

Unitedworld Institute Of Technology

B.Tech. Computer Science & Engineering

Semester : 3rd

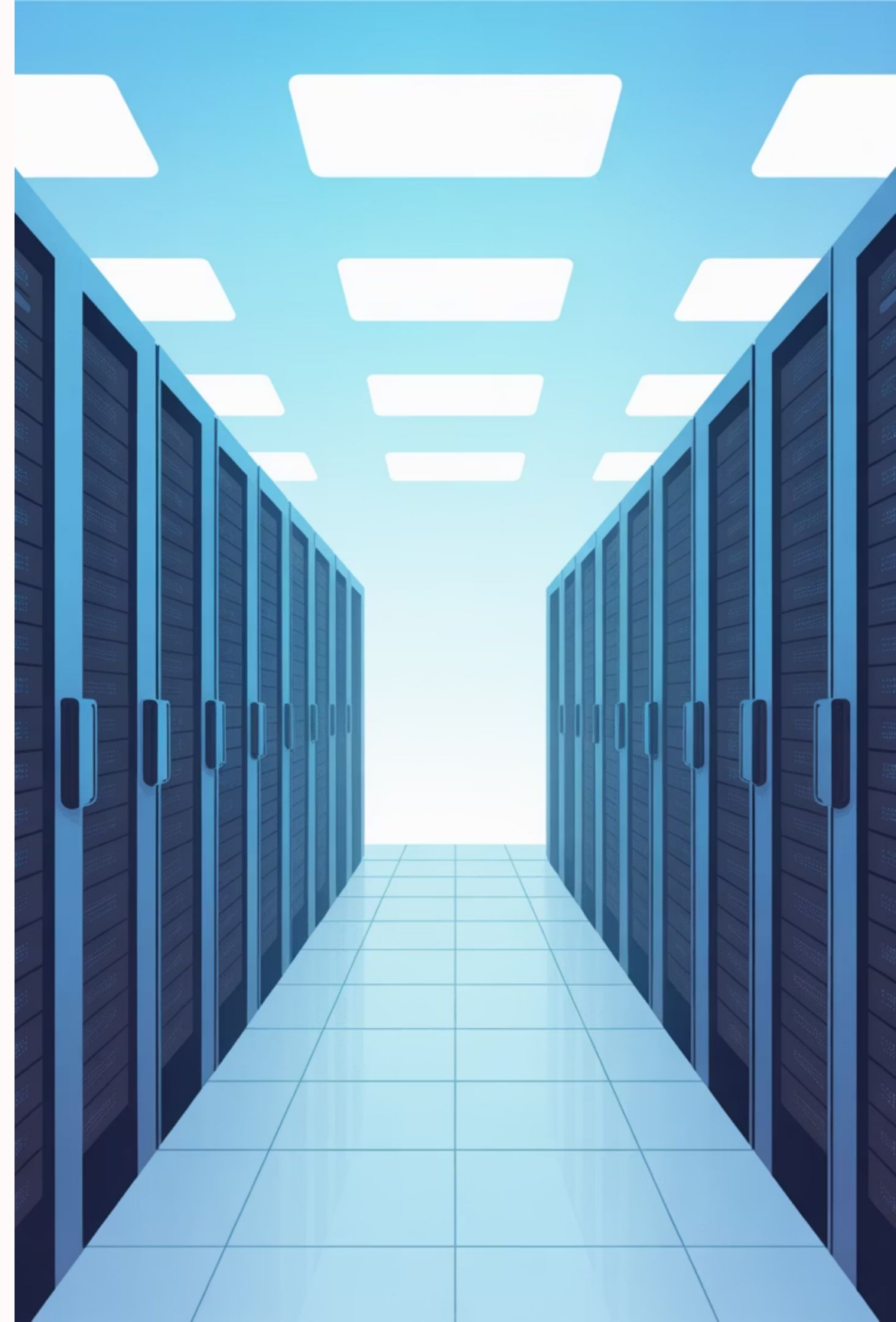
Introduction To Database Management System
Course Code: 71203002003

UNIT IV: TRANSACTIONS, CONCURRENCY CONTROL AND DEADLOCKS

Prepared by:
Mr. Utsav Kapadiya
Assistant Professor (UIT)

Concurrency Control – Lock Based Techniques

Understanding how databases handle multiple users safely and efficiently



What is Concurrency Control?

In today's world, databases aren't used by just one person at a time. Imagine hundreds or even thousands of users simultaneously booking concert tickets, transferring money, or updating records. When multiple transactions run at the same time in a database, they often need to access and modify the same pieces of data.

Without proper management, chaos ensues—data becomes inconsistent, calculations go wrong, and users see incorrect information. This is where Concurrency Control comes in: it's the traffic controller of the database world, ensuring that simultaneous operations don't crash into each other and corrupt your data.

Think of it as the rules that prevent two people from editing the same document at the exact same moment in conflicting ways. The database needs intelligent mechanisms to coordinate access and maintain data integrity.

The Problem: When Things Go Wrong

1

Both transactions start

T1 and T2 both read balance = ₹1000 from the same account at nearly the same time

2

T1 processes withdrawal

T1 withdraws ₹500 and calculates new balance as ₹500

3

T2 processes withdrawal

T2 withdraws ₹400 and calculates new balance as ₹600 (using the original ₹1000)

4

Conflicting results

Both write back their results—final balance might be ₹600 or ₹500, but never the correct ₹100!

📌 **The Lost Update Problem:** This classic scenario demonstrates why uncontrolled concurrent access leads to incorrect results. The correct final balance should be $₹1000 - ₹500 - ₹400 = ₹100$, but instead one withdrawal completely disappears from the calculation.

What is a Lock?



A Lock is a fundamental database mechanism—think of it as a temporary restriction or reservation placed on a specific piece of data (a row, a table, or even an entire database). When a transaction locks a data item, it's essentially putting up a "Do Not Disturb" sign. The lock prevents other transactions from accessing that data in ways that could cause conflicts. It's the database's way of saying: *"Hold on! Someone is currently working with this data. Please wait your turn, or you can look but not touch."*

Locks are automatically managed by the database management system (DBMS) based on the operations being performed, though developers can sometimes explicitly request them.

Goals of Locking



Prevent Conflicts

Stop transactions from interfering with each other's operations and overwriting changes



Maintain Consistency

Ensure data remains accurate and valid throughout all concurrent operations



Enable Safe Concurrency

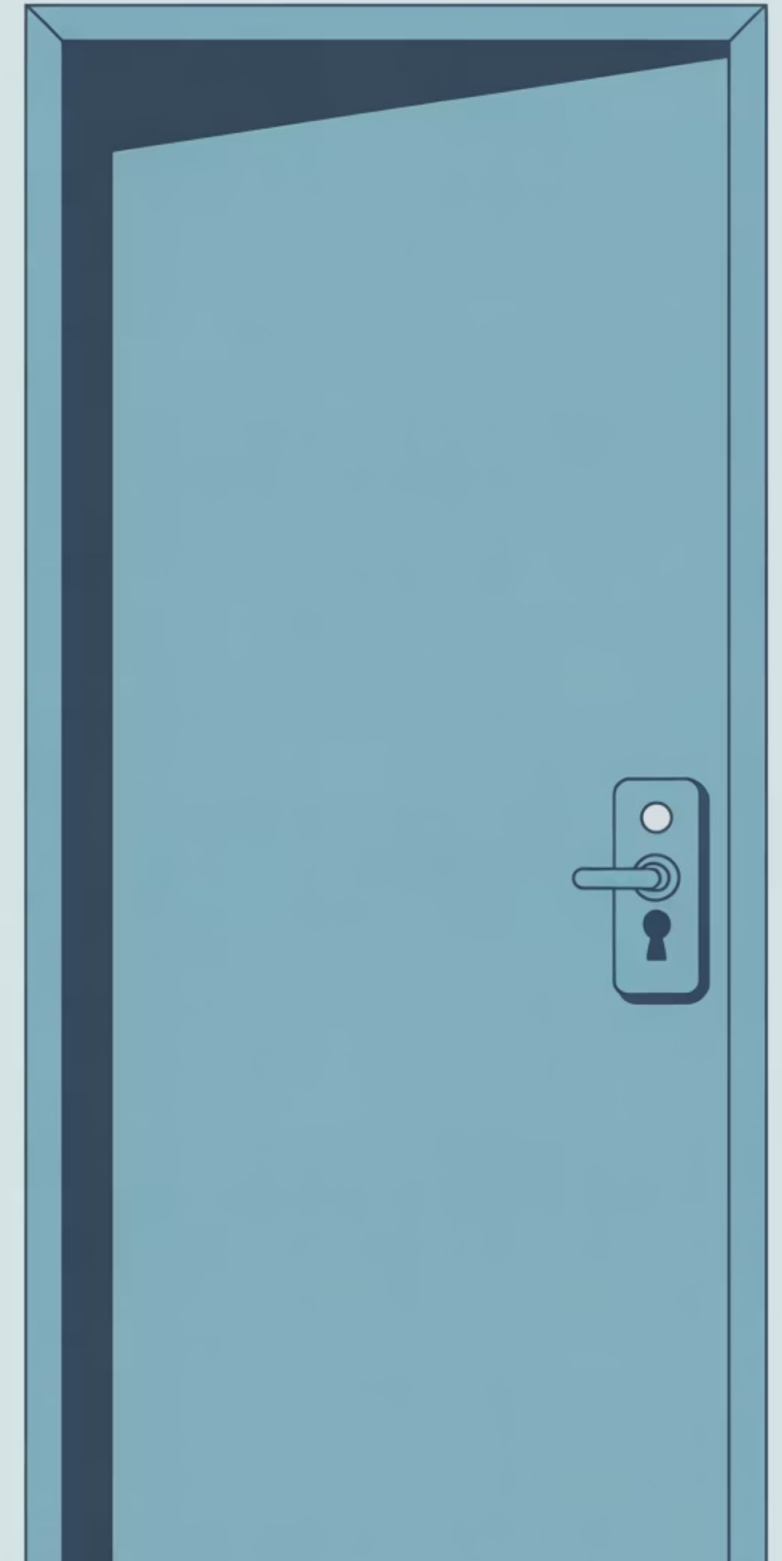
Allow multiple users to work simultaneously without compromising data integrity

Understanding Locks: The Bathroom Analogy

Here's an analogy that makes locks crystal clear: Imagine a shared bathroom in a college hostel,

- 1 Person Enters**
When Transaction T1 goes inside, they lock the door from the inside
- 2 Others Wait**
T2, T3, and other transactions must wait outside until T1 finishes and unlocks
- 3 Safe Usage**
Only one person uses the bathroom at a time, preventing chaos and conflicts

Without the lock, if two people tried to use the bathroom simultaneously, the result would be chaotic and unacceptable! The same principle applies to databases—locks ensure orderly, safe access to shared resources. Just as the bathroom lock prevents embarrassing collisions, database locks prevent data corruption.



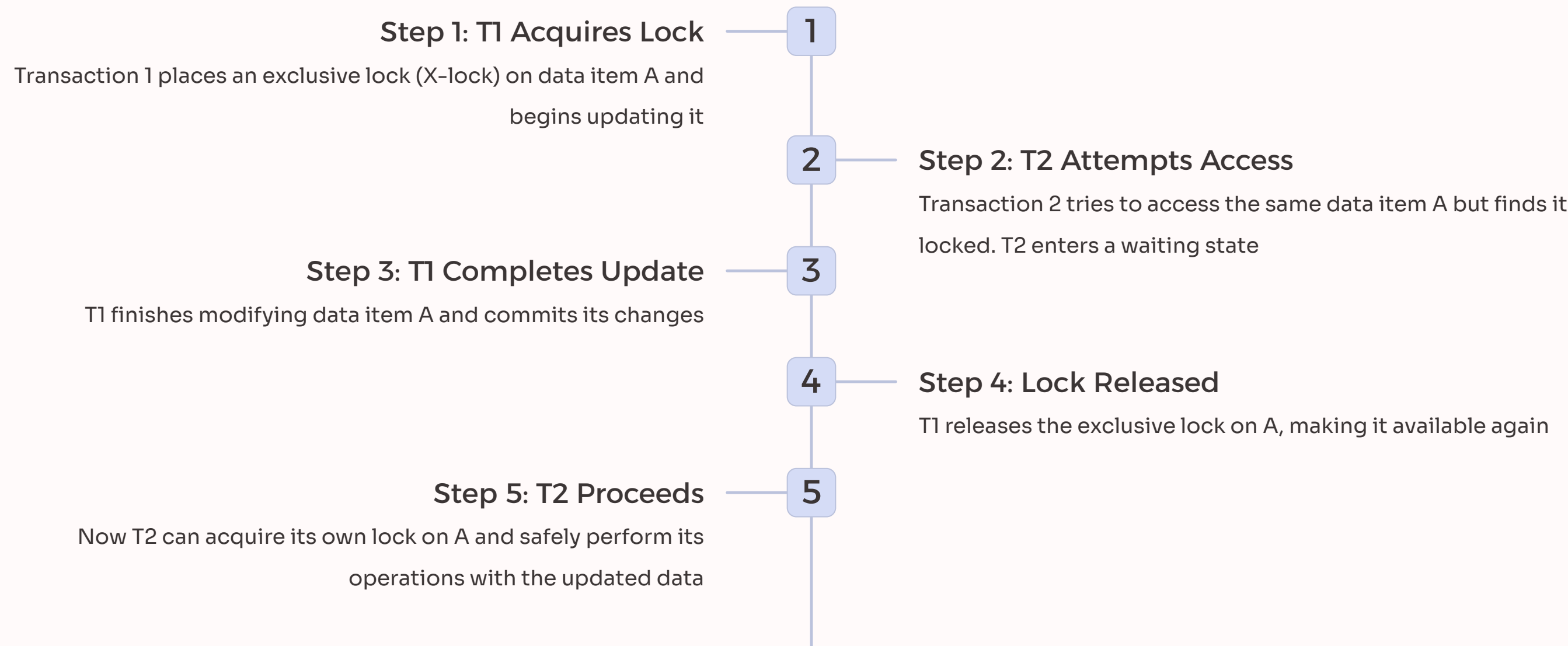
Types of Locks

Shared Lock (S-Lock)	Allows <i>only reading</i> of data. Multiple transactions can hold shared locks on the same data simultaneously since reading doesn't modify anything.	Multiple students can view exam results at the same time—everyone can read, but nobody can change the data.
Exclusive Lock (X-Lock)	Allows <i>both reading and writing</i> . No other transaction can access that data in any way—not even to read it—while an exclusive lock is held.	Only one administrator can update student marks at a time. Others must wait until the update is complete and the lock is released.

📌 **Key Insight:** Shared locks are compatible with each other (multiple readers can coexist), but exclusive locks are compatible with nothing—they demand solitude to safely modify data.

Lock Coordination in Action

Let's walk through a practical example showing how locks coordinate database access between two competing transactions:

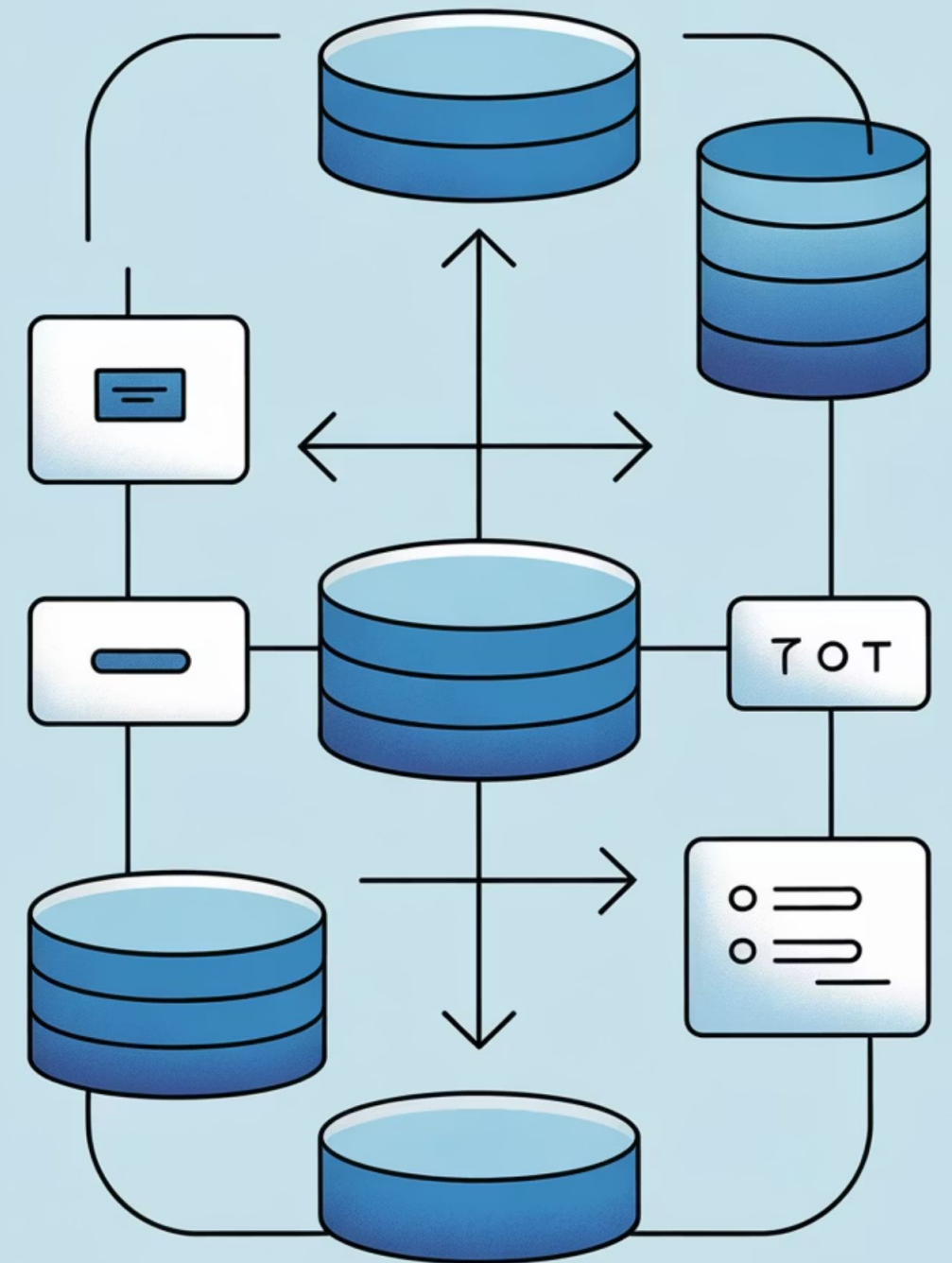


This orderly process ensures that T2 always works with the correct, updated data from T1. Without locks, T2 might read stale or partially-updated information, leading to incorrect results.

Lock-Based Protocols

Simply having locks isn't enough—we need rules and protocols for when to acquire them, how long to hold them, and when to release them. These protocols ensure that locking actually achieves its goals of consistency and serializability.

The most important and widely-used protocol is the Two-Phase Locking Protocol (2PL), which forms the foundation of concurrency control in most modern database systems. Let's explore how it works and why it's so effective.



Two-Phase Locking Protocol (2PL)

Phase 1: Growing Phase

The transaction can acquire (obtain) locks on data items as needed

It cannot release any locks during this phase

The number of locks held grows or stays the same

Phase 2: Shrinking Phase

The transaction can release locks that it no longer needs

It cannot acquire any new locks during this phase

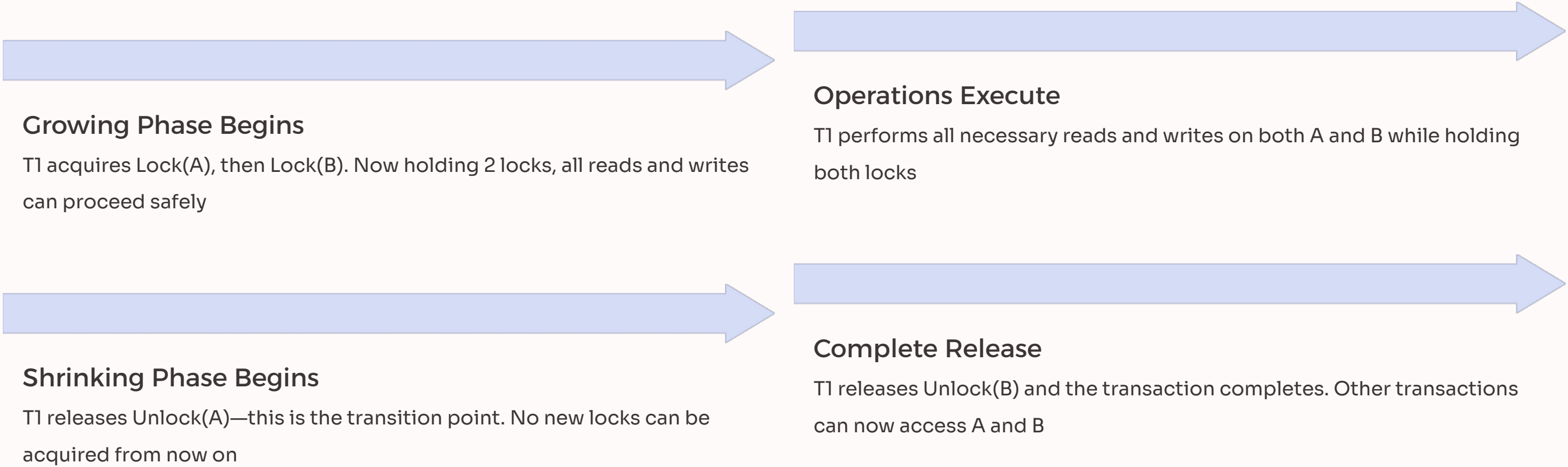
The number of locks held shrinks until none remain

📌 **The Critical Rule:** Once a transaction releases its first lock (entering the shrinking phase), it can never acquire another lock. This strict rule is what guarantees serializability.

2PL Example Walkthrough

Here's a concrete example of Two-Phase Locking in action with a transaction T1 that needs to work with data items A and B:

```
T1: Lock(A) → Lock(B) → Read(A) → Write(A) → Read(B) →      Write(B) → Unlock(A) → Unlock(B)
```



Notice the clear boundary: once T1 releases its first lock (A), it cannot request any new locks. This seemingly simple rule has profound implications for maintaining database consistency.

Why 2PL is Important

Key Benefits

Guarantees Serializability: Any schedule produced using 2PL will have the same result as if all transactions ran one after another in some sequential order

Prevents Lost Updates: Exclusive locks ensure that concurrent writes don't overwrite each other's changes

Prevents Dirty Reads: Transactions can't read uncommitted data that might be rolled back

Widely Implemented: Most commercial DBMS use variants of 2PL because it's proven and reliable



Serializability is the gold standard in database theory—it means the concurrent execution is equivalent to some serial execution, guaranteeing correctness.

Problems Prevented by Locking

Lost Update	Two transactions read the same value, both modify it, and the second write overwrites the first—one update is "lost"	Exclusive locks ensure only one transaction can write at a time, preserving all updates
Dirty Read	One transaction reads data that another transaction has modified but not yet committed—if that transaction rolls back, the read data was never valid	Transactions hold shared locks until commit, preventing reads of uncommitted data
Unrepeatable Read	A transaction reads the same data twice and gets different values because another transaction modified it in between	Locks prevent other transactions from modifying data while it's being read multiple times

These three problems are among the most common concurrency anomalies. Lock-based protocols effectively prevent all of them when properly implemented, which is why locking remains the dominant approach in production database systems.

Deadlock: The Dark Side of Locking

While locks solve many problems, they introduce a new challenge: **deadlock**. This occurs when two or more transactions are waiting for each other to release locks, creating a cycle of waiting that can never resolve itself.

Deadlock Scenario

- Transaction T1 locks data item A and needs B
- Transaction T2 locks data item B and needs A
- T1 waits for T2 to release B
- T2 waits for T1 to release A
- Both wait forever = Deadlock!

DBMS Solutions

Deadlock Detection: The DBMS periodically checks for cycles in the wait-for graph and breaks them by aborting one transaction

Timeout: If a transaction waits too long for a lock, the system assumes deadlock and automatically aborts it

Prevention: Some systems require transactions to acquire all locks upfront, avoiding circular waiting

Quick Reference Summary

Lock	A mechanism that restricts access to data items to prevent conflicts
Shared Lock (S-Lock)	Multiple transactions can read, but none can write
Exclusive Lock (X-Lock)	Only one transaction can read or write; all others are blocked
Two-Phase Locking (2PL)	Protocol with a growing phase (acquire locks) and shrinking phase (release locks)
Deadlock	Circular waiting condition where transactions wait for each other indefinitely
Serializability	Concurrent execution produces the same result as some serial execution

Real-Life Example: College ERP System

Let's see lock-based concurrency control in a familiar context: a college ERP (Enterprise Resource Planning) system where multiple faculty members access student records.

The Scenario:

- Faculty A opens the record for student Roll_101 to update midterm exam marks
- The system places an exclusive lock on this record
- At the same moment, Faculty B tries to update the same student's assignment marks

Faculty B sees a message: *"Record is locked by another user. Please try again later."*

- Once Faculty A saves and closes, the lock releases
- Now Faculty B can access and update the record safely



This is **lock-based concurrency control in action**—transparent to users but crucial for preventing errors like overwriting updates or calculating incorrect GPAs. Without locks, Faculty B might accidentally overwrite Faculty A's changes, or both might be working with outdated totals.

Benefits of Lock-Based Concurrency Control



Maintains Data Consistency

Ensures all transactions see a coherent, valid view of the database without anomalies or corruption



Prevents Lost Updates

Guarantees that concurrent writes don't overwrite each other, preserving all changes



Prevents Dirty Reads

Stops transactions from reading uncommitted data that might be rolled back



Enables Safe Multi-User Access

Allows hundreds or thousands of users to work simultaneously without conflicts



Maintains Serializability

Guarantees that concurrent execution is equivalent to some serial order, the gold standard for correctness



Industry Standard

Widely implemented in major DBMS like Oracle, MySQL, PostgreSQL, and SQL Server—proven and reliable

Lock-based concurrency control remains the foundation of modern database systems because it provides a practical, proven solution to the complex challenge of coordinating simultaneous access to shared data. While newer optimistic approaches exist, locking's reliability and predictability make it indispensable for systems where consistency is paramount.

The background features a complex, abstract design. It includes several overlapping, wavy, horizontal bands in shades of light blue and grey. Superimposed on these bands is a network diagram consisting of numerous small, circular nodes in white, light blue, and grey. These nodes are interconnected by thin, grey lines that form a web-like structure across the entire image. The overall aesthetic is modern and technical.

Thank You!