Carnegie Mellon University

# Two-Phase Locking

Lecture #17

Database Systems
15-445/15-645
Fall 2017

**Andy Pavlo**
Computer Science Dept.
Carnegie Mellon Univ.

# UPCOMING DATABASE EVENTS

## QuasarDB Talk

→ Thursday Nov 2nd @ 12pm
→ CIC 4th Floor

## Peloton Hack-a-thon

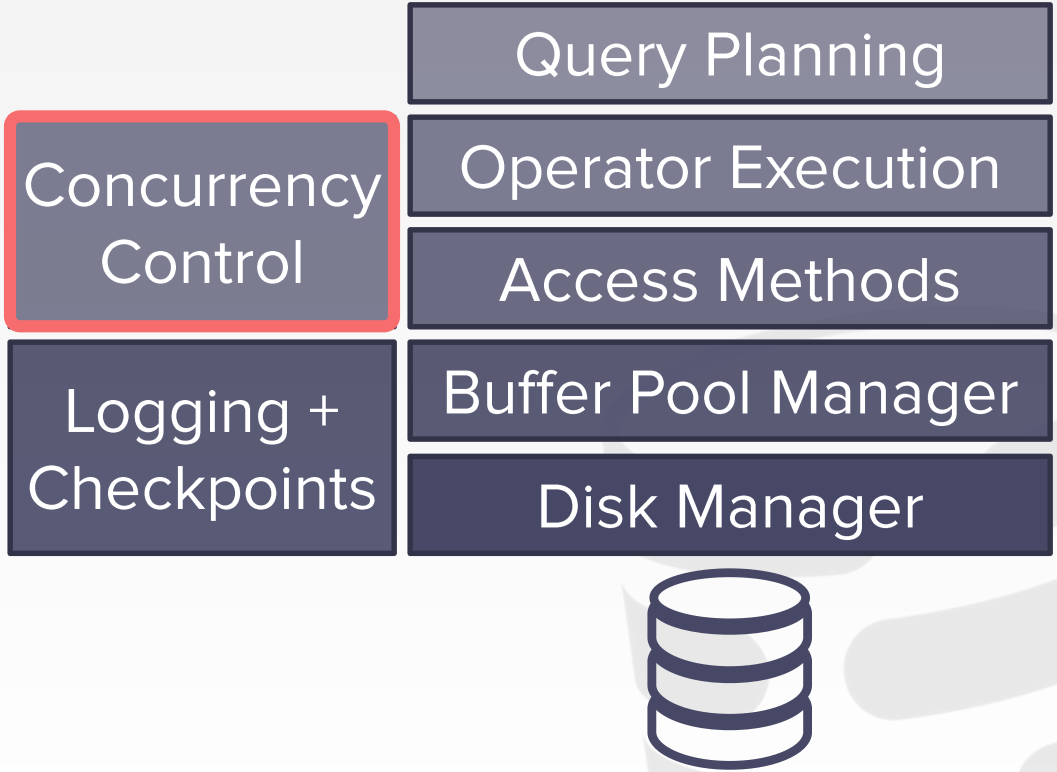
→ Friday Nov 10th @ 9:30am
→ GHC 8102

## TimescaleDB Talk

→ Thursday @ Nov 16th @ 12:00pm
→ CIC 4th Floor

# STATUS

A DBMS's concurrency control and recovery components permeate throughout the design of its entire architecture.



# LAST CLASS

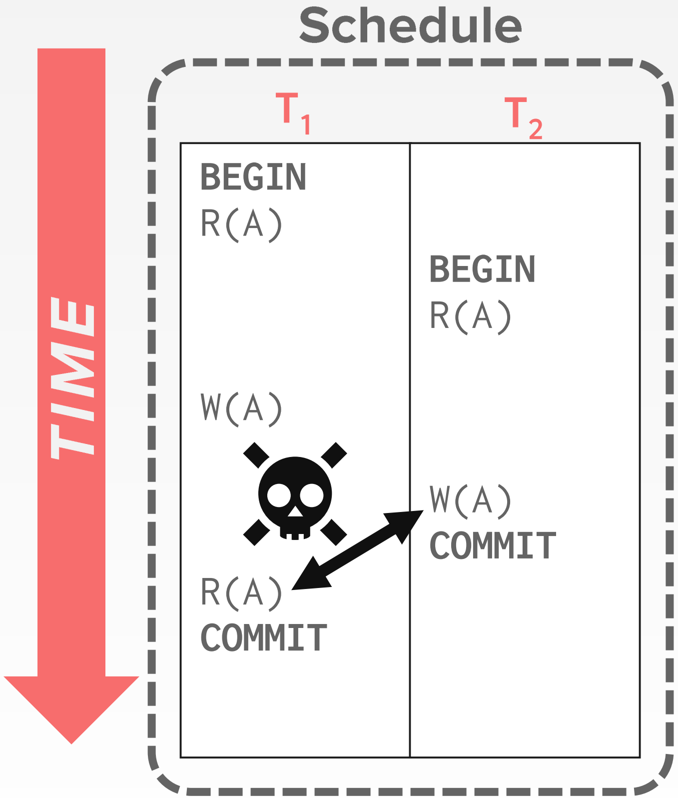
## Conflict Serializable

→ Verify using either the "swapping" method or dependency graphs.
→ Any DBMS that says that they support "serializable" isolation does this.

## View Serializable

→ No efficient way to verify.
→ Andy doesn't know of any DBMS that supports this.

# EXAMPLE

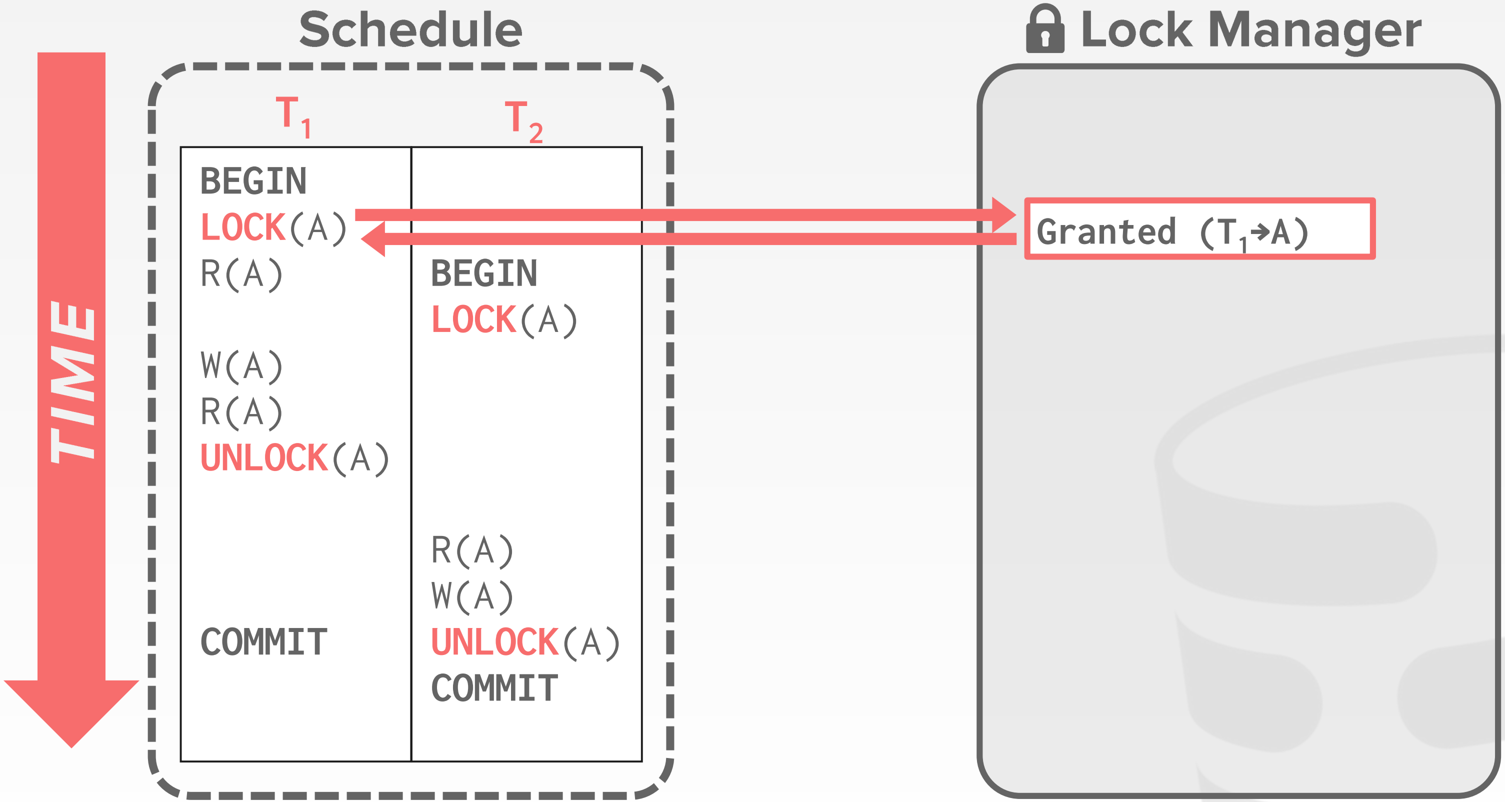


# OBSERVATION

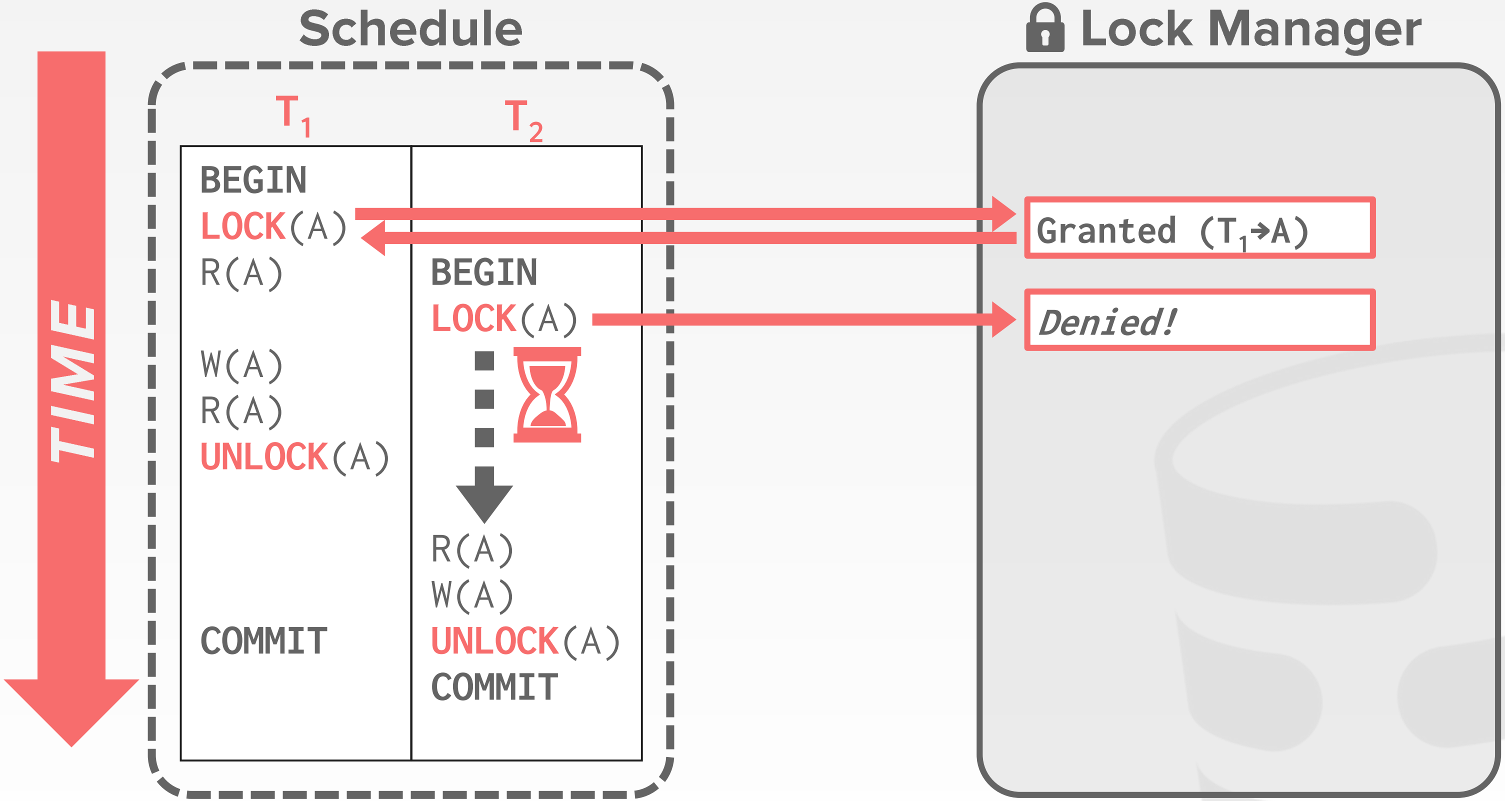
How could you guarantee that all resulting schedules are correct (i.e., serializable)?

Use **locks** to protect database objects.

# EXECUTING WITH LOCKS



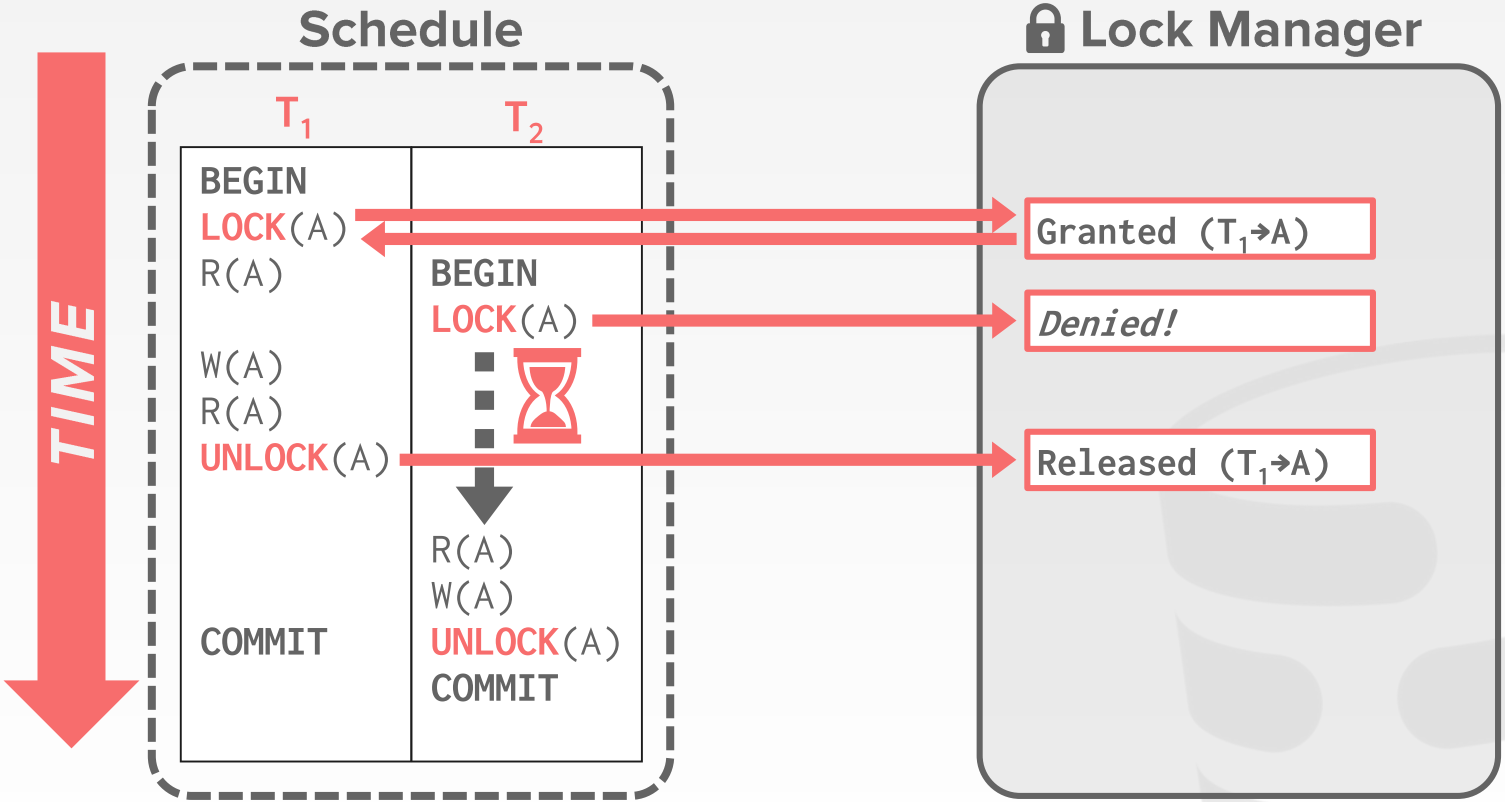
# EXECUTING WITH LOCKS



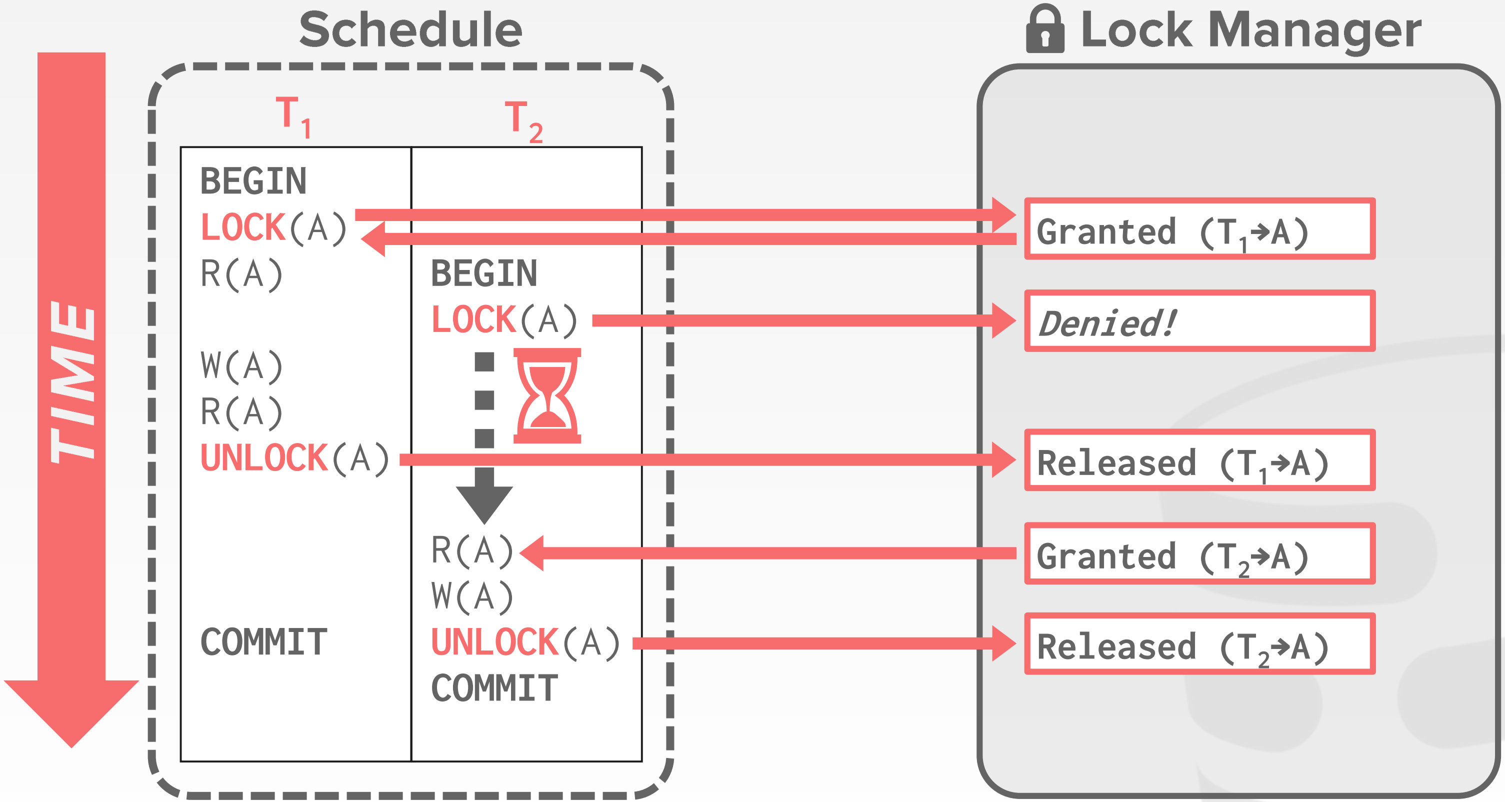


CARNEGIE MELLON
DATABASE GROUP

# EXECUTING WITH LOCKS



# EXECUTING WITH LOCKS





CARNEGIE MELLON DATABASE GROUP

# TODAY'S AGENDA

Lock Types

Two-Phase Locking

Deadlock Detection + Prevention

Hierarchical Locking

# LOCKS VS. LATCHES

## Locks

→ Protects the index's logical contents from other txns.
→ Held for txn duration.
→ Need to be able to rollback changes.

## Latches

→ Protects the critical sections of the index's internal data structure from other threads.
→ Held for operation duration.
→ Do not need to be able to rollback changes.

# 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 | Read, Write |
| **Deadlock** | Detection & Resolution | Avoidance |
| **…by…** | Waits-for, Timeout, Aborts | Coding Discipline |
| **Kept in…** | Lock Manager | Protected Data Structure |

Source: Goetz Graefe

# BASIC LOCK TYPES

S-LOCK: **Shared** Locks for reads.

X-LOCK: **Exclusive** Locks for writes.

**Compatibility Matrix**

| | Shared | Exclusive |
|---|---|---|
| Shared | ✓ | X |
| Exclusive | X | X |

# EXECUTING WITH LOCKS

Transactions request locks (or upgrades)

Lock manager grants or blocks requests

Transactions release locks

Lock manager updates its internal lock-table

# EXECUTING WITH LOCKS

## Schedule

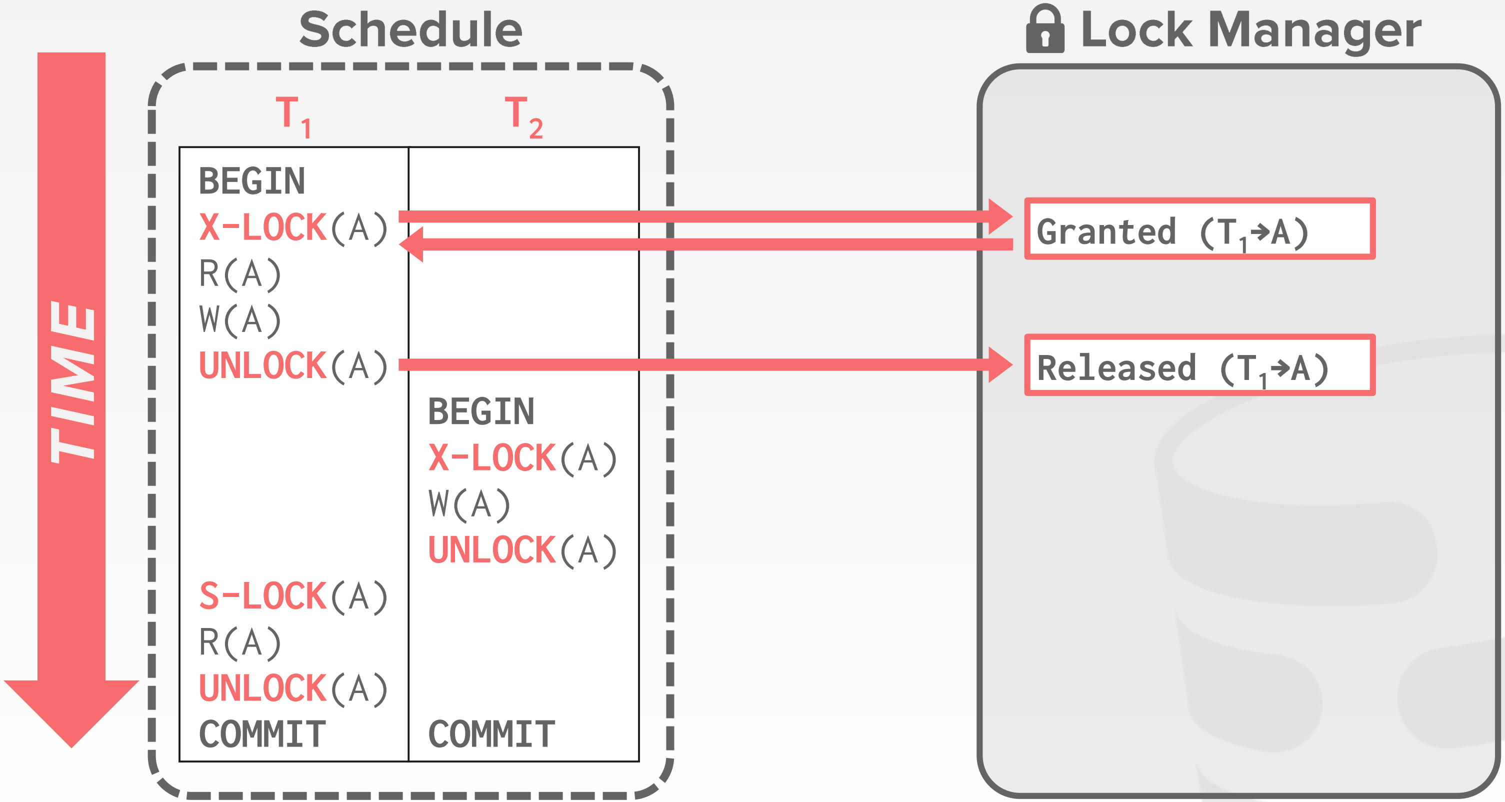
TIME

| $T_1$ | $T_2$ |
|---|---|
| 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 |

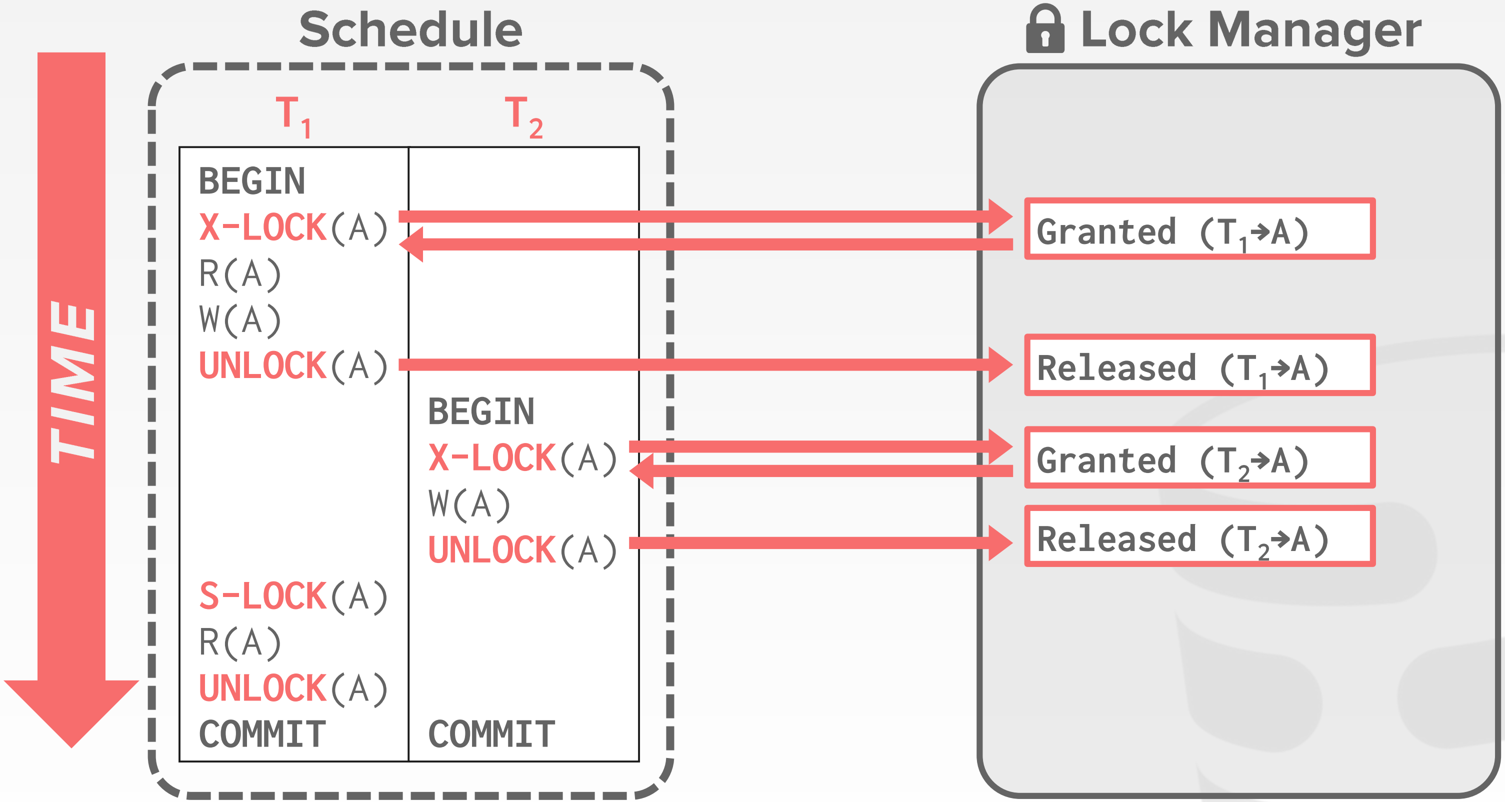
## Lock Manager

Granted ($T_1$→A)

# EXECUTING WITH LOCKS



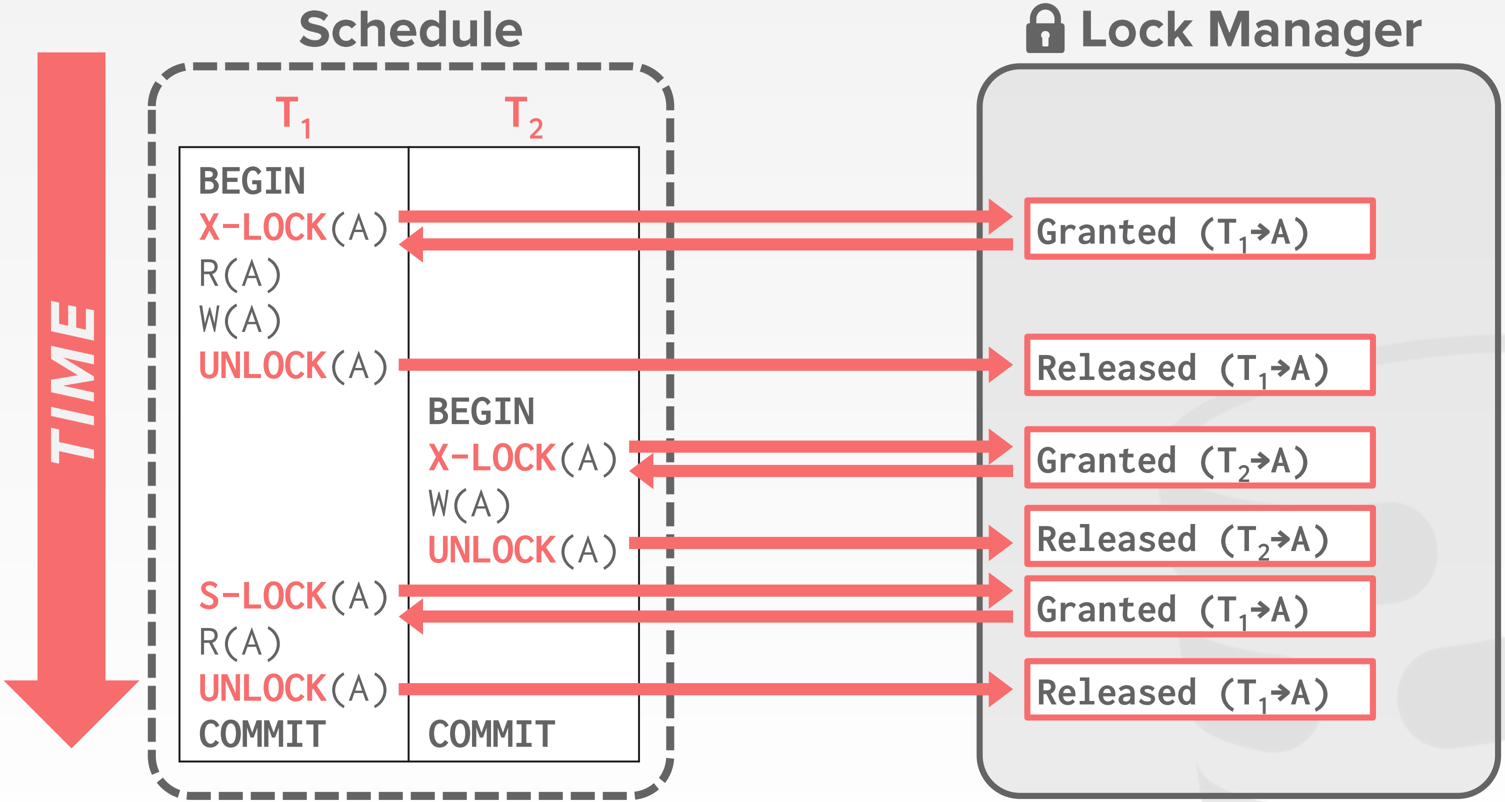
# EXECUTING WITH LOCKS



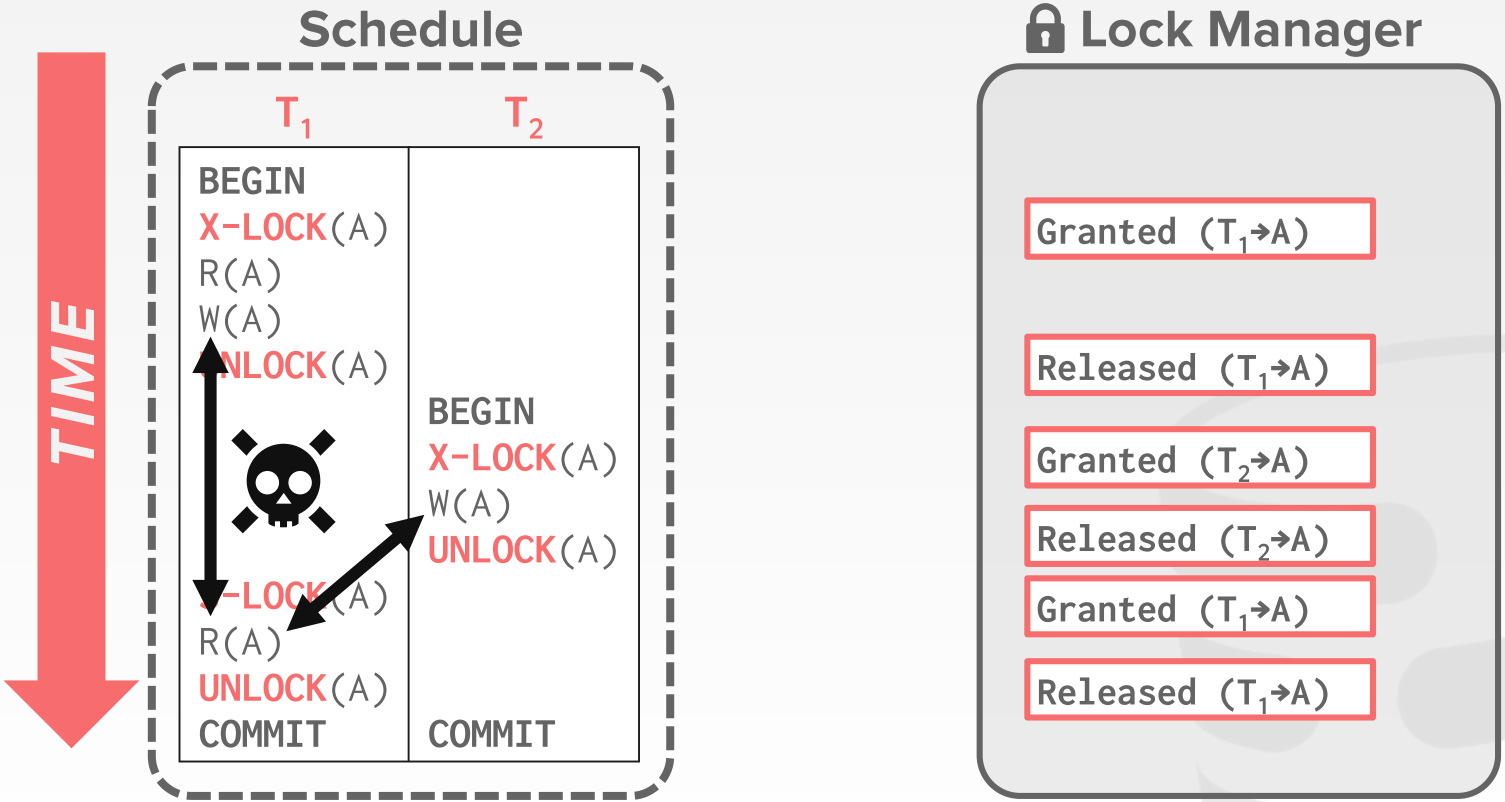



CARNEGIE MELLON
DATABASE GROUP

# EXECUTING WITH LOCKS



# EXECUTING WITH LOCKS



# CONCURRENCY CONTROL PROTOCOL

Two-phase locking (2PL) is a concurrency control protocol that determines whether a txn is allowed to access an object in the database on the fly.

The protocol does not need to know all of the queries that a txn will execute ahead of time.

# TWO-PHASE LOCKING

## Phase 1: Growing

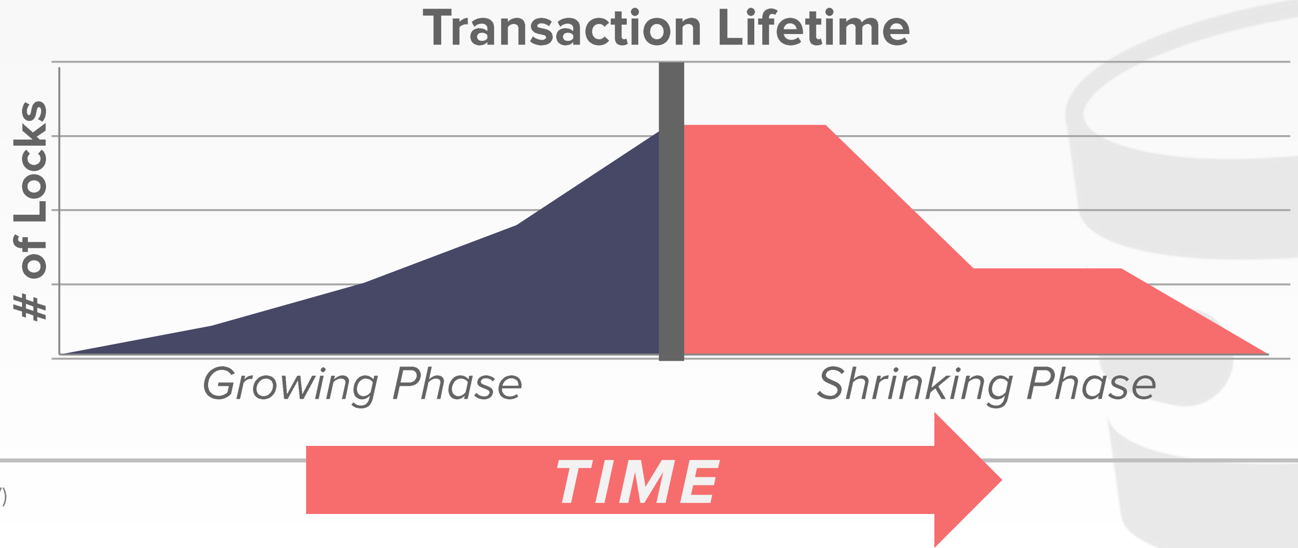
→ Each txn requests the locks that it needs from the DBMS's lock manager.
→ The lock manager grants/denies lock requests.

## Phase 2: Shrinking

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

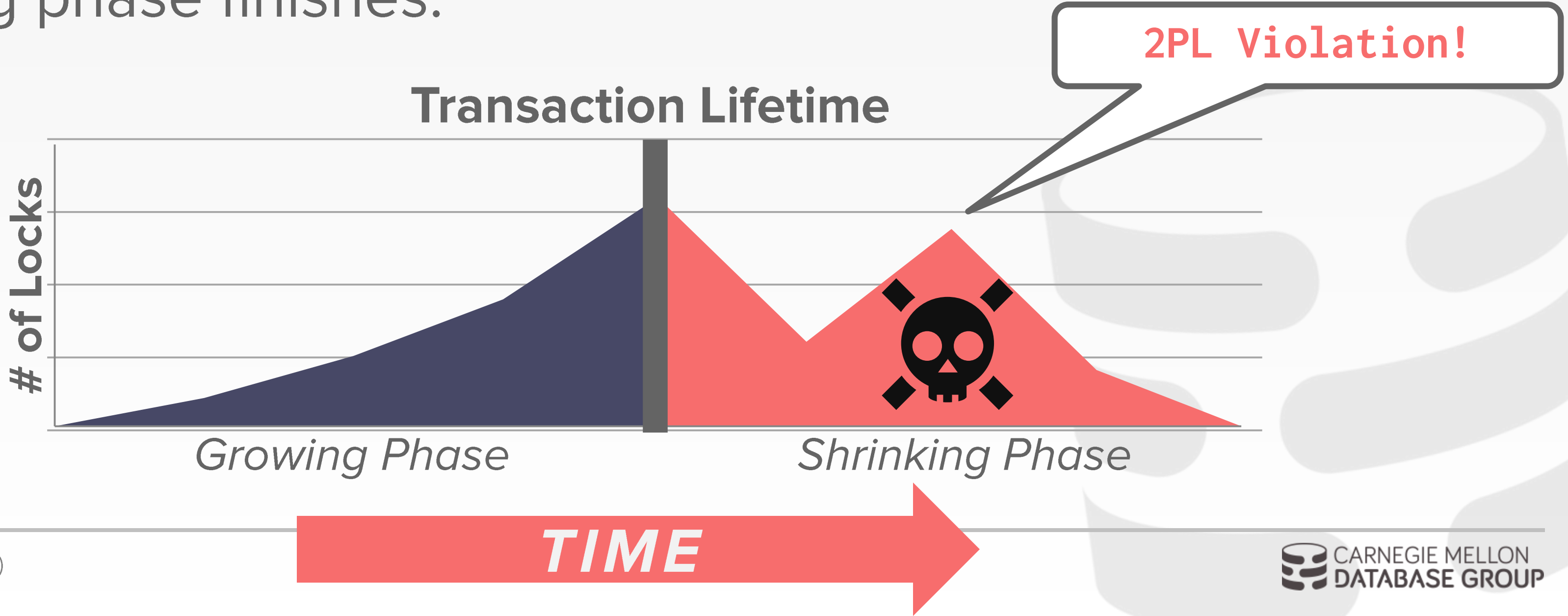
# TWO-PHASE LOCKING

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



# TWO-PHASE LOCKING

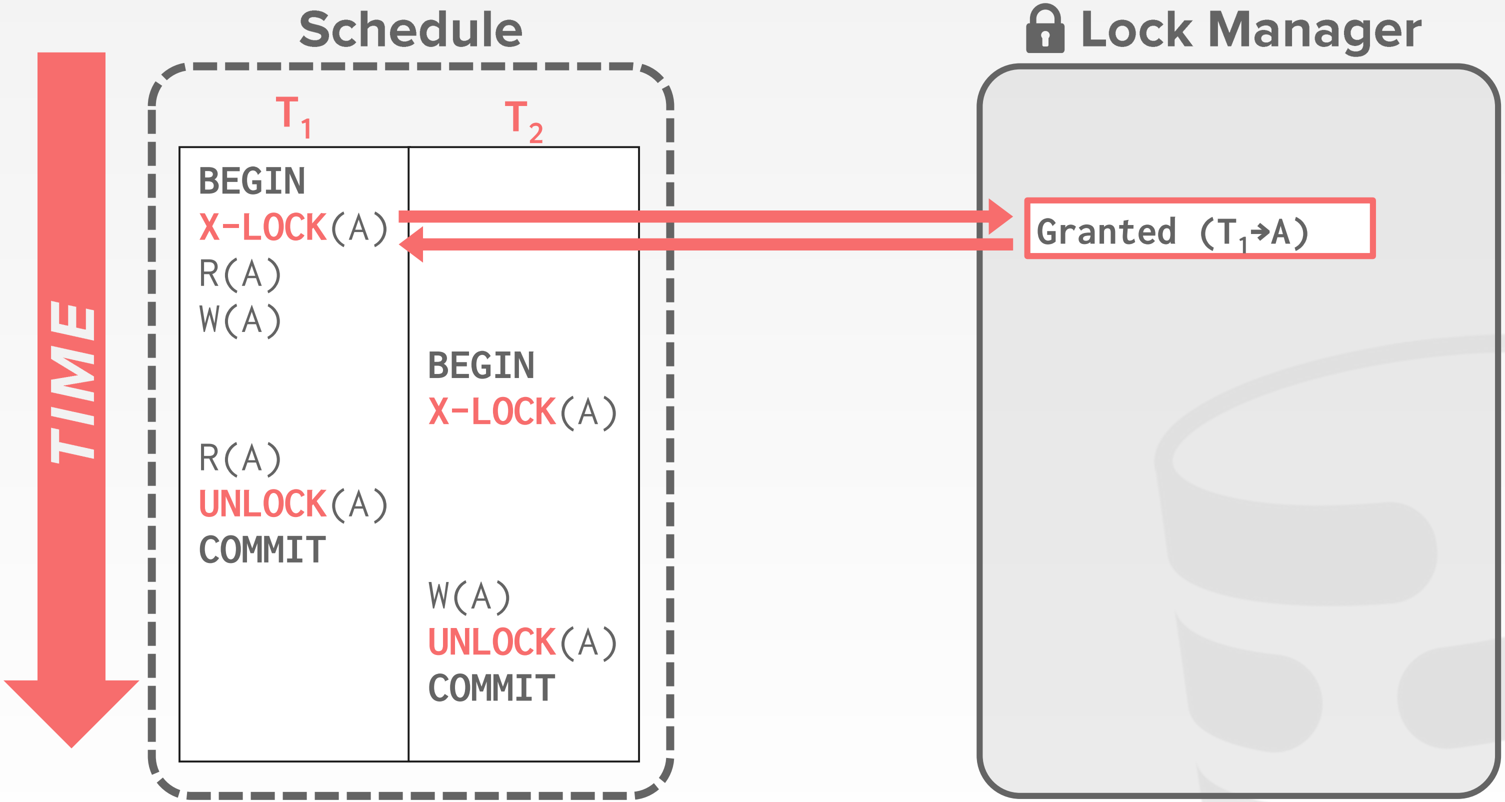
The txn is not allowed to acquire/upgrade locks after the growing phase finishes.



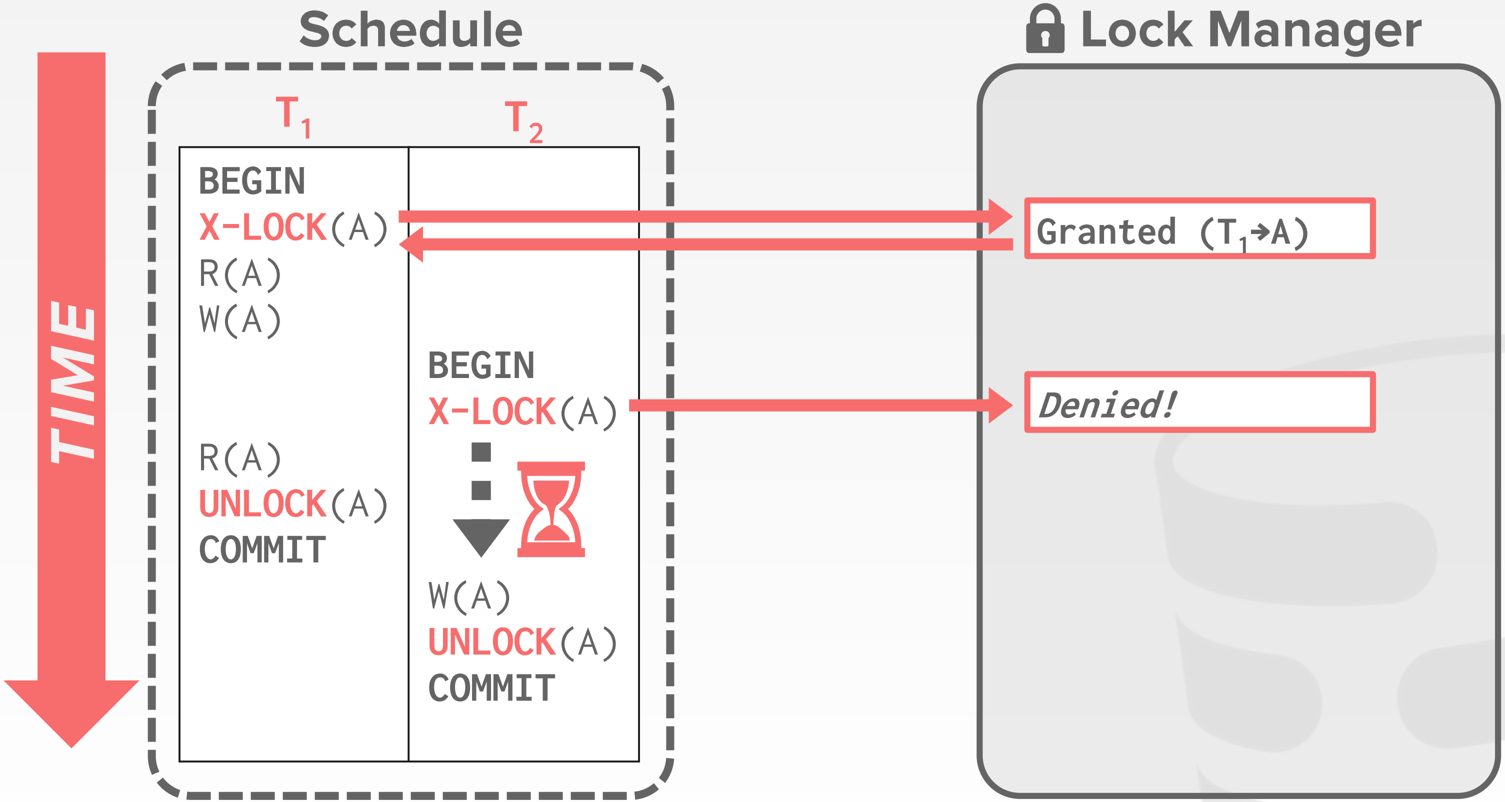


CARNEGIE MELLON DATABASE GROUP

# EXECUTING WITH 2PL



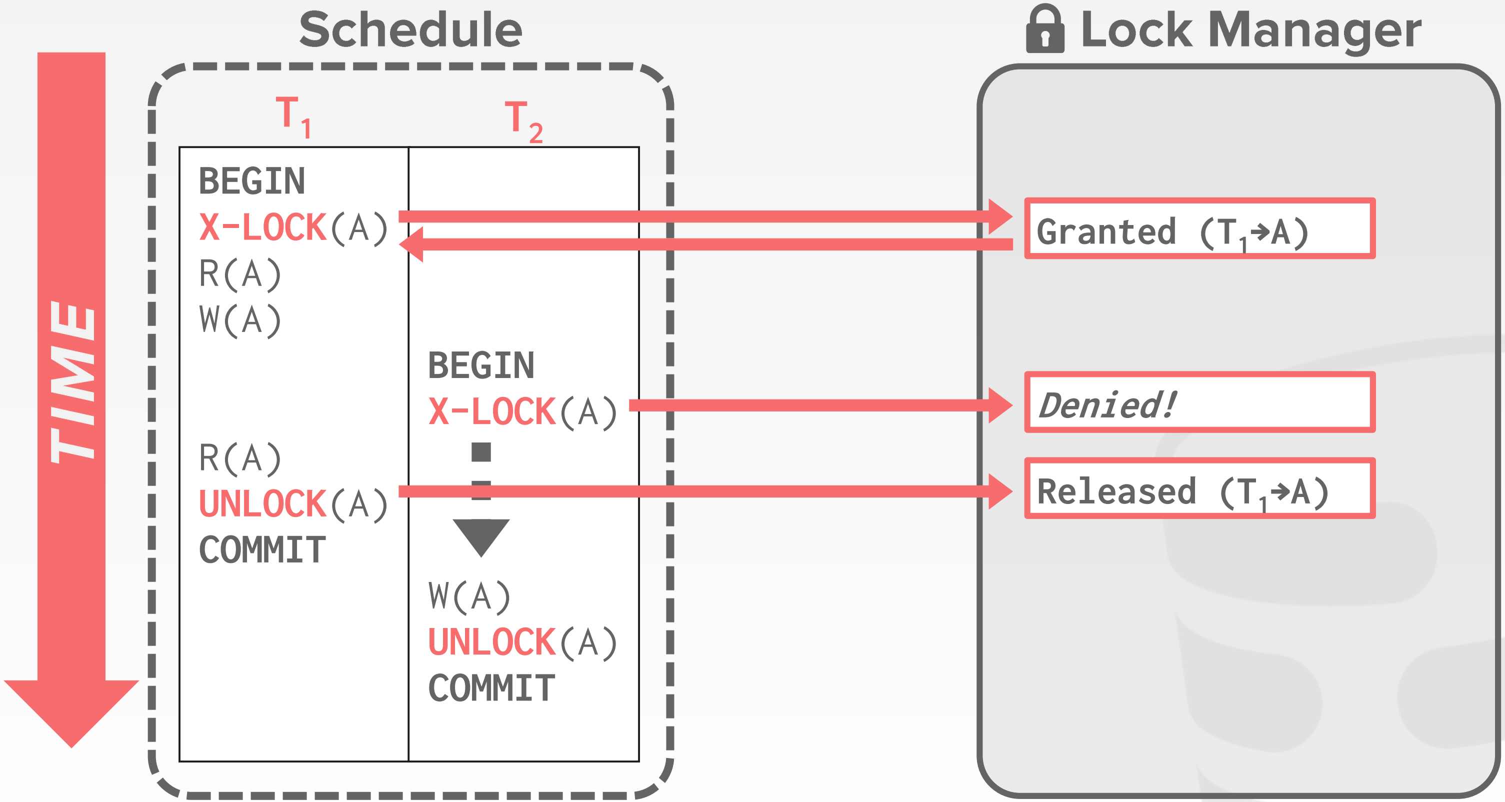
# EXECUTING WITH 2PL






CARNEGIE MELLON
DATABASE GROUP

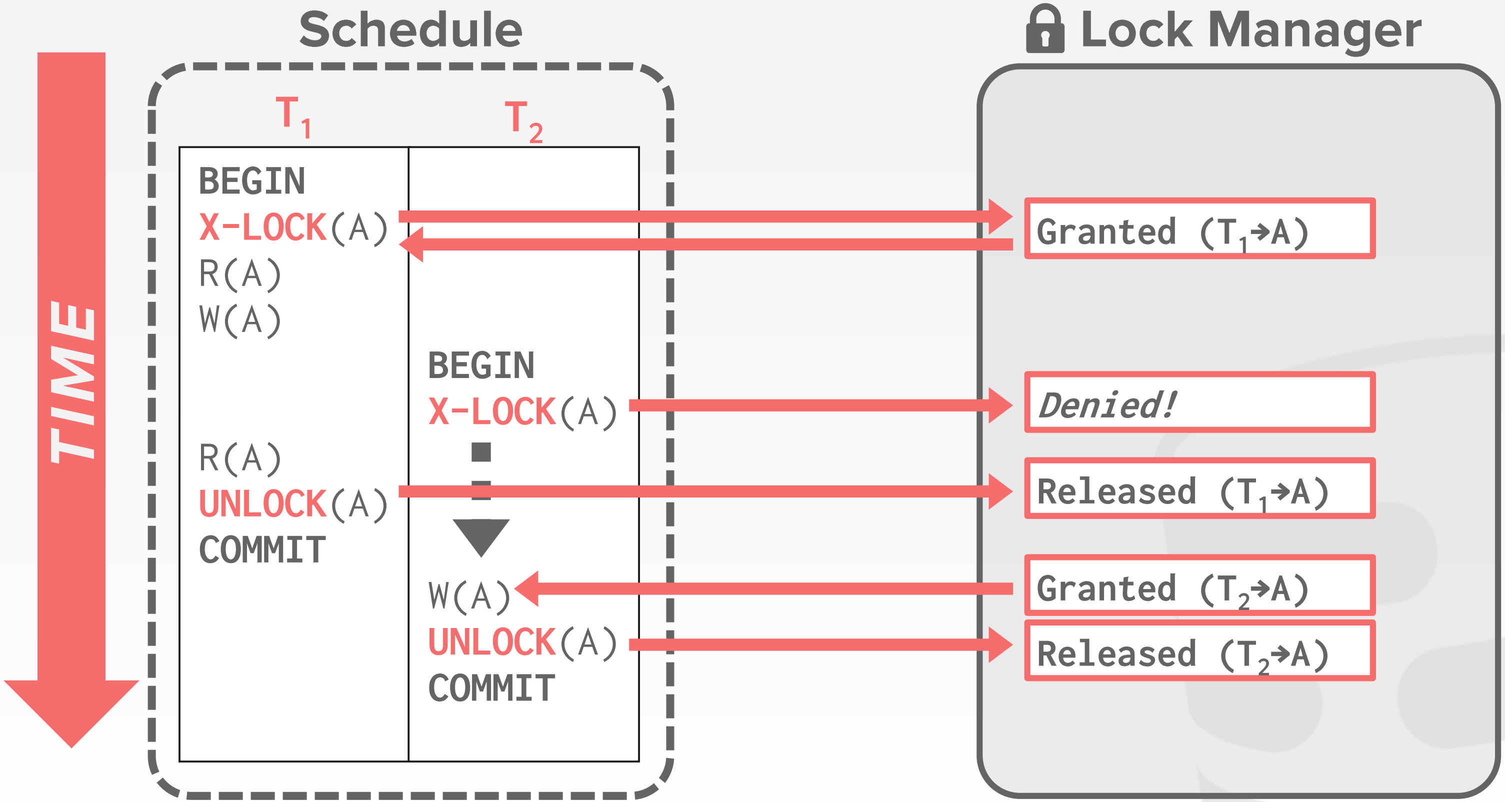
# EXECUTING WITH 2PL






CARNEGIE MELLON DATABASE GROUP

# EXECUTING WITH 2PL





CARNEGIE MELLON DATABASE GROUP

# TWO-PHASE LOCKING

2PL on its own is sufficient to guarantee conflict serializability.

→ It generates schedules whose precedence graph is acyclic.

But it is subject to **cascading aborts**.

# 2PL – CASCADING ABORTS

Schedule

TIME

| $T_1$ | $T_2$ |
|---|---|
| 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 | |

This is all wasted work!

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

# 2PL OBSERVATIONS

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

→ Locking limits concurrency.

May still have "dirty reads".

→ Solution: **Strict 2PL**

May lead to deadlocks.

→ Solution: **Detection** or **Prevention**

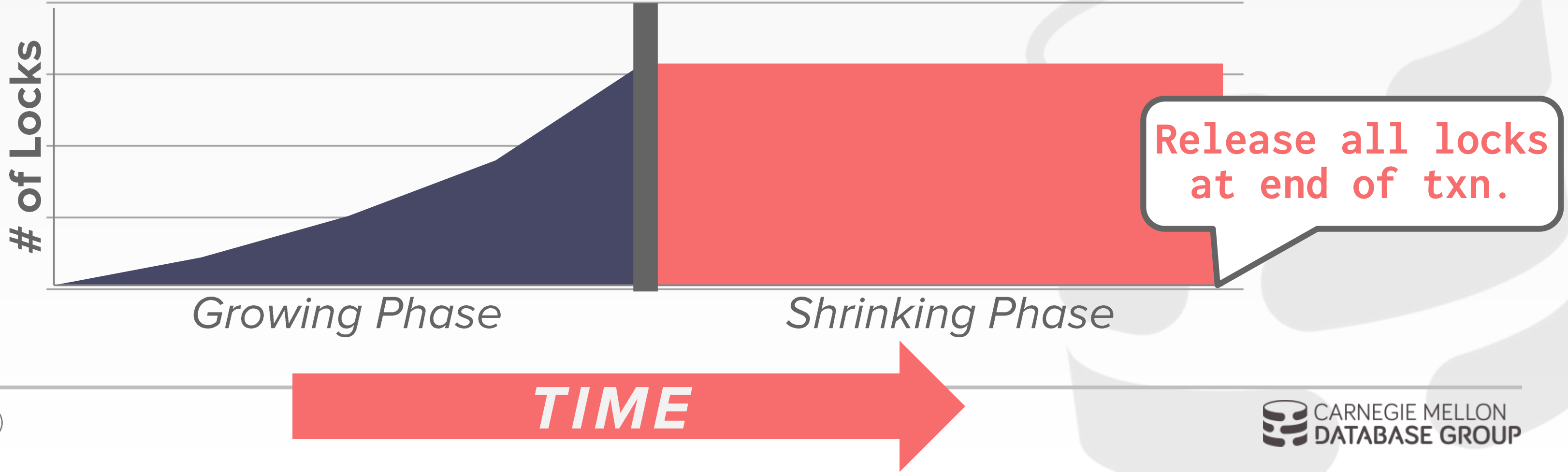


CARNEGIE MELLON DATABASE GROUP

# 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 actually stronger than needed.



# STRICT TWO-PHASE LOCKING

A schedule is **strict** if a value written by a txn is not read or overwritten by other txns until that txn finishes.

Advantages:

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

# EXAMPLES

$T_1$ – Move $50 from Andy's account to his bookie's account.

$T_2$ – Compute the total amount in all accounts and return it to the application.

Legend:

→ A ➔ Andy's account.

→ B ➔ The bookie's account.

# NON-2PL EXAMPLE

## Schedule

TIME

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | BEGIN |
| X-LOCK(A) | |
| R(A) | |
| | S-LOCK(A) |
| A=A-50 | |
| 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+50 | COMMIT |
| W(B) | |
| UNLOCK(B) | |
| COMMIT | |

# NON-2PL EXAMPLE

Schedule

TIME

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | BEGIN |
| X-LOCK(A) | |
| R(A) | |
| | S-LOCK(A) |
| A=A-50 | |
| 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+50 | COMMIT |
| W(B) | |
| UNLOCK(B) | |
| COMMIT | |

**Initial Database State**

A=100, B=100

**$T_2$ Output**

A+B=150

# 2PL EXAMPLE

Schedule

TIME

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | BEGIN |
| X-LOCK(A) | |
| R(A) | S-LOCK(A) |
| A=A-50 | |
| W(A) | |
| X-LOCK(B) | |
| UNLOCK(A) | R(A) |
| | S-LOCK(B) |
| R(B) | |
| B=B+50 | |
| W(B) | |
| UNLOCK(B) | R(B) |
| COMMIT | UNLOCK(A) |
| | UNLOCK(B) |
| | ECHO A+B |
| | COMMIT |

Initial Database State

A=100, B=100

$T_2$ Output

A+B=200

# STRICT 2PL EXAMPLE

## Schedule

TIME

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | BEGIN |
| X-LOCK(A) | |
| R(A) | S-LOCK(A) |
| A=A-50 | |
| W(A) | |
| X-LOCK(B) | |
| R(B) | |
| B=B+50 | |
| 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=100, B=100

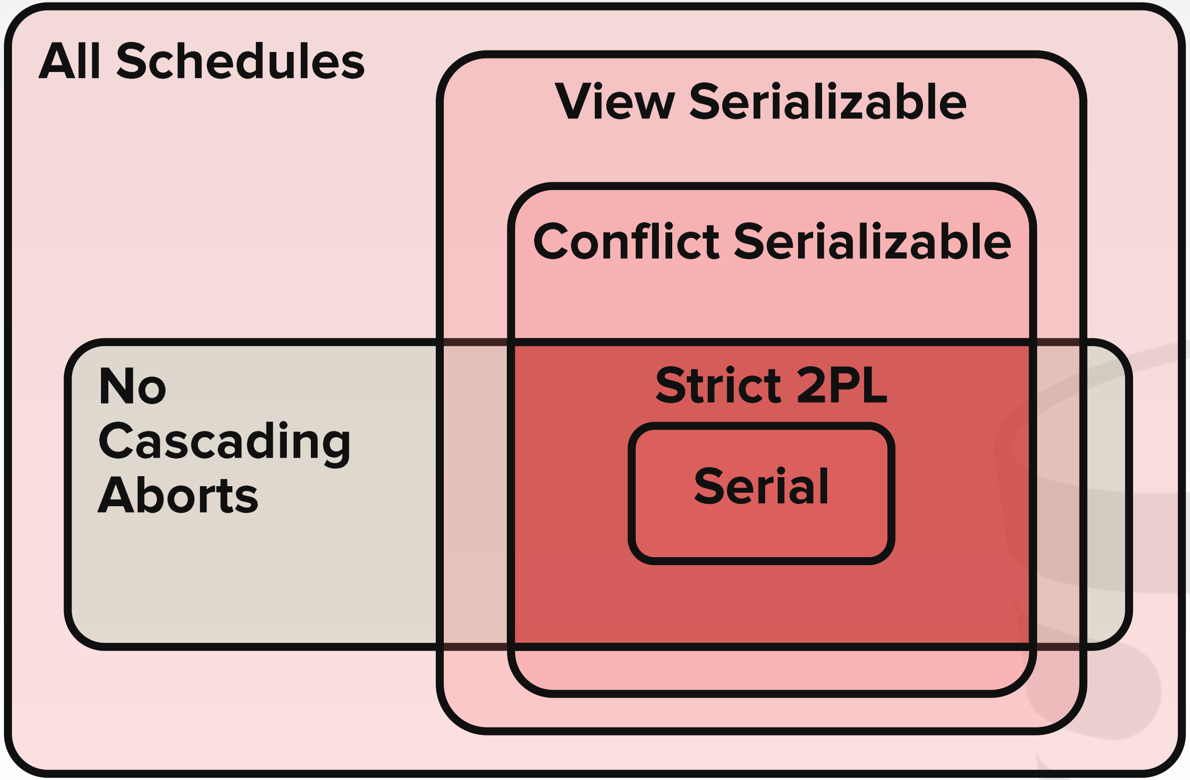
## $T_2$ Output

A+B=200



CARNEGIE MELLON DATABASE GROUP

# UNIVERSE OF SCHEDULES






CARNEGIE MELLON DATABASE GROUP

# 2PL OBSERVATIONS

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

→ Locking limits concurrency.

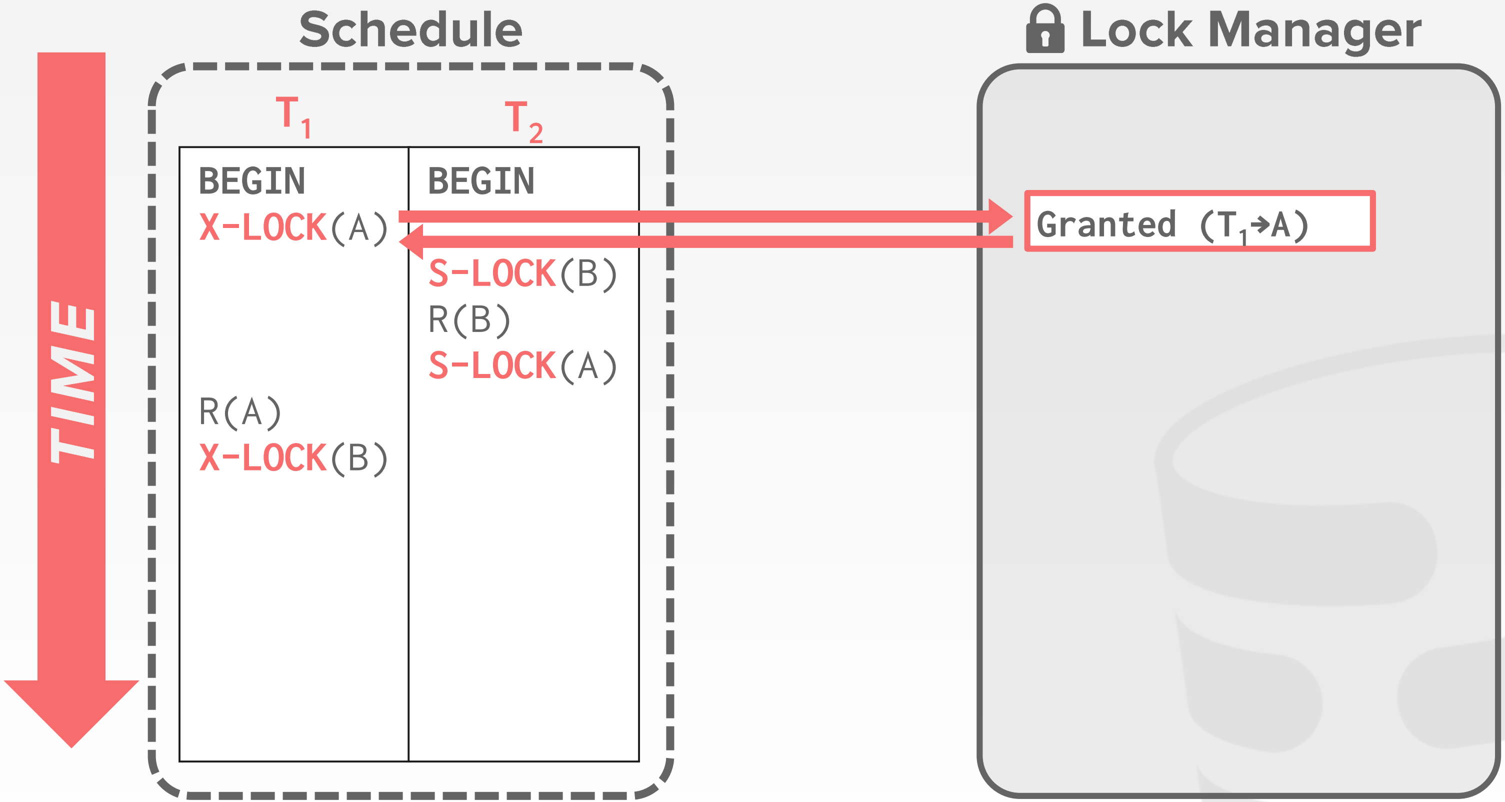
May still have "dirty reads".

→ Solution: **Strict 2PL**

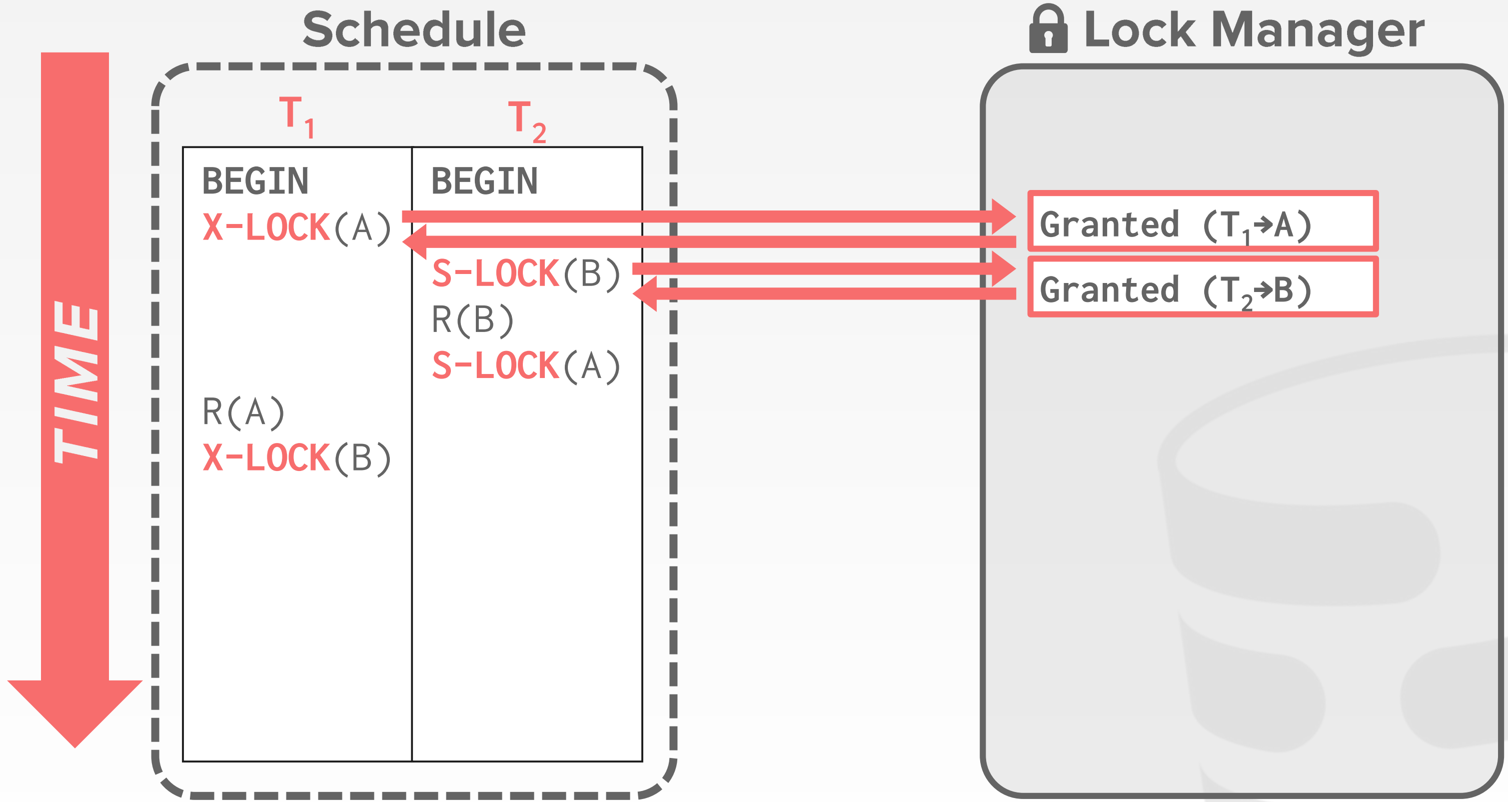
May lead to deadlocks.

→ Solution: **Detection** or **Prevention**

# SHIT JUST GOT REAL, SON



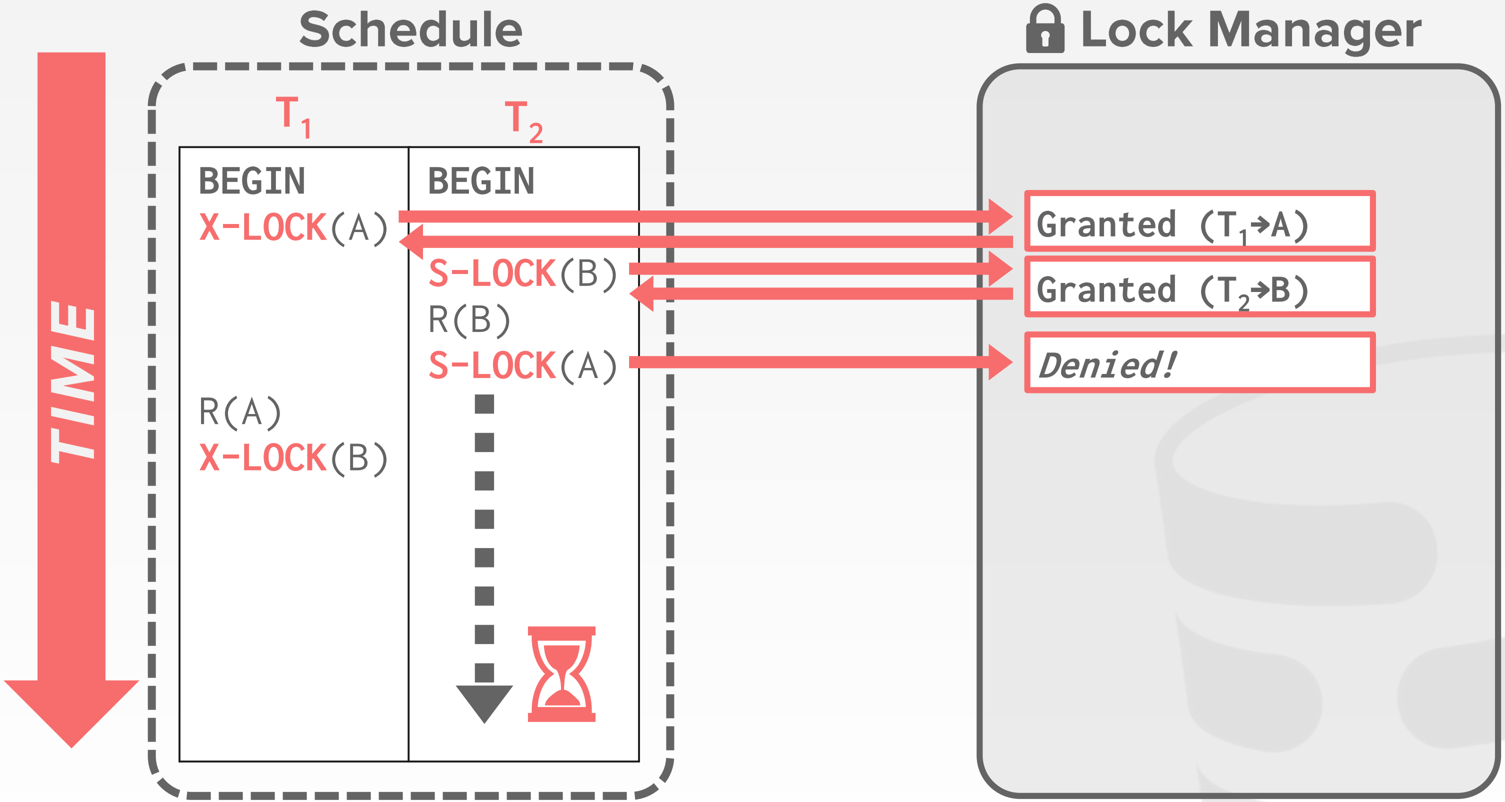
# SHIT JUST GOT REAL, SON



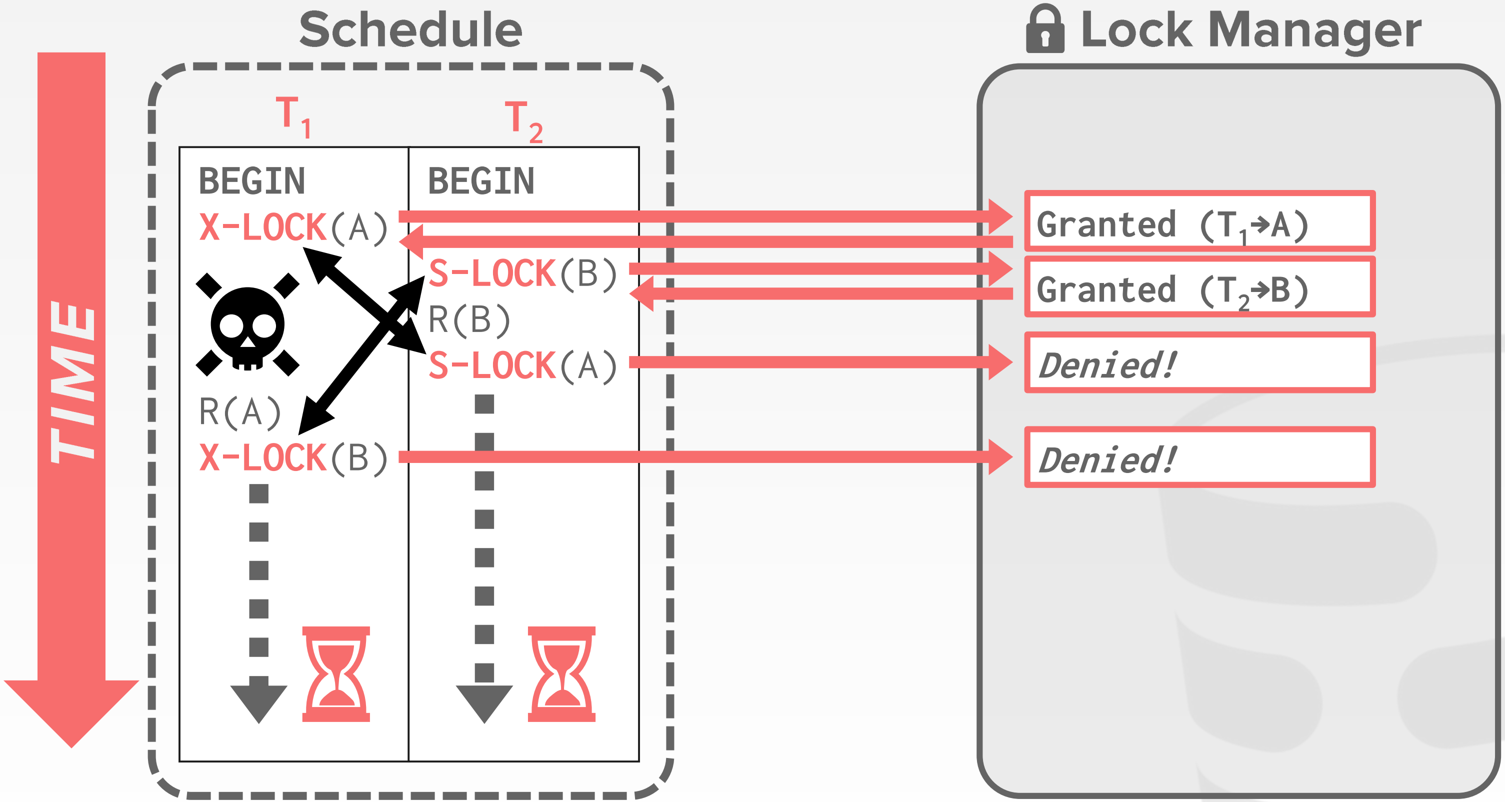


CARNEGIE MELLON DATABASE GROUP

# SHIT JUST GOT REAL, SON



# SHIT JUST GOT REAL, SON






CARNEGIE MELLON
DATABASE GROUP

# 2PL DEADLOCKS

A **deadlock** is a cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

→ **Approach #1: Deadlock Detection**

→ **Approach #2: Deadlock Prevention**

# DEADLOCK DETECTION

The DBMS creates 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.

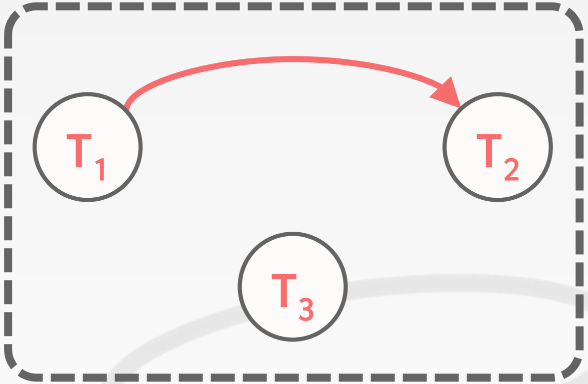
The system will periodically check for cycles in waits-for graph and then make a decision on how to break it.

# DEADLOCK DETECTION

## Schedule

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

## *Waits-For* Graph




TIME

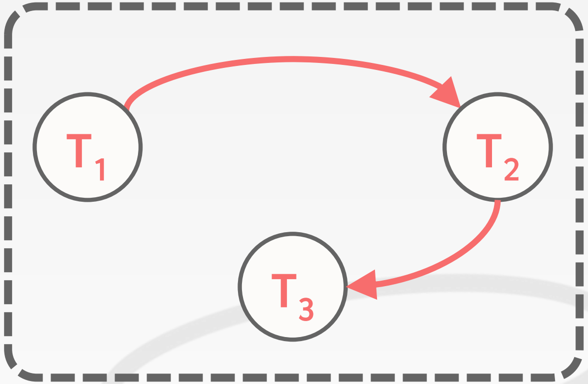
# DEADLOCK DETECTION

**Schedule**

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

TIME

***Waits-For* Graph**






CARNEGIE MELLON DATABASE GROUP

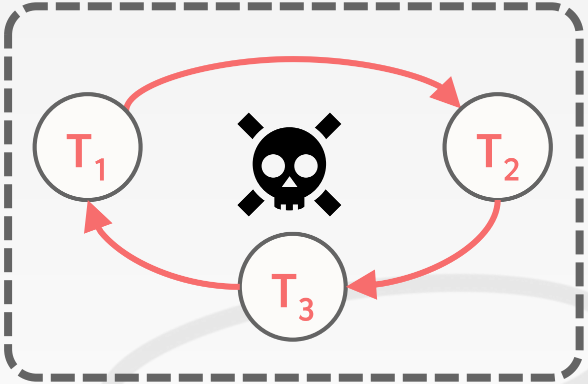
# DEADLOCK DETECTION

TIME

## Schedule

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

## *Waits-For* Graph





CARNEGIE MELLON DATABASE GROUP

# DEADLOCK DETECTION

How often should we run the algorithm?

How many txns are typically involved?

What do we do when we find a deadlock?

# DEADLOCK HANDLING

When the DBMS detects a deadlock, it will select a "victim" txn to rollback to break the cycle.

The victim txn will either restart or abort depending on how the application invoked it.

# DEADLOCK HANDLING: VICTIM SELECTION

Selecting the proper victim depends on a lot of different variables….

→ By age (lowest timestamp)
→ By progress (least/most queries executed)
→ By the # of items already locked
→ By the # of txns that we have to rollback with it

We also should consider the # of times a txn has been restarted in the past.

# DEADLOCK HANDLING: ROLLBACK LENGTH

After selecting a victim txn to abort, the DBMS can also decide on how far to rollback the txn's changes.

**Approach #1: Completely**

**Approach #2: Minimally**

# DEADLOCK PREVENTION

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

No *waits-for* graph or detection algorithm.

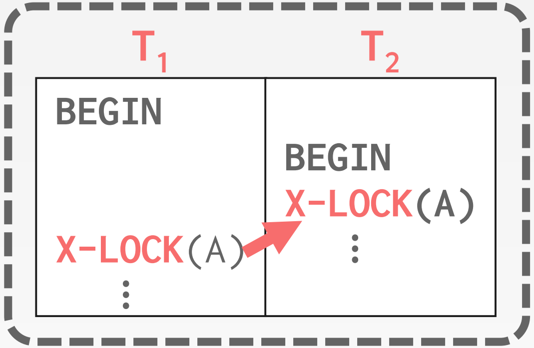
# DEADLOCK PREVENTION

Assign priorities based on timestamps:
→ Older ➔ higher priority (e.g., $T_1 > T_2$)

**Wait-Die ("Old Waits for Young")**
→ If $T_1$ has higher priority, $T_1$ waits for $T_2$.
→ Otherwise $T_1$ aborts.

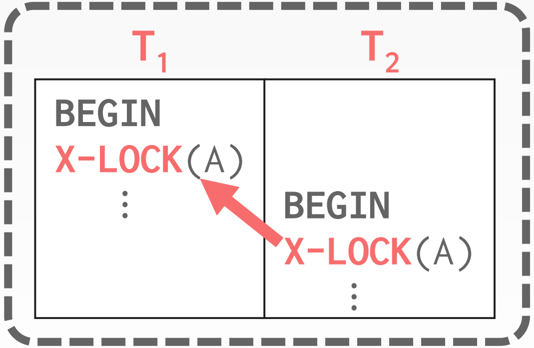
**Wound-Wait ("Young Waits for Old")**
→ If $T_1$ has higher priority, $T_2$ aborts.
→ Otherwise $T_1$ waits.

# DEADLOCK PREVENTION




**Wait-Die**: $T_1$ waits

**Wound-Wait**: $T_2$ aborts




**Wait-Die**: $T_2$ aborts

**Wound-Wait**: $T_2$ 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?

# OBSERVATION

All of these examples have a one-to-one mapping from database objects to locks.

If a txn wants to update one billion tuples, then it has to acquire one billion locks.

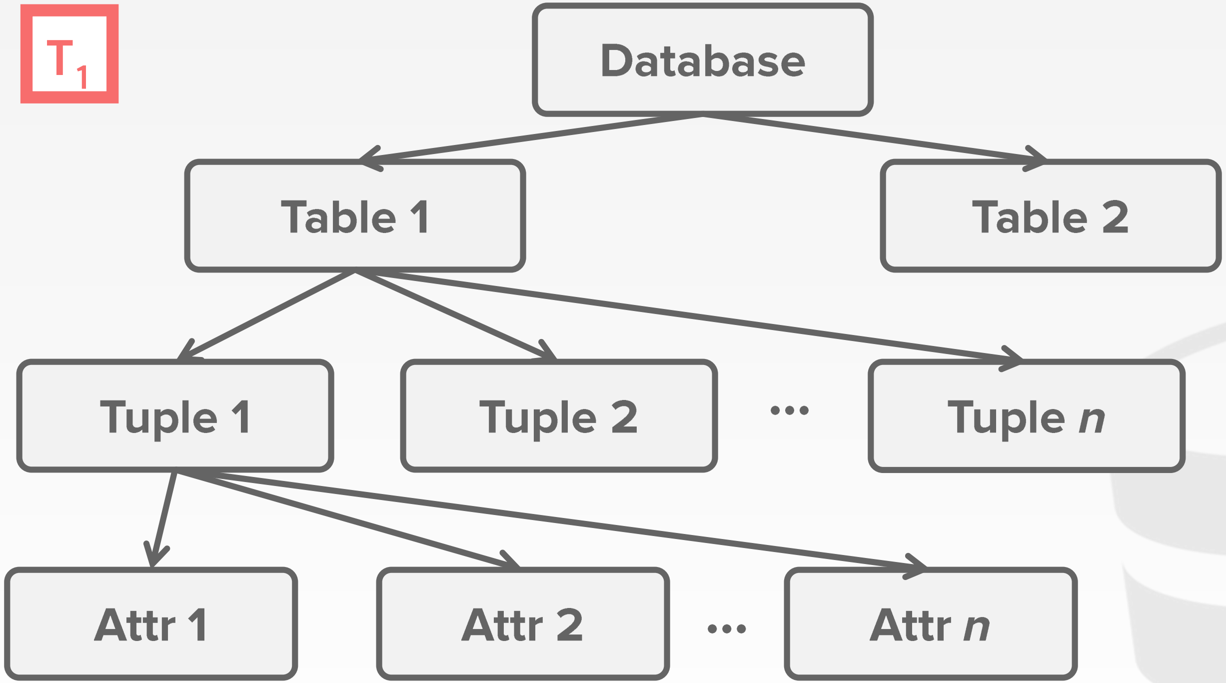
# LOCK GRANULARITIES

When we say that a txn acquires a "lock", what does that actually mean?

→ On an Attribute? Tuple? Page? Table?

Ideally, each txn should obtain fewest number of locks that is needed...

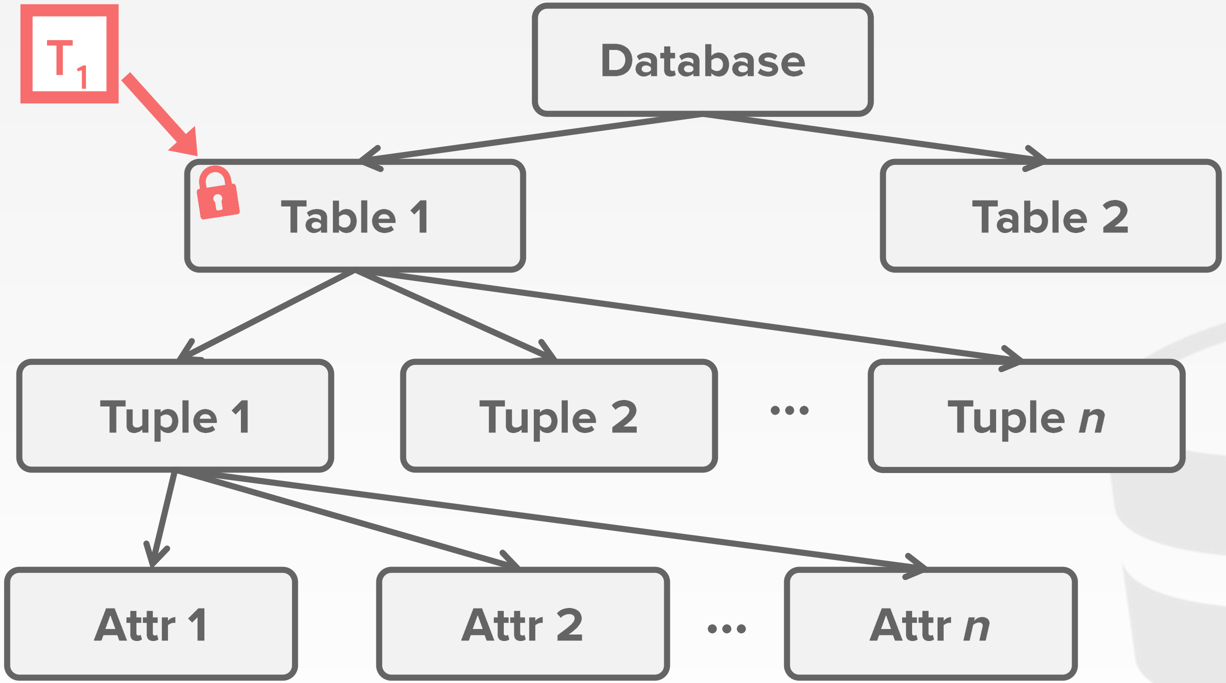
# DATABASE LOCK HIERARCHY






CARNEGIE MELLON DATABASE GROUP

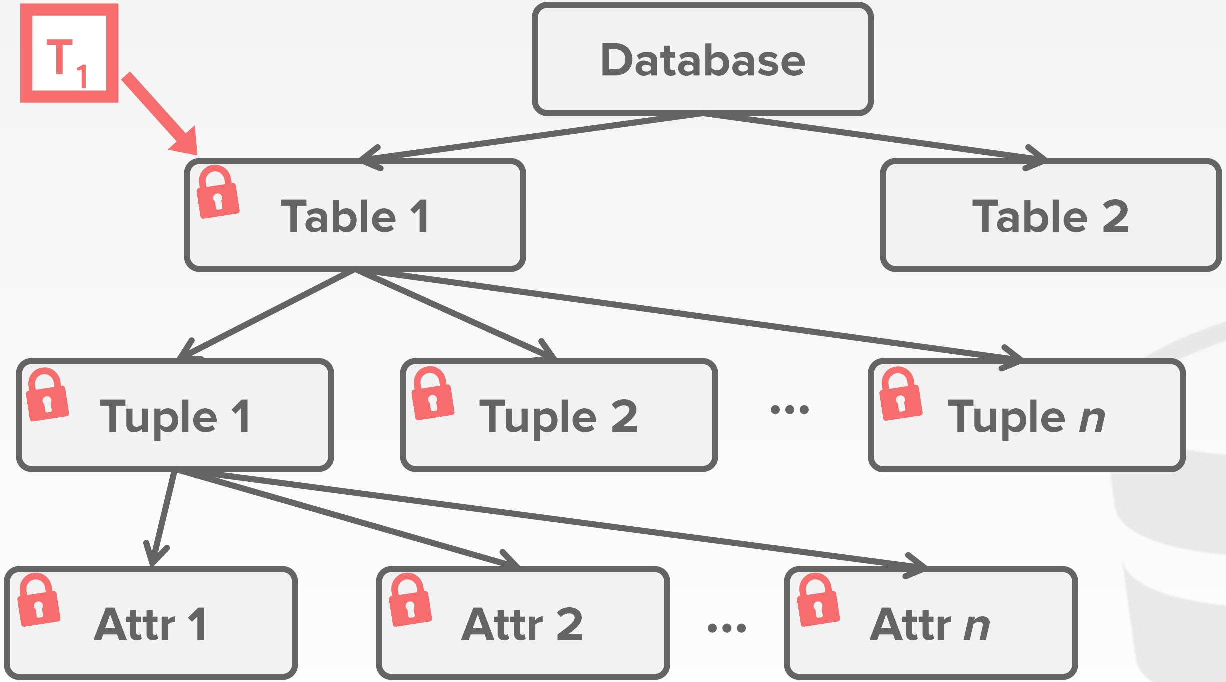
# DATABASE LOCK HIERARCHY






CARNEGIE MELLON DATABASE GROUP

# DATABASE LOCK HIERARCHY



# EXAMPLE

$T_1$ – Get the balance of Andy's shady off-shore bank account.

$T_2$ – Increase Joy's bank account balance by 1%.

**What locks should they obtain?**

# EXAMPLE

$T_1$ – Get the balance of Andy’s shady off-shore bank account.

$T_2$ – Increase Joy's bank account balance by 1%.

**What locks should they obtain?**

Multiple:

→ **Exclusive** + **Shared** for leafs of lock tree.
→ Special **Intention** locks for higher levels.

# INTENTION LOCKS

An **intention lock** allows a higher level node to be locked in **shared** or **exclusive** mode without having to check all descendent nodes.

If a node is in an intention mode, then explicit locking is being done at a lower level in the tree.

# INTENTION LOCKS

**Intention-Shared (IS)**
→ Indicates explicit locking at a lower level with shared locks.

**Intention-Exclusive (IX)**
→ Indicates locking at lower level with exclusive or shared locks.

# INTENTION LOCKS

## Shared+Intention-Exclusive (SIX)

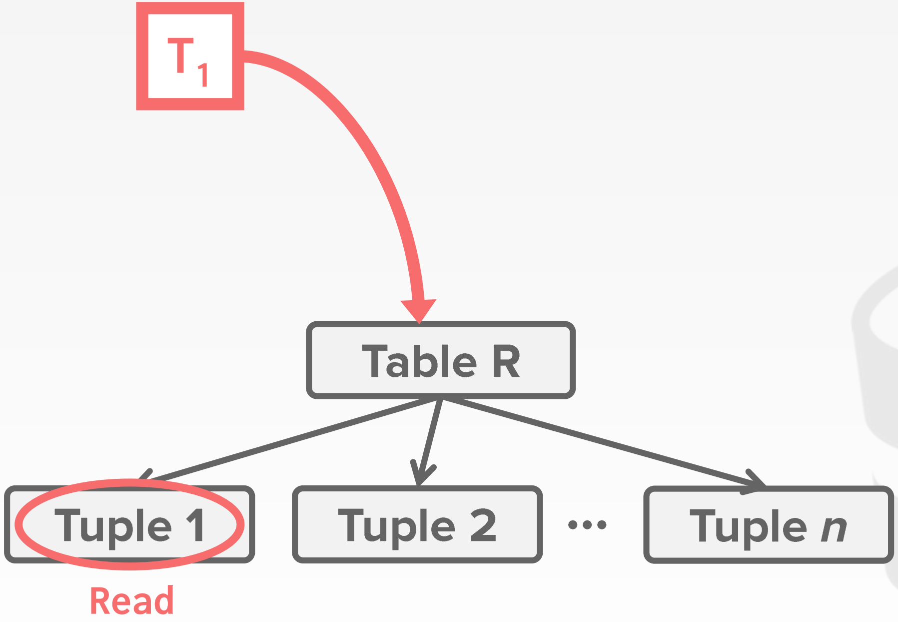
→ The subtree rooted by that node is locked explicitly in **shared** mode and explicit locking is being done at a lower level with **exclusive-mode** locks.

# COMPATIBILITY MATRIX

| $T_1$ Holds \ $T_2$ Wants | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | ✓ | ✓ | ✓ | ✓ | ✗ |
| IX | ✓ | ✓ | ✗ | ✗ | ✗ |
| S | ✓ | ✗ | ✓ | ✗ | ✗ |
| SIX | ✓ | ✗ | ✗ | ✗ | ✗ |
| X | ✗ | ✗ | ✗ | ✗ | ✗ |

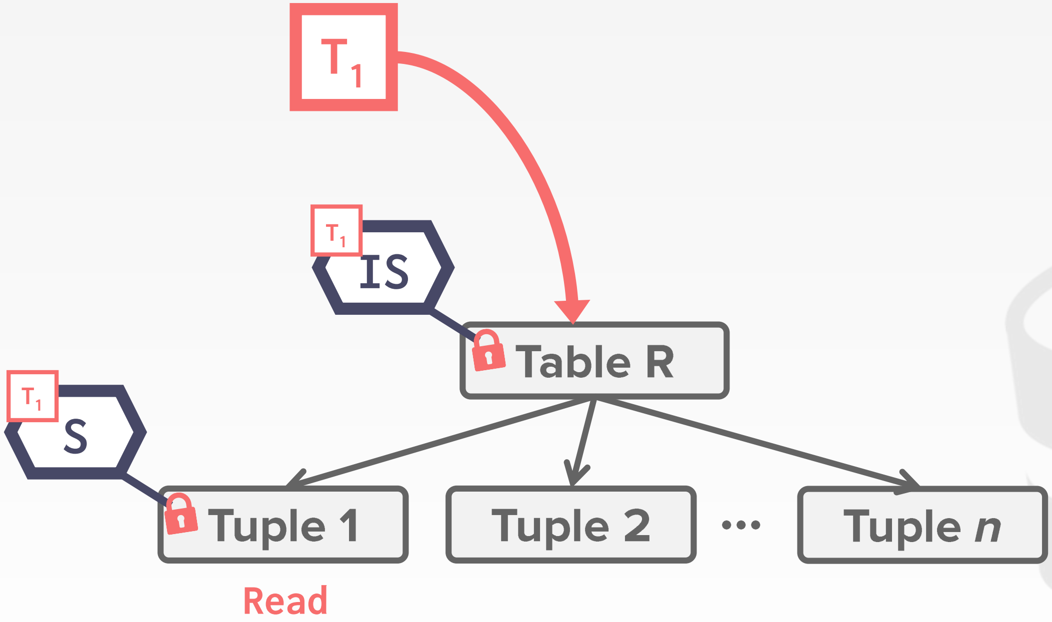
# EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in **R**.



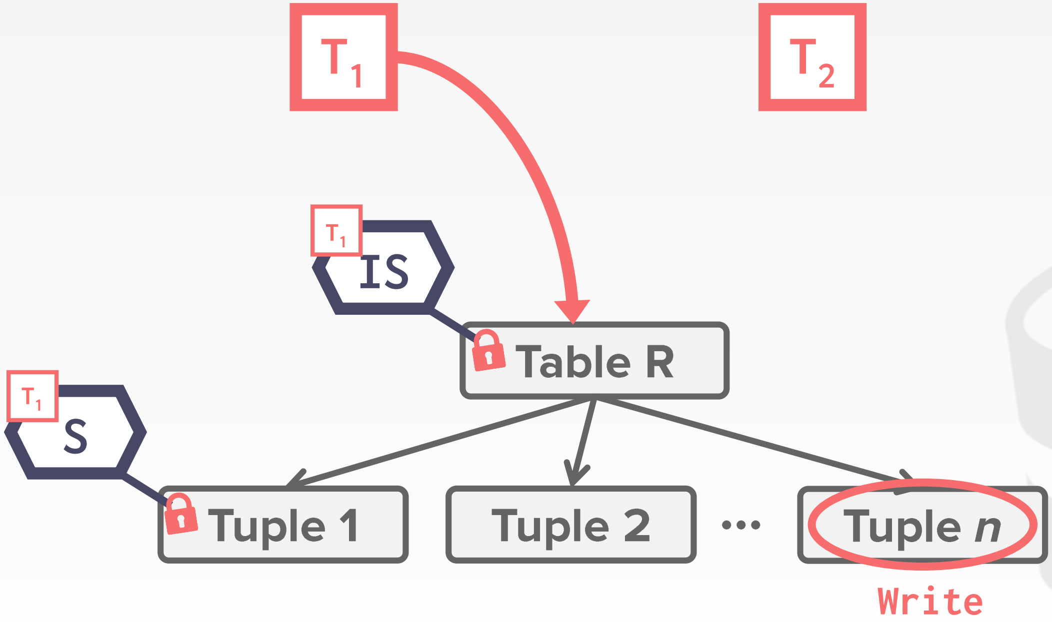
# EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in **R**.



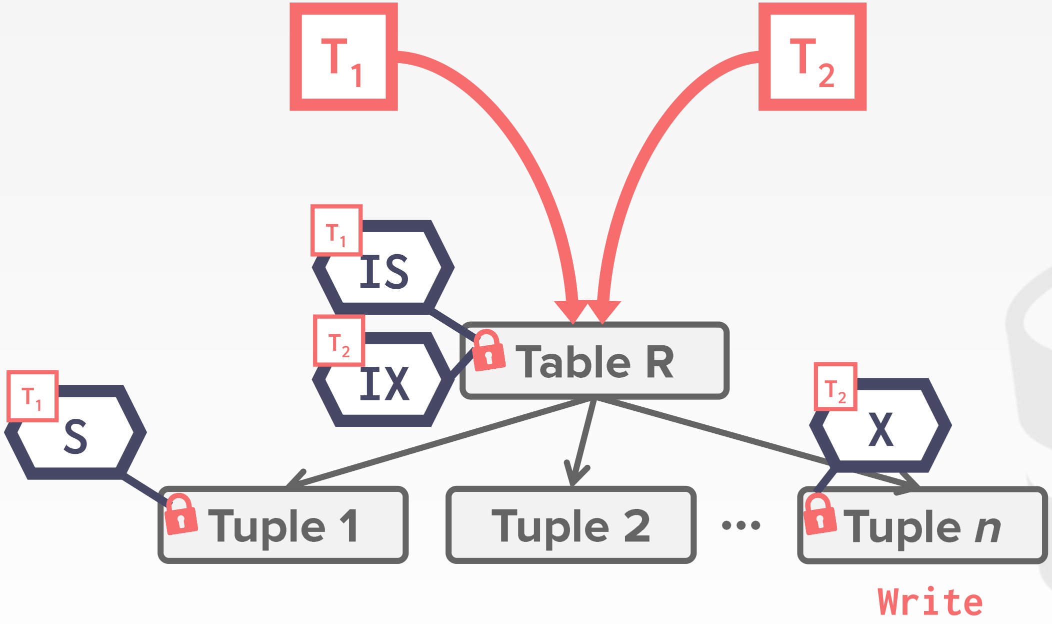
# EXAMPLE – TWO-LEVEL HIERARCHY

Update Joy's record in **R**.



# EXAMPLE – TWO-LEVEL HIERARCHY

Update Joy's record in **R**.



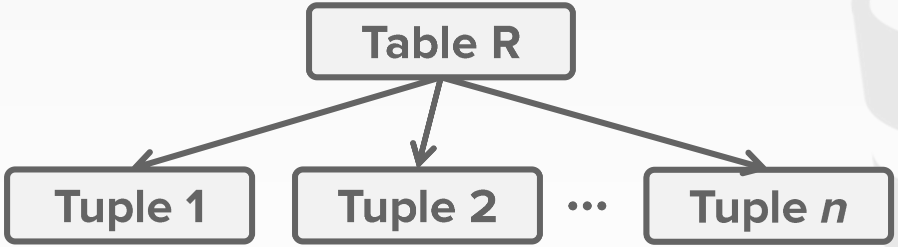



CARNEGIE MELLON DATABASE GROUP

# 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**.

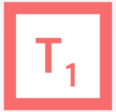


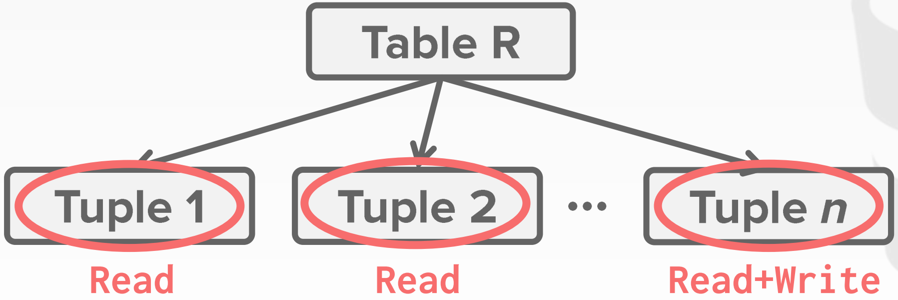



CARNEGIE MELLON DATABASE GROUP

# EXAMPLE – THREESOME

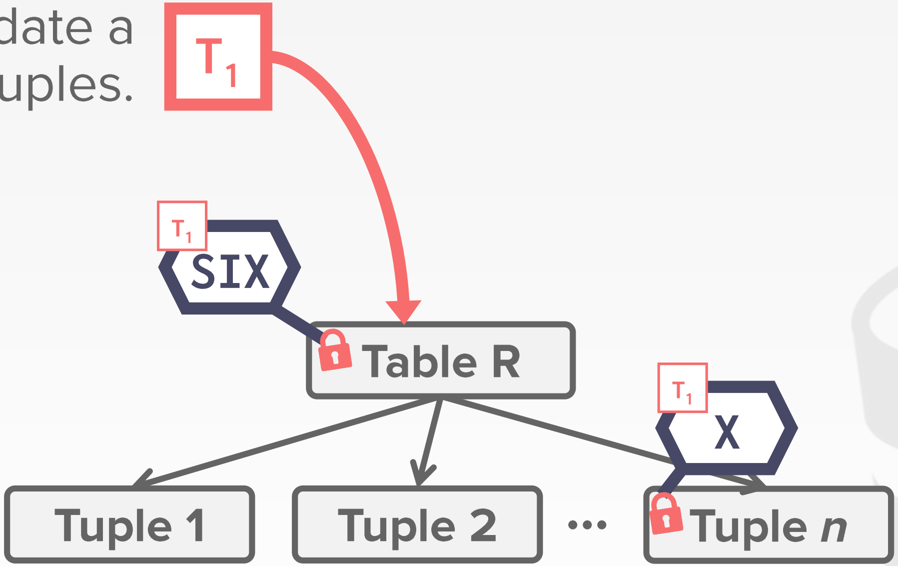
Scan R and update a few tuples.



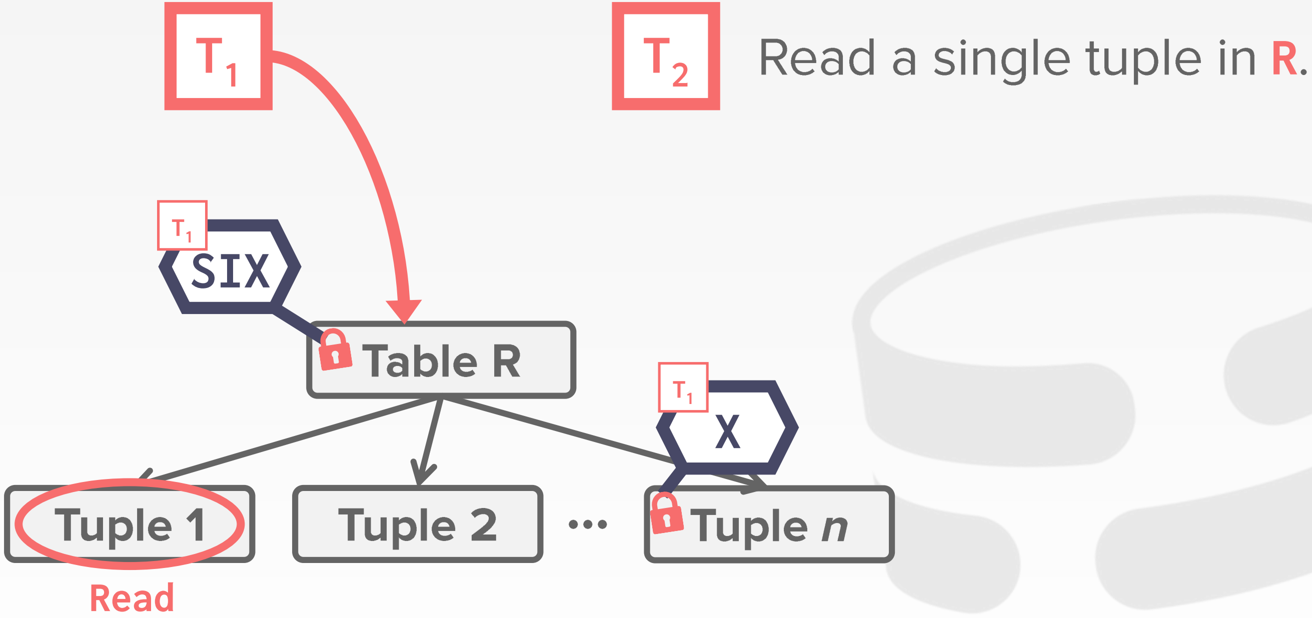



# EXAMPLE – THREESOME

Scan R and update a few tuples.



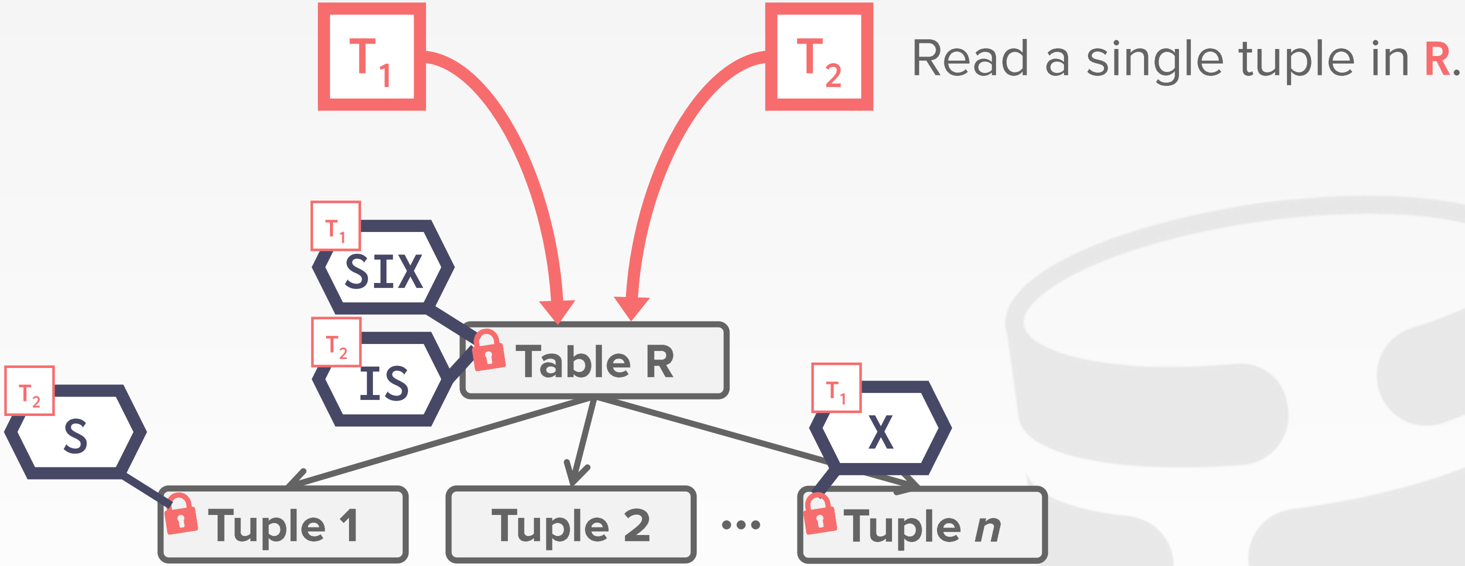
# EXAMPLE – THREESOME



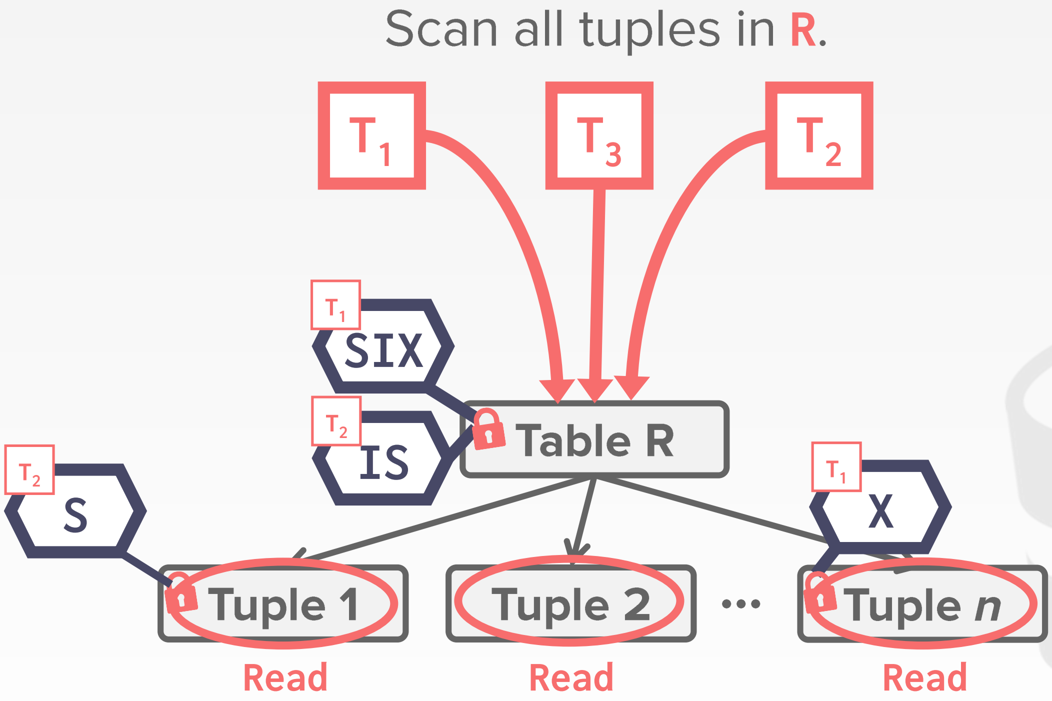



CARNEGIE MELLON DATABASE GROUP

# EXAMPLE – THREESOME

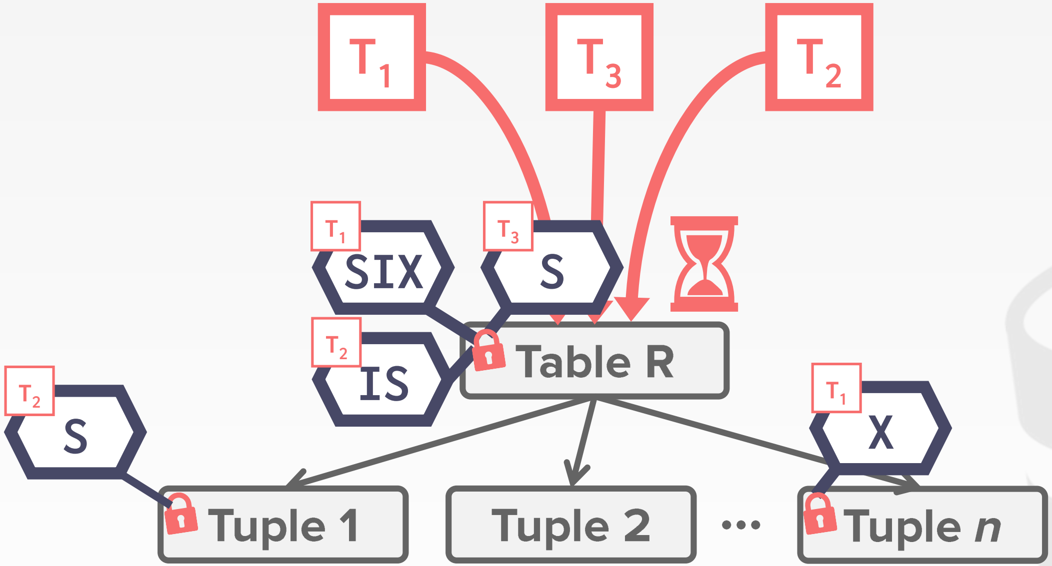


# EXAMPLE – THREESOME



# EXAMPLE – THREESOME

Scan all tuples in **R**.






CARNEGIE MELLON
DATABASE GROUP

# MULTIPLE LOCK GRANULARITIES

Useful in practice as each txn only needs a few locks.

Intention locks help improve concurrency:

→ **Intention-Shared (IS)**: Intent to get S lock(s) at finer granularity.
→ **Intention-Exclusive (IX)**: Intent to get X lock(s) at finer granularity.
→ **Shared+Intention-Exclusive (SIX)**: Like S and IX at the same time.

# LOCKING PROTOCOL

Each txn obtains appropriate lock at highest level of the database hierarchy.

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.

# LOCK ESCALATION

Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired.

This reduces the number of requests that the lock manager has to process.

# LOCKING IN PRACTICE

You typically don't set locks manually.

Sometimes you will need to provide the DBMS with hints to help it to improve concurrency.

Also useful for doing major changes.

# LOCK TABLE

Explicitly locks a table.

Not part of the SQL standard.

→ Postgres/DB2/Oracle Modes: **SHARE**, **EXCLUSIVE**

→ MySQL Modes: **READ**, **WRITE**

PostgreSQL, ORACLE, IBM DB2

```
LOCK TABLE <table> IN <mode> MODE;
```

Microsoft SQL Server

```
SELECT 1 FROM <table> WITH (TABLOCK, <mode>);
```

MySQL

```
LOCK TABLE <table> <mode>;
```

# SELECT...FOR UPDATE

Perform a select and then sets an exclusive lock on the matching tuples.

Can also set shared locks:

→ Postgres: FOR SHARE

→ MySQL: LOCK IN SHARE MODE

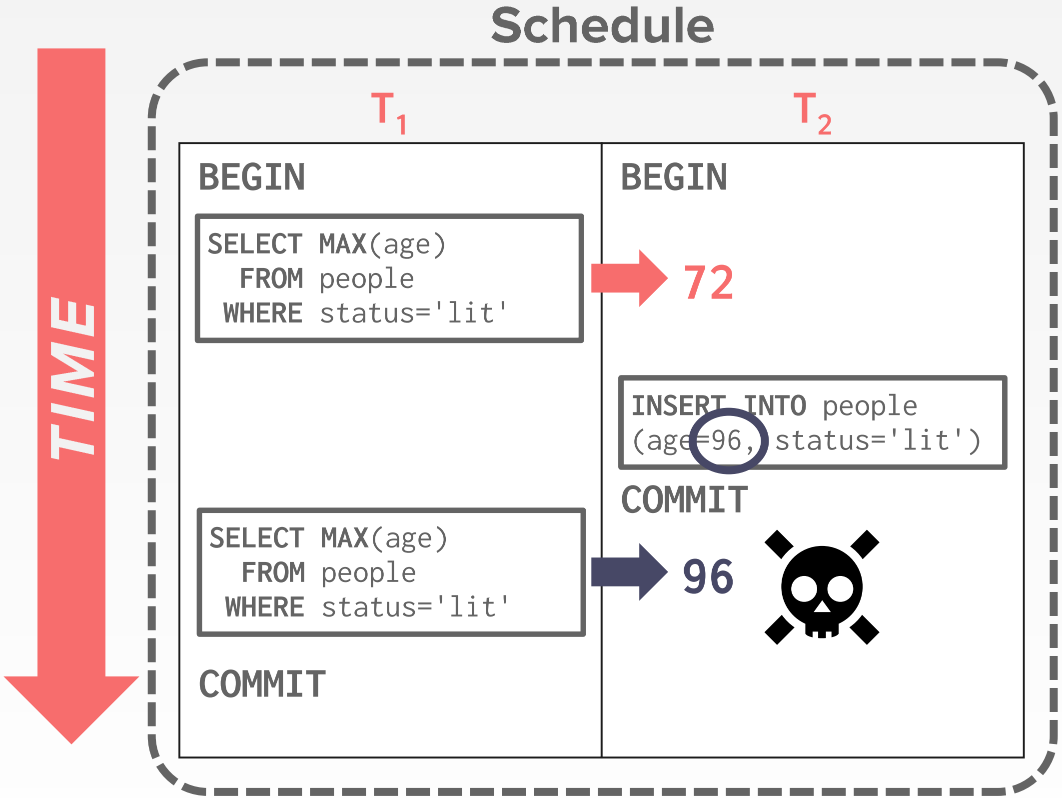
```
SELECT * FROM <table>
 WHERE <qualification> FOR UPDATE;
```

# DYNAMIC DATABASES

Recall that so far we have only dealing with transactions that read and update data.

But now if we have insertions, updates, and deletions, we have new problems…

# THE PHANTOM PROBLEM




```
CREATE TABLE people (
  id SERIAL,
  name VARCHAR,
  age INT,
  status VARCHAR
);
```

# WTF?

How did this happen?

→ Because $T_1$ locked only existing records and not ones under way!

Conflict serializability on reads and writes of individual items guarantees serializability **only** if the set of objects is fixed.

We will solve this problem in the next class.

# CONCLUSION

2PL is used in almost DBMS.

Automatically correct interleavings:

→ Locks + protocol (2PL, S2PL ...)
→ Deadlock detection + handling
→ Deadlock prevention

# NEXT CLASS

Two-Phase Locking

Isolation Levels