THE UNIVERSITY of EDINBURGH

Advanced Databases

Spring 2020

Lecture #13:

Two-Phase Locking

Milos Nikolic

Query Scheduler

How to guarantee only serializable schedules in DBMS?

Problem: user does not need to specify the full transaction at once

Goal: build a query scheduler that always emits serializable schedules

Pessimistic (locking)

Use locks to protect database objects

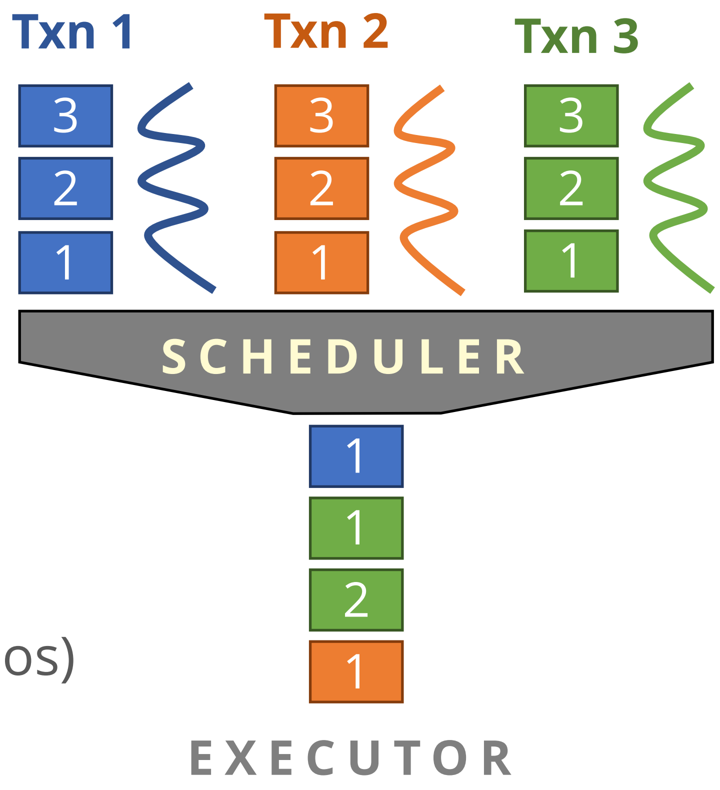
Standard approach if conflicts are frequent

Optimistic (versioning)

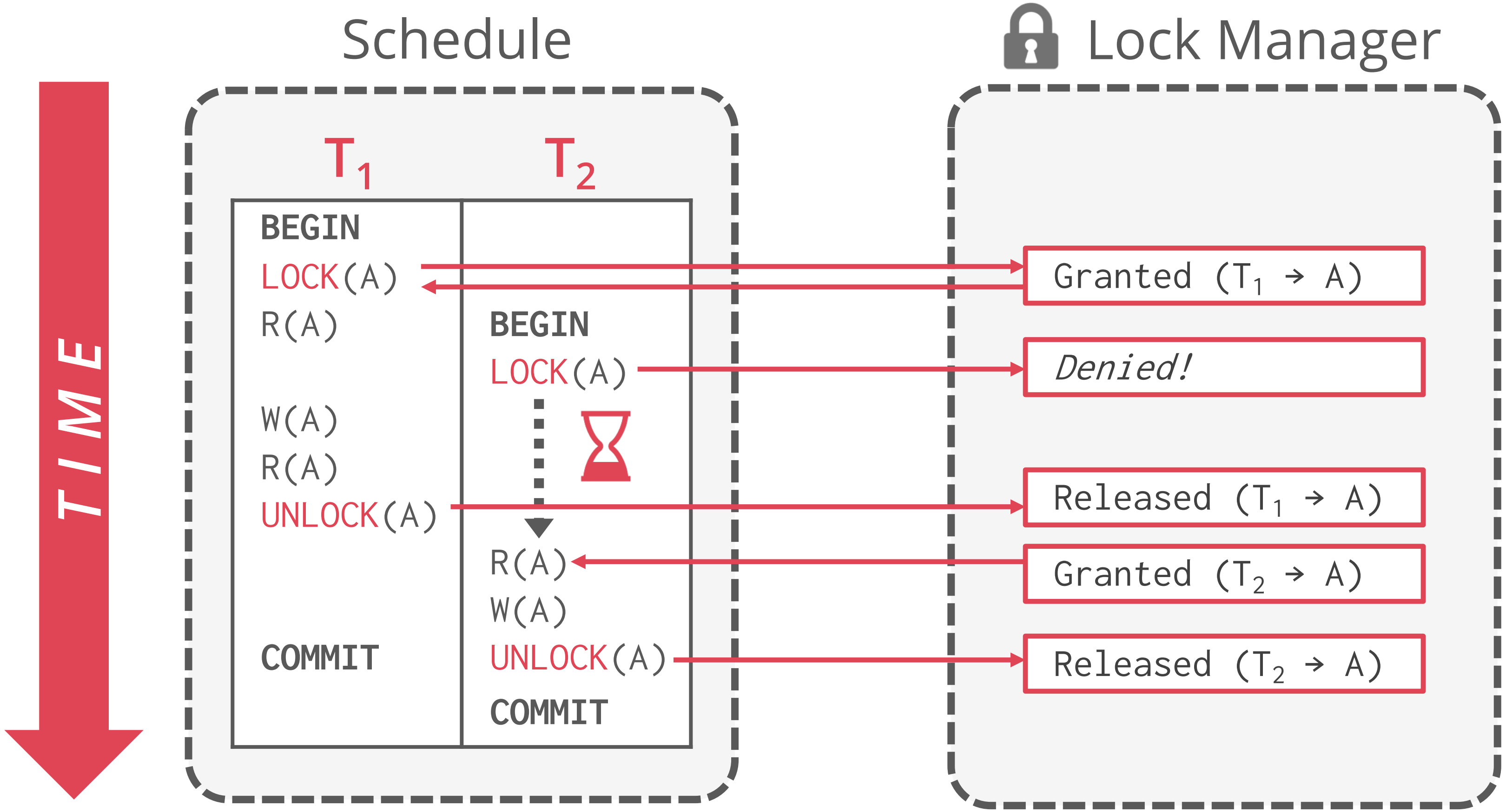
Record changes for each txn individually

Validate and possibly rollback on commit

Used if conflicts are rare (e.g., write-once-read-many scenarios)



Executing with Locks



Executing with Locks

Basic lock types:

- **S-LOCK**: Shared locks for reads
- **X-LOCK**: Exclusive locks for writes

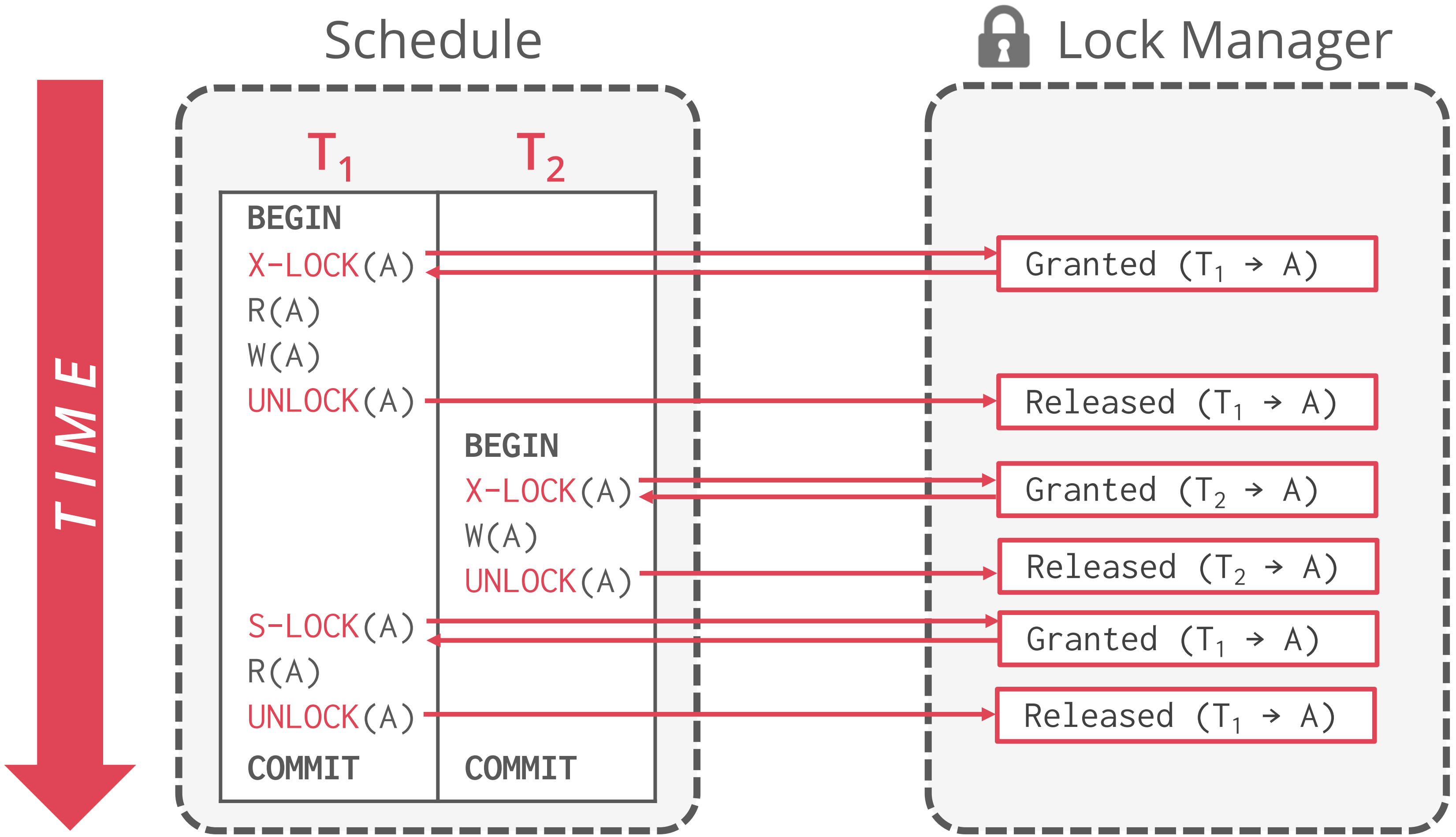
Compatibility Matrix

	Shared	Exclusive
Shared	✓	✗
Exclusive	✗	✗

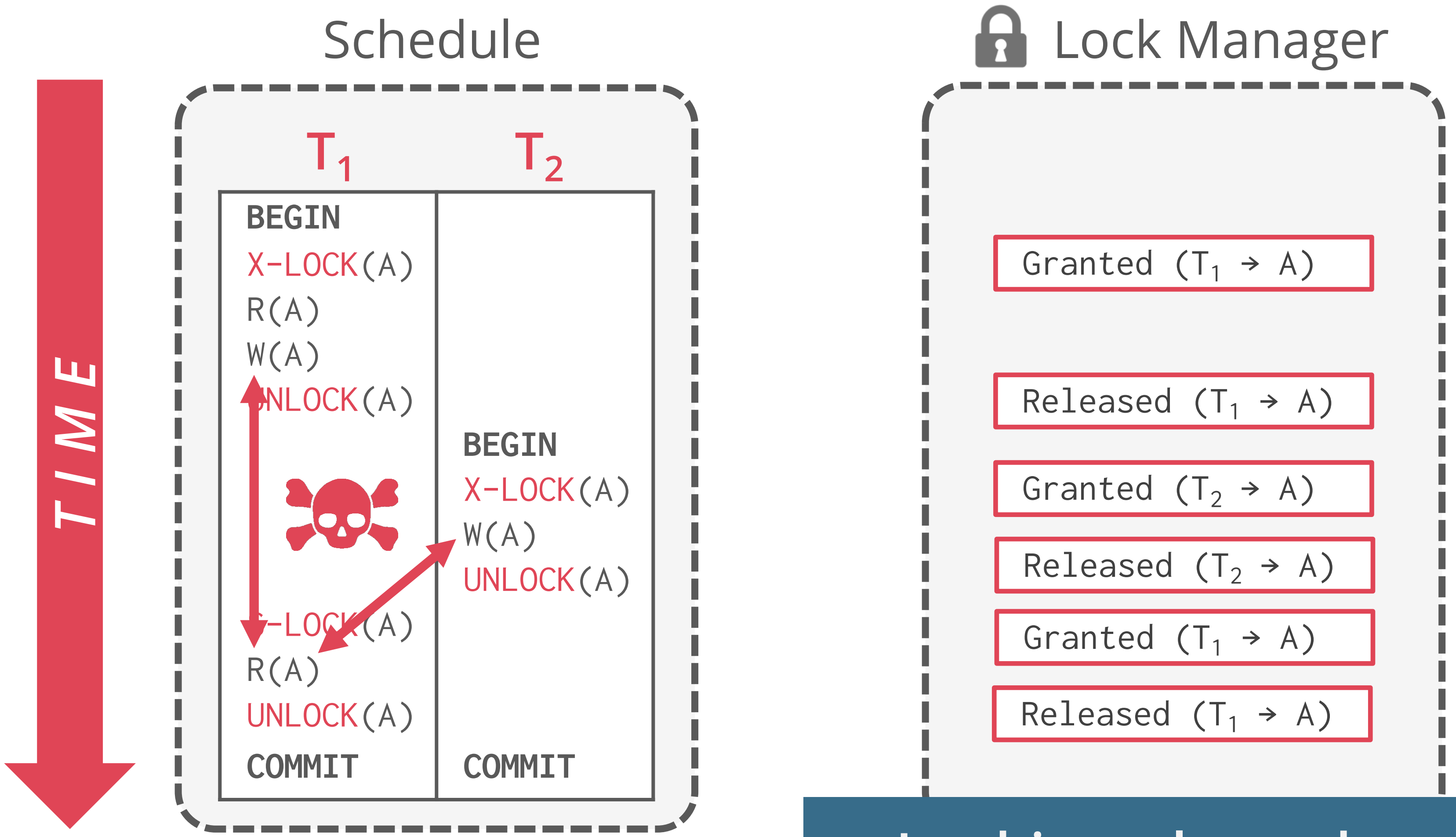
Steps:

- Transactions request locks (or upgrades) before accessing objects
- Lock manager grants or blocks requests
- Transactions release locks
- Lock manager updates its internal lock-table

Executing with Locks



EXECUTING WITH LOCKS



Not serializable

Locking alone does not enforce serializable schedules

Two-Phase Locking

Locks + concurrency control protocol

- Determines if a txn is allowed to access an object in the database on the fly
- Does not need to know all of the queries that a txn will execute ahead of time

Phase 1: Growing

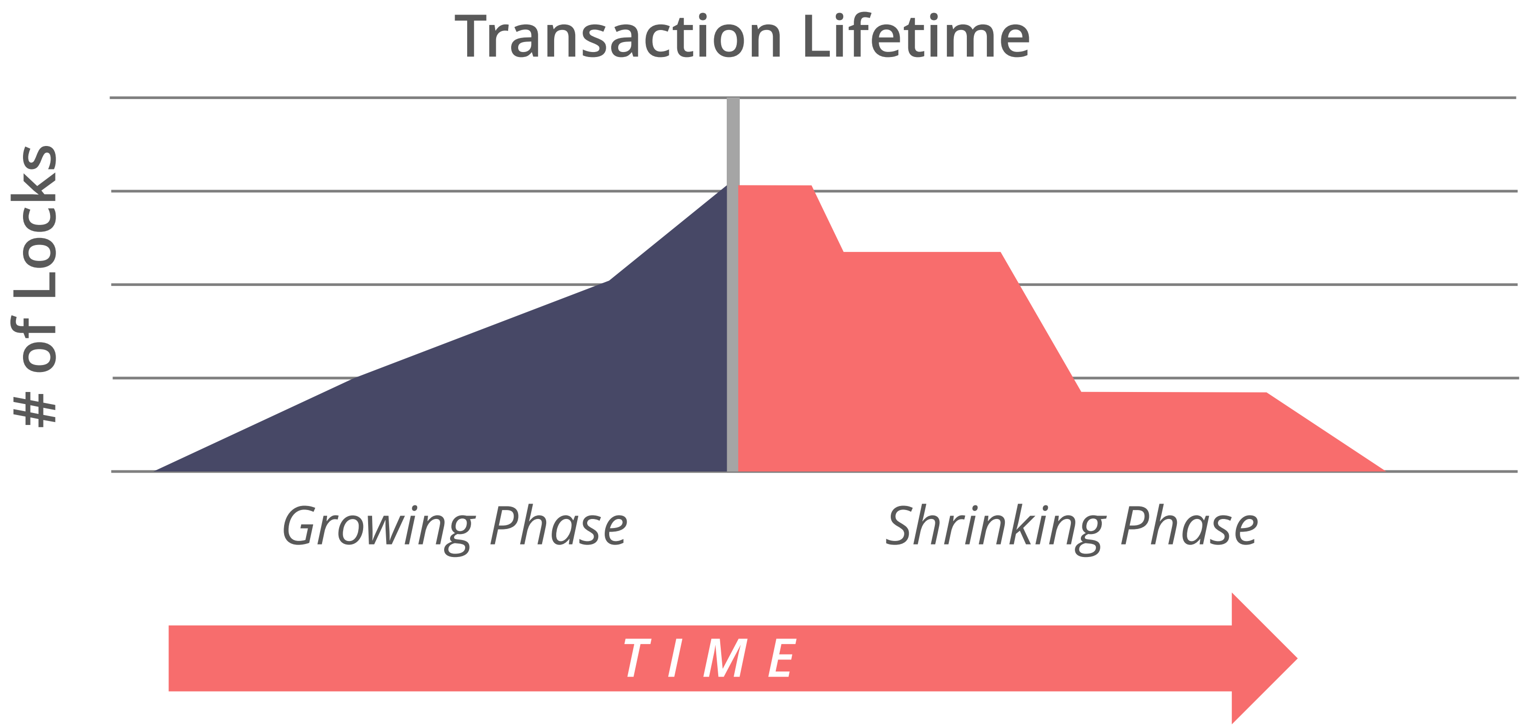
- Each txn requests the locks that it needs from the lock manager
- The lock manager grants/denies lock requests

Phase 2: Shrinking

- The txn is allowed to only release locks that it previously acquired
- It cannot acquire new locks

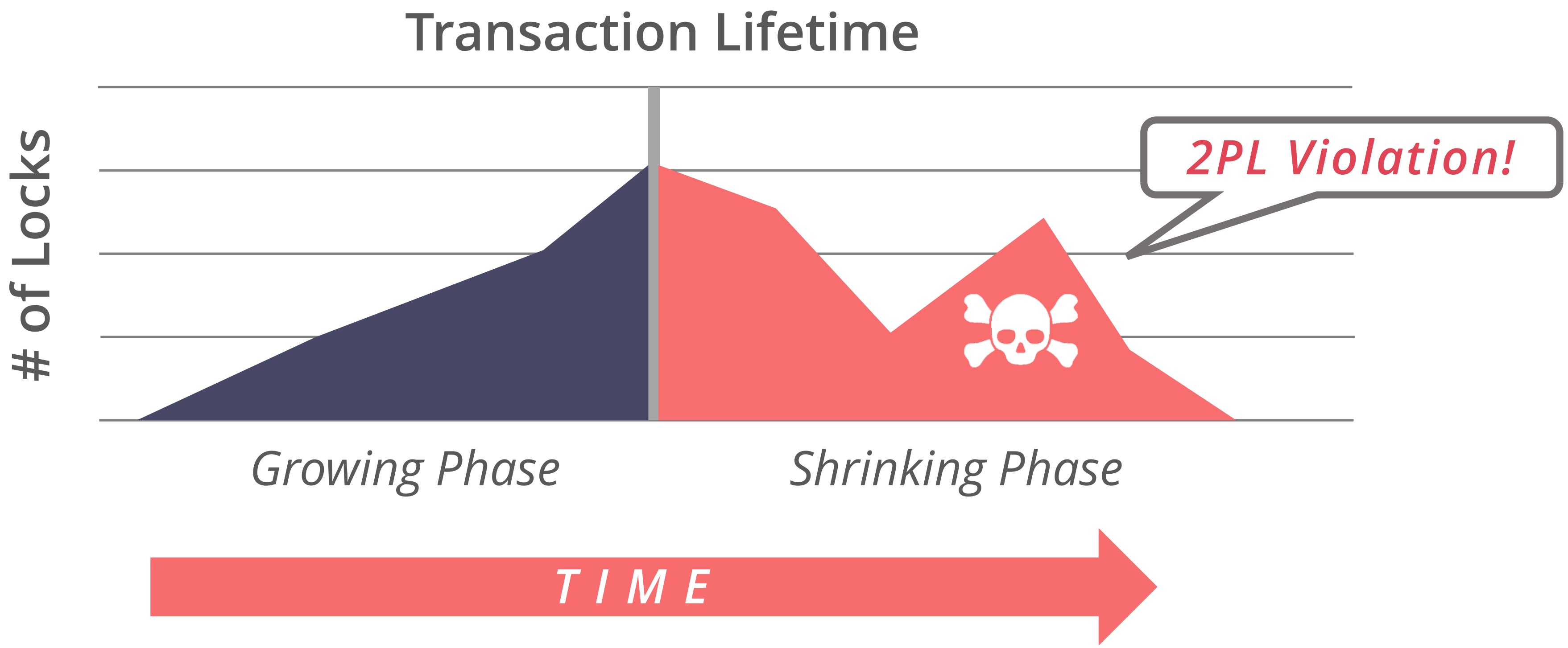
Two-Phase Locking

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

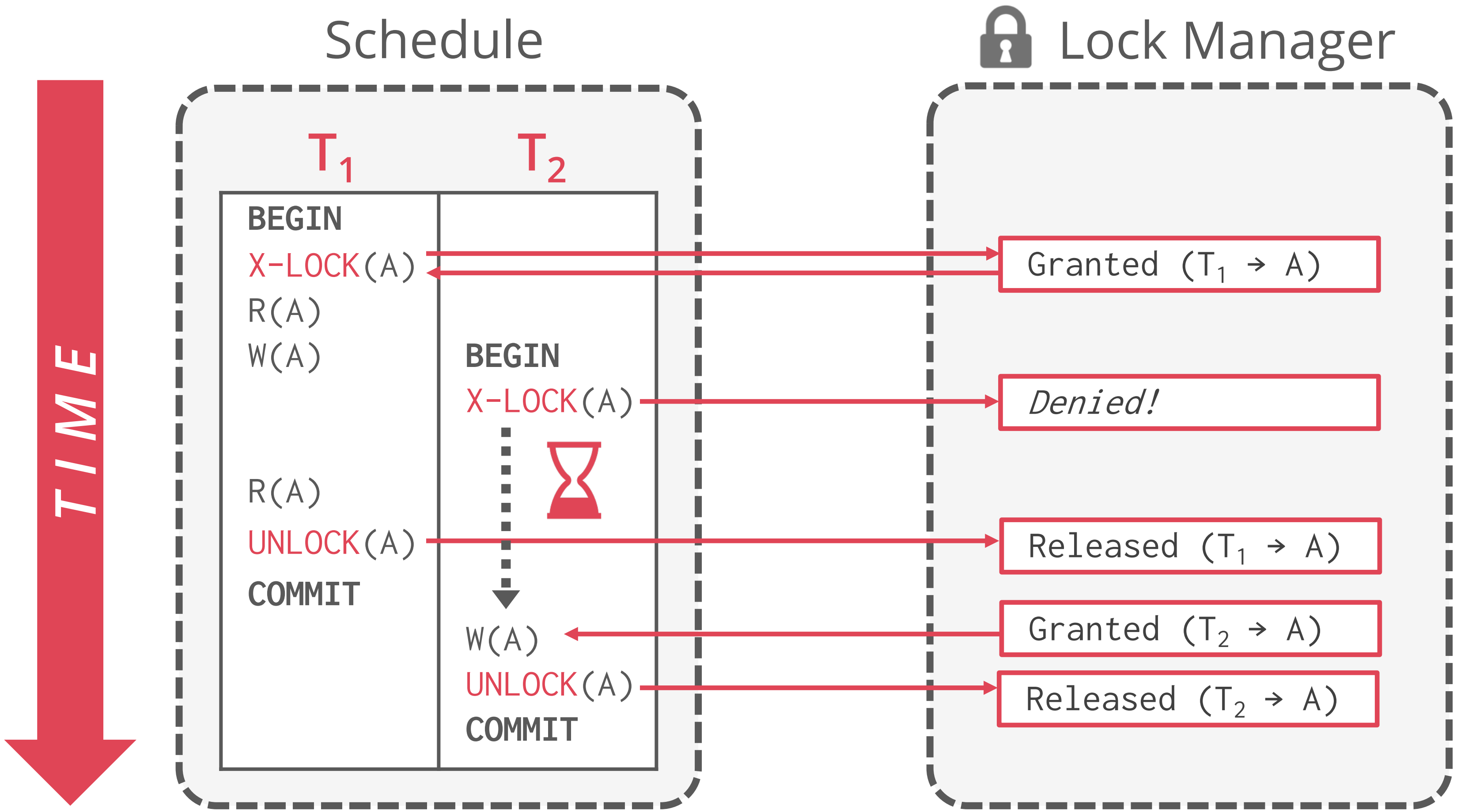


TWO-PHASE LOCKING

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



Executing with Locks

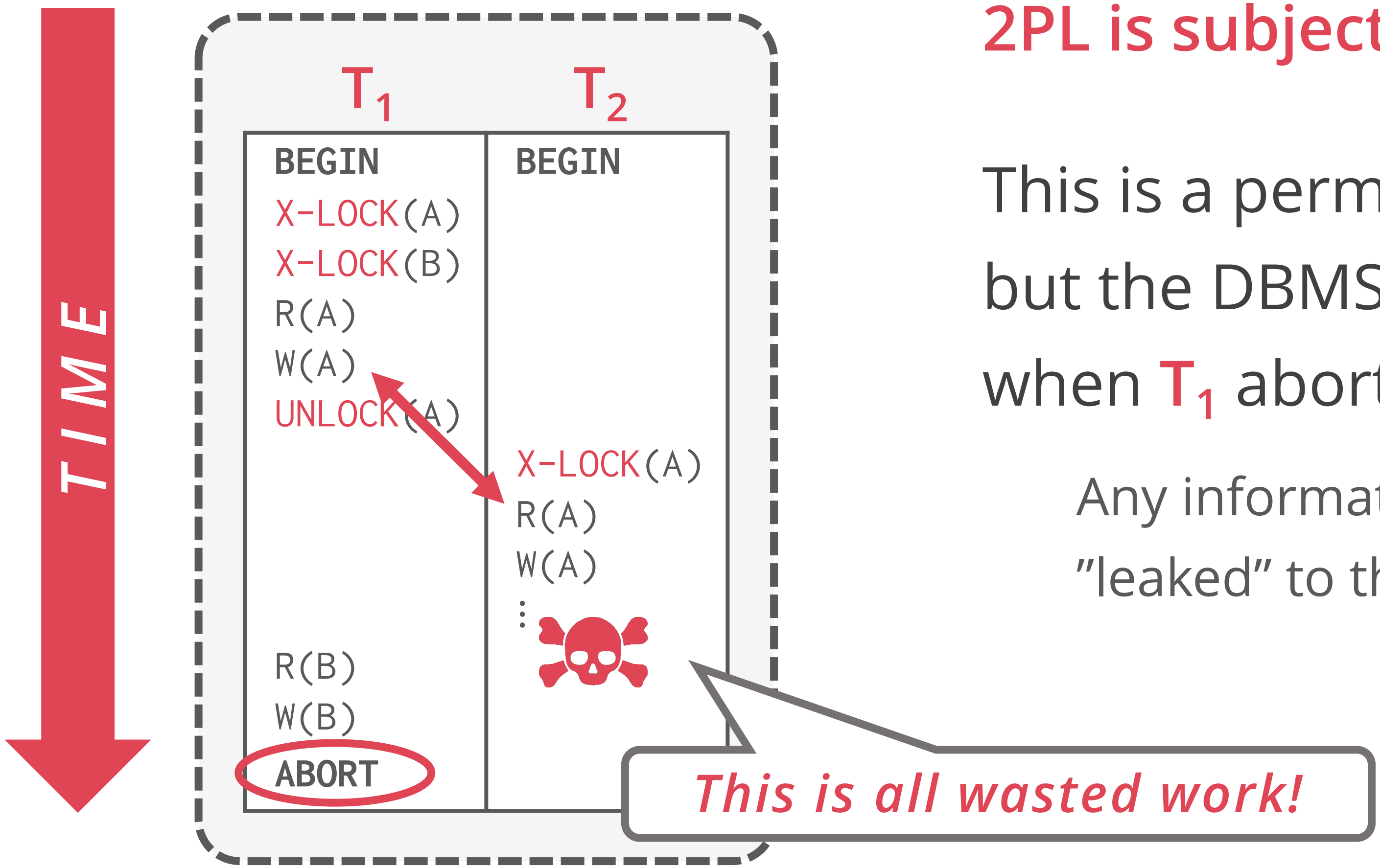


2PL is sufficient to guarantee conflict-serializability

(generates schedules whose precedence graph is acyclic)

2PL – Cascading Aborts

Schedule



2PL is subject to 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

2PL Observations

There are schedules that are serializable but not be allowed by 2PL

- Locking limits concurrency

May require **cascading aborts**

- **Solution**: Strict 2PL

May still have **”dirty reads”**

- **Solution**: Strict 2PL

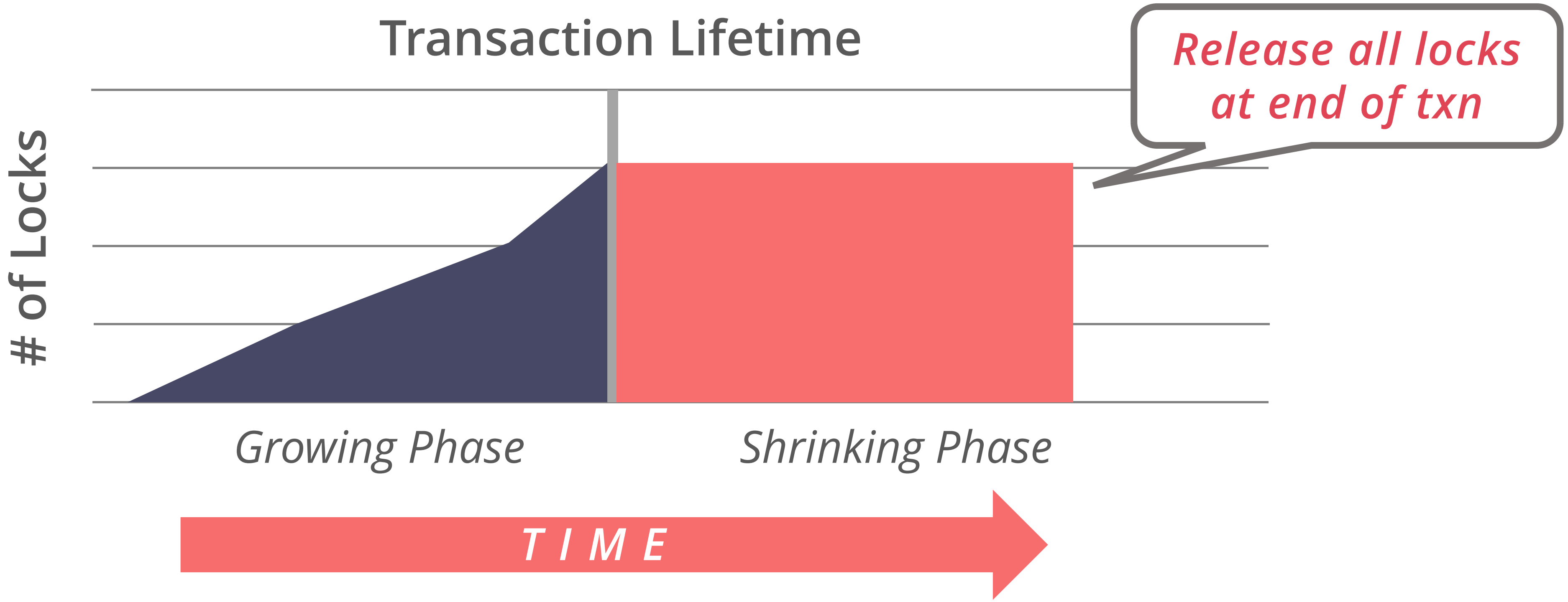
May lead to **deadlocks**

- **Solution**: Detection or Prevention

Strict Two-Phase Locking

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

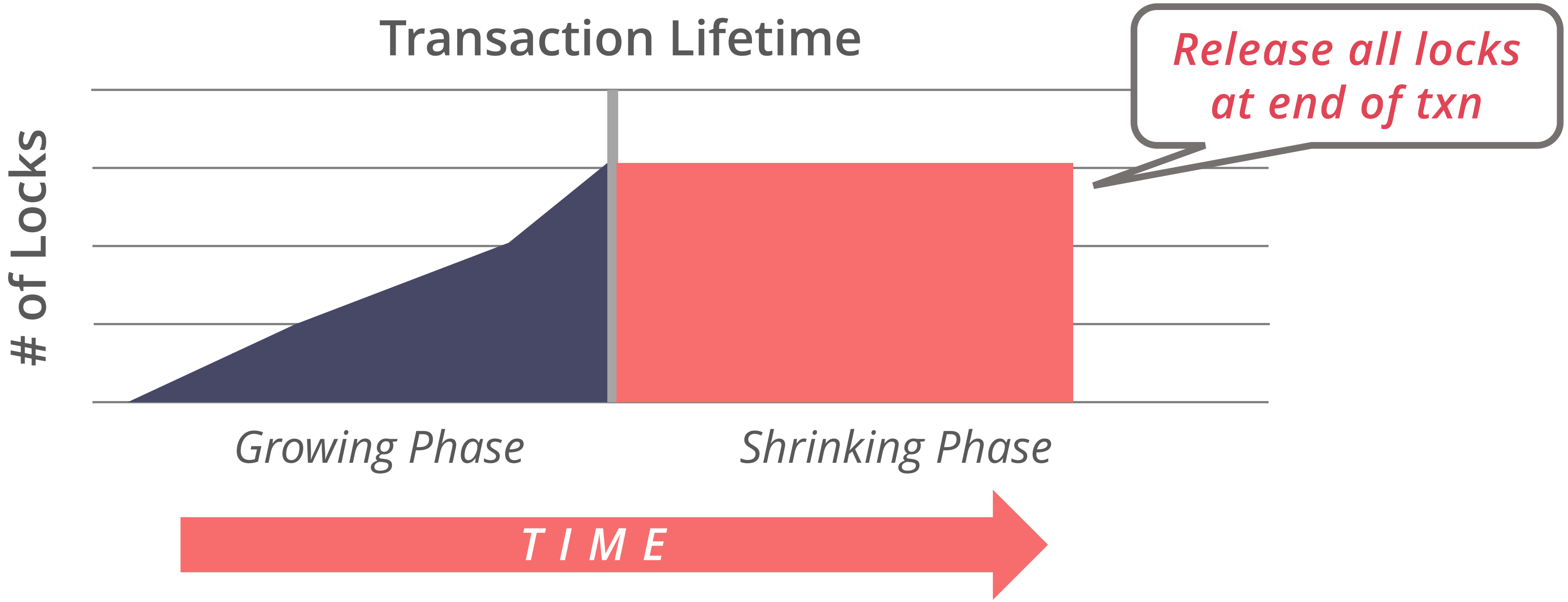
Allows only conflict-serializable schedules, but it is often stronger than needed for some applications



Strict Two-Phase Locking

Advantages:

- Does not incur cascading aborts
- Aborted txns can be undone by just restoring original values of modified tuples



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	

T_1 – move £100 from account A to account B

T_2 – compute the total amount in all accounts and return it to the application

Initial Database State

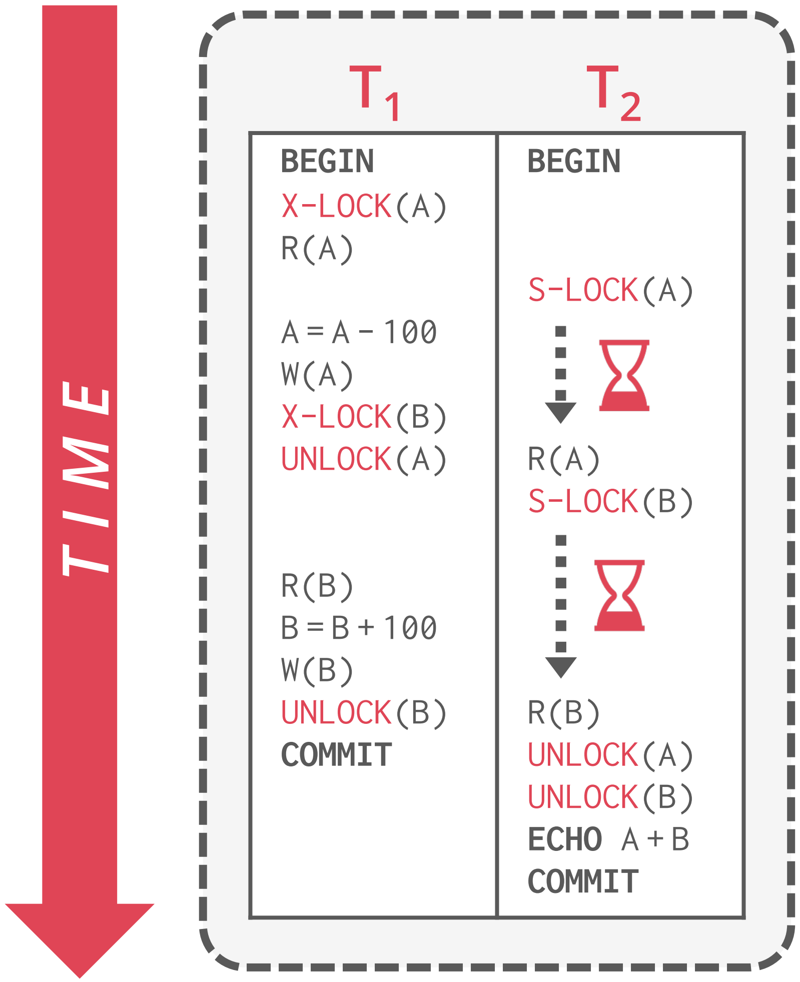
A = 1000, **B** = 1000

T_2 Output

A + **B** = 1900

2PL Example

Schedule



T_1 – move £100 from account A to account B

T_2 – compute the total amount in all accounts and return it to the application

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)	R(A)
UNLOCK(B)	S-LOCK(B)
COMMIT	R(B)
	ECHO A + B
	UNLOCK(A)
	UNLOCK(B)
	COMMIT

T_1 – move £100 from account A to account B

T_2 – compute the total amount in all accounts and return it to the application

Initial Database State

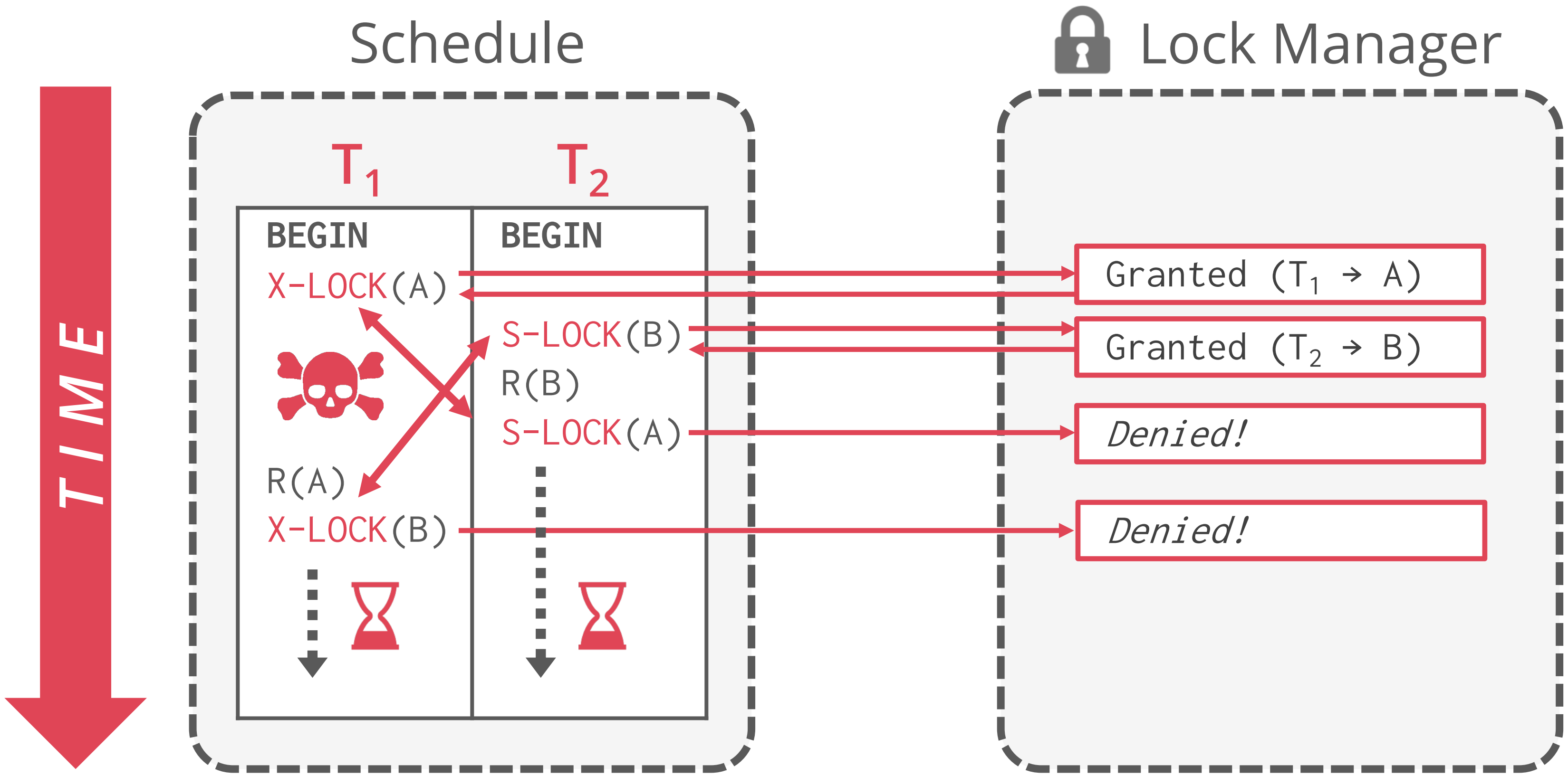
A = 1000, **B** = 1000

T_2 Output

A + **B** = 2000

SCHEDULING: DEADLOCKS

Two-phase locking has the risk of **deadlock situations**



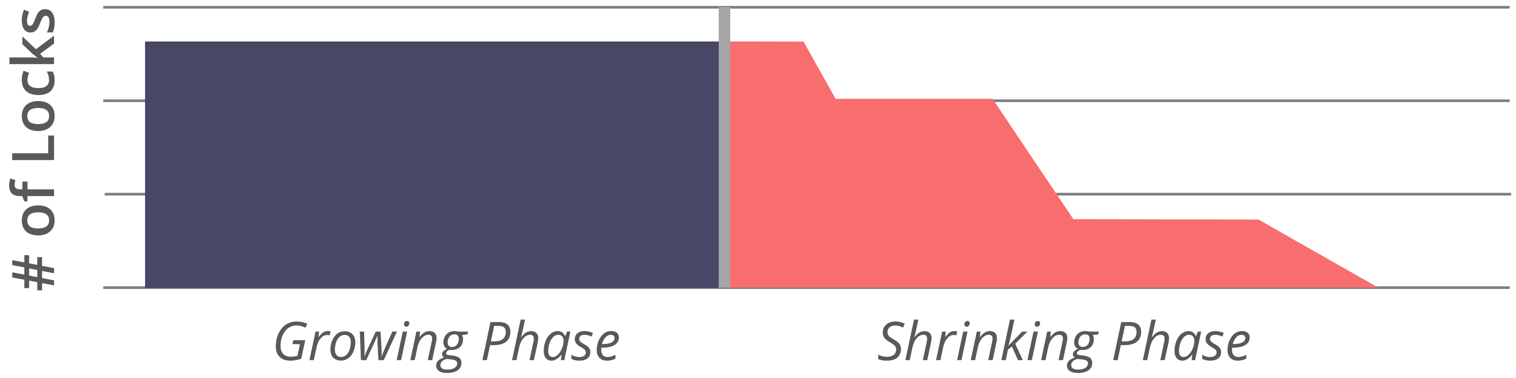
2PL DEADLOCKS

Deadlock = a cycle of txns waiting for locks to be released by each other

Two ways of dealing with deadlocks:

- **Deadlock Detection**
- **Deadlock Prevention**

Conservative (or "preclaiming") 2PL also prevents deadlocks. Why?



Deadlock Detection

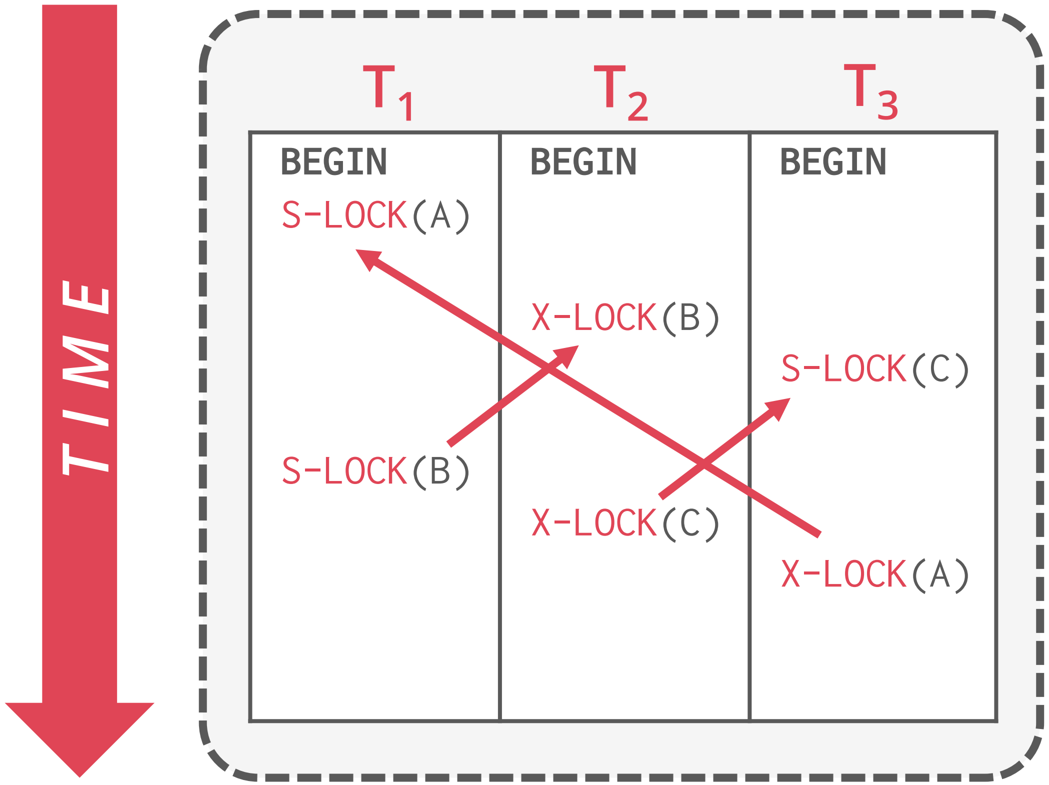
The DBMS creates a **waits-for** graph to keep track of what locks each transaction is waiting to acquire:

- Nodes are transactions
- Edge from T_i to T_j if T_i is waiting for T_j to release a lock

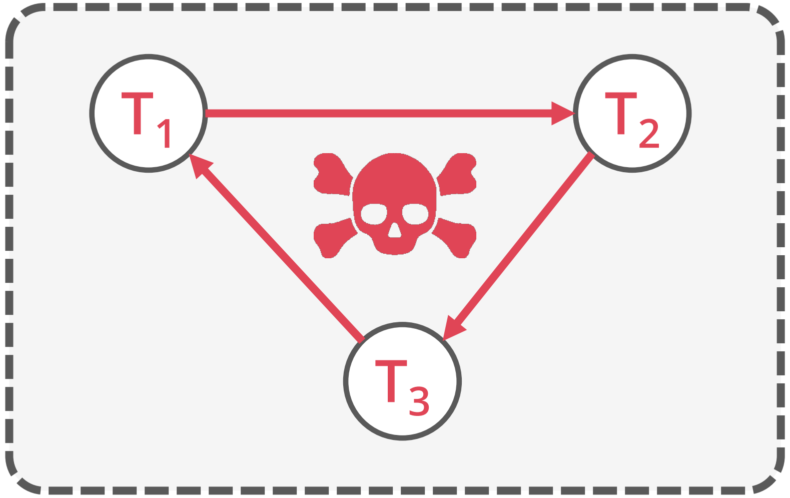
The system periodically checks for cycles in waits-for graph and then make a decision on how to break it

Deadlock Detection

Schedule



Waits-For Graph



Deadlock Handling

Upon detecting a deadlock, the DBMS selects a "victim" transaction to rollback to break the cycle

- Selecting a "victim" transaction might depend on:
  - age (lowest timestamp)
  - progress (least/most executed queries)
  - # of items already locked
  - # of txns that we have to rollback with it
  - # of previous restarts (to prevent starvation)

There is a trade-off between the frequency of checking for deadlocks and how long transactions have to wait before deadlocks are broken

Deadlock Prevention

When a transaction tries to acquire a lock that is held by another transaction, kill one of them to prevent a deadlock

No waits-for graph or detection algorithm

Assign **priorities** based on timestamps

- Older ⇒ higher priority (e.g., $T_1 > T_2$)

Two deadlock prevention policies:

- **Wait-Die** (“Old Waits for Young”)
- **Wound-Wait** (“Young Waits for Old”)

Deadlock Prevention

Wait-Die ("Old Waits for Young")

If *requesting* txn has higher priority than *holding* txn

Then *requesting* txn **waits** for *holding* txn

Else *requesting* txn **aborts**

Wound-Wait ("Young Waits for Old")

If *requesting* txn has higher priority than *holding* txn

Then *holding* txn **aborts** and releases locks

Else *requesting* txn **waits**

Deadlock Prevention

$T_{req} > T_{hold}$?

Wait

:

Die

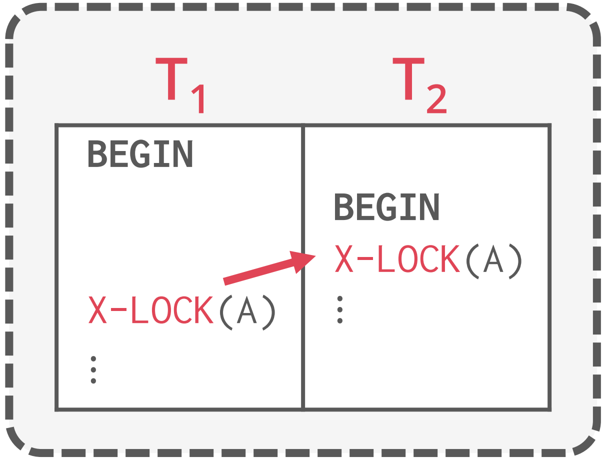
$T_{req} > T_{hold}$?

Wound

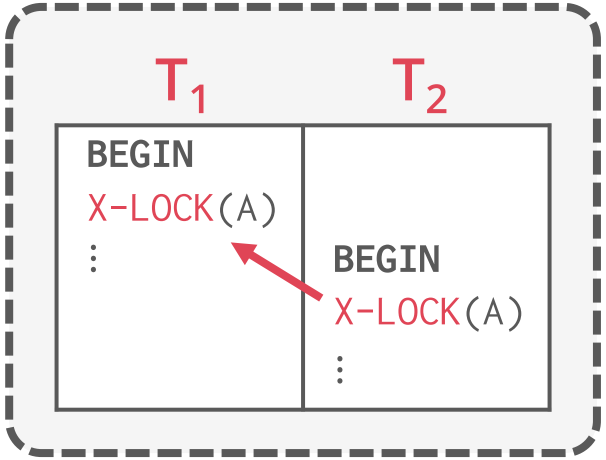
:

Wait

Deadlock Prevention



Wait-Die	Wound-Wait
T_1 waits	T_2 aborts



Wait-Die	Wound-Wait
T_2 aborts	T_2 waits

Deadlock Prevention

Why do these schemes guarantee no deadlocks?

Only one “type” of direction allowed when waiting for a lock

When a transaction restarts, what is its (new) priority?

Its original timestamp. Why?

Conclusion

ACID Transactions

Atomicity: All or nothing

Consistency: Only valid data

Isolation: No interference

Durability: Committed data persists

Serializability

Serializable schedules

Conflict & view serializability

Checking for conflict serializability

Concurrency Control

Prevent anomalous schedules

Locks + protocol (2PL, Strict 2PL) guarantees conflict serializability

Deadlock detection and deadlock prevention