# CMPT 606

Read Chapters 20 and 21

# Database Concurrency Control

**Dr. Abdelkarim Erradi**

Department of Computer Science and Engineering

**QU**

# Outline

- Transaction Management
- Concurrency control
- Lock-based Concurrency control
- Transaction Isolation Levels
- Optimistic Concurrency Control

# Transaction Management

< Back

# Concept of Transaction

- Transaction = sequence of one or more read / write operations (e.g., SQL queries) on a shared database to perform a **single** logical unit of work that must either commit or abort
    - Transfer money from one bank account to another
    - Checkout: place order and process payment
- They are the basic unit of change in a DBMS. Partial transactions are not allowed.
- Transaction boundaries are defined by the database user / application programmer
- **Atomicity, Consistency and Isolation are achieved using DB Transactions**

# Definition

- Database = a set of named data objects (A;B;C)
- Transaction = a sequence of read and write operations (R(A); W(B))
- The outcome of a transaction is either COMMIT or ROLLBACK:
  - If COMMIT, all of the transaction’s modifications are saved to the database.
  - If ROLLBACK, all of the transaction’s changes are undone as if the transaction never happened.

# SQL Transaction

- By default, each SQL statement (any query or modification of the database or its schema) is treated as a separate transaction
- Transactions can also be defined explicitly

```
START TRANSACTION;
        <sequence of SQL statements>
COMMIT;
or ROLLBACK;
```

- COMMIT makes all modifications of the transaction permanent, ROLLBACK undoes all DB modifications made

# Correctness Criteria: ACID

- **Atomicity**: “All or Nothing”
- **Consistency**: the database is guaranteed to be consistent when the transaction completes
- **Isolation**: The execution of one transaction is isolated from that of other transactions.

  “As if alone”
- **Durability**: If a transaction commits, then its effects on the database persist.

# How ACID are Achieved?

- **Atomicity**:
  - **Logging**: DBMS logs all actions so that it can undo the actions of aborted transactions.
  - **Shadow Paging**: DBMS makes copies of pages and transactions make changes to those copies. When the transaction commits, the page made visible to others.
- **Consistency**: Transactions violating integrity constraints are rolled back
- **Isolation**: using concurrency control protocols:
  - **Pessimistic**: *locking* to avoid transactions conflict (readers block writers and writers block readers)
  - **Optimistic**: *Timestamp Ordering* or *Multi-Version Concurrency Control (MVCC)*. Assumes that conflicts between transactions are rare, so it deals with conflicts when they happen
- **Durability**: DBMS can either use **logging** or **shadow paging** to ensure that all changes are durable.

# Concurrency control

Back

# Concurrency Control

- Multiple **concurrent** transactions $T_1$, $T_2$, … may read/write Data Items $A_1$, $A_2$, … concurrently
- Concurrency Control is the process of managing concurrent operations performed on shared data so that data manipulation does not generate inconsistent databases or produce wrong results
- Objective
  - Maximise throughput (i.e., work performed)
  - Minimize response time
- Constraint: But we also would like **correctness**
  - Avoid **interference** between transactions

# Three Concurrency Anomalies

- **Lost update** (some changes to DB get overwritten)
  - Two transactions $T_1$ and $T_2$ both modify the same data
  - $T_1$ and $T_2$ both commit
  - Final state shows effects of only $T_1$ but not of $T_2$

- **Dirty read**
  - $T_1$ reads data written by $T_2$ while $T_2$ has not committed
  - If $T_2$ aborts then $T_1$ will have dirty data

- **Unrepeatable read**
  - Getting different values when a read operation is re-executed within a Transaction T

# Illustrative Example

- **Example (to illustrate consistency issues that can be introduced by concurrent updates)**
- Ali at ATM1 withdraws $100
- Sara at ATM2 withdraws $50
- Initial balance = $400, final balance = ?
    - Should be $250 no matter who goes first

```
Read(balance);
If balance > withdrawalAmount {
    balance = balance - withdrawalAmount;
    Write(balance);

}
```

# No concurrent transactions scenario

Ali withdraws $100:

```
read balance;  => $400
if balance > amount then
    balance = balance - amount; => $300
    write balance; => $300
```

Sara withdraws $50:

```
read balance; => $300
if balance > amount then
    balance = balance - amount; => $250
    write balance; => $250
```

# Lost update problem

Ali withdraws $100:

Sara withdraws $50:

```
read balance; => $400
```

```
read balance; => $400
if balance > amount then
 balance = balance - amount; => $350
 write balance; => $350
```

```
if balance > amount then
  balance = balance - amount; => $300
  write balance; => $300
```

**Lost update problem** => DB is in inconsistent state

# Lost update problem

Ali withdraws $100:

Sara withdraws $50:

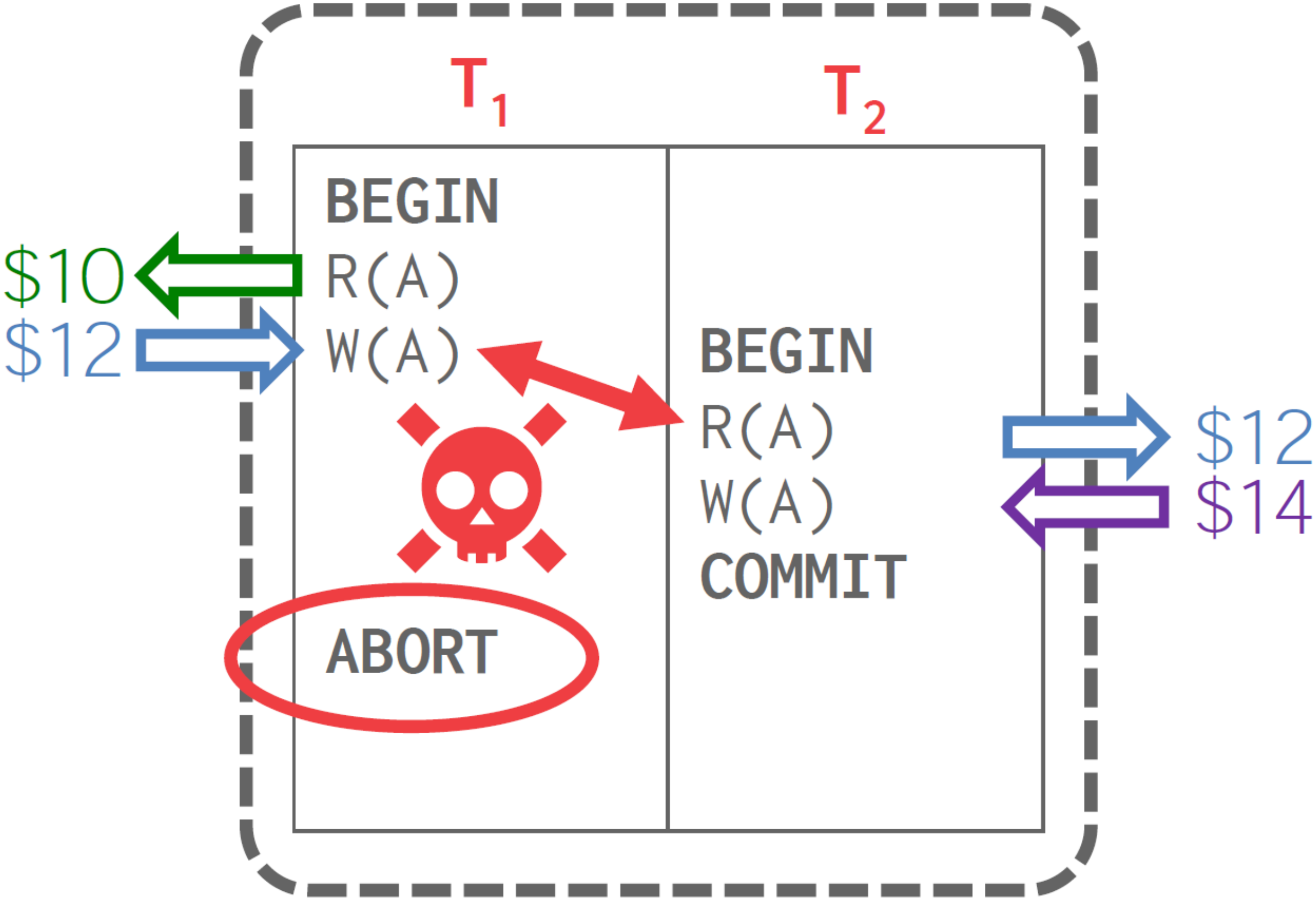
```
read balance; => $400
                                      read balance; => $400

if balance > amount then
  balance = balance - amount; => $300
  write balance; => $300

                                 if balance > amount then
                                   balance = balance - amount; => $350
                                   write balance; => $350
```

**Lost update problem** => DB is in inconsistent state

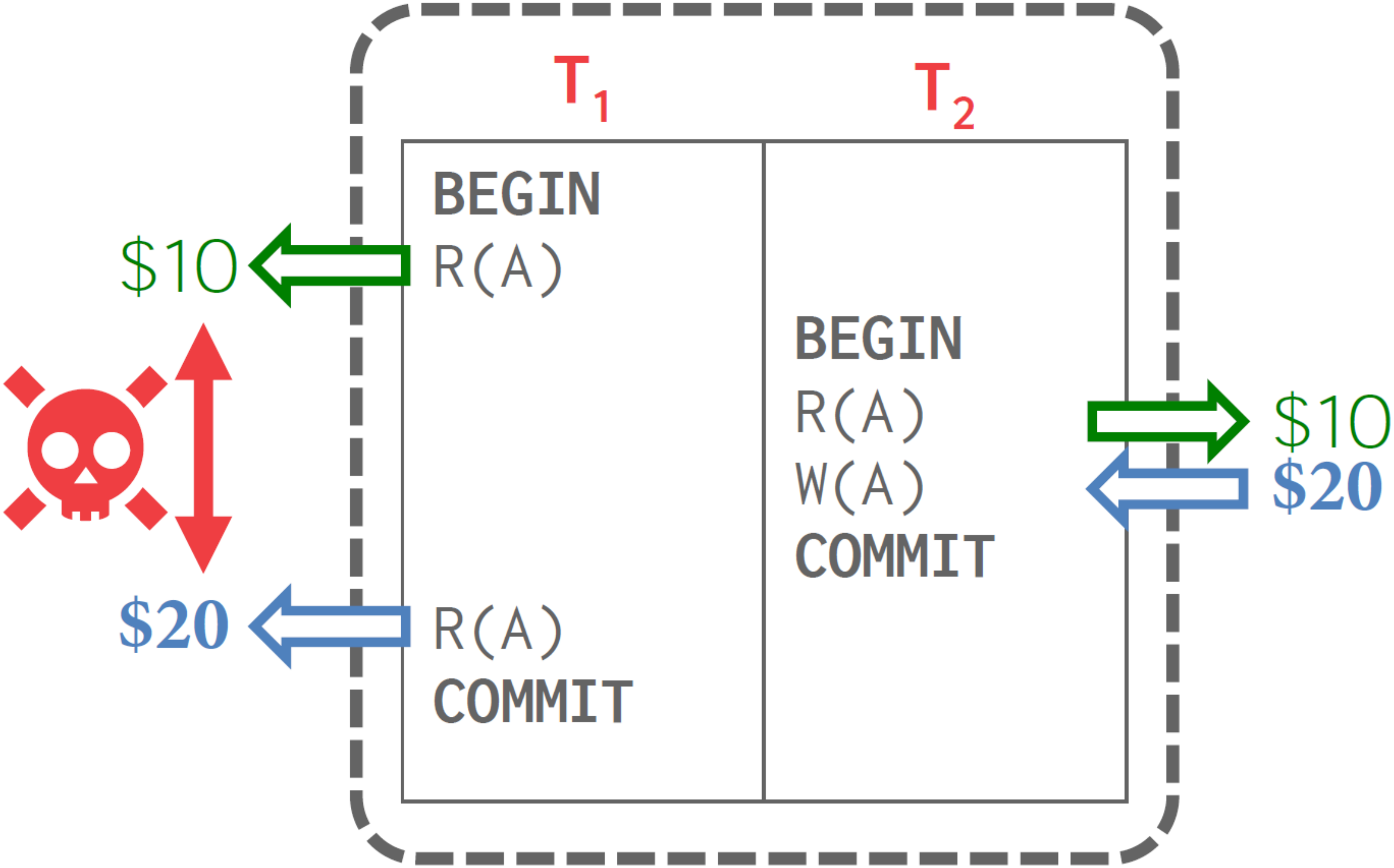
# Dirty Read



| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| R(A) | |
| W(A) | |
| | BEGIN |
| | R(A) |
| | W(A) |
| | COMMIT |
| ABORT | |

Source: CMU DB Course https://15445.courses.cs.cmu.edu/fall2019/

# Unrepeatable Reads



| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| R(A) | |
| | BEGIN |
| | R(A) |
| | W(A) |
| | COMMIT |
| R(A) | |
| COMMIT | |

Source: CMU DB Course https://15445.courses.cs.cmu.edu/fall2019/

# Serializable Schedule

- The order in which the DBMS executes operations is called an **execution schedule**
- The goal of a concurrency control protocol is to generate a correct (i.e., **Conflict-Serializable**) Schedule that is equivalent to some serial execution:
    - **Serial Schedule**: A schedule that does not interleave the operations of different transactions.
    - **Serializable Schedule**: A schedule with interleaving but is equivalent to some serial schedule

    => yields the same results as a serial schedule

# Read and Write Conflicting Operation

- Read-Write Conflicts (**R-W**)
- Write-Read Conflicts (**W-R**)
- Write-Write Conflicts (**W-W**)

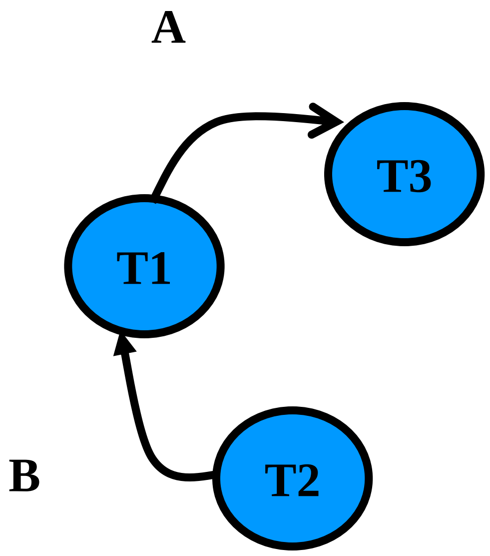
| *Operations of different transactions* | | *Conflict* | *Reason* |
|---|---|---|---|
| *read* | *read* | No | Because the effect of a pair of *read* operations does not depend on the order in which they are executed |
| *read* | *write* | Yes | Because the effect of a *read* and a *write* operation depends on the order of their execution |
| *write* | *write* | Yes | Because the effect of a pair of *write* operations depends on the order of their execution |

# Verifying Conflict-Serializability using a Precedence Graph

- Precedence Graph = {**Nodes**: transactions, **Edges**: r/w or w/w conflicts}
- The precedence graph for a schedule S contains:
  - A node for each transaction in S
  - An **edge** from $T_i$ to $T_j$ if an operation of $T_i$ **precedes and conflicts with** one of $T_j$'s operations.
- The **schedule S is serializable if and only if the precedence graph has no cycles**.

# Example 1

| T1 | T2 | T3 |
|---|---|---|
| Read(A) | | |
| … | | |
| Write(A) | | |
| | | Read(A) |
| | | … |
| | | Write(A) |
| | Read(B) | |
| | … | |
| | Write(B) | |
| Read(B) | | |
| … | | |
| Write(B) | | |



**serial execution?**

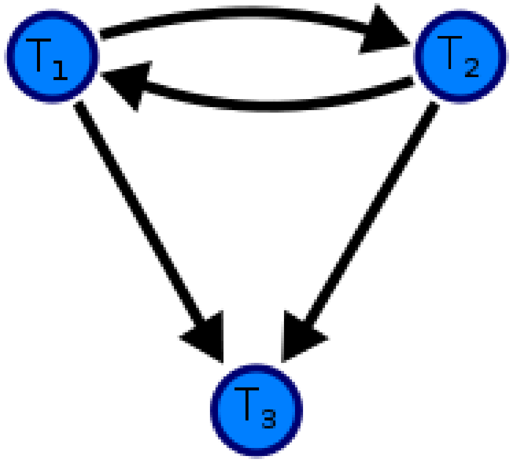
Conflict-serializable Schedule: T2, T1, T3

(Notice that T3 should go after T2, although it starts before it!) => always ‘read before write’

# Precedence Graph Example 2

$$D = \begin{bmatrix} T1 & T2 & T3 \\ R(A) & & \\ & W(A) & \\ & & \\ W(A) & & \\ & & W(A) \\ & & \end{bmatrix}$$

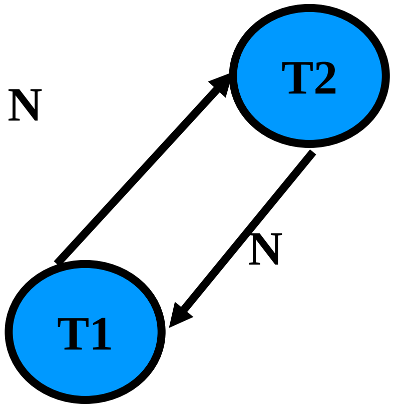
D = $R_1(A)\ W_2(A)\ W_1(A)\ W_3(A)$



**As $T_1$ and $T_2$ constitute a cycle the above schedule is not conflict-serializable.**

# Example 3 - ‘Lost-update’ problem

| T1 | T2 |
|---|---|
| Read(N) | |
| | Read(N) |
| N=N-1 | |
| | N= N-1 |
| Write(N) | |
| | Write(N) |



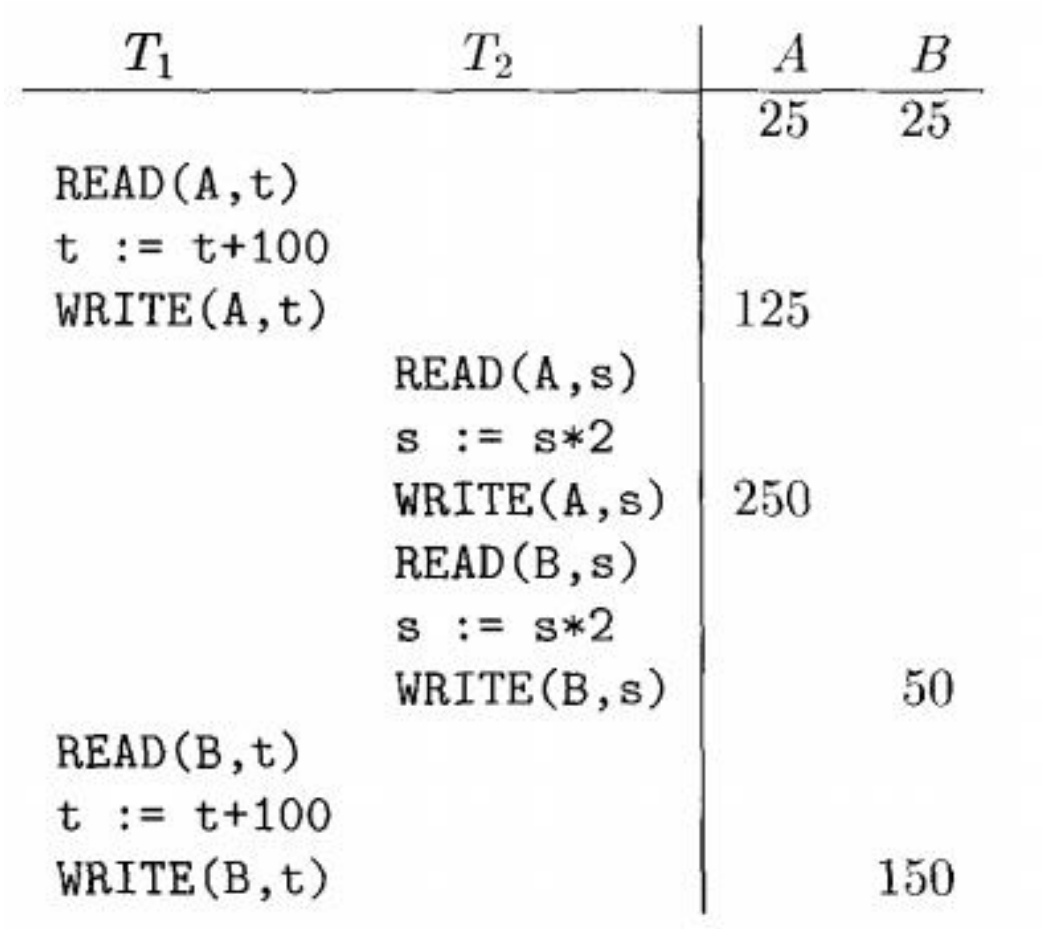
**Cycle -> not conflict-serializable**

**Not equivalent to any serial execution (why not?)**

**We can draw a precedence graph to prove this**

***Here, $T_1$ changes N, hence $T_2$ should have either run first (read and write) or after (reading the changed value)***

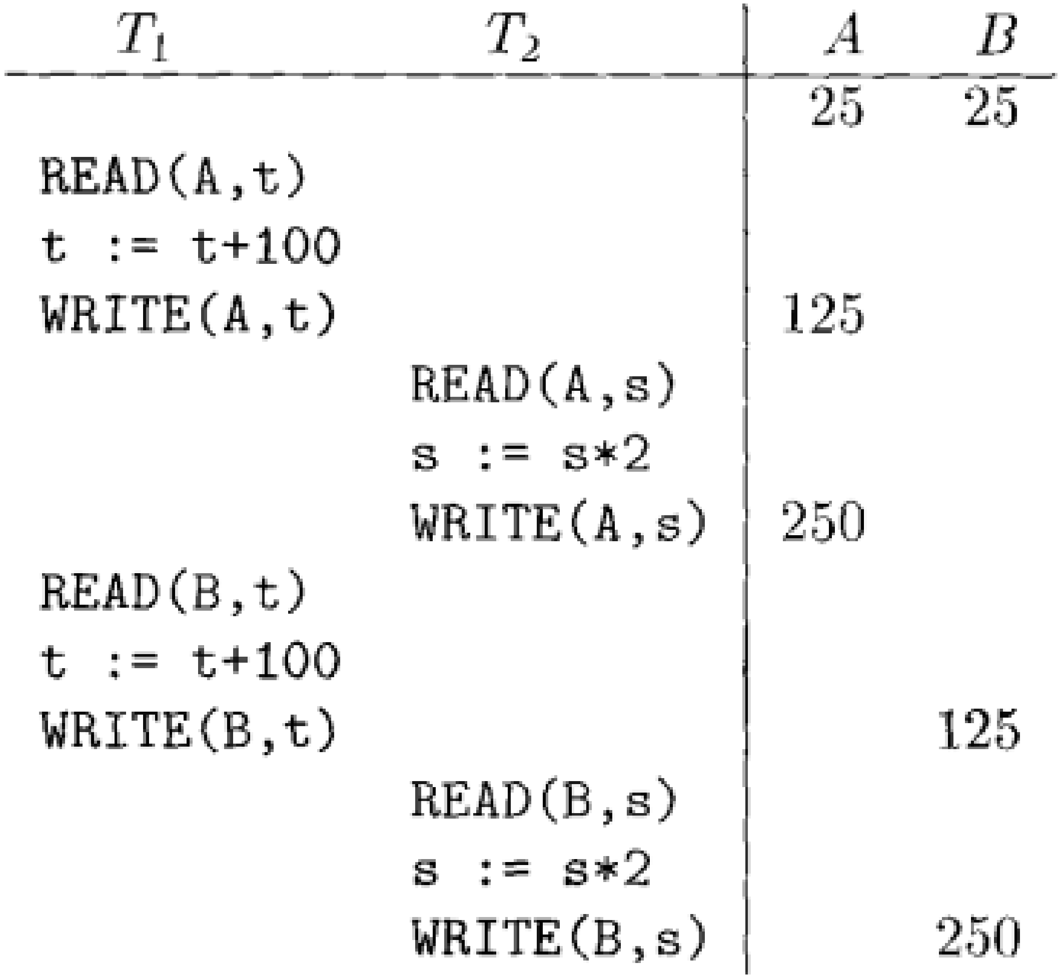
# Non-serializable schedule



| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| READ(A,t) | | | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| | READ(A,s) | | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 50 |
| READ(B,t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 150 |

**Non-serializable schedule as the precedence graph has a cycle**

# A serializable schedule



| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| READ(A,t) | | | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| | READ(A,s) | | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| READ(B,t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 125 |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 250 |

**A serializable schedule as the precedence graph is acyclic**

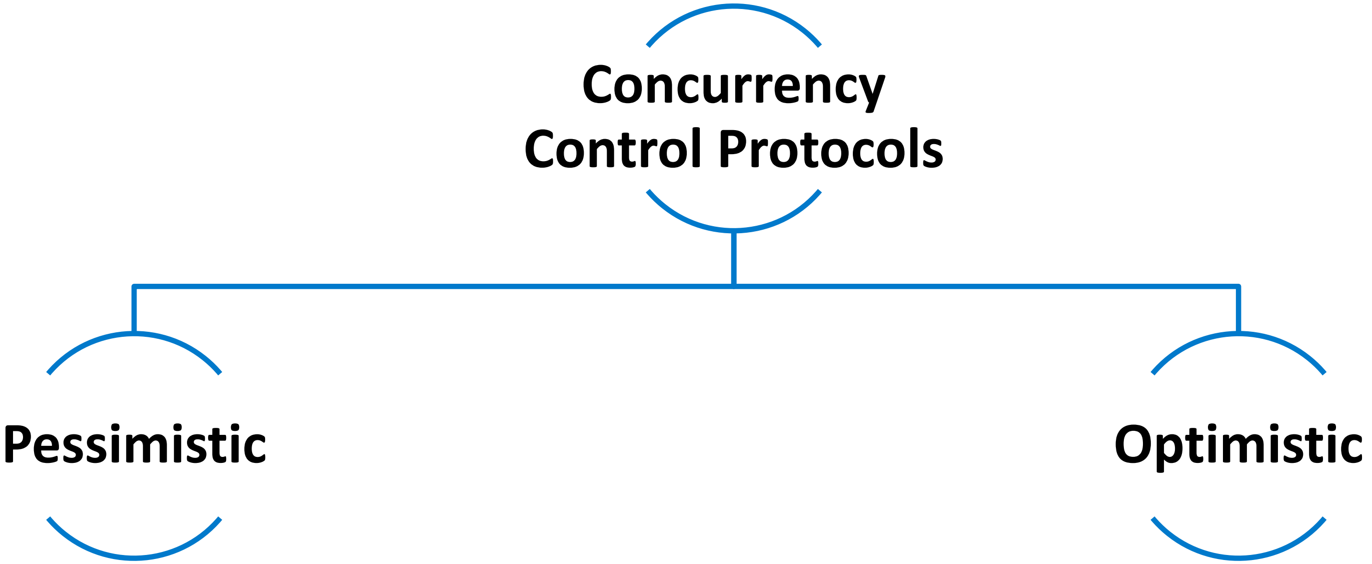
# Recap

- Concurrency control motivation
  - If only one transaction can execute at a time, in serial order, then performance will be poor.
- **Concurrency Control is a method for controlling or scheduling the operations of transactions in such a way that concurrent transactions can be executed safely**
  - without causing the database to reach an inconsistent state
- If we do concurrency control properly, then we can maximize *transaction throughput* while avoiding Concurrency Anomalies

# Lock-based Concurrency control

# Concurrency Control Protocols

- A concurrency control protocol is how the DBMS decides the proper interleaving of operations from multiple transactions



**Concurrency Control Protocols**

**Pessimistic**

- Assumes that transactions will conflict, so it doesn't let problems arise in the first place
- **Lock & change** current database state

**Optimistic**

- Assumes that conflicts between transactions are rare, so it chooses to deal with conflicts when they
- First perform changes in a private area & then change current database state

# Enforcing Serializability by Locks

- We need a way to guarantee that all execution schedules are correct (i.e., serializable) without knowing the entire schedule ahead of time
- Solution: Use **locks** to protect database objects
- **Locks** = most popular solution for concurrency control
- Lock manager: grants/denies lock requests
- Locking mechanisms prevent conflicts
  - Readers block writers
  - Writers block readers

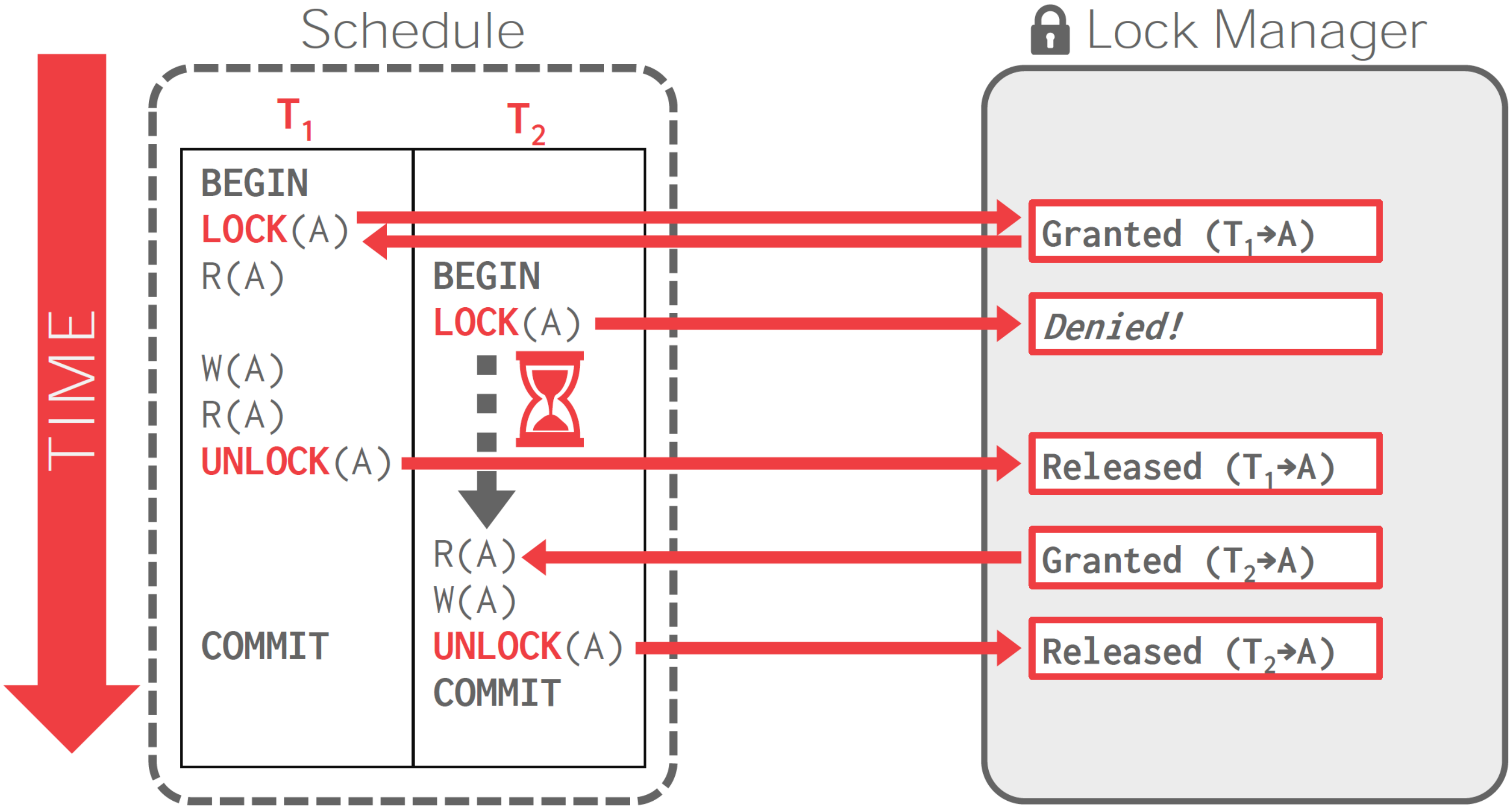
# How Locks Work?

1. Transactions request a lock from the **lock manager** before read or write operation
2. The lock manager grants the requested lock if not currently held by other transactions

Otherwise the requesting transaction blocks and waits for the lock

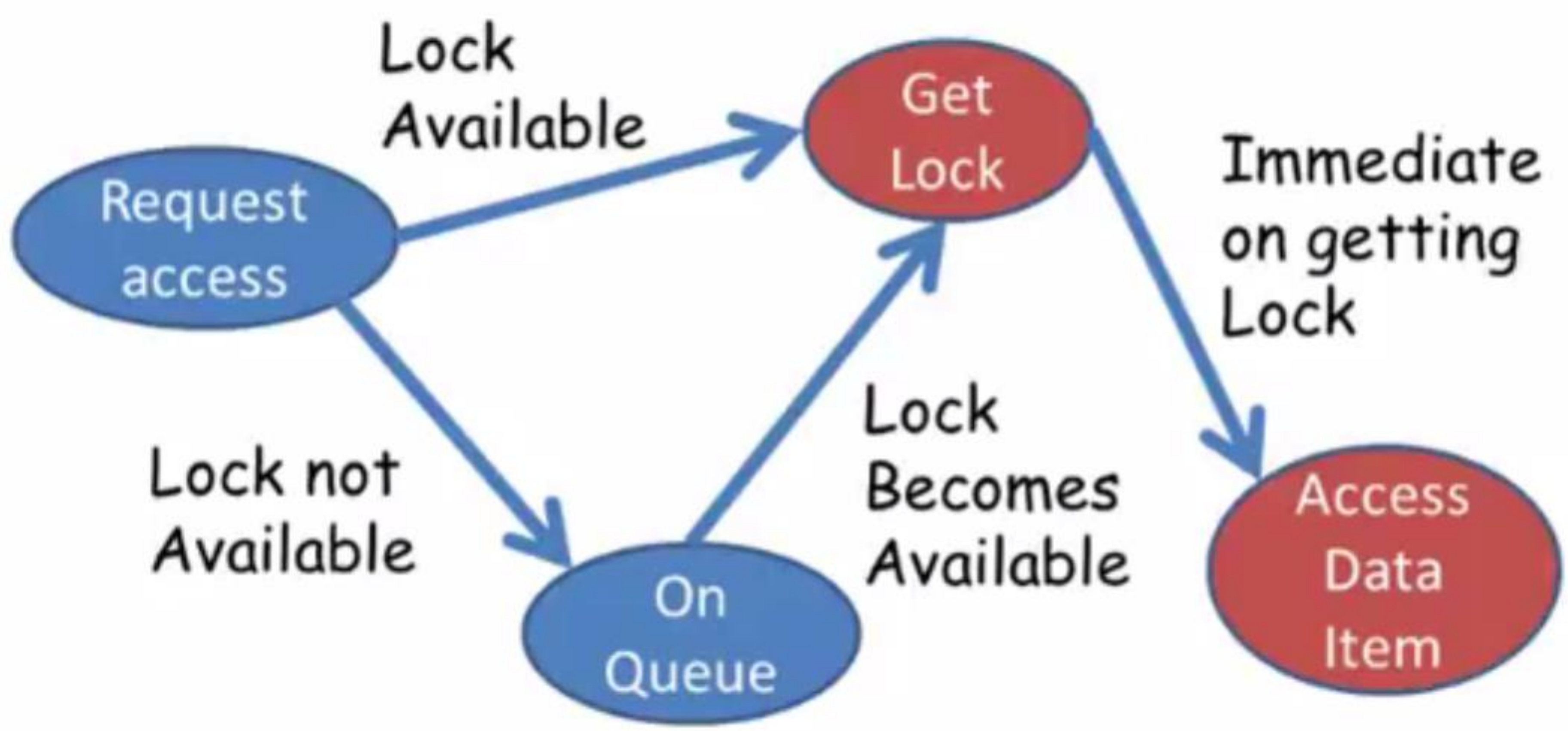
3. Transactions release locks when they no longer need them.
4. The lock manager updates its internal lock-table and then gives locks to waiting transactions.

# How Locks Work?



Source: CMU DB Course https://15445.courses.cs.cmu.edu/fall2019/

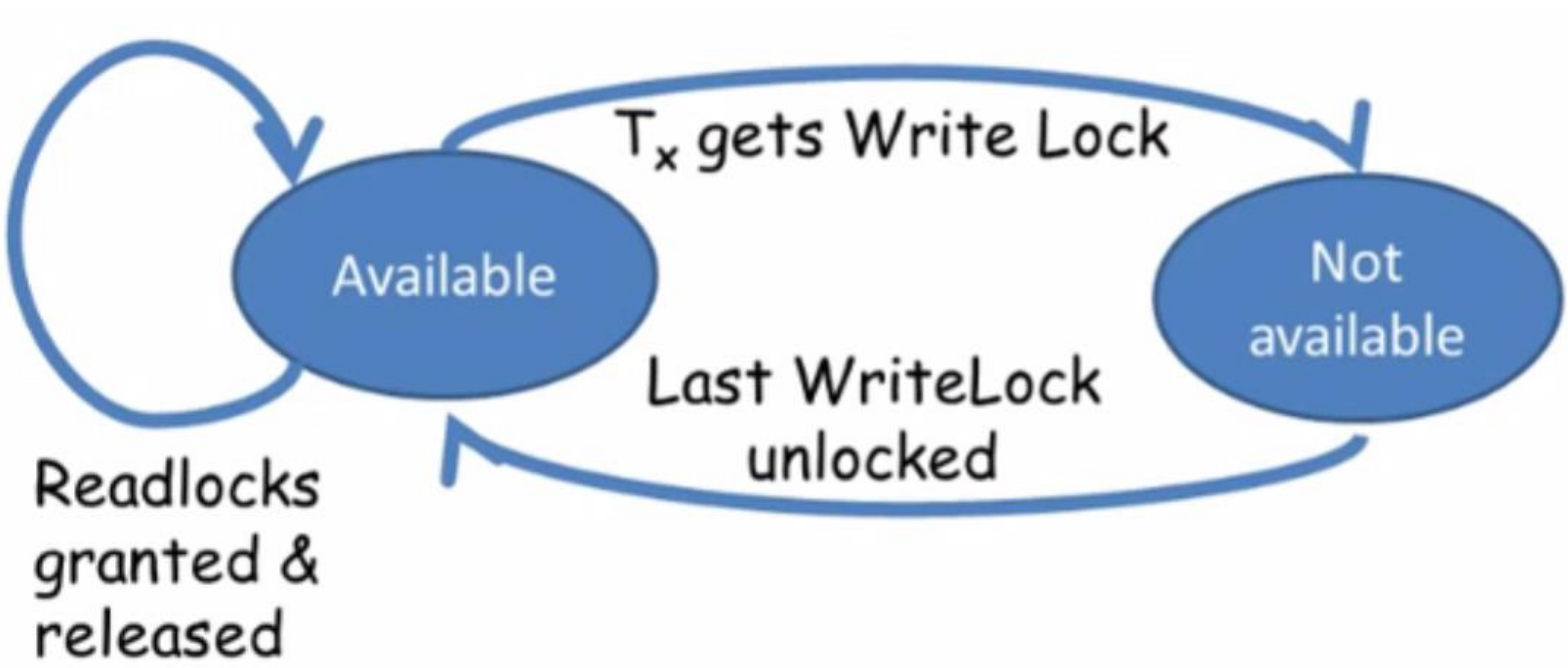
# Lock State Diagram



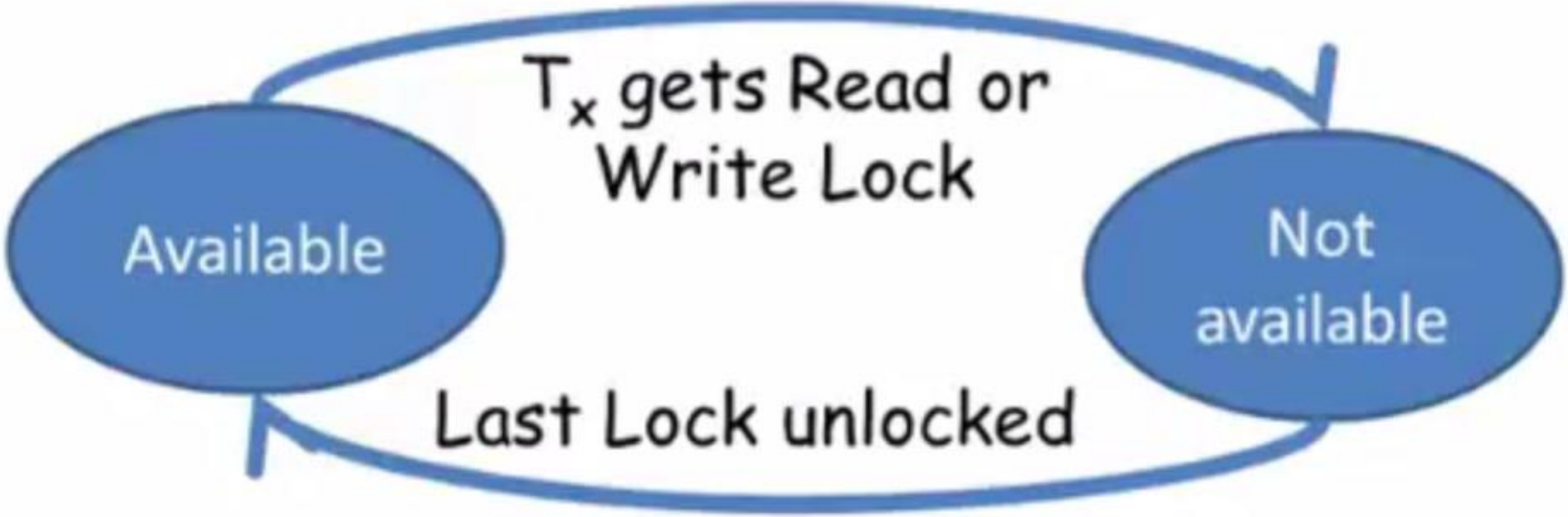
# Lock Types

- Shared Lock (**S-LOCK**) for reads:
  - Allows multiple transactions to read the same object at the same time.
  - If one transaction holds a shared lock, then another transaction can also acquire that same shared lock.
- Exclusive Lock (**X-LOCK**) for writes:
  - Allows a transaction to modify an object.
  - This lock is not compatible for any other lock. Only one transaction can hold an exclusive lock at a time as:
    - 2 Write Locks lead to race condition: lost update
    - Sharing for Read leads to dirty read or non-repeatable read

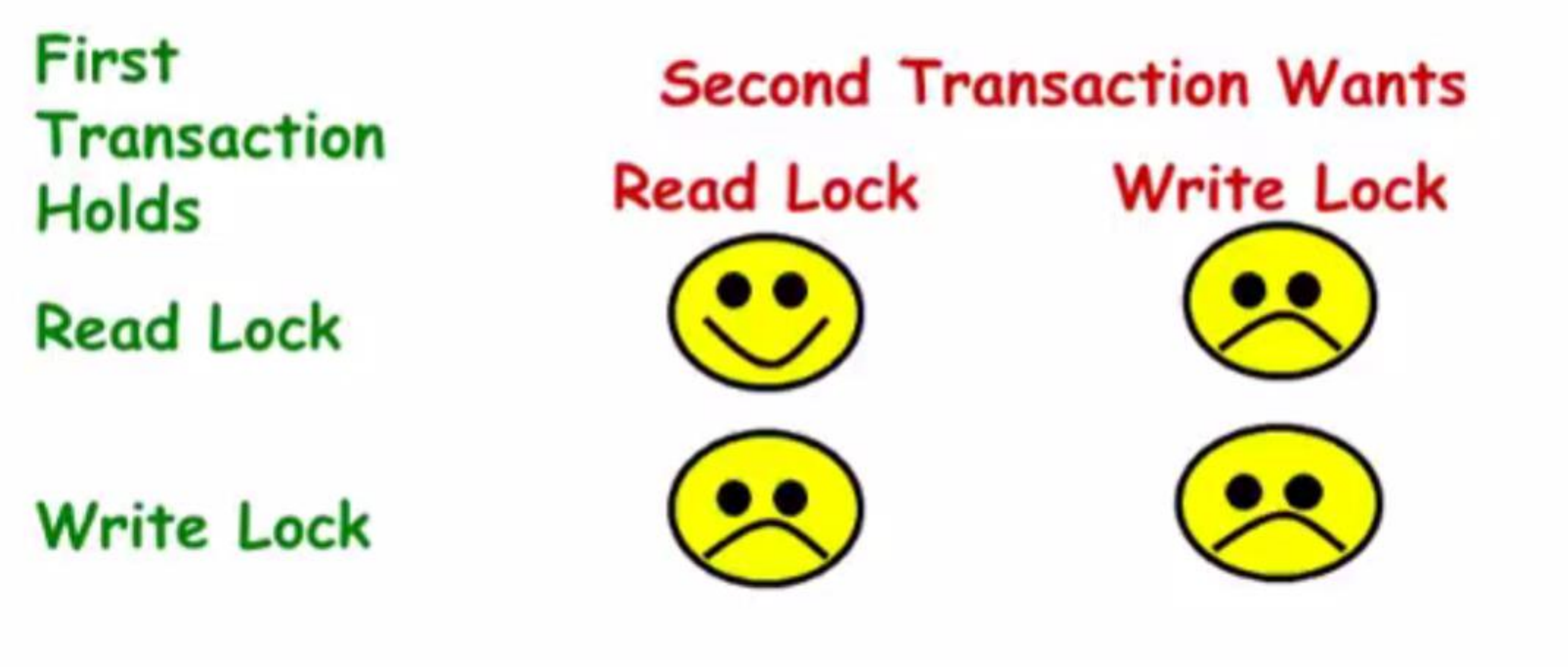
# Read Lock State Diagram



# Write Lock State Diagram



# ReadLock WriteLock Compatibility Matrix

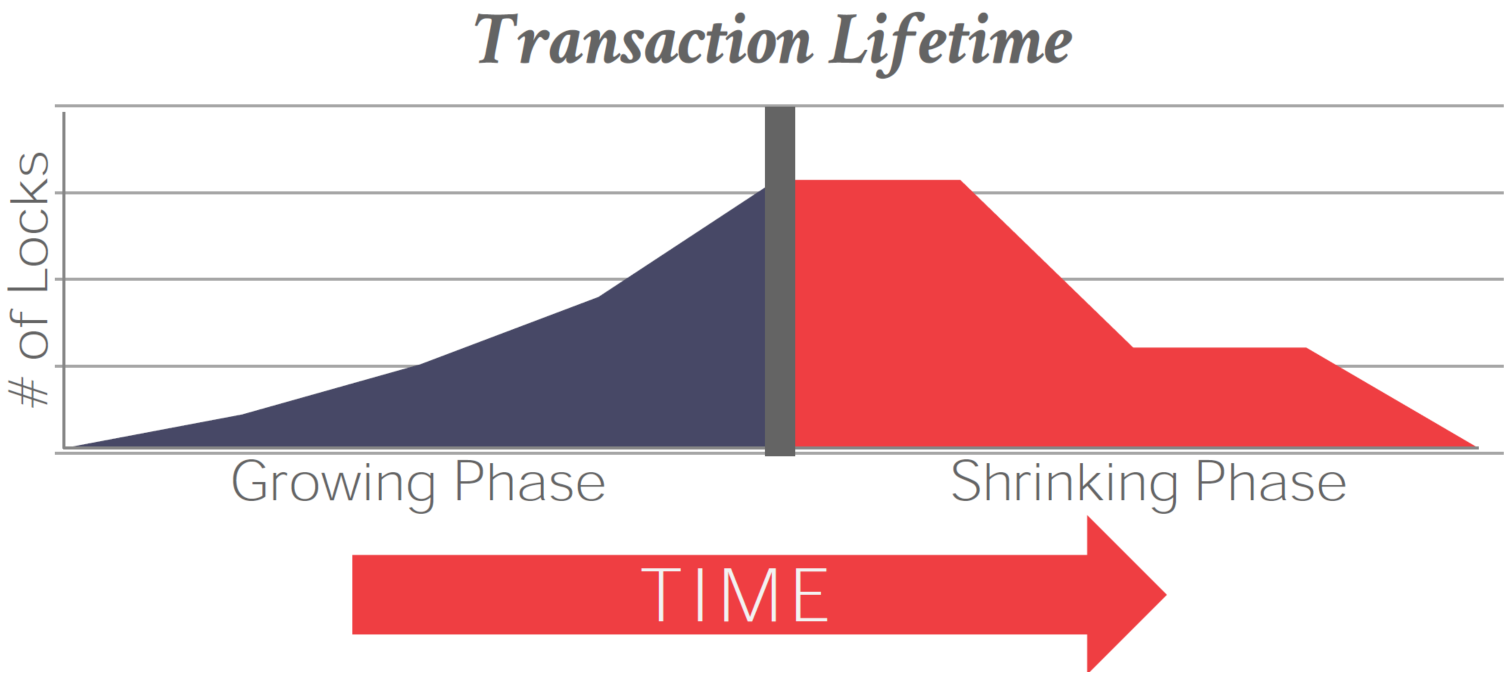


| First Transaction Holds | Second Transaction Wants | |
|---|---|---|
| | Read Lock | Write Lock |
| Read Lock | ☺ | ☹ |
| Write Lock | ☹ | ☹ |

# Two-Phase Locking (2PL)

- Two-Phase locking (2PL) is a concurrency control protocol that to guarantee conflict serializability. It has:
- Phase #1: Growing
    - Each transaction requests the locks that it needs from the DBMS’s lock manager.
    - The lock manager grants/denies lock requests.
- Phase #2: Shrinking
    - The transaction enters this phase immediately after it releases its first lock.
    - The transaction is allowed to only release locks that it previously acquired. It cannot acquire new locks in this phase.

# Two-Phase Locking

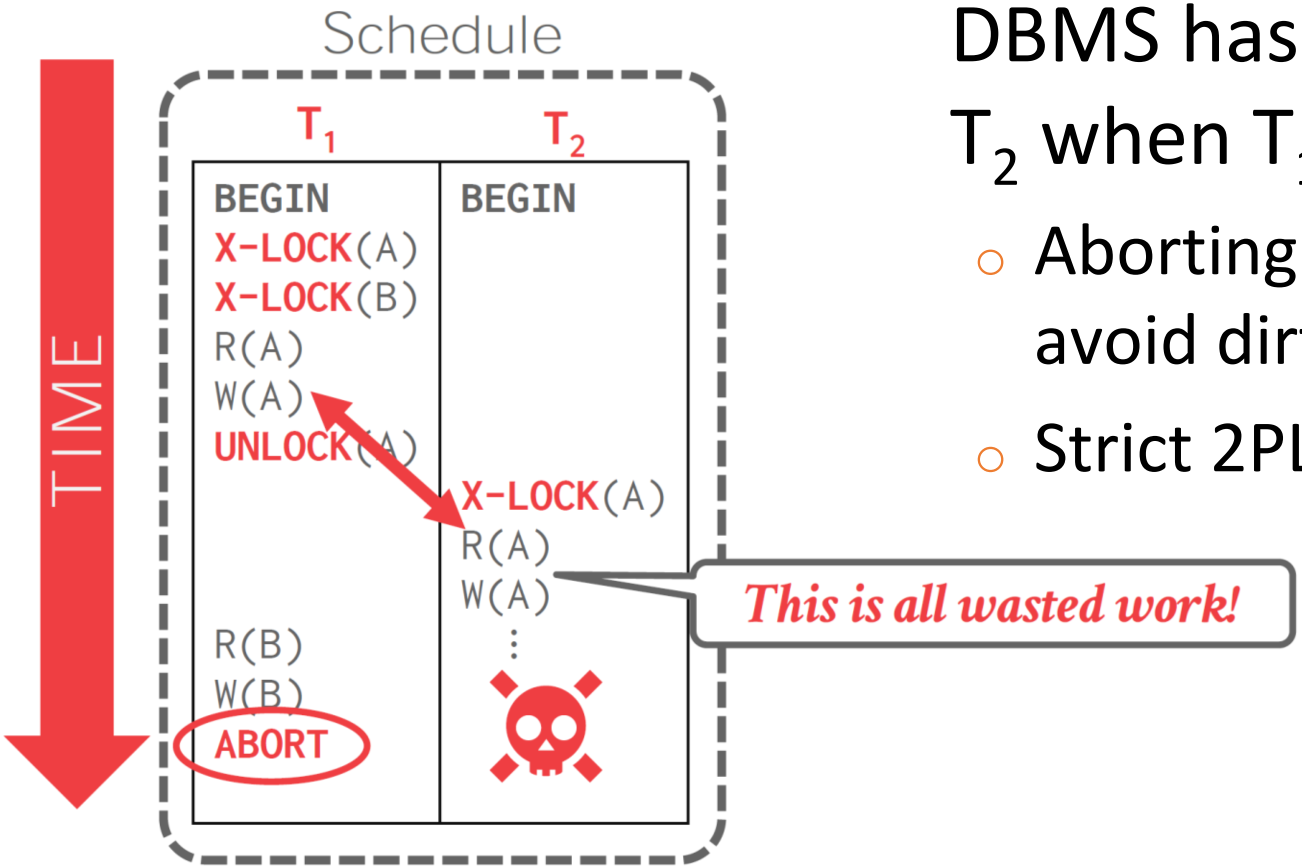


But 2PL can cause:

- **Cascading aborts**: when a transaction aborts and it may cause aborting another transaction
- **Deadlocks**: transactions waiting for each other to release the locks

Source: CMU DB Course https://15445.courses.cs.cmu.edu/fall2019/

# 2PL Cascading aborts

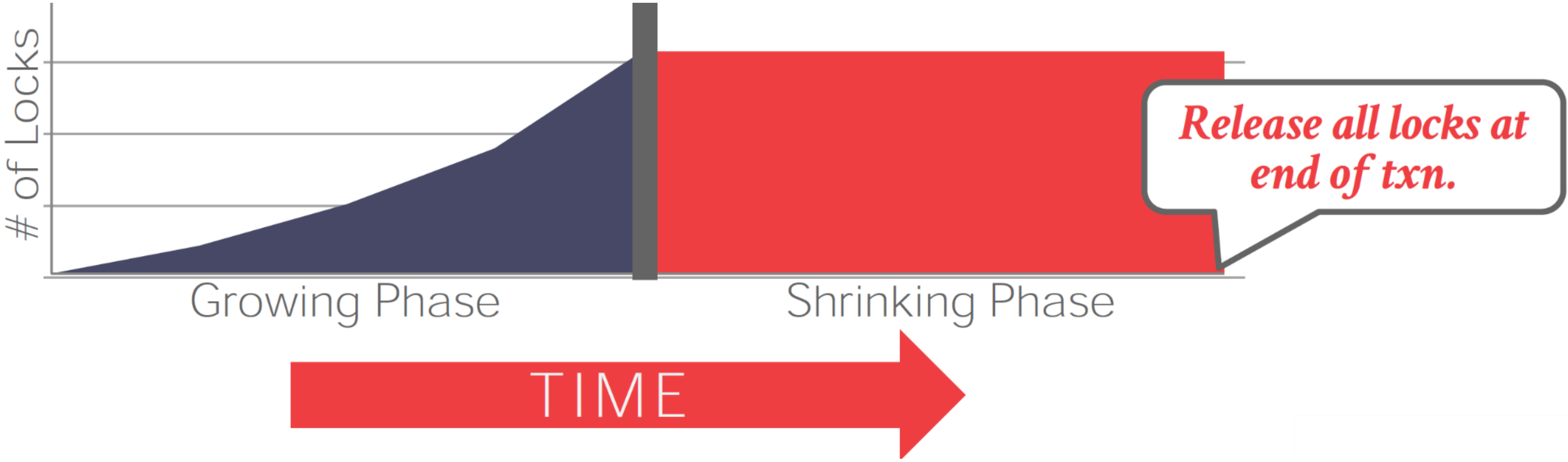


- This is a permissible schedule in 2PL, but the DBMS has to also abort $T_2$ when $T_1$ aborts
  - Aborting $T_2$ is a must to avoid dirty read
  - Strict 2PL can avoid this

Source: CMU DB Course https://15445.courses.cs.cmu.edu/fall2019/

# Strict 2PL

- Strict 2PL is a variant of 2PL where the transaction only releases locks when it finishes.
  - A schedule is strict if a value written by a transaction is not read or overwritten by other transactions until that transaction finishes
  - Solves 2PL **Cascading aborts** **but** limits concurrency

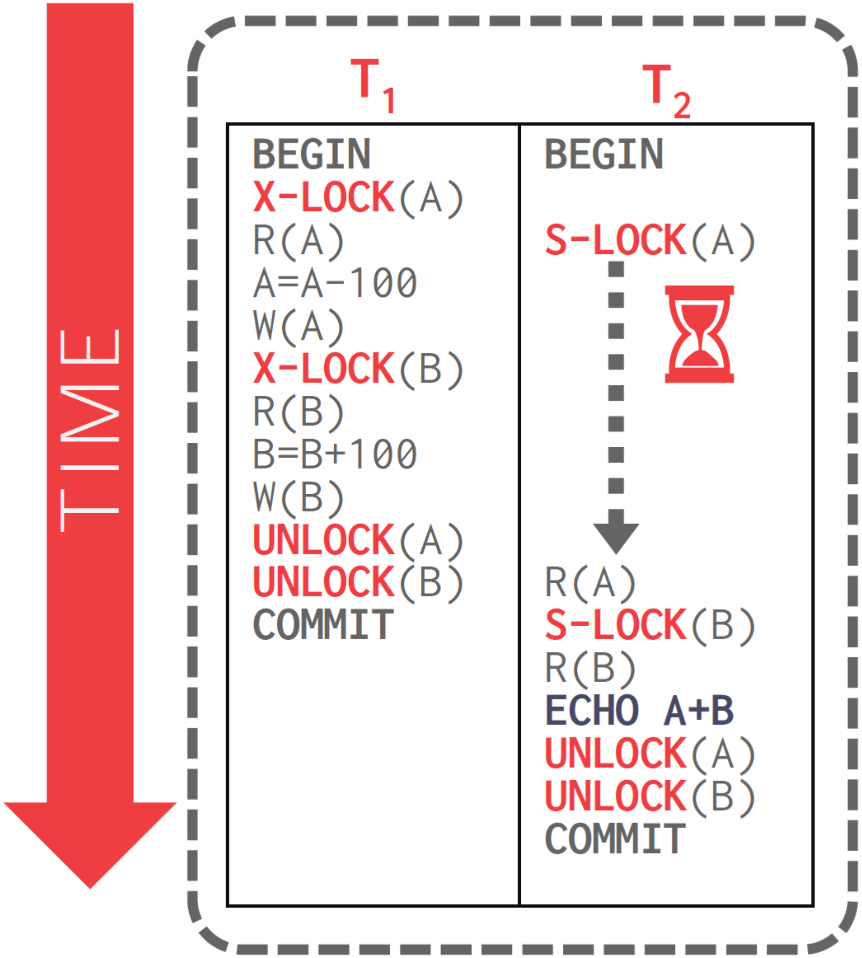


*Release all locks at end of txn.*

Source: CMU DB Course https://15445.courses.cs.cmu.edu/fall2019/

# Strict 2PL Example

Schedule



| $T_1$ | $T_2$ |
|---|---|
| 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) | |
| UNLOCK(B) | R(A) |
| COMMIT | S-LOCK(B) |
| | R(B) |
| | ECHO A+B |
| | UNLOCK(A) |
| | UNLOCK(B) |
| | COMMIT |

Initial Database State

A=1000, B=1000

$T_2$ Output

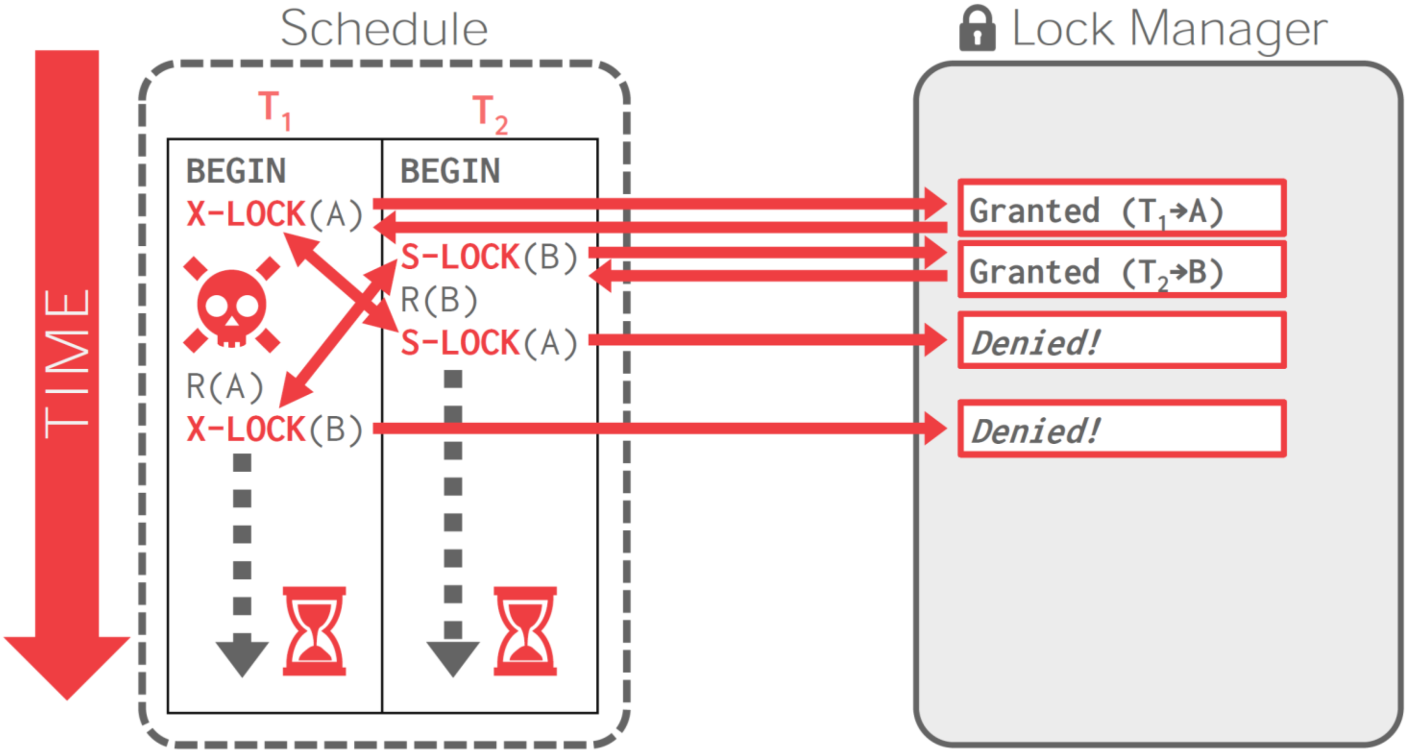
A+B=2000

Source: CMU DB Course https://15445.courses.cs.cmu.edu/fall2019/

# Deadlocks

- A deadlock is 2 transactions waiting for locks to be released by each other.
- Most databases periodically run a deadlock and breaks the deadlock by simply aborting one of the transactions involved
- How do you choose which transaction to abort?
  - By age (newest or oldest timestamp)
  - By progress (least/most queries executed)
  - By the # of items already locked
  - …

# Deadlock Example



- Auto-detection and “resolution”

```
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 232) was deadlocked on lock resources with
another process and has been chosen as the deadlock victim. Rerun the
transaction
```

Source: CMU DB Course https://15445.courses.cs.cmu.edu/fall2019/

# Transaction Isolation Levels

< Back

# Transaction Isolation Levels

- The *isolation level* of a transaction defines what data *that* transaction may see

4 standard levels:

Isolation (High→***Low***)

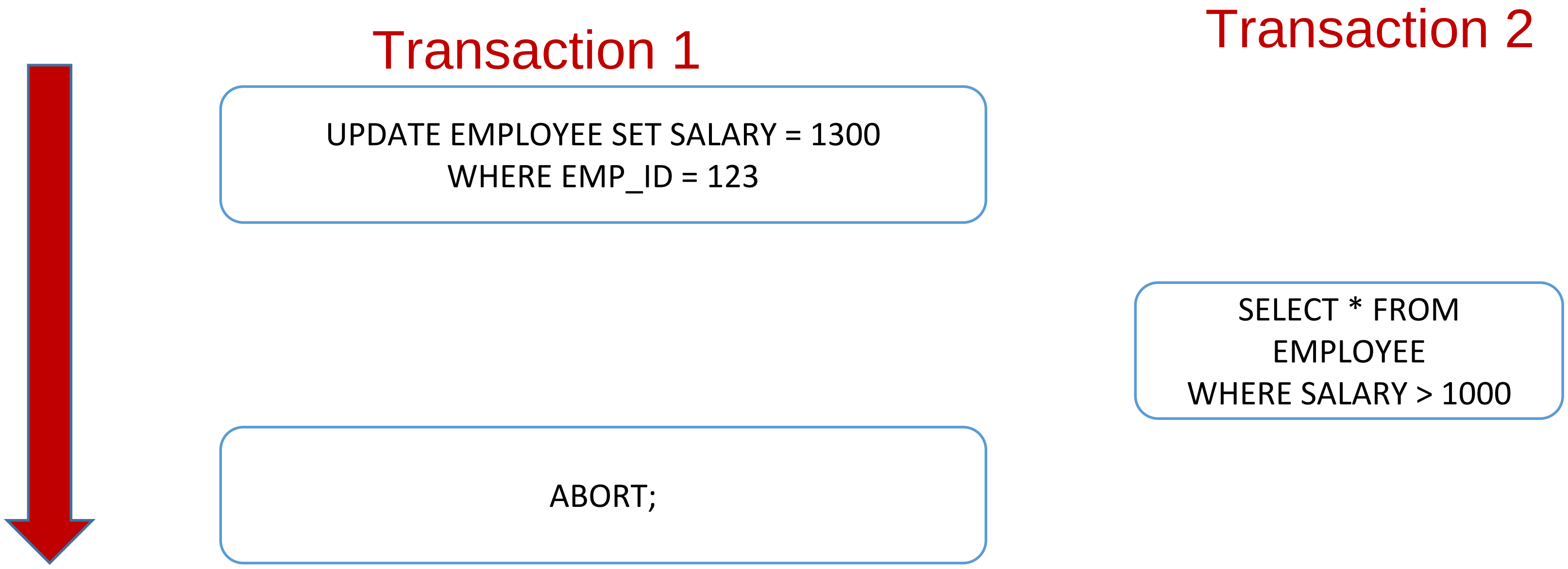
- **SERIALIZABLE**: No phantoms, all reads repeatable, no dirty reads.
- **REPEATABLEREADS**: Phantoms may happen.
- **READCOMMITTED**: Phantoms and unrepeatable reads may happen.
- **READUNCOMMITTED**: All of them may happen.

# Read Uncommitted

- Does not honour locks
- Can read uncommitted data
  - *Dirty data* is data that has been modified by a transaction that has not yet committed.
  - If that transaction is rolled back after another transaction has read its dirty data, inconsistency is introduced
- Sacrificing consistency in favour of high concurrency
- Useful for reporting applications
- Be very careful!

# Read Uncommitted

- With Read Uncommitted, dirty reads might occurs



Transaction 1

Transaction 2

UPDATE EMPLOYEE SET SALARY = 1300
WHERE EMP_ID = 123

SELECT * FROM
EMPLOYEE
WHERE SALARY > 1000

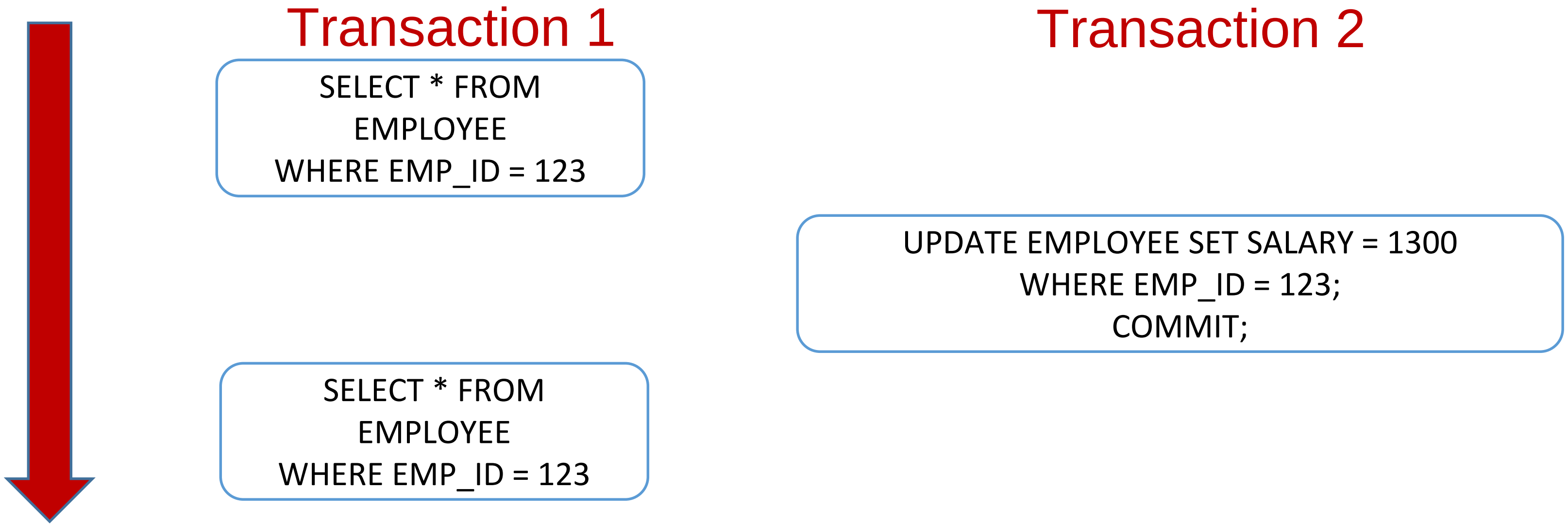
ABORT;

# Read Committed (the default)

- Cannot read uncommitted data
- Share lock before reading data and released after processing is complete
- Dirty reads do not happen but non-repeatable reads can happen

# Read Committed

- Read Committed
  - Non-repeatable reads might occur



Transaction 1

Transaction 2

SELECT * FROM
EMPLOYEE
WHERE EMP_ID = 123

UPDATE EMPLOYEE SET SALARY = 1300
WHERE EMP_ID = 123;
COMMIT;

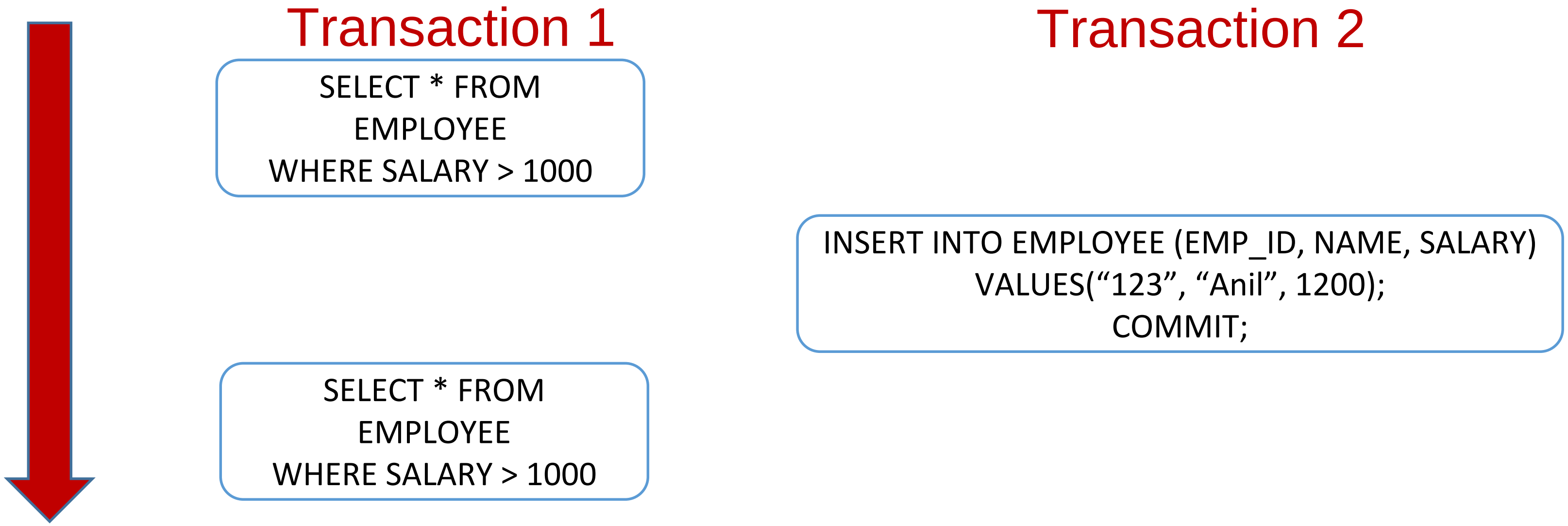
SELECT * FROM
EMPLOYEE
WHERE EMP_ID = 123

# Repeatable Read

- All **data** records read by a SELECT statement cannot be changed
- Holds shared locks on data read until transaction commit/rollback
    - Reduce concurrency and degrade performance
- Rows that have been read can be read again with confidence they won’t have changed
    - A query running more than once within the same transaction returns same values
- But if the SELECT statement contains any ranged WHERE clauses, phantom reads can occur

# Repeatable Reads

- Repeatable Reads
  - Phantom Reads might occur



Transaction 1

SELECT * FROM EMPLOYEE WHERE SALARY > 1000

Transaction 2

INSERT INTO EMPLOYEE (EMP_ID, NAME, SALARY) VALUES(“123”, “Anil”, 1200);
COMMIT;

Transaction 1

SELECT * FROM EMPLOYEE WHERE SALARY > 1000

# Phantom Problem

- A “phantom” is a tuple that is invisible during part of a transaction execution but appears when the query is re-executed

- In our example:
    - T1: reads list of products
    - T2: inserts a new product
    - T1: re-reads: a new product appears !

# Serializable

- This isolation level specifies that all transactions occur in a completely isolated fashion
    - *i.e.*, as if all transactions in the system had executed serially, one after the other
- Locks index ranges as well as rows or table locks
- Phantom rows will not appear if the same query is issued twice within a transaction
- Greatly reduces concurrency
    - Dealing with phantoms is expensive !

# Concurrency and Consistency

| Isolation Table | Dirty Read | Non-Repeatable Read | Phantom Read |
|---|---|---|---|
| **Read Uncommitted** | Possible | Possible | Possible |
| **Read Committed** | Not Possible | Possible | Possible |
| **Repeatable Read** | Not Possible | Not Possible | Possible |
| **Serializable** | Not Possible | Not Possible | Not Possible |

- Concurrency and consistency are mutually opposing goals

# How Isolation Levels are Achieved?

- **SERIALIZABLE**: Obtain all locks first; plus index locks, plus strict 2PL.
- **REPEATABLEREADS**: Same as above, but no index locks.
- **READCOMMITTED**: Same as above, but **S** locks are released immediately.
- **READUNCOMMITTED**: Same as above, but allows dirty reads (no **S** locks).

# Optimistic Concurrency Control

< Back

# Optimistic Concurrency Control

- In an *optimistic concurrency control (OCC) protocol, we assume that most of the time,* transactions will not conflict thus all of the locking is not necessary while the transaction is executing
  - Writers don’t block the readers. Readers don’t block the writers
- Multi-Version Concurrency Control (MVCC) is the most widely used OCC scheme in DBMS
  - The DBMS maintains multiple physical versions of a single logical object in the database
  - Also known as Snapshot Isolation

# OOC Phases

- **Modify Phase**:
  - Every transaction has its own private workspace not visible to other transactions
  - Any object read is copied into the transaction’s workspace
  - All modifications are applied to the private workspace
  - System tracks the read/write sets of transactions
- **Validation Phase**: When a transaction commits, check whether other transactions have modified data that this transaction has used (read or written)
- **Commit Phase**: If validation succeeds, apply private changes to the database. Otherwise abort the transaction

# Validation Phase

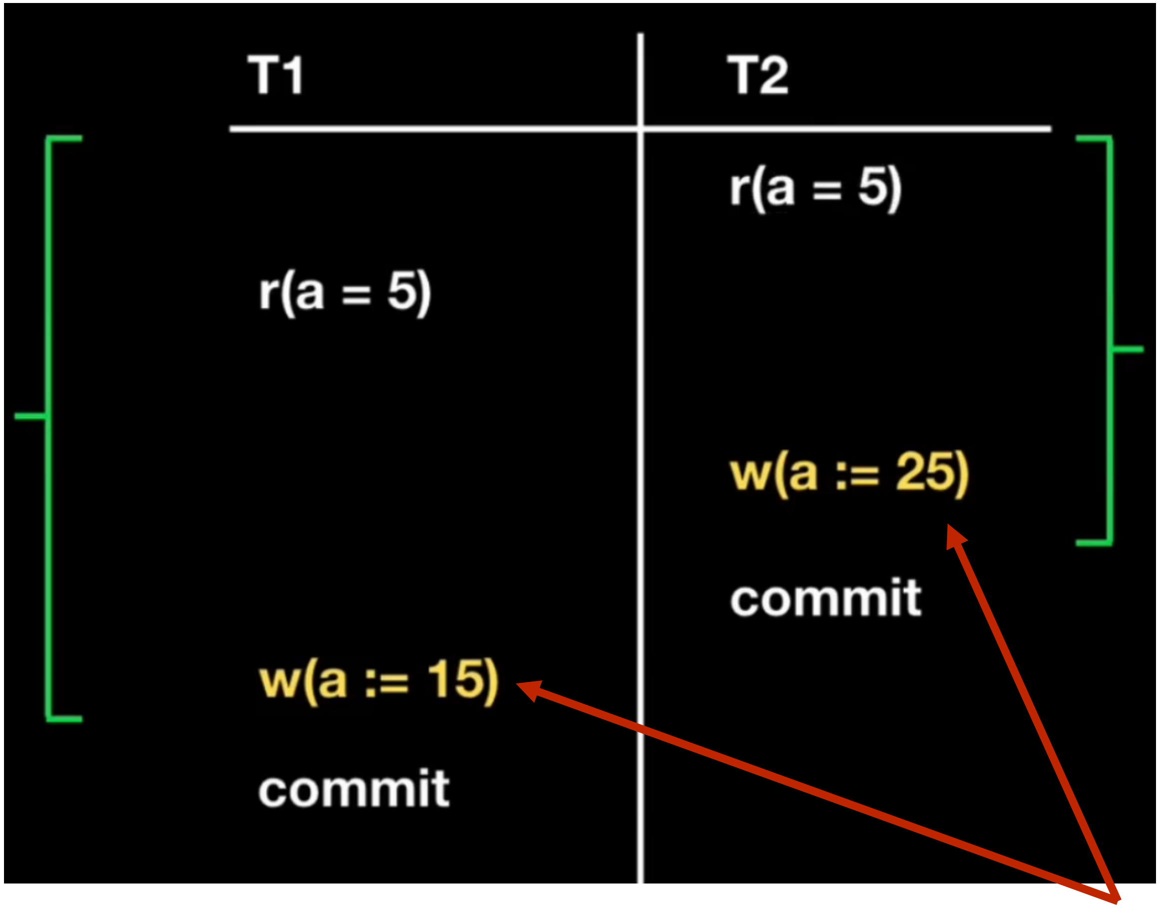
- Check whether the read/write sets of the committing txn $\mathbf{T_i}$ intersects with the read/write sets of concurrent txns that committed while $\mathbf{T_i}$ was running
- Ti will commit if the following conditions are met:

$$\mathbf{WriteSet(T_i)} \cap \mathbf{WriteSet(T_{j \neq i})} = \emptyset$$

$$\mathbf{ReadSet(T_i)} \cap \mathbf{WriteSet(T_{j \neq i})} = \emptyset$$

$\mathbf{T_{j \neq i}}$ concurrent txns that committed while $\mathbf{T_i}$ was running

# Modify Phase



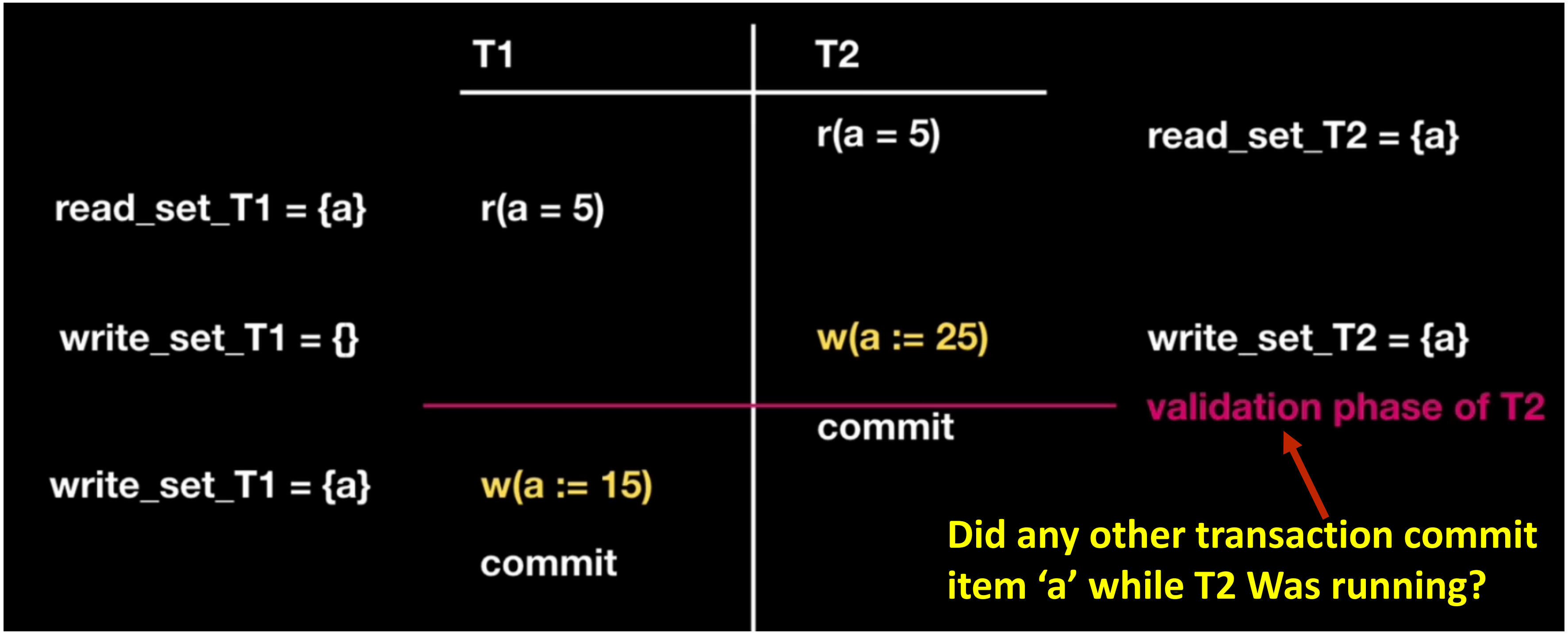
| T1 | T2 |
|---|---|
| | r(a = 5) |
| r(a = 5) | |
| | w(a := 25) |
| | commit |
| w(a := 15) | |
| commit | |

$T_1$ private workspace

$T_2$ private workspace

**These updates are performed in a private workspace on local copy but NOT on the current database-state**

# $T_2$ Validation Phase



| | T1 | T2 | |
|---|---|---|---|
| | | r(a = 5) | read_set_T2 = {a} |
| read_set_T1 = {a} | r(a = 5) | | |
| write_set_T1 = {} | | w(a := 25) | write_set_T2 = {a} |
| | | | validation phase of T2 |
| | | commit | |
| write_set_T1 = {a} | w(a := 15) | | |
| | commit | | |

**Did any other transaction commit item 'a' while T2 Was running?**

No other txns committed changes to 'a' while T2 was running

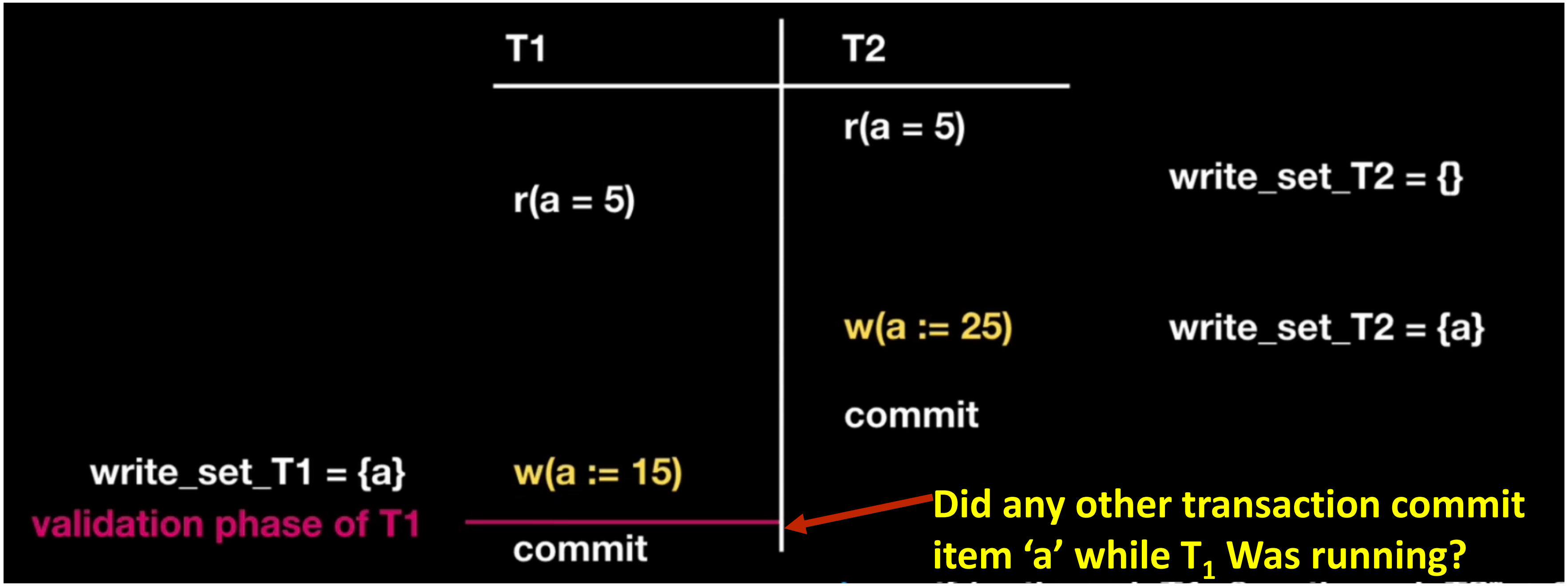
$$\mathbf{WriteSet(T_2)} \cap \mathbf{WriteSet(T_1)} = \varnothing$$

$$\mathbf{ReadSet(T_2)} \cap \mathbf{WriteSet(T_1)} = \varnothing$$

=> proceed to **Commit Phase**

Local version of T2 becomes the current global version (a := 25)

# $T_1$ Validation Phase



| | T1 | T2 | |
|---|---|---|---|
| | | r(a = 5) | |
| | | | write_set_T2 = {} |
| | r(a = 5) | | |
| | | w(a := 25) | write_set_T2 = {a} |
| | | commit | |
| write_set_T1 = {a} | w(a := 15) | | |
| validation phase of T1 | | | |
| | commit | | |

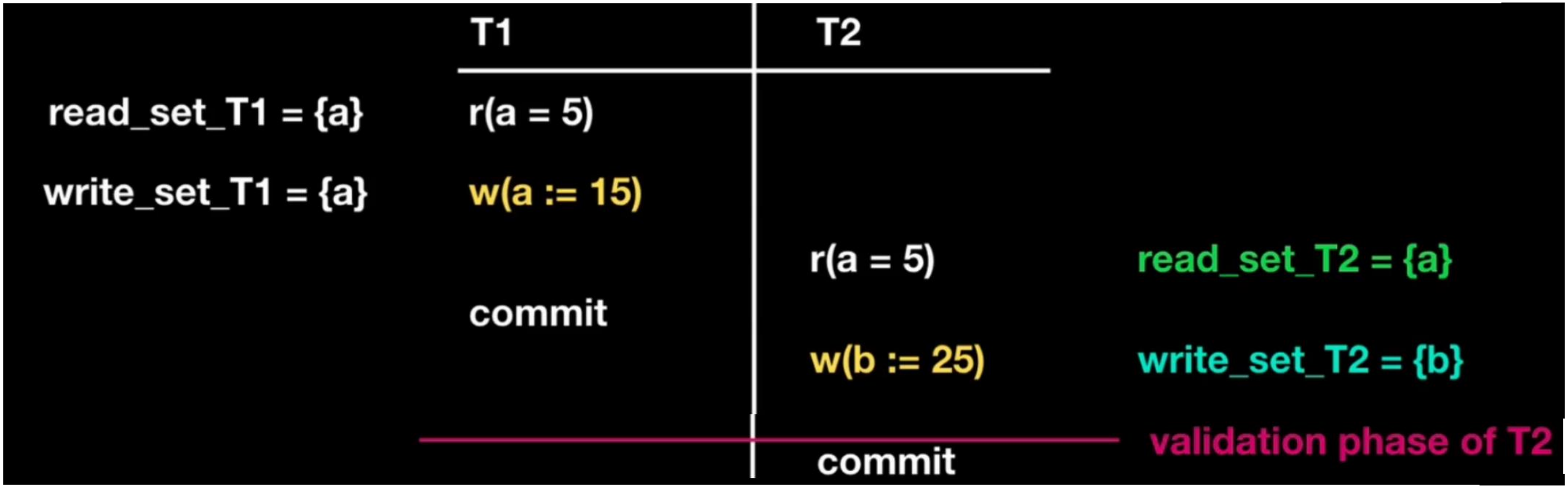
Did any other transaction commit item ‘a’ while $T_1$ Was running?

$$\mathrm{WriteSet}(T_1) \cap \mathrm{WriteSet}(T_2) \neq \varnothing$$

=> Abort $T_1$

`In practice: intersect with WriteSet of every transaction that committed during the lifetime of T₁`

# Example 2 - $T_2$ Validation Phase



| | T1 | T2 | |
|---|---|---|---|
| read_set_T1 = {a} | r(a = 5) | | |
| write_set_T1 = {a} | w(a := 15) | | |
| | | r(a = 5) | read_set_T2 = {a} |
| | commit | | |
| | | w(b := 25) | write_set_T2 = {b} |
| | | | validation phase of T2 |
| | | commit | |

**$\text{WriteSet}(T_2) \cap \text{WriteSet}(T_1) = \varnothing$**

**$\text{ReadSet}(T_2) \cap \text{WriteSet}(T_1) \neq \varnothing$**

=> Abort $T_2$

**In practice: intersect with WriteSet of every transaction that committed during the lifetime of $T_2$**

# OCC Observations

- Advantages:
  - OCC works well when the number of conflicts is low: Txns access disjoint subsets of data
  - A low probability of conflict makes locking wasteful

- Limitations:
  - High overhead for copying data locally
  - Validation/Write phase bottlenecks
  - Aborts are more wasteful than in 2PL because they only occur after a txn has already executed

# Summary

- A transaction consists of a sequence of read / write operations that must be performed as a single logical unit
- The DBMS guarantees the atomicity, consistency, isolation and durability of transactions
- **Concurrency-control manager** controls the interaction among the concurrent transactions to ensure the consistency of the database using:
  - Pessimistic techniques: don’t let problems arise in the first place
  - Optimistic techniques: assume conflicts are rare, deal with them *after* they happen