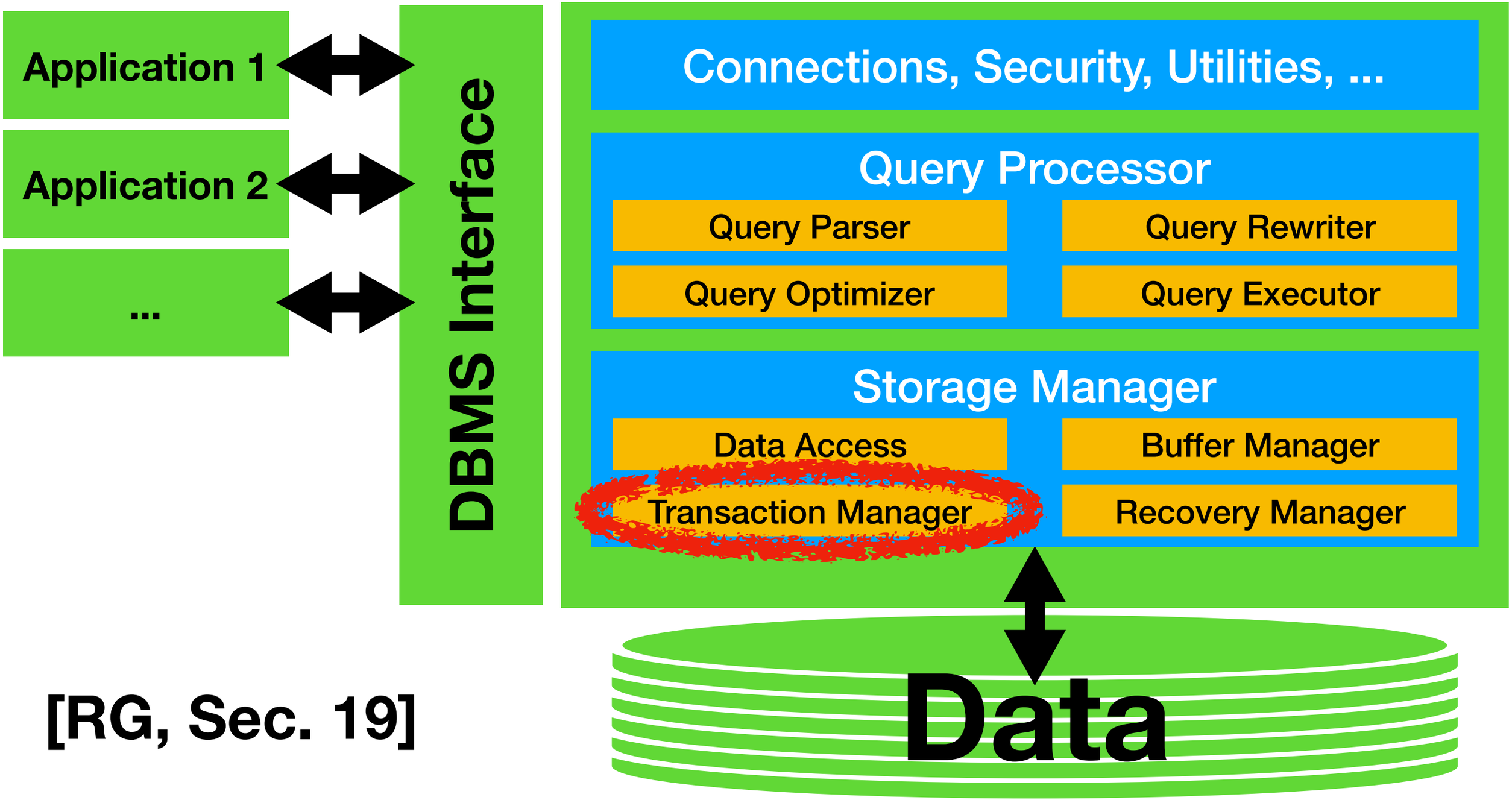
Two-Phase Locking

Immanuel Trummer
itrummer@cornell.edu
www.itrummer.org

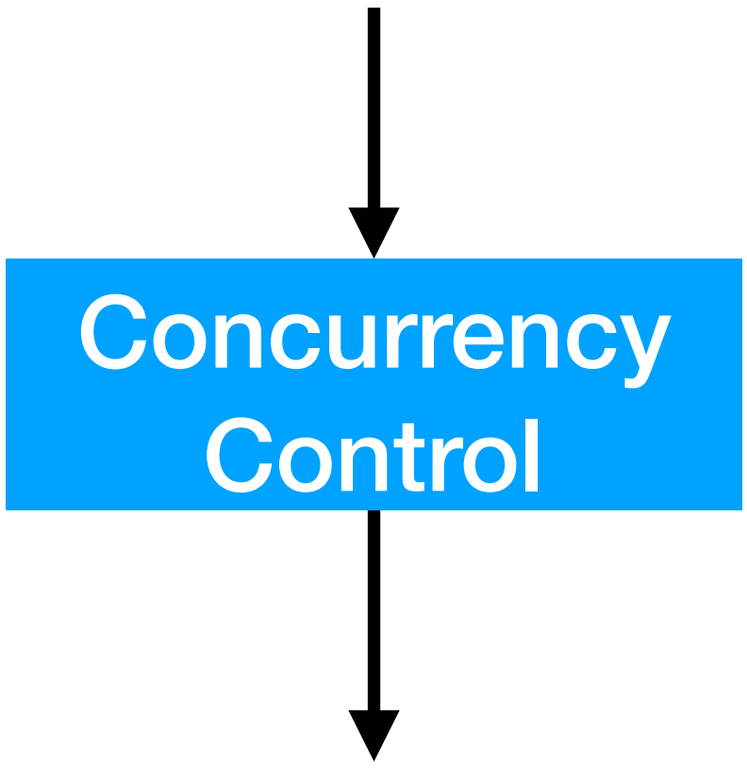
Database Management Systems (DBMS)



[RG, Sec. 19]

Concurrency Control

Transactions



Picks cheapest schedule among good ones

Schedule

(Ordered Transaction Steps)

Concurrency Control Protocols

- Have seen **desirable properties** of schedules
  - **Conflict serializability**: efficient and quite permissive
    - Want **recoverable** schedules, possibly ACA or strict
- Now discuss **protocols** to enforce such schedules
  - Allowing more schedules: **more optimization** possible
  - Ok with less schedules if mechanism **more efficient**

Lock-Based CC

- **Lock**: permission to operate on specific objects
  - Transactions need lock **to work** on object
    - Transactions obtain locks via a **lock request**
    - May have to **wait** until desired lock is granted
- **Lock manager** component grants locks
  - **Keeps track** of which transaction holds which locks

Simple Locking Strategy

- Use **one lock** for the entire database
- Transactions requests lock at **transactions start**
- Transaction gives back lock at **transaction end**
- **Only one** transaction can hold at the same time

How Does This Perform?

Refining Lock Granularity

- Transactions can work on **different objects** in parallel
- Enable by locking **specific DB objects** (instead of DB)
- Locking **protocol** summary:
  - Transaction requests locks on all its objects **at start**
  - **Waits** until all locks have been granted
  - Transaction executes and releases locks **at end**

Introducing Lock Types

- All conflicts involve some **write operation**
- Multiple transactions can **read objects** without conflicts
- Idea: distinguish between **read and write locks**
  - **Read (aka shared) locks** allow only read access
  - **Write (aka exclusive)** lock allow read+write access
- Transactions specifically request either **read or write** lock
- Lock manager may grant **multiple read locks** on same object

Release Locks Early

- So far: transactions request locks at **start**, release at **end**
- **Releasing locks earlier** may increase parallelism
  - Release lock **after last operation** on associated object
- But doing so may lead to **cascading aborts**, e.g.:
  - W1(A) **[Lock on A from 1 → 2]** R2(A) A1

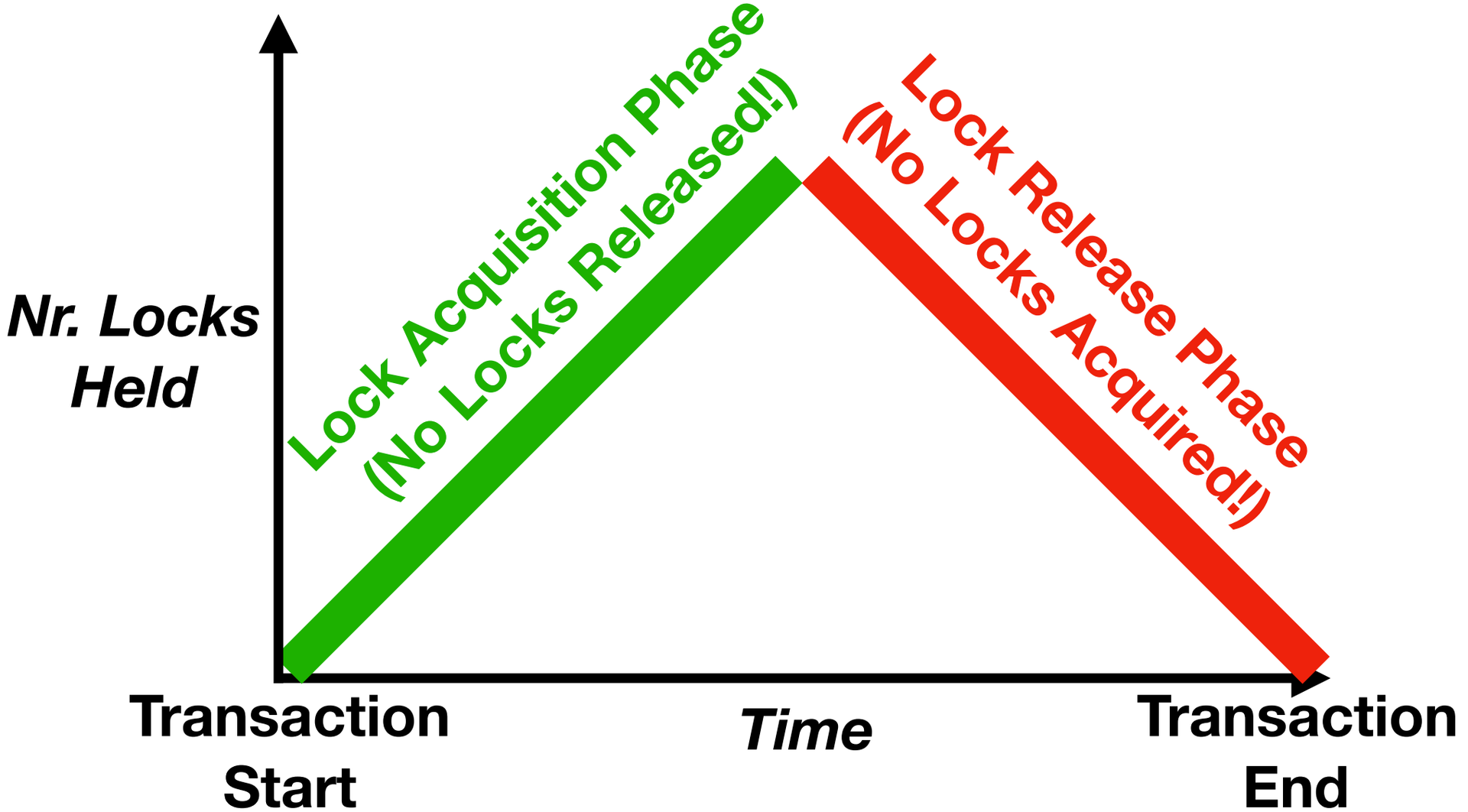
Acquire Locks Late

- Acquire locks **directly before** read or write operation
  - (So far: acquired all locks at transaction start)
- May improve performance by **increasing parallelism**
- May however lead to **deadlocks**:
  - Transaction 1 acquires **lock on A**, now **waiting for B**
  - Transaction 2 acquires **lock on B**, now **waiting for A**
  - Transaction are both waiting for each other, **no progress**

Two-Phase Locking

- **Combines all** of the aforementioned optimizations
  - **Fine-grained locks** on single objects
  - Distinguishes different **lock types**
  - Locks may be **acquired late** (depends on 2PL variant)
  - Locks may be **released early** (depends on 2PL variant)
  - But **restrictions** on when locks are acquired/released

The Two Phases of 2PL



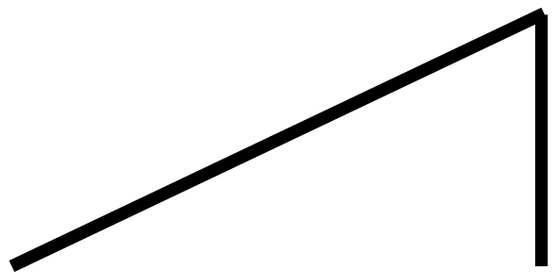
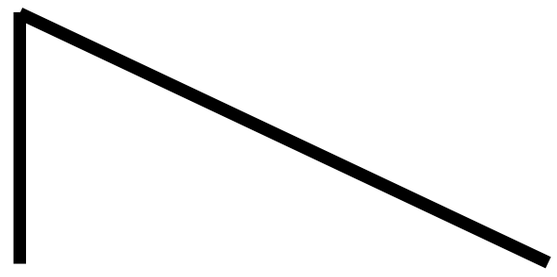
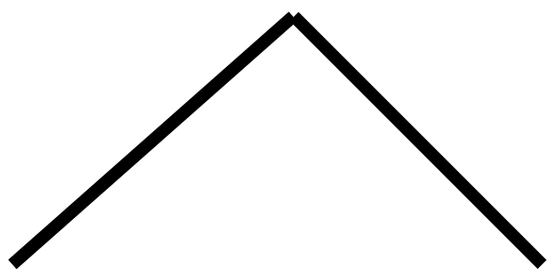
Two Phases Summary

- Each transaction has **two separate phases** with 2PL
- First phase: transaction may **acquire** locks but no release
- Second phase: transaction may only **release** locks
- Will see later that this restriction is necessary!
  - Guarantees **conflict-serializable** schedules

Two Phase Locking Variants

- **Conservative 2PL:** acquire all locks at transaction start
- **Strict 2PL:** release all locks at transaction end
- Can also **combine** the two (conservative strict 2PL)
- **Plain 2PL** makes no restrictions on locking periods

Illustration of 2PL Variants

	Conservative	Non-Conservative
Strict		
Non-Strict		

Pros and Const of Variants

- Being non-conservative or non-strict is **more permissive**
  - Allows more transactions to proceed **in parallel**
- **Conservative** 2PL prevents deadlocks
- **Strict** 2PL prevents cascading aborts
- Optimal variant **depends on workload**
  - E.g., how likely are deadlocks and cascading aborts?

Analyzing 2PL Schedules

- Agreed on aiming for **conflict-serializable** schedules

- Will prove that 2PL **generates** such schedules

Proof Overview

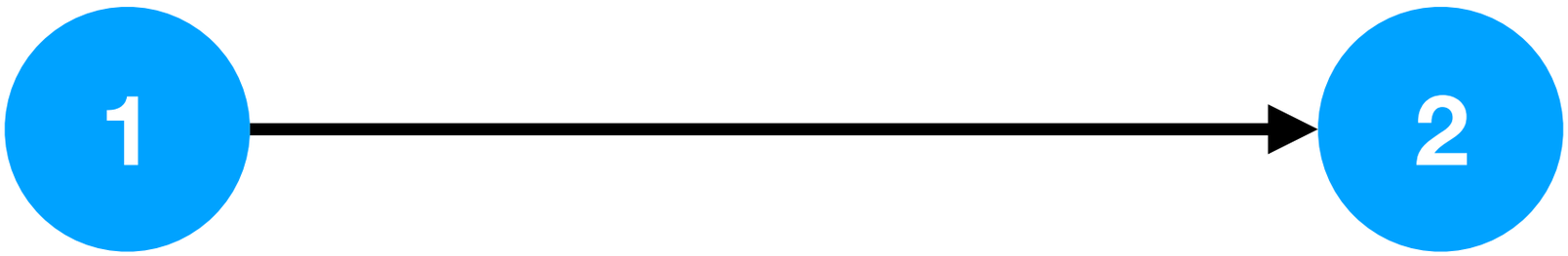
- Assume schedule was generated **using 2PL**
- Now imagine **conflict graph** of schedule
- Schedule is conflict serializable if it is **acyclic**
- Will show: assuming **cycle leads to contradiction**
  - Based on **lemma** introduced next

Release First Lemma

- Lemma: if conflict graph has **path from transaction T1 to transaction T2** then T1 **releases** some lock before T2 **acquires** some lock

- Will prove that **via induction**
  - **Induction start**: holds for paths of length 1
  - **Induction step**: from paths of length I to i+1

Induction Start



(Two transactions with conflict)

Induction Start

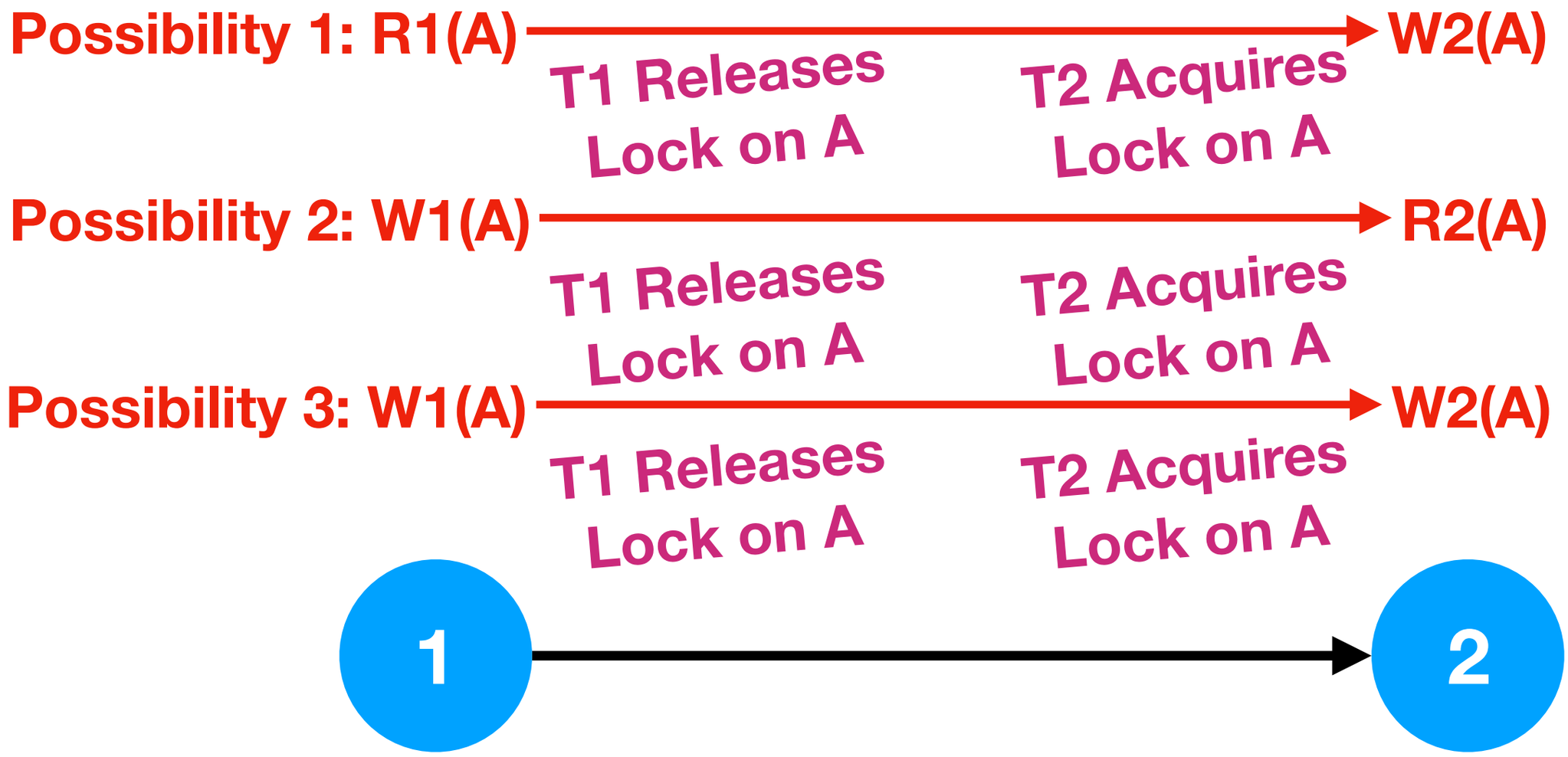
Possibility 1: R1(A) → W2(A)

Possibility 2: W1(A) → R2(A)

Possibility 3: W1(A) → W2(A)

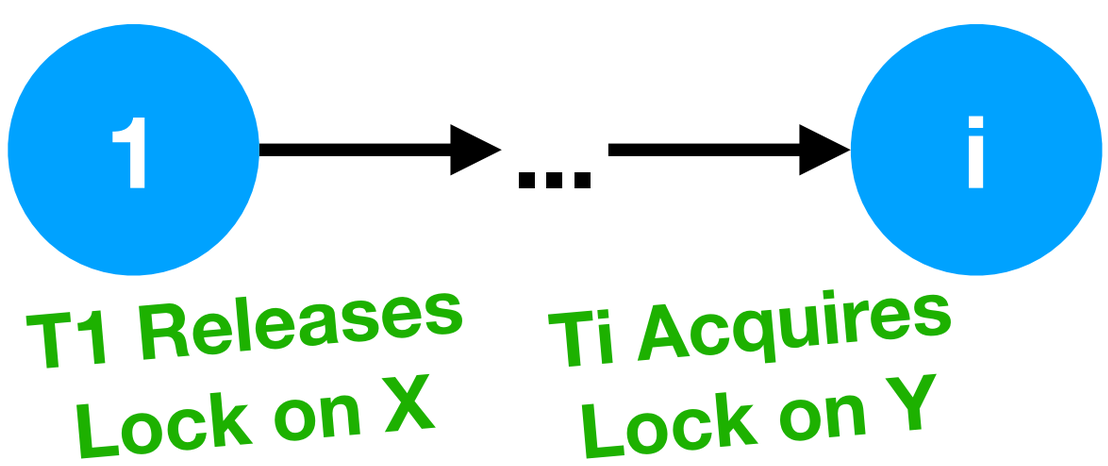
(Two transactions with conflict)

Induction Start

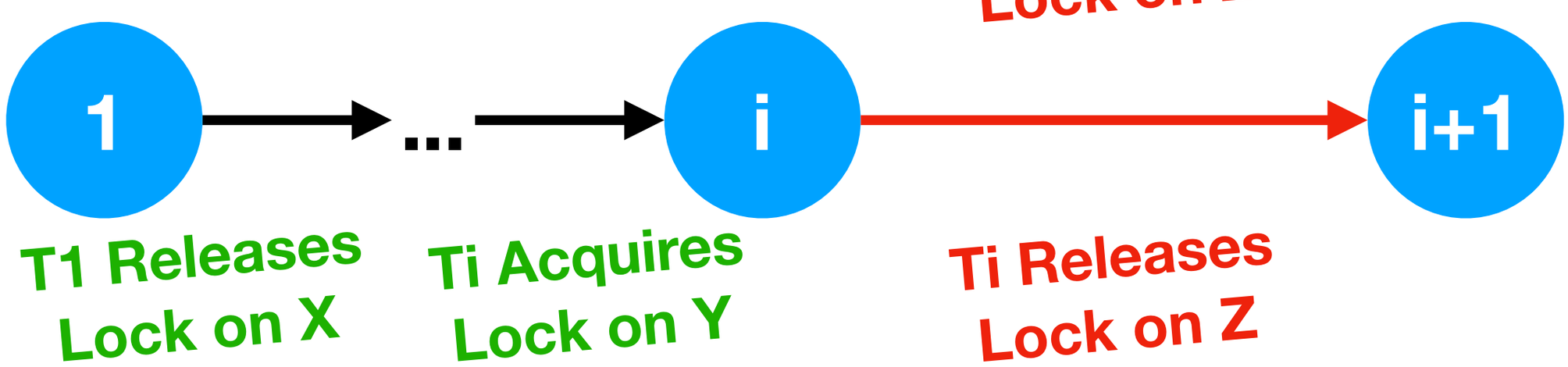


(Two transactions with conflict)

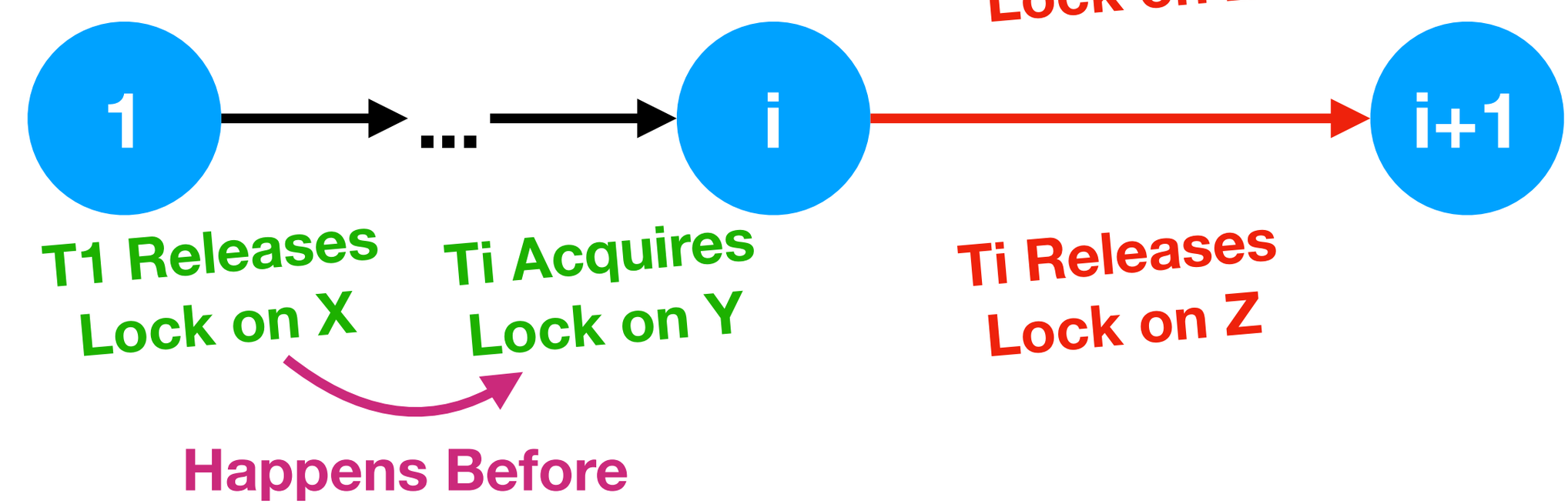
Induction Step



Induction Step

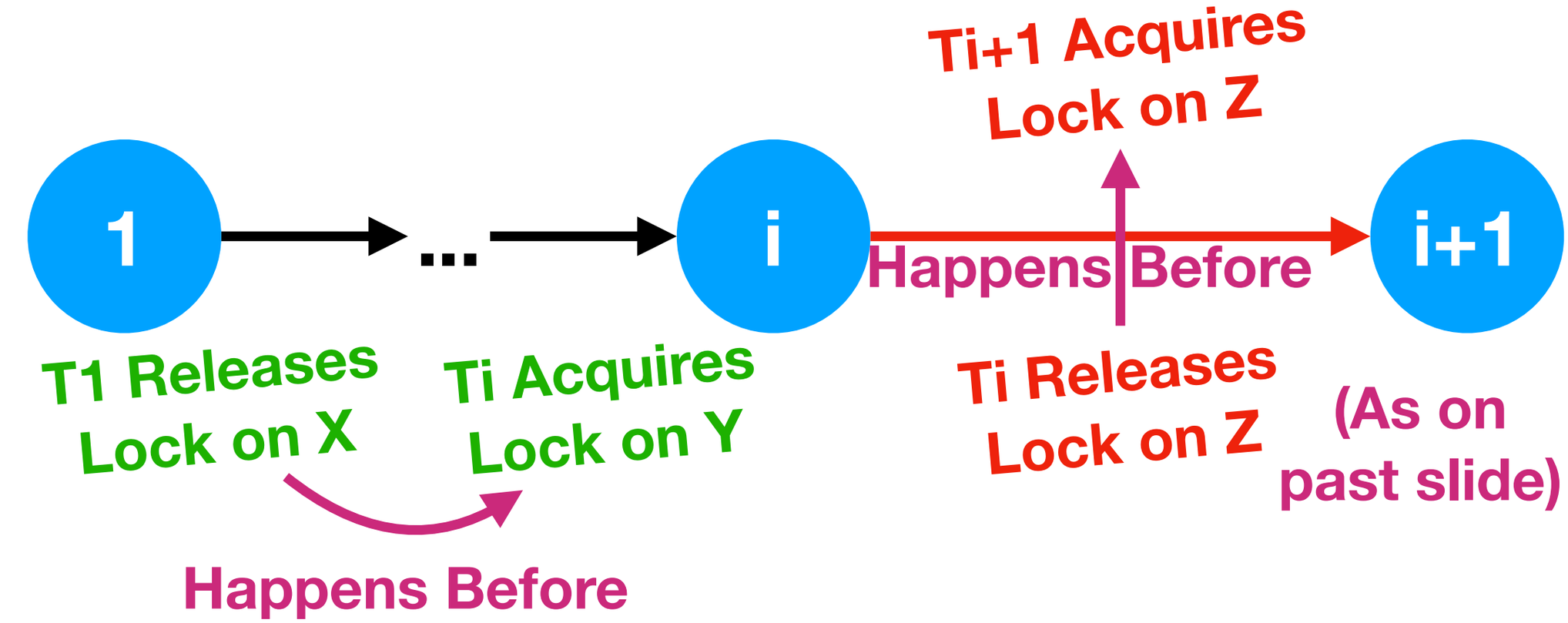


Induction Step



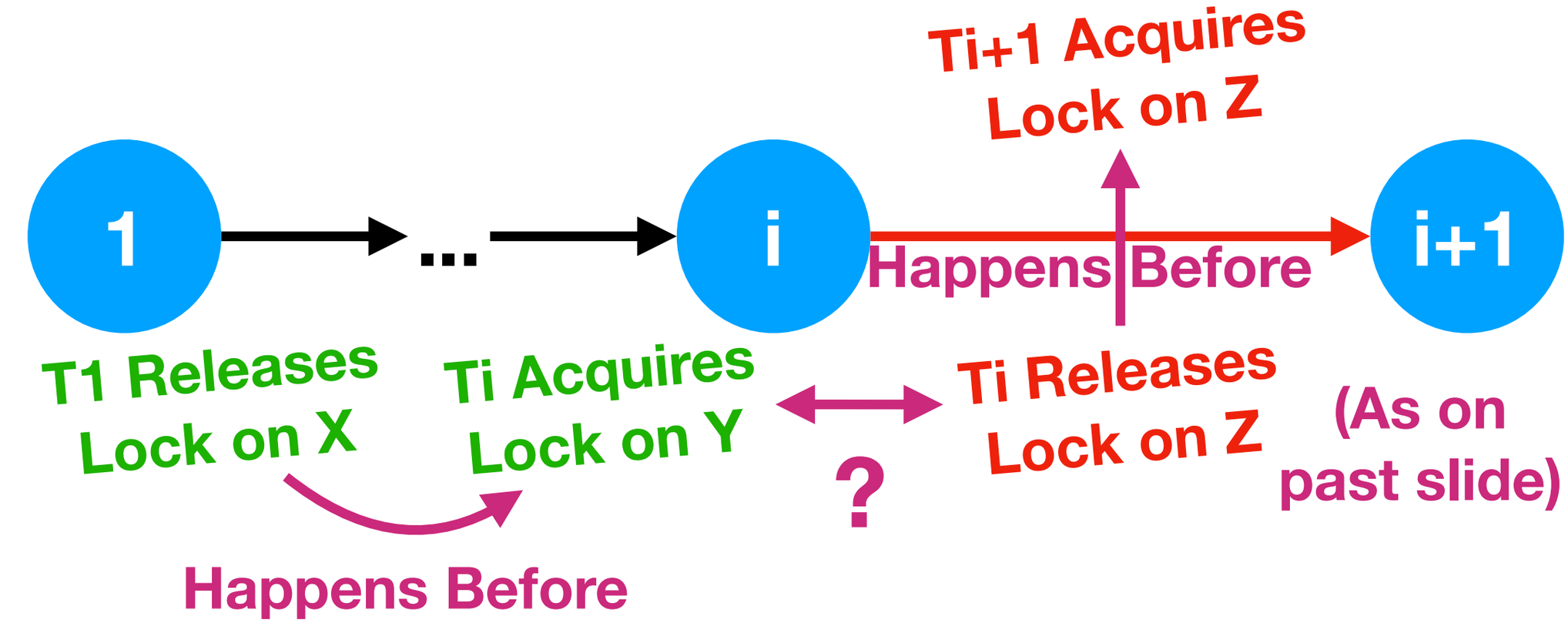
(Lemma for paths up to length i)

Induction Step



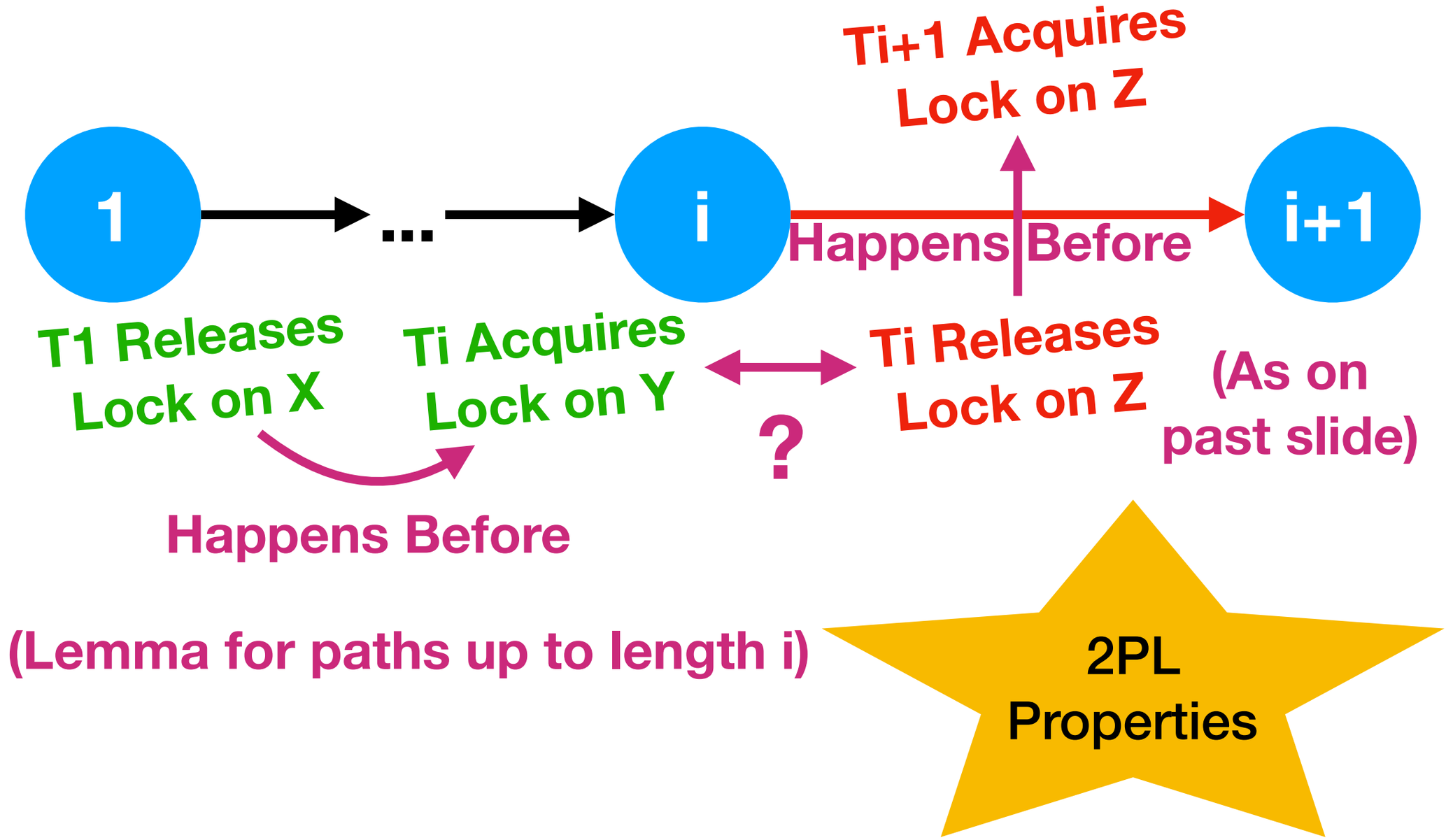
(Lemma for paths up to length i)

Induction Step

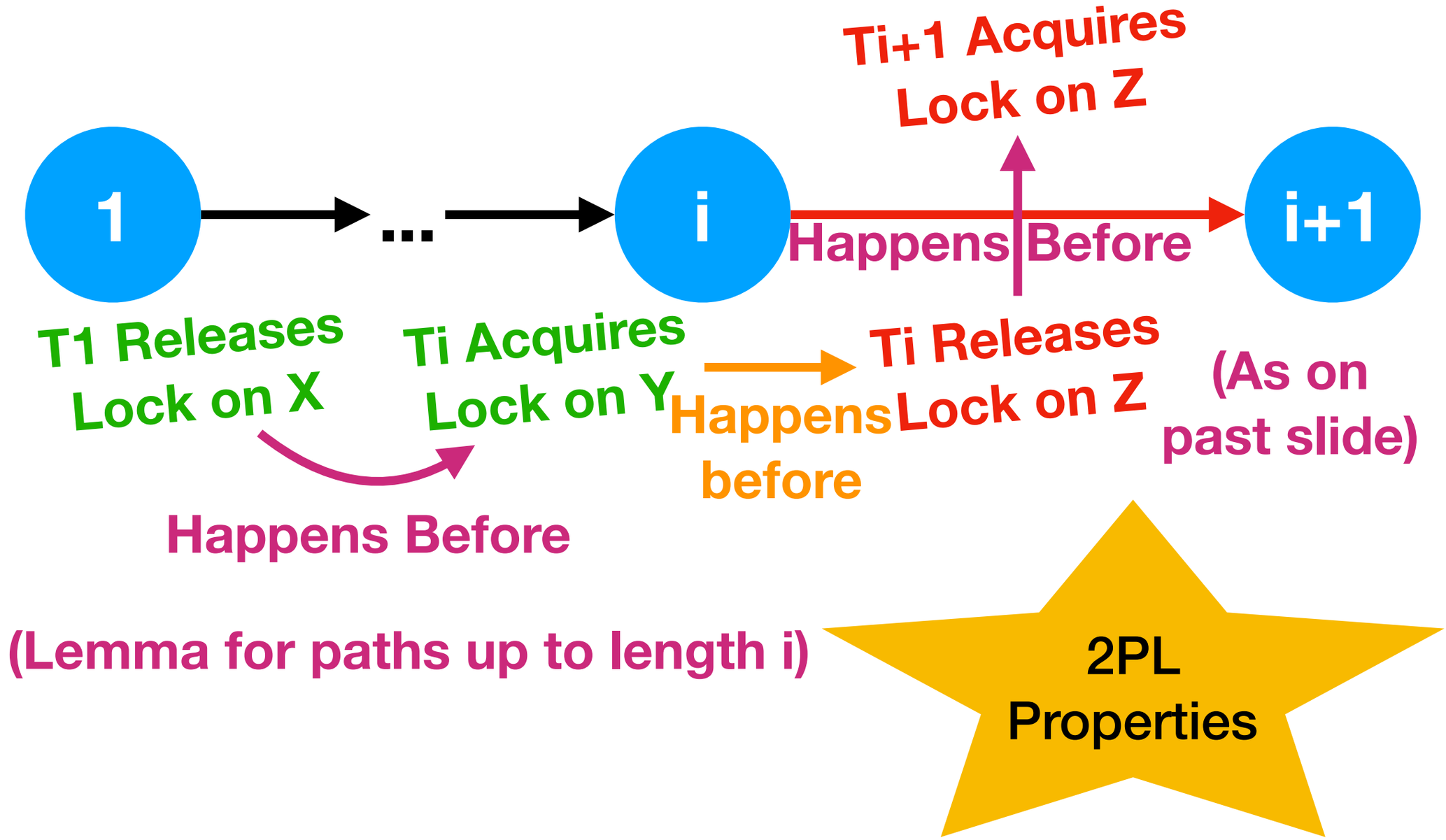


(Lemma for paths up to length i)

Induction Step



Induction Step



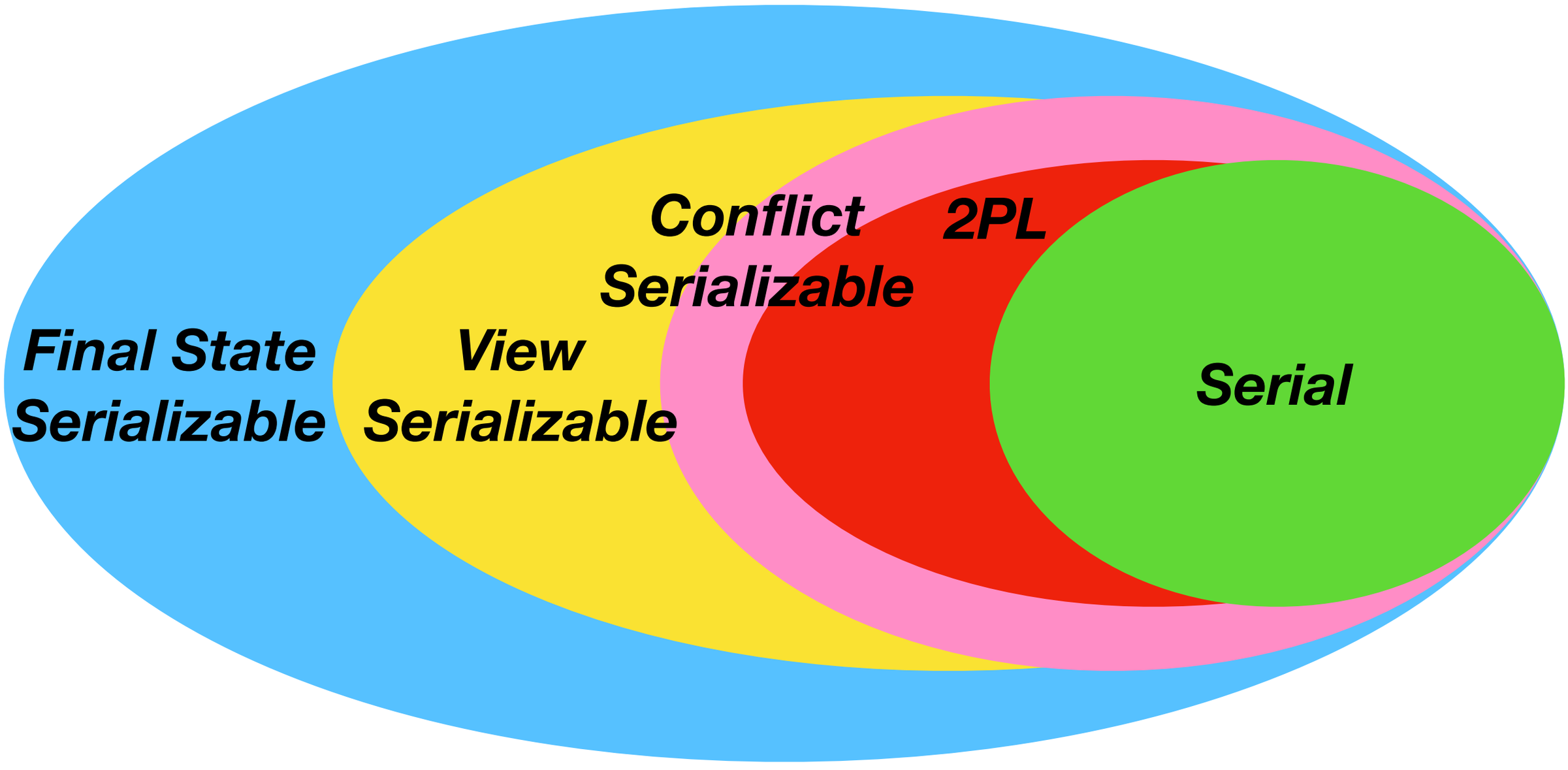
Wrapping Up Proof

- **Lemma**: path from T1 to T2 - T1 releases lock before T2 acquires lock
- **Cycle** means T1 releases lock before T1 acquires lock
- 2PL **does not acquire lock** after releasing them!
- Hence, we **cannot have a cycle** in conflict graph
- Hence, 2PL produces **conflict serializable** schedules

2PL vs. Conflict Serializable

- 2PL **only** produces conflict serializable schedules
- But can 2PL produce **all** conflict serializable schedules?
- The answer is "No" as **demonstrated** below:
  - **W1(A) R2(A) C2 R3(B) C3 W1(B) C1**
  - Conflict graph has three nodes, two edges → **no cycle**
  - ***Could this have been produced by 2PL?***

Classes of Schedules



All Schedules