CSE 541: Database Systems I

Locking

Slides Credit: Andy Pavlo, Tianzheng Wang

Locking Basics

- Each database object (e.g., a record or a table) is associated a lock
- A lock can be held in two modes
  - Shared (S): allow read
  - Exclusive (X): allow read and write
- General Idea: use a lock to protect the object from non-compatible accesses
  - Reads are compatible with each other
  - Read-write, write-write are not compatible with each other
- Lock upgrade: an S-lock can be upgraded to an X-lock
- Lock downgrade: an X-lock can be downgraded to an S-lock

Assumptions (for now)

- The database does not grow or shrink
  - Only read and write (i.e., update) operations are allowed
  - Scan is modeled after reading each record involved
- We consider inserts, deletion, and scan in later lectures

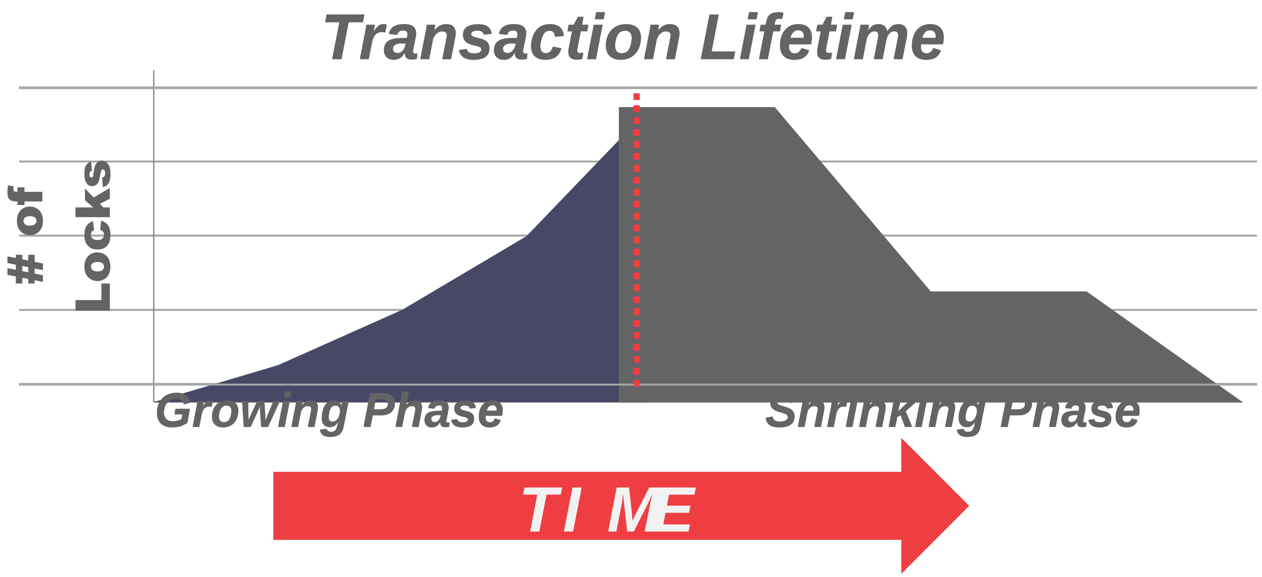
Two-Phase Locking (2PL)

- A concurrency control protocol that determines whether a txn can access an object in the database on the fly.
- The protocol does **NOT** need to know all the queries that a txn will execute ahead of time.

- **Phase #1: Growing**
  - Each txn requests the locks that it needs from the DBMS's lock manager.
  - The lock manager grants/denies lock requests.
- **Phase #2: Shrinking**
  - The txn is allowed to only release locks that it previously acquired. It cannot acquire new locks.

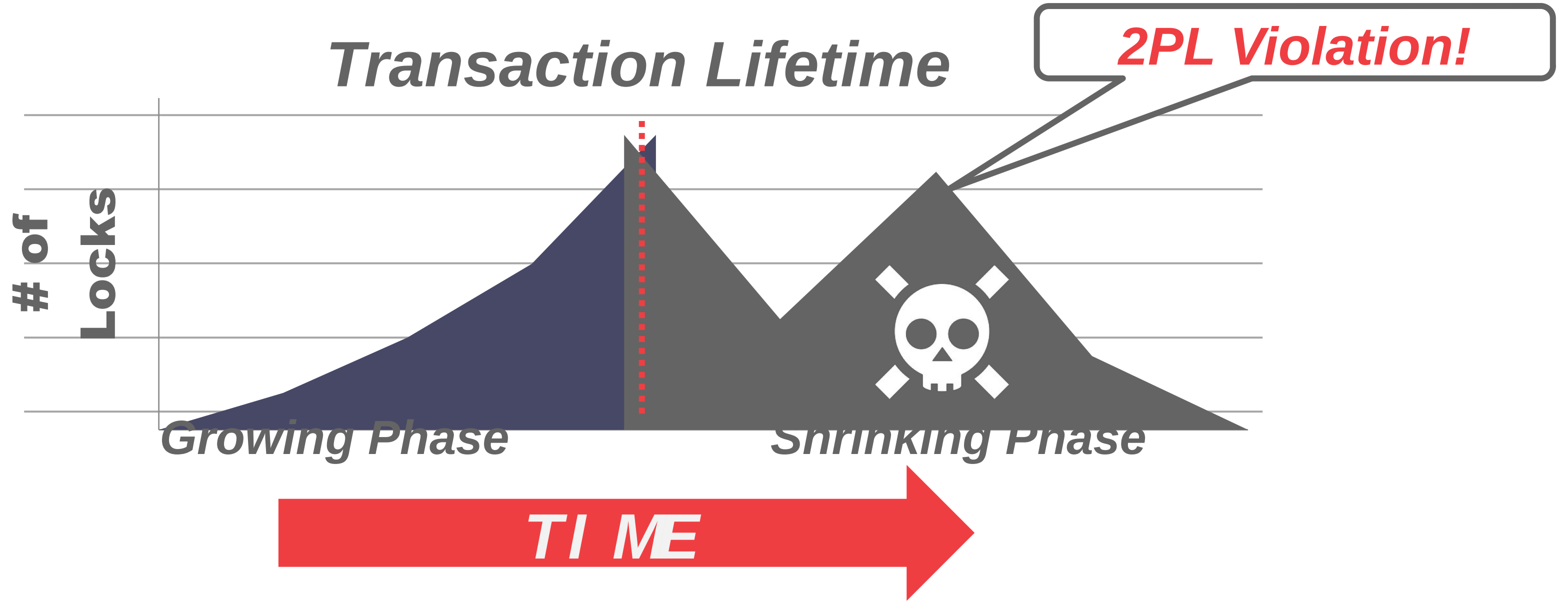
Two Phase Locking

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

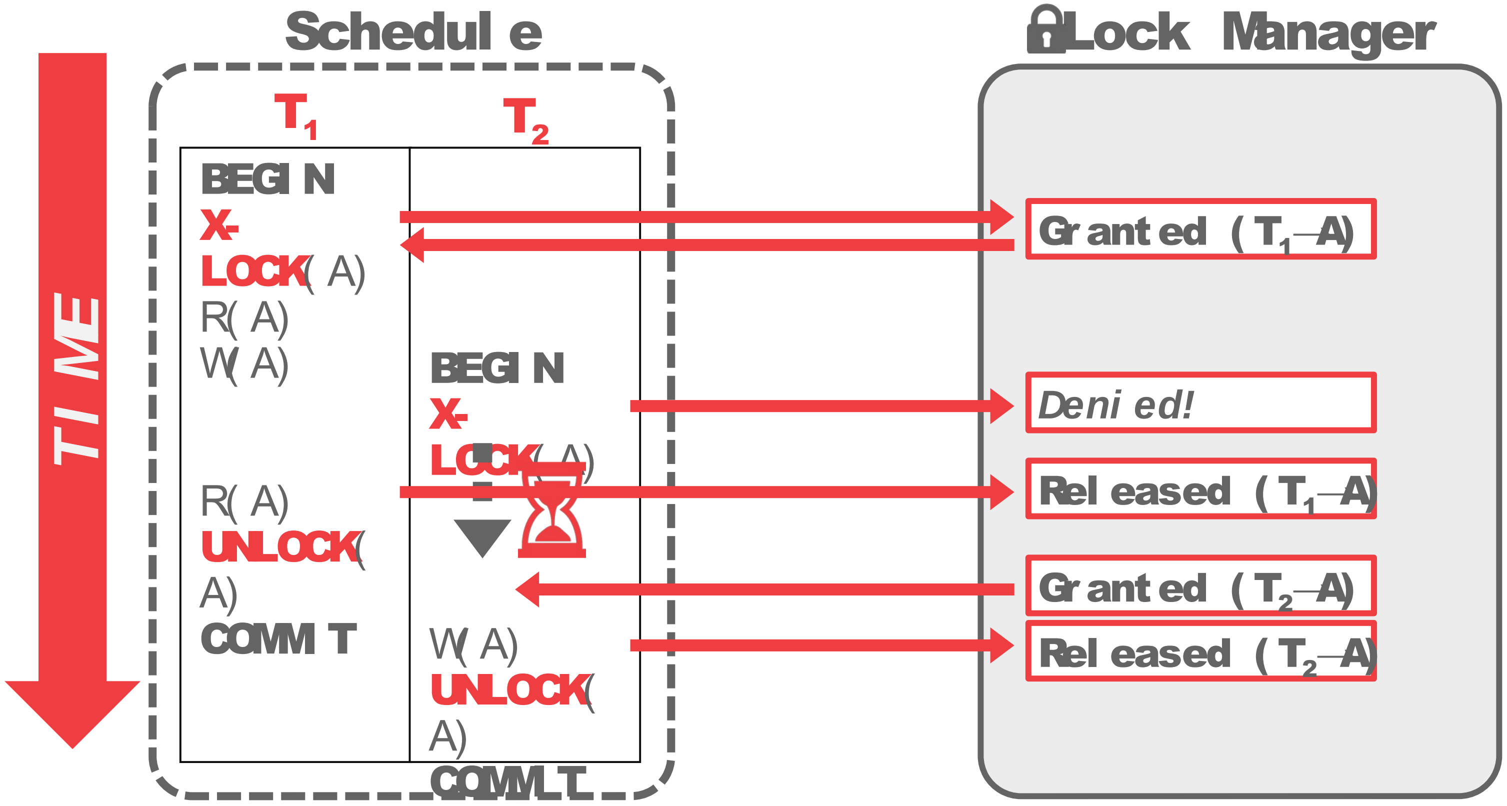


Two Phase Locking

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



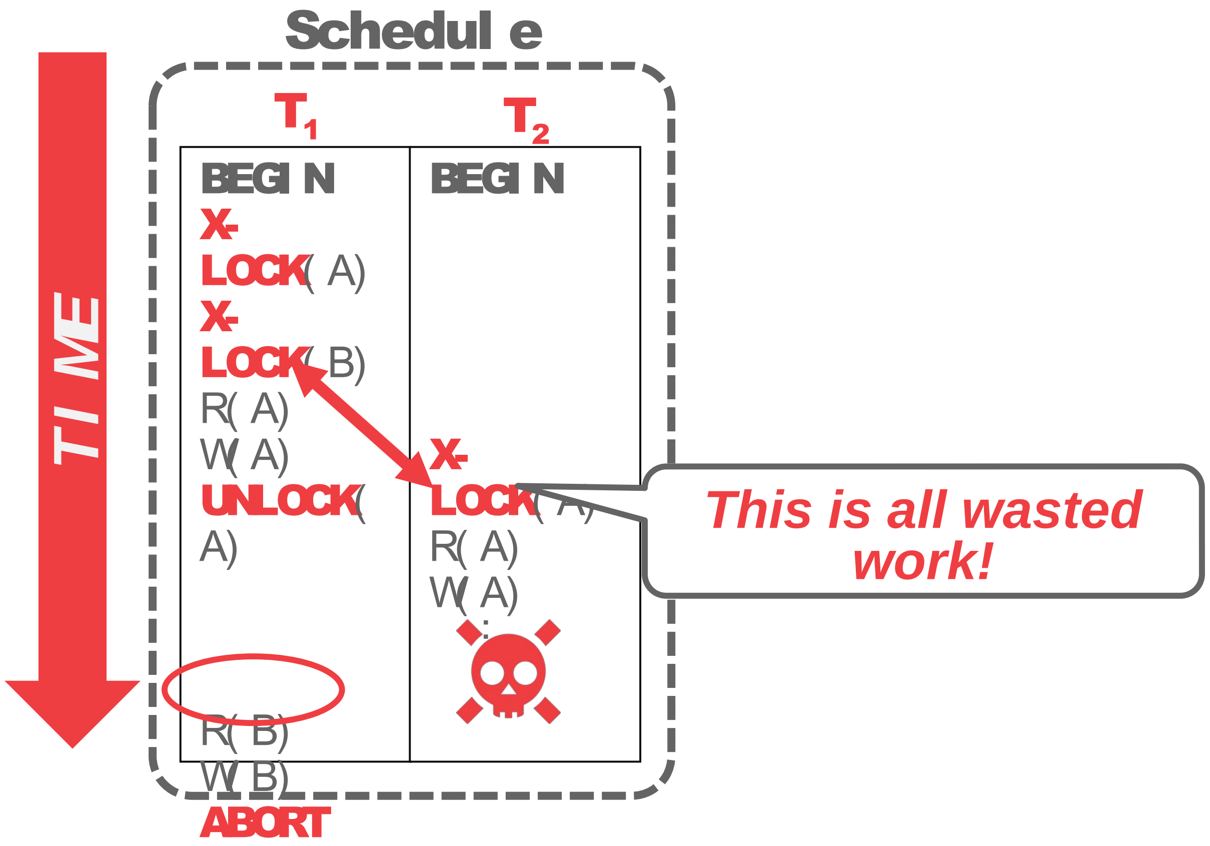
Executing with 2PL



2PL Guarantees

- What property does 2PL guarantees?
- Conflict Serializability → Acyclic Dependency Graphs
- Will this save us totally from CC?
  - NO.
  - Concurrency is limited. (Serializable schedules but not allowed by 2PL)
  - Subject to **cascading aborts** and **deadlocks**.

Cascading Aborts



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

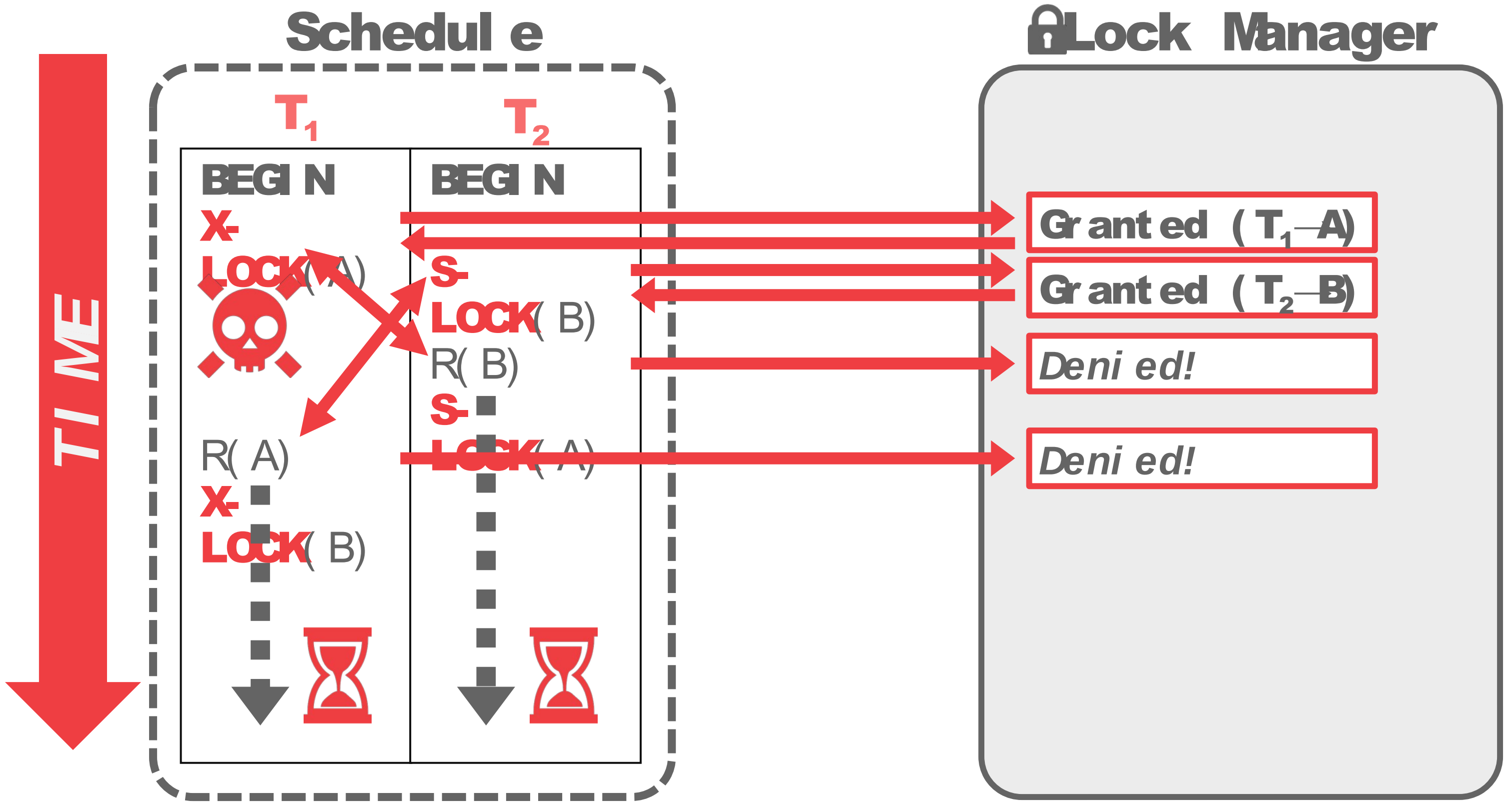
- Any information about (aborted) T_1 cannot be "leaked" to the outside world!

Solution to Cascading Abort: SS2PL

- SS2PL: Strong Strict Two-Phase Locking
- 2PL but will only release all locks at the end of txns.

- Slightly better but still guarantees free of cascading aborts?
- Are we deadlock free yet?

Deadlocks



TIME
Schedule
T1
T2
BEGIN
X-LOCK(A)
R(A)
X-LOCK(B)
BEGIN
S-LOCK(B)
R(B)
S-LOCK(A)
Lock Manager
Granted (T1→A)
Granted (T2→B)
Denied!
Denied!

Deadlock Theory

Definition: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does.

Necessary and sufficient conditions for deadlocks:

- **Mutual exclusion:** Resources cannot be shared
- **Hold and wait**: A thread is both holding a resource and waiting on another resource to become free
- **No preemption**: Once a thread gets a resource, it cannot be taken away
- **Circular wait**: There is a cycle in the graph (resource allocation graph/waits-for-graph) of who has what and who wants what…

Eliminate deadlock by eliminating any one condition

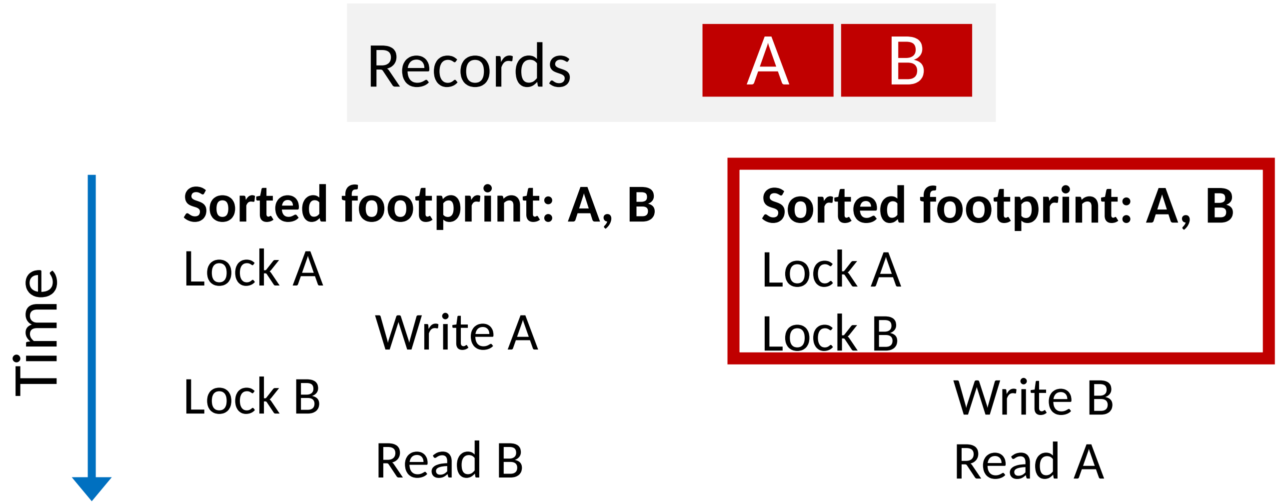
Dealing with Deadlocks

- Prevention (aka avoidance)
  - Make sure deadlocks will never happen in the first place
  - Trivial for transactions that access a single record
  - Multiple ways
    - With workload knowledge
    - Wait-die
    - Wound-wait
- Resolution
  - Resolve deadlocks after they are formed
  - Use a waits-for graph to detect cycles and abort transactions (thus releasing locks) on a cycle

Deadlock Prevention

Alternative 1: **Know workload access patterns (footprint)**

- Exact footprint given a priori – *not possible in many cases*
- Sort all locks in some consistent order that is obeyed by all transactions
- Acquire locks one by one following the consistent order
  - Block until a lock can be granted



Deadlock Prevention

Alternative 2: Prioritize transactions

- Lower prioritized transactions not allowed to wait for higher prioritized transactions
- Priority represented by timestamps
  - Assigned upon transaction startup
  - Lower timestamp ➔ higher priority

Method 1: Wait-die (can wait if priority is high)

  - T_i will wait for T_j if T_i has higher priority, otherwise T_i is aborted

Method 2: Wound-wait (abort the low-priority transaction)

  - T_i will cause T_j to be aborted if T_i has higher priority, otherwise T_i will wait

- Retry transactions with the original timestamp

Deadlock Resolution

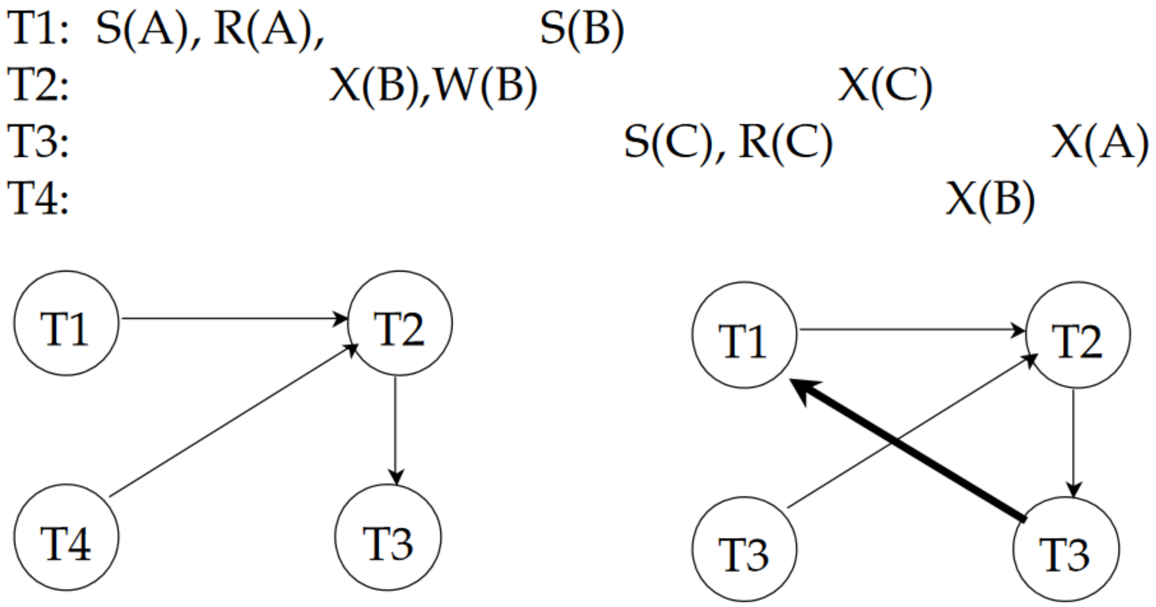
Alternative 1: Timeout

- Give up (i.e., abort self) if still cannot get lock after a certain amount of time
  - Easy to implement, but hard to know/set the right timeout value

Alternative 2: Use a system-wide *waits-for* graph

- Nodes: transactions
- Directed edges: from node T_i to node T_j if Ti is waiting for T_j to release a lock
- Cycle indicates deadlock
- Periodically traverses the waits-for graph to find cycles

Waits-For-Graph



- High overhead
  - Concurrent updates to a centralized data structure
  - Latch contention on the graph, too much inter-connect traffic

Deadlock Resolution

Alternative 3: Dreadlock*

- Avoid centralized bookkeeping of a waits-for graph
- Each thread/transaction publishes a "digest"
  - Summarizes "who am I waiting for"
    - Directly and indirectly
  - Initially contains only self
  - Always contains self
- Keep checking (e.g., spin on) lock owner's digest
  - Abort if find self in the owner's digest
    - i.e., the owner is waiting for me
  - Back-propagate changes
    - i.e., my new digest = union <owner's digest, my digest>

* Eric Koskinen and Maurice Herlihy. Dreadlocks: efficient deadlock detection. SPAA 2008.

Dreadlock Algorithm

```
public class TTASLock implements Lock {
  AtomicReference<Set<Thread>> state =
    new AtomicReference<Set<Thread>>();
  public void lock() {
    Thread me = Thread.currentThread();
    while (true) {
      // spin while lock looks busy
      while ((owner = state.get()) != null) {
        if (owner.contains(me)) {
          throw new AbortedException();
        } else if (owner.changed()) {
          // back-propagate digest
          me.digest.setUnion(owner, me);
        }
      }
      // lock looks free, try to acquire
      if (state.compareAndSet(null, me)) {
        me.digest.setSingle(me);
        return;
      }
    }
  }
}
```

Gradually build up the digest that contains

- Myself
- Who am I waiting for directly
- Who am I waiting for indirectly

Pro: No need for a centralized waits-for graph

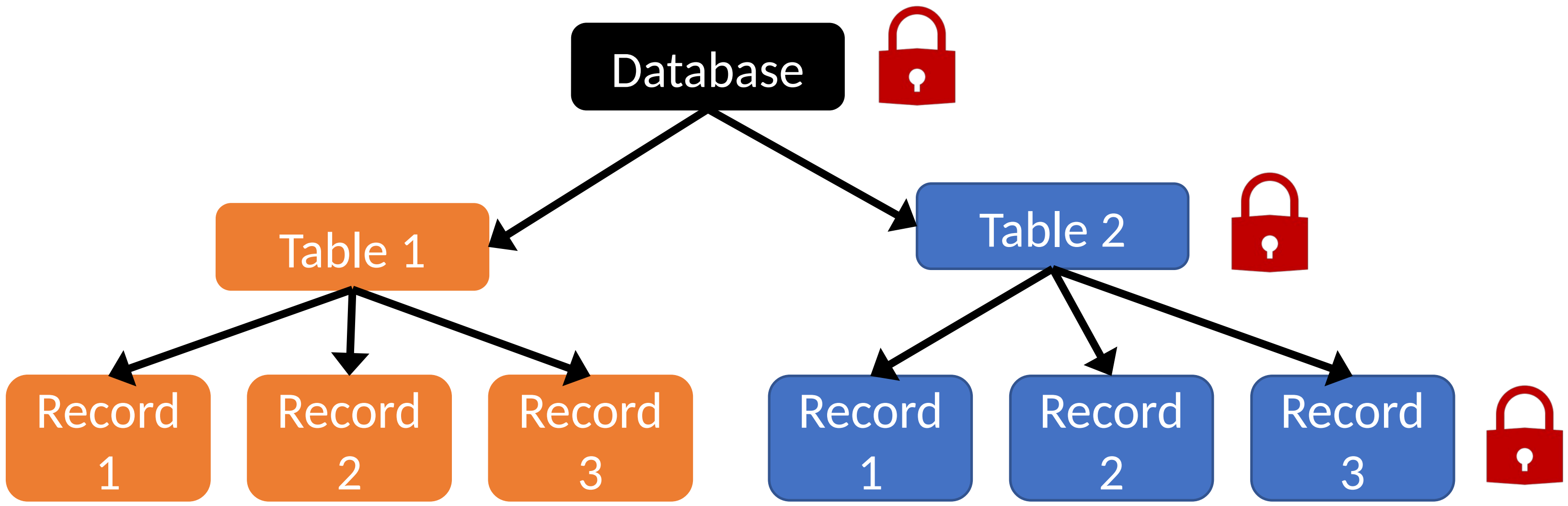
Con: Still need inter-thread communication, can become a problem on large machines

What we have covered so far…

Row-level Locking: each record is associated with a lock

- Sometimes too much overhead
  - Scan a table of 1 million records, 1 million locks to take!
- Observation: Data is organized in a hierarchical way
  - Field $\sqsubset$ Record $\sqsubset$ Table/File $\sqsubset$ Database $\sqsubset$ Storage device
  - Lock large objects (table, database level)
    - Need fewer locks but low concurrency
  - Lock small objects (record level)
    - Need a lot of locks, but high concurrency
  - Can we get the benefits of both?

Hierarchical Locking



- Shared and exclusive locks can be held at different levels
  - Locking a higher level implicitly locks lower-level objects
- New “intention” lock modes
  - Intention to lock a lower-level object
  - Intention Shared (IS), Intention Exclusive (IX)
  - Shared Intention Exclusion (SIX)

Hierarchical Locking Rules

Begin from the root and take:

- Appropriate intention locks in higher levels
  - “I will lock something this element contains”
  - E.g., taking an IS lock on the table to read a record.
- Shared or Exclusive lock at the target object’s level
  - E.g., taking an S lock on a database tuple

Steps:

1. If at the object we want to lock, take the S or X lock
2. If still at a higher level, take IS or IX lock and proceed to the next level

Locks are taken according to a compatibility matrix

Lock Compatibility

Element already locked by **another** transaction in
this mode, can I lock it in this mode on the same element?

	S	X	IS	IX
S	Yes	No	Yes	No
X	No	No	No	No
IS	Yes	No	Yes	Yes
IX	No	No	Yes	Yes

Note: S/X locking a higher level implicitly locks lower-level objects

Lock Compatibility

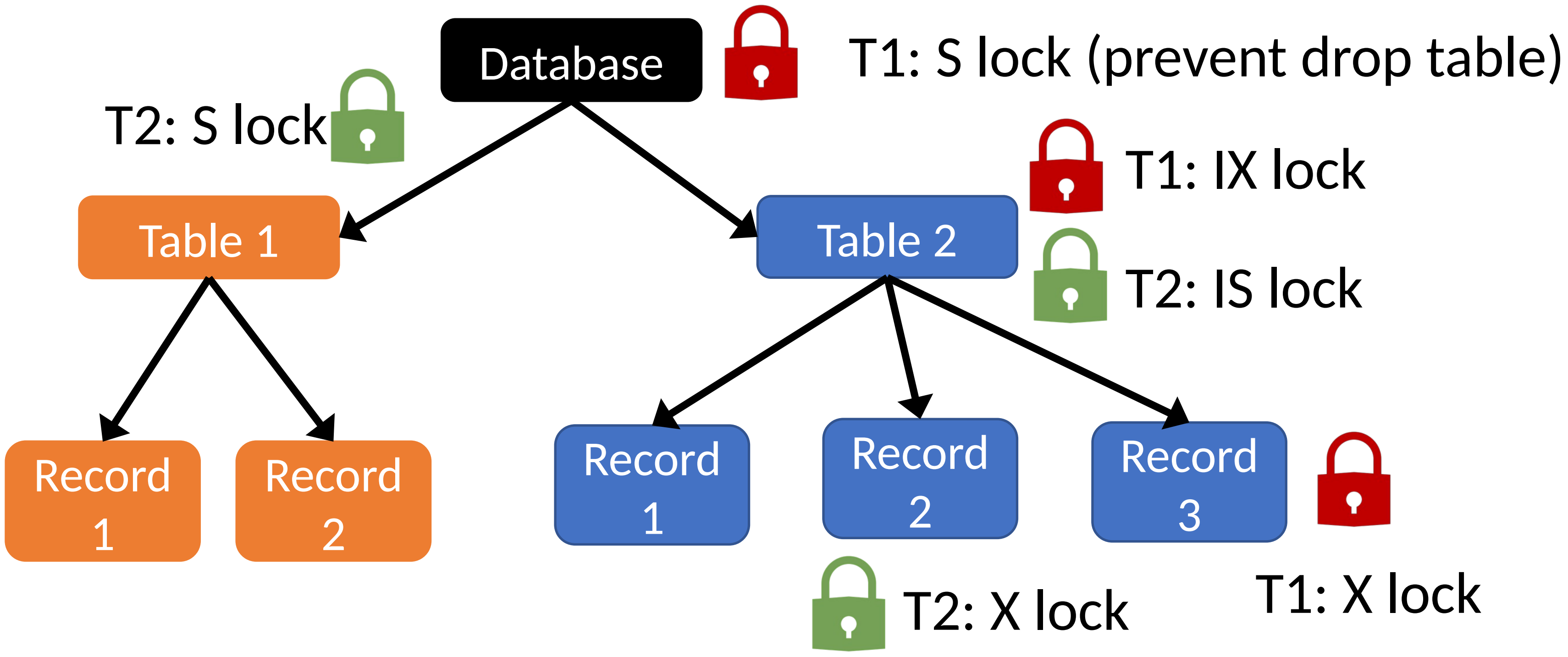
- S + IX locks: Read entire element + modify a sub-element
  - A transaction can take both S and IX locks on an element
  - But not compatible for two different transactions

	S	X	IS	IX	SIX
S	Yes	No	Yes	No	No
X	No	No	No	No	No
IS	Yes	No	Yes	Yes	Yes
IX	No	No	Yes	Yes	No
SIX	No	No	Yes	No	No

Hierarchical Locking Example

Transaction T1: lock Record 3 in Table 2 for modification

Transaction T2: lock Record 2 in Table 2 for reading



Phantoms

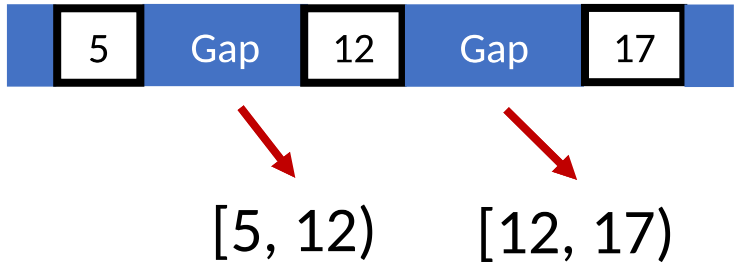
Phantom: **newly added rows by other transactions seen by a repeated range scan with the same predicate**

Time	T1	T2
	BEGIN	
	SCAN key > 1…	
	Result: {2, 4}	BEGIN
		INSERT key = 3…
	... Other ops ...	COMMIT
	SCAN key > 1…	
	Result: {2, **3**, 4}	3: phantom read
	COMMIT	

- "True" serializability requires repeatable read + no phantom
- → Need phantom protection mechanisms

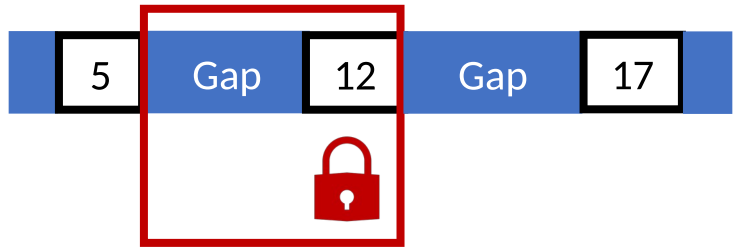
Phantom Protection

Prevent insertion in gaps between keys in leaf nodes

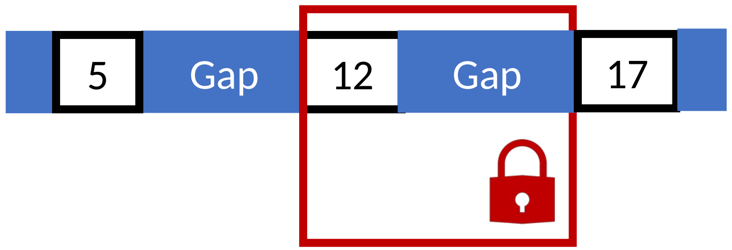


Key-range locking: key and adjacent gap locked as a unit

Next-key locking: lock the key and the gap before it

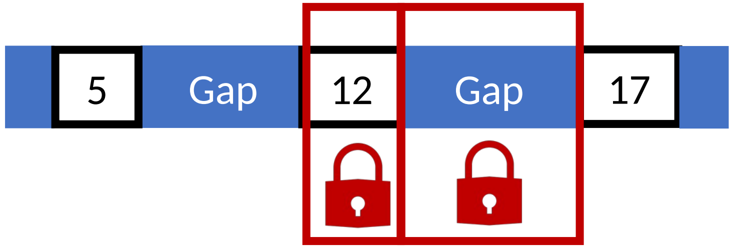


Prior-key locking: lock the key and the gap after it



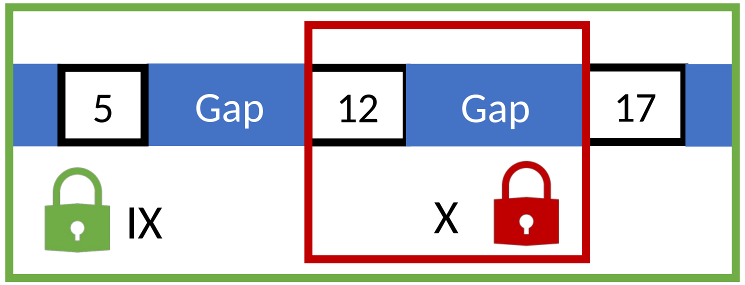
Phantom Protection

Gap locks: lock key and the gap that follows separately



- Different locking modes allowed
  - E.g., key read + gap read (allow shared reads), key read + gap write (allow exclusive insert)

Hierarchical locking: can be combined with key-range/gap locks to acquire locks on multiple ranges more easily



Latches vs. Locks

- Locks protect database elements
  - Tables, records
  - Used by transactions (“logical” level) to provide isolation between transactions
  - Users can query, see their status
- Latches protect “physical” states
  - Used in the implementation of database engines to protect data structures
    - E.g., protect a buffer pool frame; protect internal transaction metadata that is only visible inside the database engine
  - Completely invisible to users
- OS/synchronization people call latch “lock”
  - Mutex, spinlock, MCS lock…

Locks vs. Latches

	Locks	*Latches*
Separate …	User transactions	Threads
Protect …	Database contents	In-memory data structures
During …	Entire transactions	Critical sections
Modes …	Shared, exclusive, update, intention, escrow, schema, etc.	Read, writes, (perhaps) update
Deadlock …	Detection & resolution	Avoidance
… by …	Analysis of the waits-for graph, timeout, transaction abort, partial rollback, lock de-escalation	Coding discipline, "lock leveling"
Kept in …	Lock manager's hash table	Protected data structure

* Goetz Graefe, A survey of B-tree locking techniques, *ACM TODS*, 2010.

Managing Locks

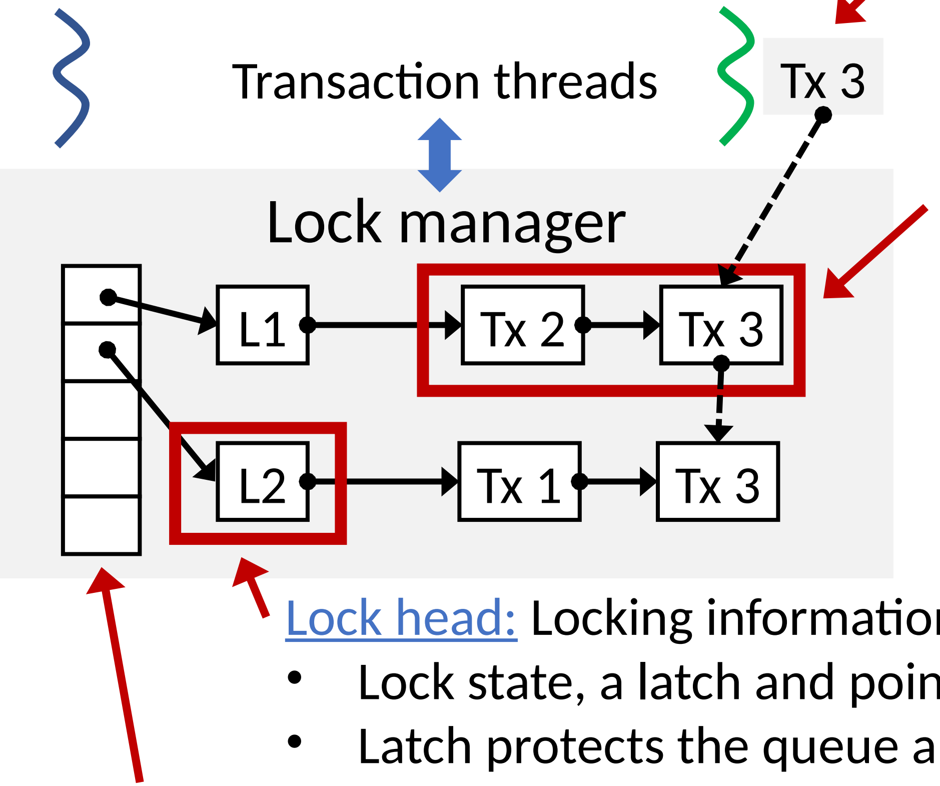
Lock manager: a **centralized** DBMS component that

- Provides an interface for transactions to
  - Acquire locks, release locks, upgrade/downgrade locks
- Manages lock states
- Handles deadlocks

Q: Why do we need a lock manager at all? What about co-locating locks with records?

A: To improve performance, locks need to be memory-resident. In disk-based systems records could be evicted and accessing locks on evicted records will incur many (slow) random accesses.

Lock Manager



Transaction lock head:

- Chain all the transaction's lock requests

Lock request queue:

- Queue of transactions that want to acquire the lock
- Must be able to handle concurrent accesses

Lock head: Locking information about the element

- Lock state, a latch and pointer to lock request queue
- Latch protects the queue and lock head itself

Lock table:

- Map database element to a **lock head**
- Typically implemented using a hash table
- Must be able to handle concurrent accesses

Handling Lock Request

Acquiring a lock:

- If the lock is free (i.e., request queue is empty):
  - Grant the lock by:
    - Incrementing the number of transactions by 1
    - Setting the lock mode (e.g., shared or exclusive)
- If the lock is currently being held in mode M:
  - For fair scheduling (FIFO), also need to check whether there is any conflicting requests among predecessors
  - Grant the lock only if all the three conditions hold:
    - M is in shared mode, and
    - All predecessors (if any) are granted in shared mode, and
    - The requesting mode is shared mode
  - Otherwise transaction waits for the lock to be granted
    - Busy spin or sleep on a condition variable

Lock compatibility matrix

	S	X
S	Yes	No
X	No	No

Handling Lock Request

Releasing a lock:

- Update the lock head
  - Decrement number of transactions holding the lock
- Examine the first requester in the lock queue
  - Grant the lock if possible
    - Wake up the waiting transaction(s)
    - Increment the number of lock-holding transactions
  - If there are multiple readers in the front of the queue, grant S-lock to them all

Lock Request Handling

T1 T2 T3

Time

BEGIN

BEGIN

BEGIN

Lined up after
T2, waiting

Read A

S-lock acquired

Read B

S-lock acquired

Write B

Lined up after
T1, waiting

Commit?

Lined up after
T3, waiting

Read B

Write A

Commit?

<u>Deadlock:</u>

- T3 waits for T2
- T2 waits for T1
- T1 waits for T3 (**FIFO lock request queue**)

Avoid starvation

Write C

Commit?

Protecting Lock Manager with Latches

- Typically mutex, spinlocks with proper blocking and spinning
  - E.g., std::mutex, MCS lock, or pthread mutex
- The lock table itself needs to support concurrency
- Per-queue latch for each lock queue
  - Multiple transactions may line up in a queue to request the lock
  - Deadlock detection/resolution traverses lock queues

Summary

- Locking basics
  - Lock modes, latch vs. locks
- 2PL for serializability
  - Grow and shrink phases: new lock not allowed once started to release lock
  - Cascading aborts possible with 2PL
  - SS2PL avoids it by keeping all locks till the end of transaction
- Deadlocks may happen
  - Because DBMS usually do not know about the workload
  - Avoidance and resolution methods
- Hierarchical locking reduces overhead of row-level locking
- Phantom protection
  - Phantom: the same scan operation returns different results (insert happened between the two scans)
  - Leverage index structure to lock gaps and keys
- Lock manager design and implementation