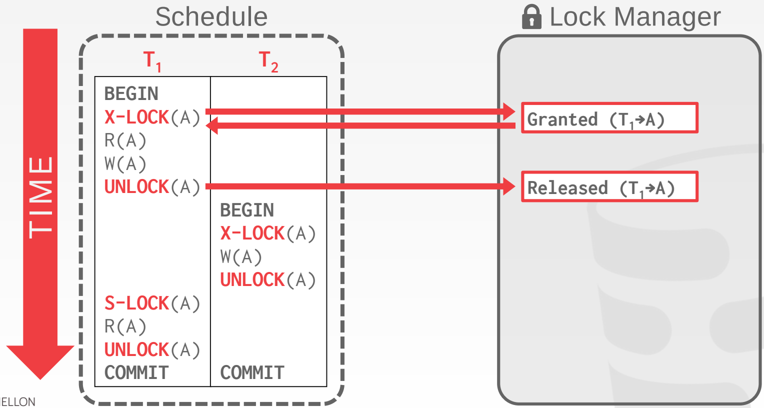
	S	X
S	true	false
X	false	false

- Transaction is granted a lock *iff* requested lock is compatible (see Figure above) with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item
- One transaction holds an exclusive lock on the item → no other transaction will be granted a lock
- Requesting transaction blocks if lock is not granted
  - System tracks status of existing, incompatible, locks held by other transactions
  - When all other locks have been released, transaction is granted lock, and resumes execution

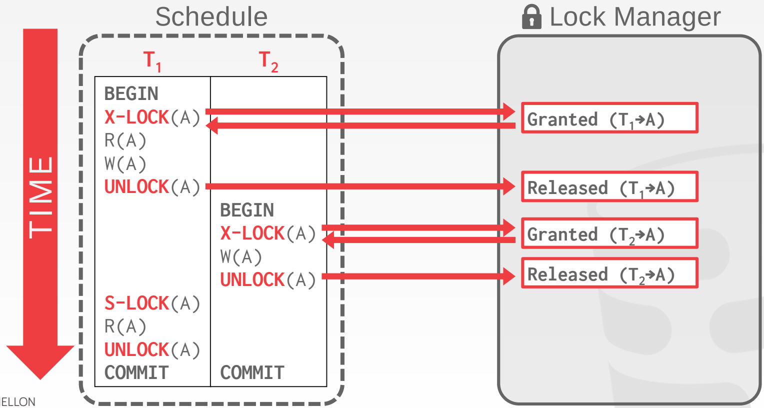
EXECUTING WITH LOCKS



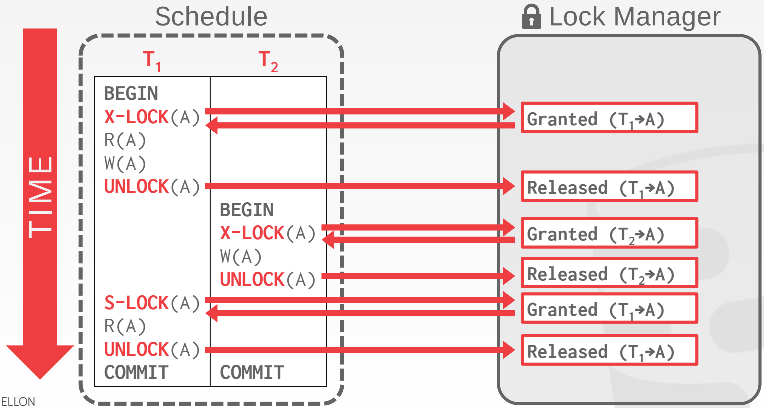
EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



Locking (By Itself) Is Not A CC Protocol!

T_2: **lock-S**(A);
  read (A);
  unlock(A);
  lock-S(B);
  read (B);
  unlock(B);
  display$(A+B)$

- Important: locking as performed in the Figure above does not guarantee serializability!
- Example: if *A* and *B* are written by another tx in-between read of *A* and *B*, displayed sum would be wrong
- We need to use a locking protocol to further restrict the set of possible schedules
  - "A set of rules followed by all transactions while requesting and releasing locks"
- Goal: a locking protocol that guarantees conflict serializability

Two-Phase Locking Protocol

- The two-phase locking protocol does guarantee conflict-serializable schedules
- Phase 1: The "growing phase"
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: The "shrinking phase"
  - Transaction may release locks
  - Transaction may not obtain locks
- This version is "vanilla" 2PL: allows transactions to release locks even before commit point
- The protocol assures serializability!
- Can be proven that the transactions can be serialized in the order of their lock points
  - The point where a transaction acquired its final lock
  - Important: "lock point" ≠ "commit point" (hold that thought)

EXECUTING WITH LOCKS

TIME

Schedule

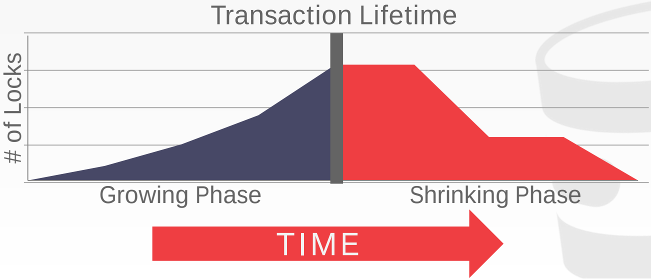
T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager

- Granted (T_1→A)
- Released (T_1→A)
- Granted (T_2→A)
- Released (T_2→A)
- Granted (T_1→A)
- Released (T_1→A)

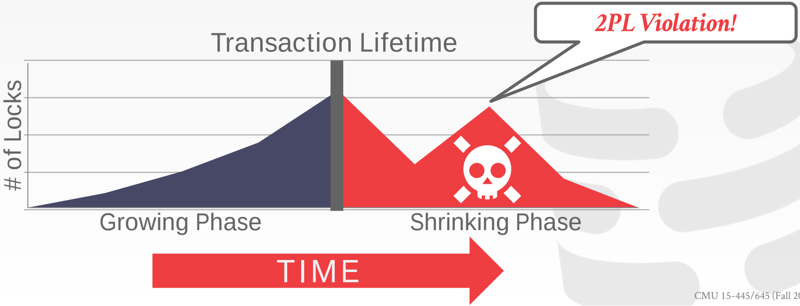
TWO-PHASE LOCKING

The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

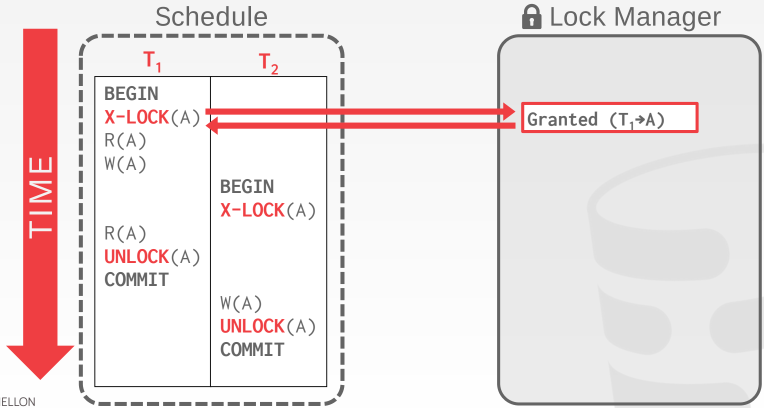


TWO-PHASE LOCKING

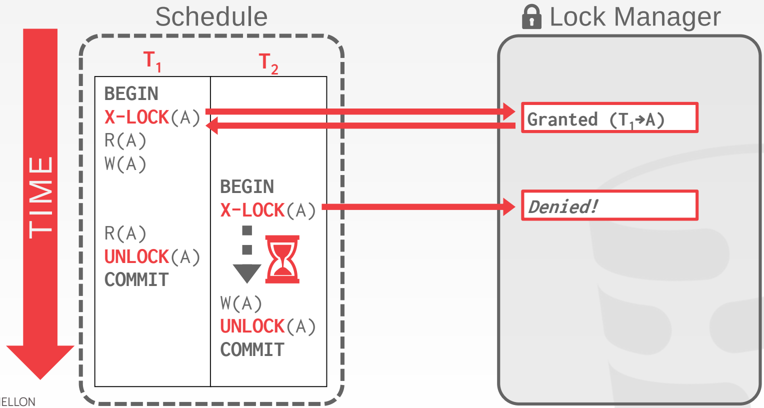
The txn is not allowed to acquire/upgrade locks after the growing phase finishes.



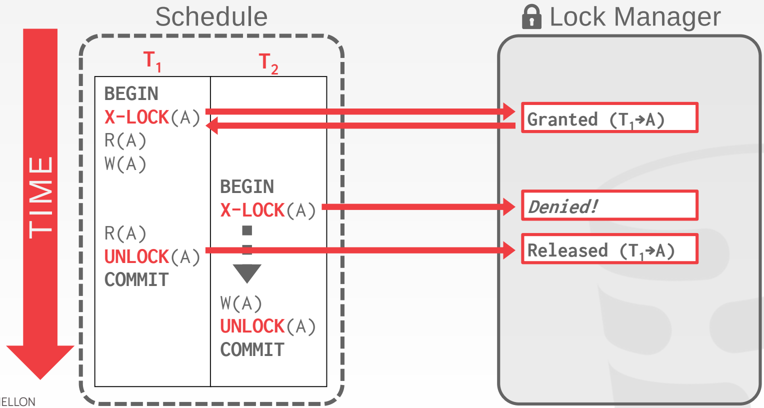
EXECUTING WITH 2PL



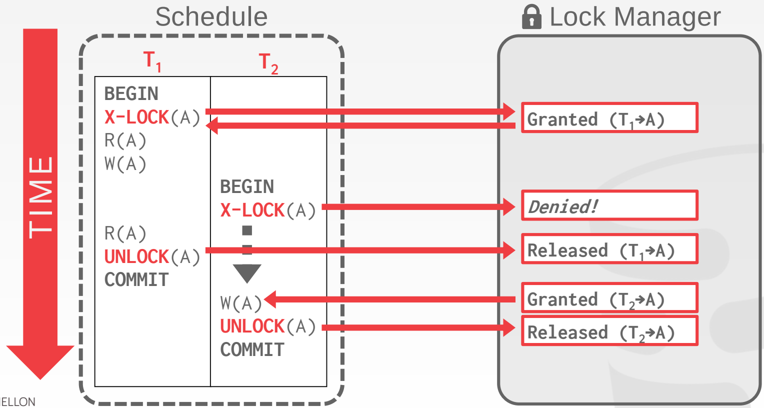
EXECUTING WITH 2PL



EXECUTING WITH 2PL



EXECUTING WITH 2PL



2PL: The Good News & The Bad

- If every transaction in a schedule follows the two-phase locking protocol, then the resulting schedule is guaranteed to be serializable
  - We can show that 2PL generates schedules whose dependency graph is acyclic
  - 2PL may limit the amount of concurrency that can occur in a schedule because some serializable schedules will be prohibited by two-phase locking protocol
- We can prove that txs in a 2PL schedule are serialized in the order of their lock points
  - Meaning: the point where a transaction acquired its final lock
- Unfortunately: 2PL schedules are subject to cascading aborts (last lecture)
  - The problem: DBMS doesn't prevent other txs from seeing the "dirty data" affected by the aborted tx

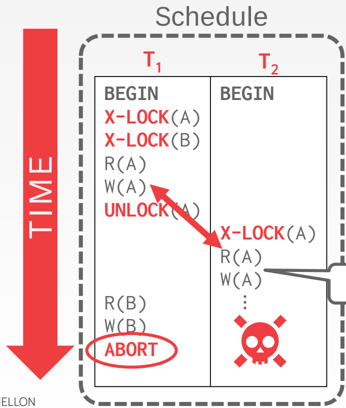
2PL – CASCADING ABORTS

TIME

Schedule

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
X-LOCK(B)	
R(A)	
W(A)	
UNLOCK(A)	
	X-LOCK(A)
	R(A)
	W(A)
R(B)	⋮
W(B)	
ABORT	

2PL – CASCADING ABORTS



This is a permissible schedule in 2PL, but the DBMS has to also abort T_2 when T_1 aborts.

→ Any information about T_1 cannot be "leaked" to the outside world.

This is all wasted work!

Variations On 2PL

- Conservative 2PL (or "static") requires that a tx lock all the items it accesses before the tx begins
    - This variant requires that the tx "pre-declare" its read-set and write-set
    - Advantage: this variant is a deadlock-free protocol
    - (More on deadlocks later)
- Strict 2PL: tx does not release exclusive locks until after it commits or aborts
    - Advantage: doesn't incur cascading aborts and ensures "recoverability"
    - Advantage: aborted txs can be undone simply by restoring original values of modified data
- Rigorous 2PL: tx holds all locks until completion
    - Txs can be serialized in the order in which they commit
- Note: most databases implement rigorous 2PL, but refer to it as simply "two-phase locking"

NON-2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	
	S-LOCK(A)
A=A-100	
W(A)	
UNLOCK(A)	
	R(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	R(B)
	UNLOCK(B)
	ECHO A+B
	COMMIT
R(B)	
B=B+100	
W(B)	
UNLOCK(B)	
COMMIT	

Initial Database State

A=1000, B=1000

NON-2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	
	S-LOCK(A)
A=A-100	
W(A)	
UNLOCK(A)	
	R(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	R(B)
	UNLOCK(B)
	ECHO A+B
	COMMIT
R(B)	
B=B+100	
W(B)	
UNLOCK(B)	
COMMIT	

Initial Database State

A=1000, B=1000

NON-2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	
	S-LOCK(A)
A=A-100	
W(A)	
UNLOCK(A)	
	R(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	R(B)
	UNLOCK(B)
R(B)	ECHO A+B
B=B+100	COMMIT
W(B)	
UNLOCK(B)	
COMMIT	

Initial Database State

A=1000, B=1000

NON-2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	
	S-LOCK(A)
A=A-100	
W(A)	
UNLOCK(A)	
	R(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	R(B)
	UNLOCK(B)
	ECHO A+B
	COMMIT
R(B)	
B=B+100	
W(B)	
UNLOCK(B)	
COMMIT	

Initial Database State

A=1000, B=1000

T_2 Output

A+B=1100

2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	S-LOCK(A)
A=A-100	
W(A)	
X-LOCK(B)	
UNLOCK(A)	R(A)
	S-LOCK(B)
R(B)	
B=B+100	
W(B)	
UNLOCK(B)	R(B)
COMMIT	UNLOCK(A)
	UNLOCK(B)
	ECHO A+B
	COMMIT

Initial Database State

A=1000, B=1000

2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	**S-LOCK**(A)
A=A-100	
W(A)	
X-LOCK(B)	
UNLOCK(A)	R(A)
	S-LOCK(B)
R(B)	
B=B+100	
W(B)	
UNLOCK(B)	R(B)
COMMIT	**UNLOCK**(A)
	UNLOCK(B)
	ECHO A+B
	COMMIT

Initial Database State

A=1000, **B**=1000

2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	**S-LOCK**(A)
A=A-100	
W(A)	
X-LOCK(B)	
UNLOCK(A)	R(A)
	S-LOCK(B)
R(B)	
B=B+100	
W(B)	
UNLOCK(B)	R(B)
COMMIT	**UNLOCK**(A)
	UNLOCK(B)
	ECHO A+B
	COMMIT

Initial Database State

A=1000, B=1000

T_2 Output

A+B=2000

STRICT 2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	S-LOCK(A)
A=A-100	
W(A)	
X-LOCK(B)	
R(B)	
B=B+100	
W(B)	
UNLOCK(A)	
UNLOCK(B)	R(A)
COMMIT	S-LOCK(B)
	R(B)
	ECHO A+B
	UNLOCK(A)
	UNLOCK(B)
	COMMIT

Initial Database State

A=1000, B=1000

STRICT 2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	S-LOCK(A)
A=A-100	
W(A)	
X-LOCK(B)	
R(B)	
B=B+100	
W(B)	
UNLOCK(A)	
UNLOCK(B)	R(A)
COMMIT	S-LOCK(B)
	R(B)
	ECHO A+B
	UNLOCK(A)
	UNLOCK(B)
	COMMIT

Initial Database State

A=1000, B=1000

STRICT 2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	S-LOCK(A)
A=A-100	
W(A)	
X-LOCK(B)	
R(B)	
B=B+100	
W(B)	
UNLOCK(A)	
UNLOCK(B)	R(A)
COMMIT	S-LOCK(B)
	R(B)
	ECHO A+B
	UNLOCK(A)
	UNLOCK(B)
	COMMIT

Initial Database State

A=1000, B=1000

T_2 Output

A+B=2000

Deadlocks

Deadlocks: A Real Problem

- A deadlock is a cycle of transactions waiting for locks to be released by each other
  - Trivially triggered by schedules that don't require txs to "pre-declare" their read-sets and write-sets
- Two ways of dealing with deadlocks
  - Deadlock detection: DBMS allows deadlocks to occur, "terminates with extreme prejudice" when it detects deadlock situation
  - Deadlock prevention: when a tx tries to acquire a lock that is currently held by another tx, DBMS kills one of them to prevent a deadlock

Deadlock Detection: "Waits-For" Graphs

- DBMS creates a "waits-for" graph to keep track of what locks each txn is waiting to acquire
  - Graph nodes are transactions
  - Graph inserts an edge from $T_i \Rightarrow T_j$ *"iff T_i is waiting for T_j to release a lock"*
- DBMS periodically check for cycles in its "waits-for" graph and then make a decision on how to break the deadlock (next slide)
- The decision: select a "victim" txn to rollback that breaks the cycle in the "waits-for" graph
  - Victim is either restarted afterwards or simply aborted
- Note the trade-off between how often the DBMS invokes the "deadlock detection" algorithm and how long txs have to wait while they're deadlocked

Deadlock Handling: "Victim" Selection

- The algorithm has many variables it can tune (or ignore)
- Examples: select the victim …
  - "By age" (most recent tx)
  - "By progress" (tx with fewest operations performed so far)
  - "By # of items already locked"
  - "By the # of txs that the DBMS will have to rollback with it"
- In addition to optimizing for "overall performance", the algorithm may choose "fairness"
  - Example: *"the number of times that the tx has previously been aborted due to deadlocks"*
  - Systems that don't consider this factor may suffer from "tx starvation"

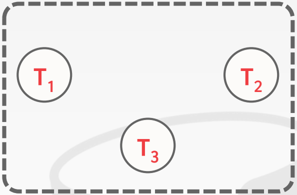
DEADLOCK DETECTION

Schedule

TIME

T_1	T_2	T_3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-For Graph



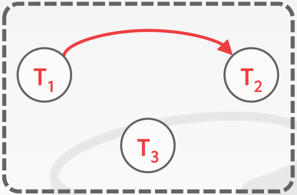
DEADLOCK DETECTION

Schedule

TIME

T_1	T_2	T_3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-For Graph



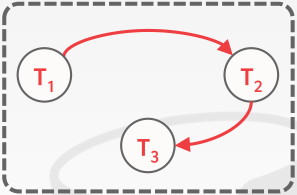
DEADLOCK DETECTION

Schedule

TIME

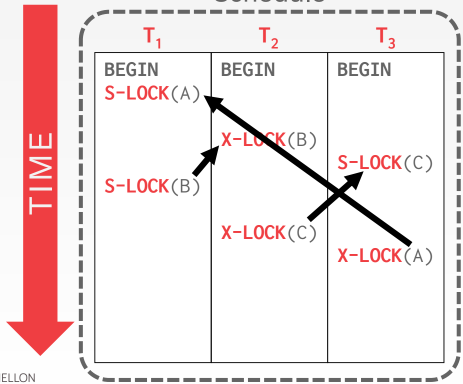
T_1	T_2	T_3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-For Graph

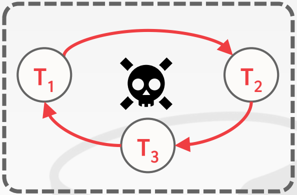


DEADLOCK DETECTION

Schedule



Waits-For Graph



Deadlock Prevention

- No need to detect a deadlock: as soon as tx_i requests a lock that is currently held by tx_j, DBMS aborts one of these txs
- But which tx should be aborted?
- Can assign priorities based on timestamps (“older tx has higher priority”)
  - Then based on such priorities, implement one of two algorithms
- Wait-Die (or “Old Waits for Young”)
  - If *requesting tx* has higher priority than *holding tx*, then *requesting tx* waits for *holding tx*
  - Otherwise: younger *requesting tx* aborts
- Wound-Wait (or “Young Waits for Old”)
  - If *requesting tx* has higher priority than *holding tx*, then *holding tx* aborts (and releases lock)
  - Otherwise: younger *requesting tx* waits

Protocol Similarities

- In both wait-die and wound-wait schemes, the rolled back tx is restarted with its original timestamp
  - Eventually, the aborted (younger) transactions will become the oldest transactions in the system and complete successfully
  - So: both protocols are therefore fair
- In both schemes older txs "beat" younger txs
- Q: can you suggest why we favor older txs this way?
- A: older txs (having run longer) typically hold more locks and have read/written more data than younger txs
  - So relatively more expensive to rollback older txs

> Both deadlock prevention schemes are based on the concept of imposing a partial ordering of all data items, then requiring that txs can only lock data items in the order specified by the partial order

Protocol Differences

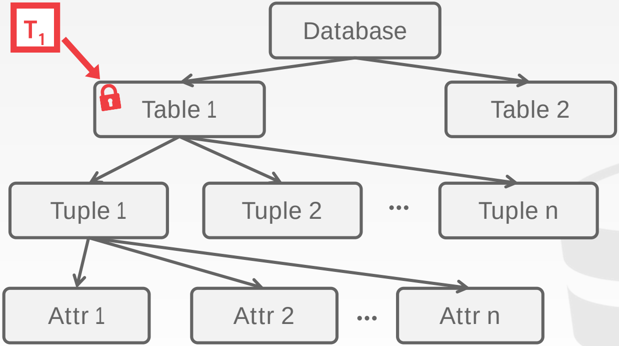
- Wait-Die: younger tx is killed when it requests a lock currently held by an older tx
- Wound-Wait: younger tx is killed when older tx requests a lock currently held by the younger tx
- Wait-die incurs more rolled back txs
  - Younger txs make relatively more lock requests than older transactions
  - But those younger txs have probably performed “less work”
- Wound-wait incurs fewer rolled back txs
  - They’ll be making (relatively more lock requests) and this protocol allows them to wait
  - Only aborted when older tx (which is less likely to make lock requests) tries to lock the younger tx’s data
  - But: in “wound-wait”, when we do rollback the younger tx, it will have done “more work” compared to transactions rolled back under the “wait-die” protocol
    - By definition: will be holding locks (and have done some work)

Lock Granularities

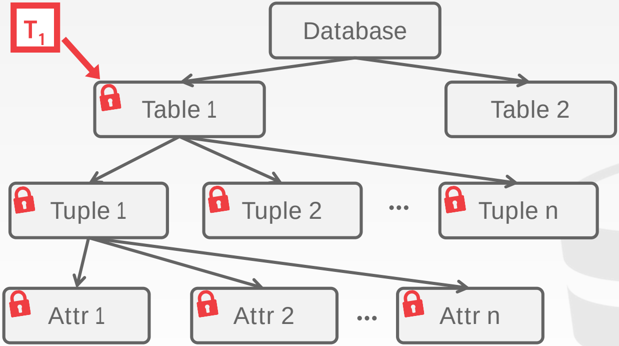
Can "Coarse Locks" Improve Performance?

- One approach to improve lock-based performance is to increase the granularity of a lock footprint
  - Note: until now, we've been assuming a one-to-one mapping from database objects to locks
  - A million item $\Rightarrow$ million locks
- What if we reduce the work that the lock manager must perform
- Specifically: model "data items" as a hierarchy of data granularities
  - Where the small granularities are nested within larger ones
  - Think of a "tree" data-structure
- Locking a "node" higher in the tree explicitly also locks lower levels of that tree implicitly
- Tradeoff:
  - Finer granularity, lower in tree $\Rightarrow$ high concurrency but high locking overhead
  - Coarse granularity, higher in tree $\Rightarrow$ low locking overhead but low concurrency

DATABASE LOCK HIERARCHY



DATABASE LOCK HIERARCHY



A New Lock Type: Intention Locks

- The motivation for coarse-granularity locking is efficiency
  - Example: by locking at the file level, no need to lock at record level
- There's a major problem lurking here: can **you spot it**?
- Scenario: T_i locked F_b explicitly, now T_j wants to lock a record that's nested in F_b
  - How does the DBMS "know" that T_i has the lock on that record?
  - Must traverse locking tree from root to this record to get that information
    - This problem is not a "deal-breaker": just an observation
- Another scenario: T_j wants to lock at a higher level than T_i lock granularity
  - Only way to block that request is for DBMS to traverse the entire tree!
- Seems to entirely negate the motivation for this idea ☺

Intention Lock Modes Solve This Problem (I)

- Intention lock modes are an efficient way to record the nested locking information
- DBMS acquires intention locks at higher levels of the tree before acquiring an explicit lock at lower levels of the tree
- Example: to acquire a row-lock, DBMS requires that
  1. An intention lock be acquired on the table
  2. After the table lock has been granted, an intention lock is acquired on the page
  3. Only after the page lock has been granted, will the DBMS allow the tx to acquire the row lock
- Locks are acquired in root-to-leaf order, released in leaf-to-root order
- Payoff: T_j detects the intention lock without having to traverse the entire tree

Intention Lock Modes Solve This Problem (II)

- Intention-Shared (IS) lock
  - Indicates explicit locking at a lower level with shared locks
- Intention-Exclusive (IX) local
  - Indicates explicit locking at lower level with exclusive or shared locks
- Shared + Intention-Exclusive (SIX)
  - Subtree rooted at that node is locked explicitly in shared mode and indicate explicit locking at a lower level with exclusive locks
- See textbook for details
- Also: bonus section in lecture

COMPATIBILITY MATRIX

T_2 Wants

T_1 Holds	IS	IX	S	SIX	X
IS	✔	✔	✔	✔	✕
IX	✔	✔	✕	✕	✕
S	✔	✕	✔	✕	✕
SIX	✔	✕	✕	✕	✕
X	✕	✕	✕	✕	✕

(Intention) Lock Protocol

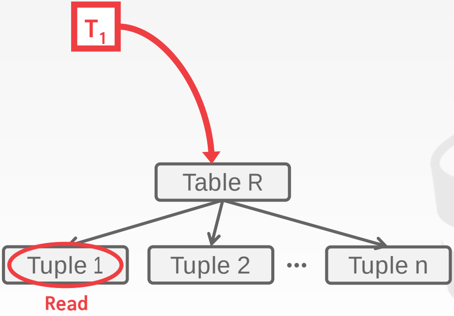
- Each tx obtains appropriate lock at highest level of the database "tree"
- To get S or IS lock on a node, the txn must hold at least IS on parent node
- To get X, IX, or SIX on a node, must hold at least IX on parent node

Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes

Bonus: Hierarchical Locking Demo

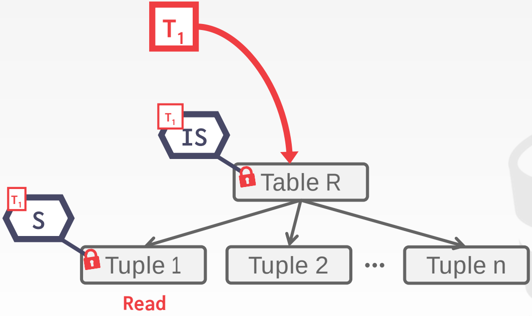
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in R.



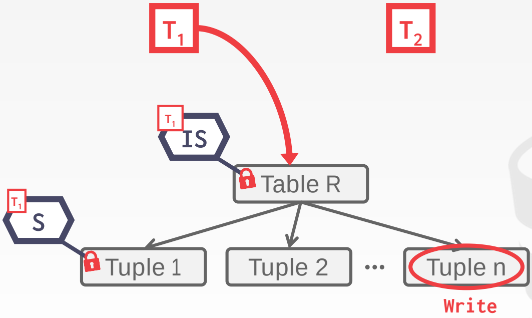
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in R.



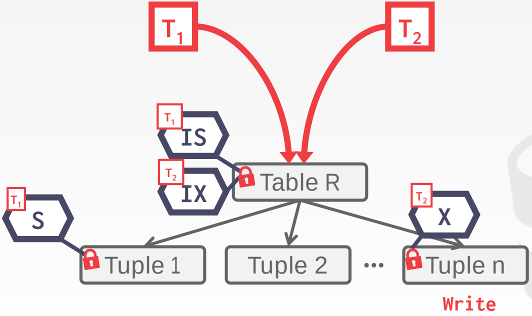
EXAMPLE – TWO-LEVEL HIERARCHY

Update Lin's record in R.



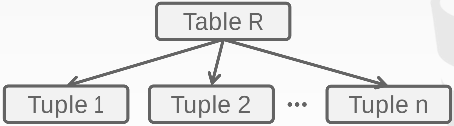
EXAMPLE – TWO-LEVEL HIERARCHY

Update Lin's record in R.



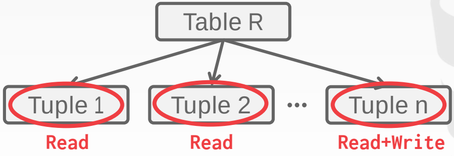
EXAMPLE – THREESOME

Assume three txns execute at same time:
→ T_1 – Scan **R** and update a few tuples.
→ T_2 – Read a single tuple in **R**.
→ T_3 – Scan all tuples in **R**.



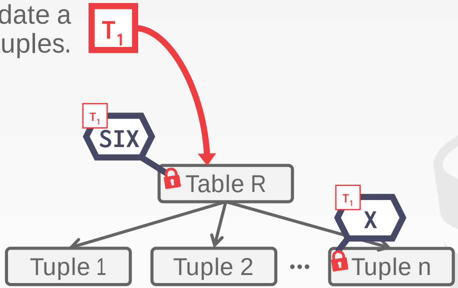
EXAMPLE – THREESOME

Scan **R** and update a few tuples. T_1



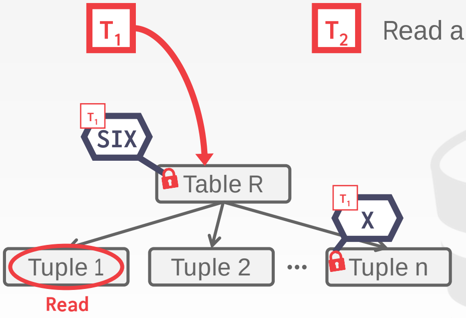
EXAMPLE – THREESOME

Scan **R** and update a few tuples.



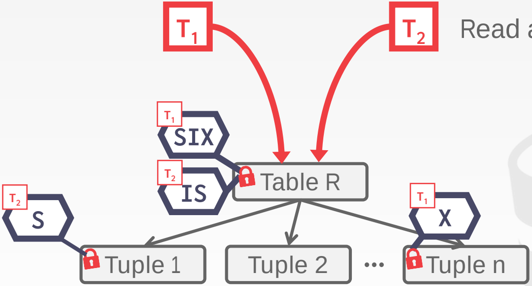
EXAMPLE – THREESOME

T_2 Read a single tuple in **R**.



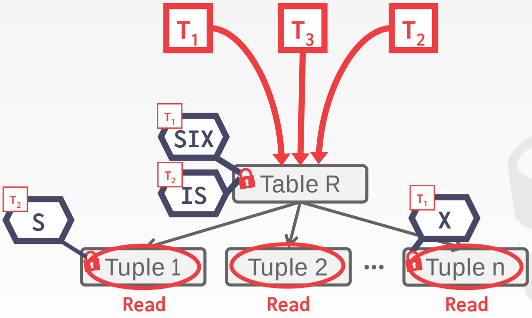
EXAMPLE – THREESOME

Read a single tuple in **R**.



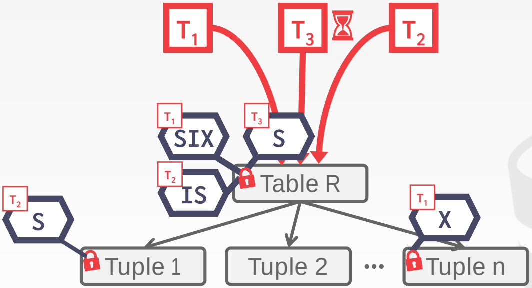
EXAMPLE – THREESOME

Scan all tuples in **R**.



EXAMPLE – THREESOME

Scan all tuples in **R**.



Lock-Based Protocols

Deadlocks

Lock Granularities

Bonus: Hierarchical Locking Demo

Readings

- Today's lecture corresponds to the textbook Chapter 18.1-18.3