CMPT 606

Database Concurrency Control

Read Chapters 20 and 21

Dr. Abdelkarim Erradi

Department of Computer Science and Engineering

QU

Outline

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

Transaction Management

Back

Concept of Transaction

- Transaction = Logical unit of work on the database
  - Transfer money from one bank account to another
  - Checkout: place order and process payment
- A transaction consists of a sequence of read / write operations that must be performed as a **single** logical unit that must either commit or abort
  - Transaction boundaries are defined by the database user / application programmer
- **Atomicity, Consistency and Isolation are achieved using DB Transactions**

Atomicity

- Consider a bank transaction T:

T:	A=A+100, B=B-100

$\longrightarrow$ time

- T is transferring $100 from B's account to A's account.
- What if there is an error right after the first statement of T has been executed, i.e. the second statement is not executed?

=> You will get **partial update** = inconsistent state

- The DBMS has to ensure that every transaction is treated as an *atomic* unit, i.e. either all succeed otherwise rollback

A transaction is a sequence of operations that must be executed as a whole

Ali

Bank Acct

$1000

Transfer $500

Sara

Bank Acct

$0

(1) Debit Ali savings acct

(2) Credit Sara checking Acct

Either both (1) and (2) happen or neither!

Every DB action takes place inside a transaction

We abstract away most of the application code when thinking about transactions

User's point of view

Transfer $500

1. Debit savings
2. Credit checking

Programmer's point of view

Read Balance1
Write Balance1
Read Balance2
Write Balance2

DB's point of view

Transaction = a sequence of DB reads (R) and writes (W)

T: R(A), W(A), R(B), W(B)

time

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

Concurrency control

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
  - 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 inconsistent results 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 from DB;
If balance > withdrawalAmount {
    balance = balance - withdrawalAmount;
    Write balance to DB;

}
```

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:

read balance; **=>** $400

Sara withdraws $50:

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 problem

What will be the final account balance?

Transaction 1: Add $100 to account A	**Transaction 2:** Add $200 to account A
R(A) W(A) FAIL	 R(A) W(A)

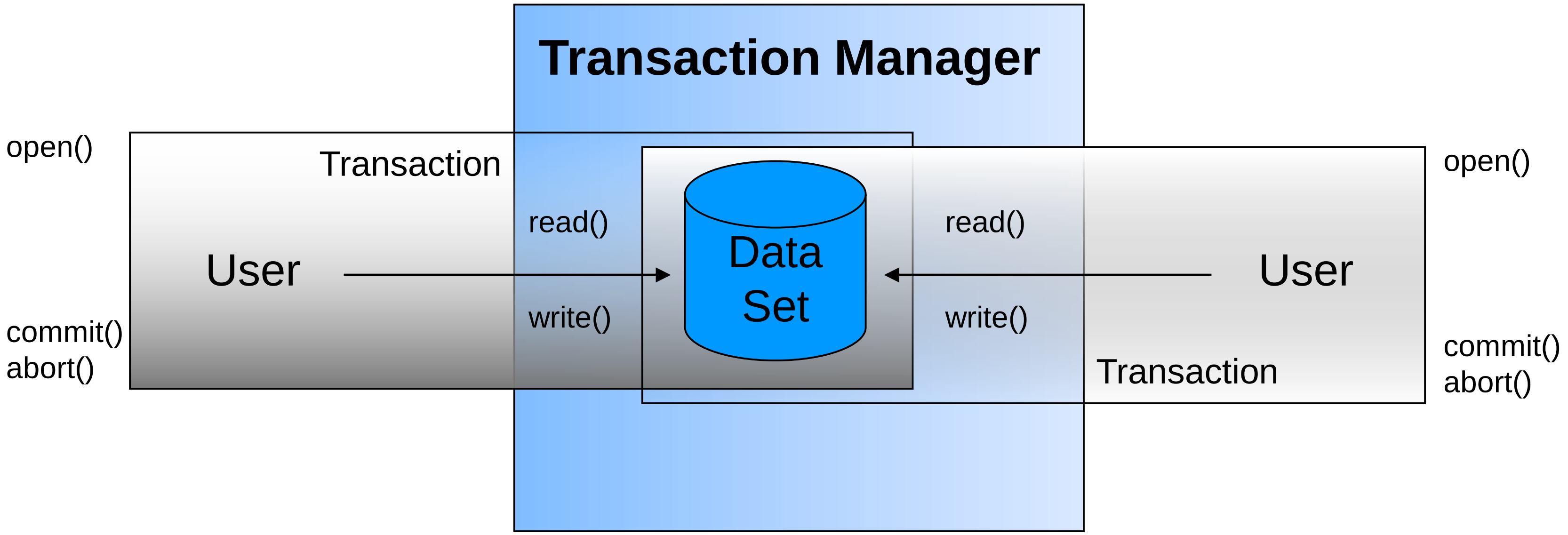
Time

Dirty read problem

Isolation

- To preserve consistency the DBMS must manage concurrency to guarantee the **Isolation** property.
  - Independence from all other transactions (serializability) => **same results as if the statements would have been executed in a single user scenario**
  - A transaction should not be affected by other concurrently running transactions

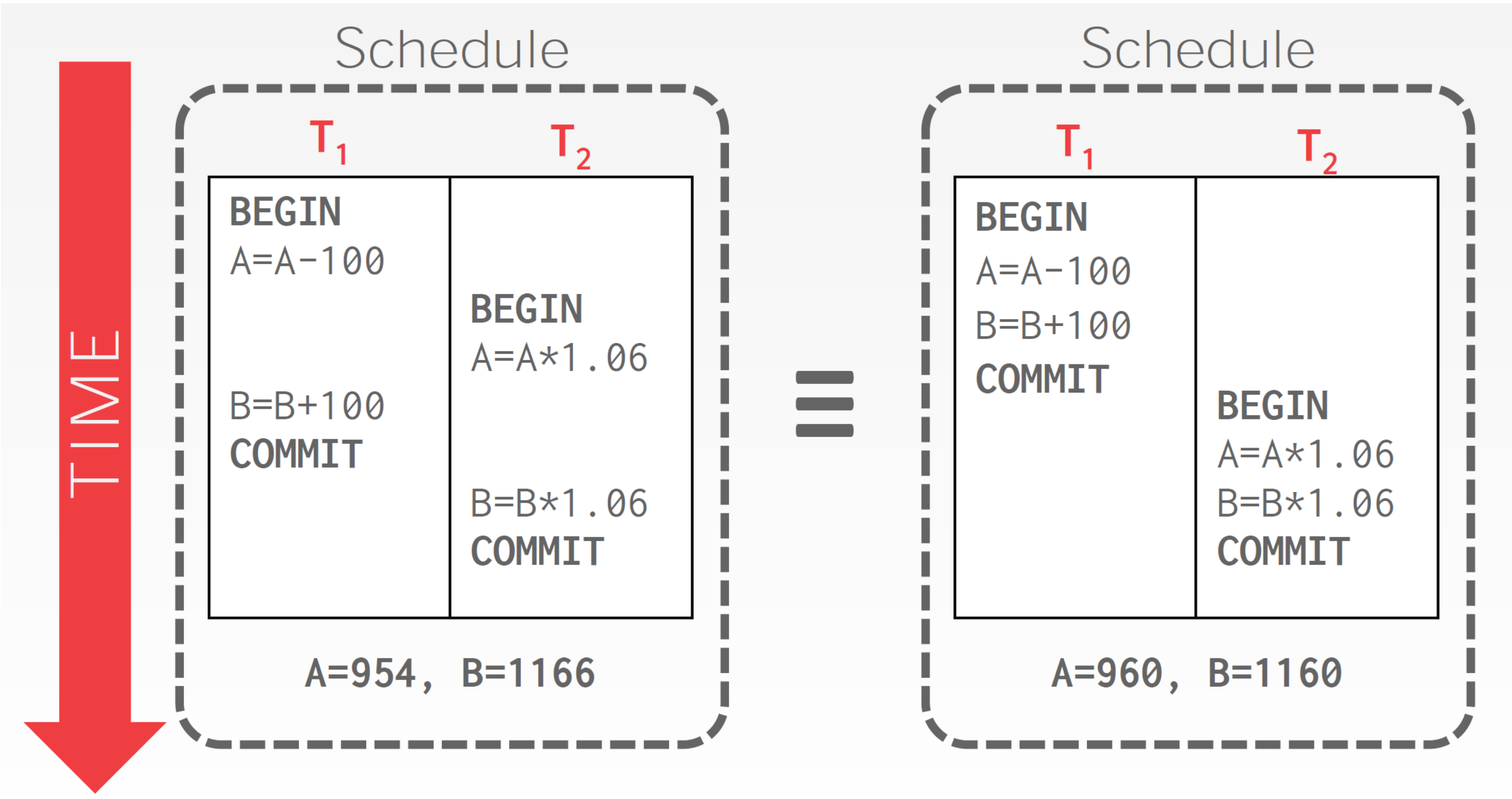
Scheduling Concurrent Transactions



- The transaction scheduler has to organise or "schedule" database read and write actions of all concurrent transactions
- A "**schedule**" is a sequence of operations from different transactions that may be executed in an interleaved fashion
  - Such a schedule may compromise the integrity / consistency of a database

Serial vs. Non-serial Schedule

Non-serial / Interleaved Schedule | **Serial Schedule**



Interleaved Transaction Schedules

- We want that the database system schedules transactions in an interleaved fashion:
  – Improve the responsiveness and increase throughput

- Interleaved schedules also create problems
  – Transactions may overwrite each others’ updates
  – Transactions may base their calculations on retrieved data that is already out-of-date or on “dirty reads”

Serial Transaction Schedules

- In order to avoid the concurrency problems described, one obvious solution would be to **schedule only one transaction at a time** for execution
- Such a completely “serialised” schedule will ensure that the **transactions are completely isolated** and cannot interfere with each other
  - But this strategy will reduce **concurrency** and **throughput**

Conflict-Serializable Schedule

- DB will try to find a non-serial schedule that is ***equivalent*** to a serial schedule:
  - Schedule is **conflict-serializable** if its effect on the database state is the same as that of some serial schedule
- A **conflict-serializable schedule** is used by the Transaction Manager to schedule operations of different transactions in a way that interference and problems such as “lost updates” are avoided

Read and *write* operation conflict rules

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

A serializable, but not serial, 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

Constraint of A = B is satisfied after this schedule

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

Constraint of A = B is violated after this schedule => Non-serializable schedule

Verifying Conflict-Serializability using a Precedence Graph?

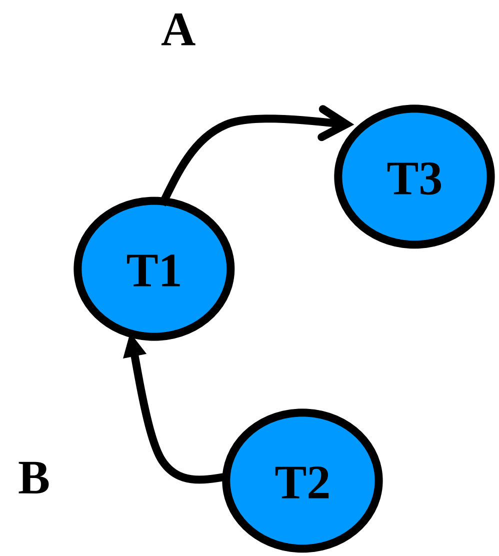
- Precedence Graph = {**Nodes**: transactions,
  Arcs: r/w or w/w conflicts}

- The precedence graph for a schedule S contains:
  - A node for each committed transaction in S
  - An **arc** from T_i to T_j if an action of T_i **precedes and conflicts with** one of T_j's actions.

- 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)		



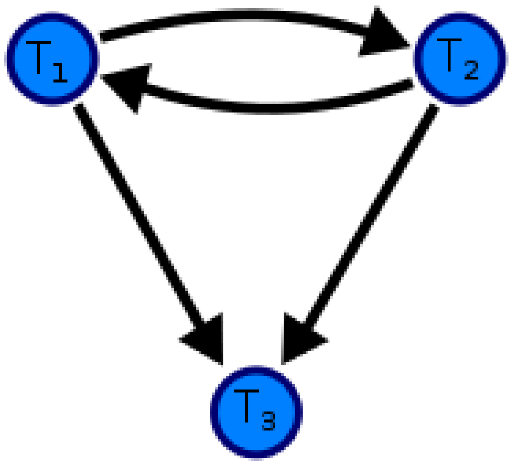
serial execution?

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

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

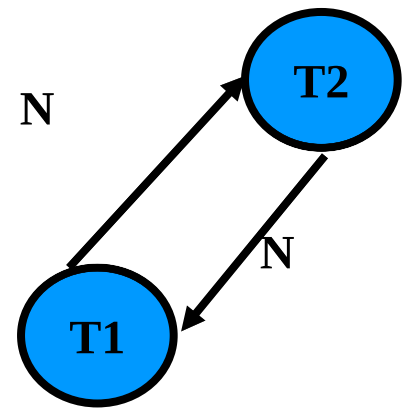


$D = R_1(A)\ W_2(A)\ W_1(A)\ W_3(A)$

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)



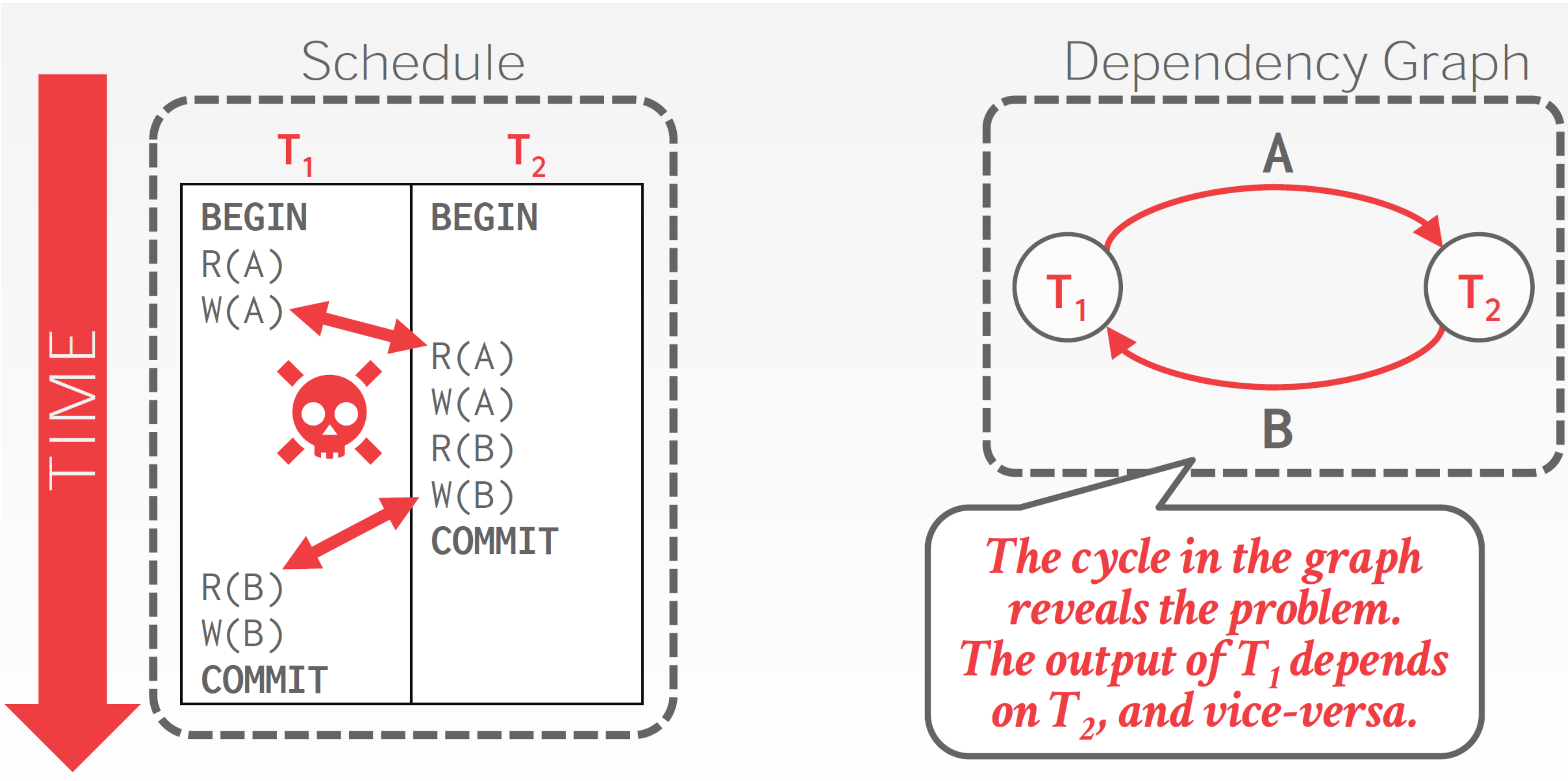
Cycle -> not conflict-serializable

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 <u>and</u> write) or after (reading the changed value)

Example 4


TIME
Schedule
T1
T2
BEGIN
R(A)
W(A)
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT
R(B)
W(B)
COMMIT
Dependency Graph
A
B
T1
T2
The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

RECAP

- Concurrency control motivation
  - If we insist 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** (i.e., without causing the database to reach an inconsistent state)
- If we do concurrency control properly, then we can maximize *transaction throughput* while avoiding any chance of corrupting the database

Concurrency Control Protocols

- The basis of concurrency control is protocols to maintain serialization in DBMSs
- E.g. protocols
  - **Two-Phase Locking (2PL)**
  - **Timestamps Ordering**
  - **Optimistic Concurrency Control (OCC)**
  - **Etc.**

Lock-based Concurrency control

Enforcing Serializability by Locks

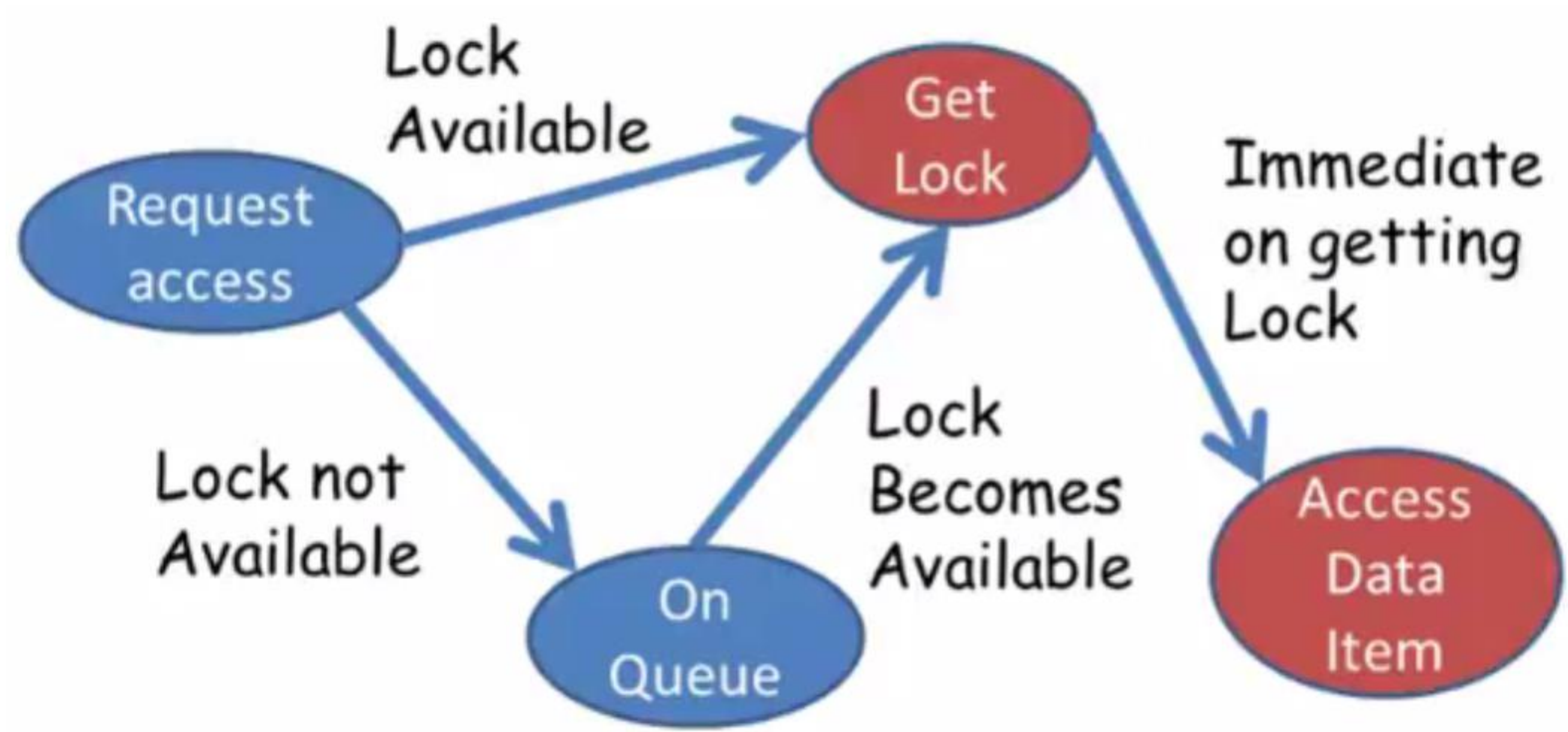
How to achieve correct concurrency control?

- **Locks** = most popular solution for concurrency control
- Lock manager: grants/denies lock requests
- Locking mechanisms prevent conflicts
  - Readers block writers
  - Writers block readers

How lock works

- Transaction needs lock before read or write
- Transaction must ask for lock
- If it does not get the lock
    - maybe another transaction already holds the lock
  - it is suspended
  - put on queue waiting for lock
- When transaction releases lock
  - Some suspended transaction awakened and given lock.

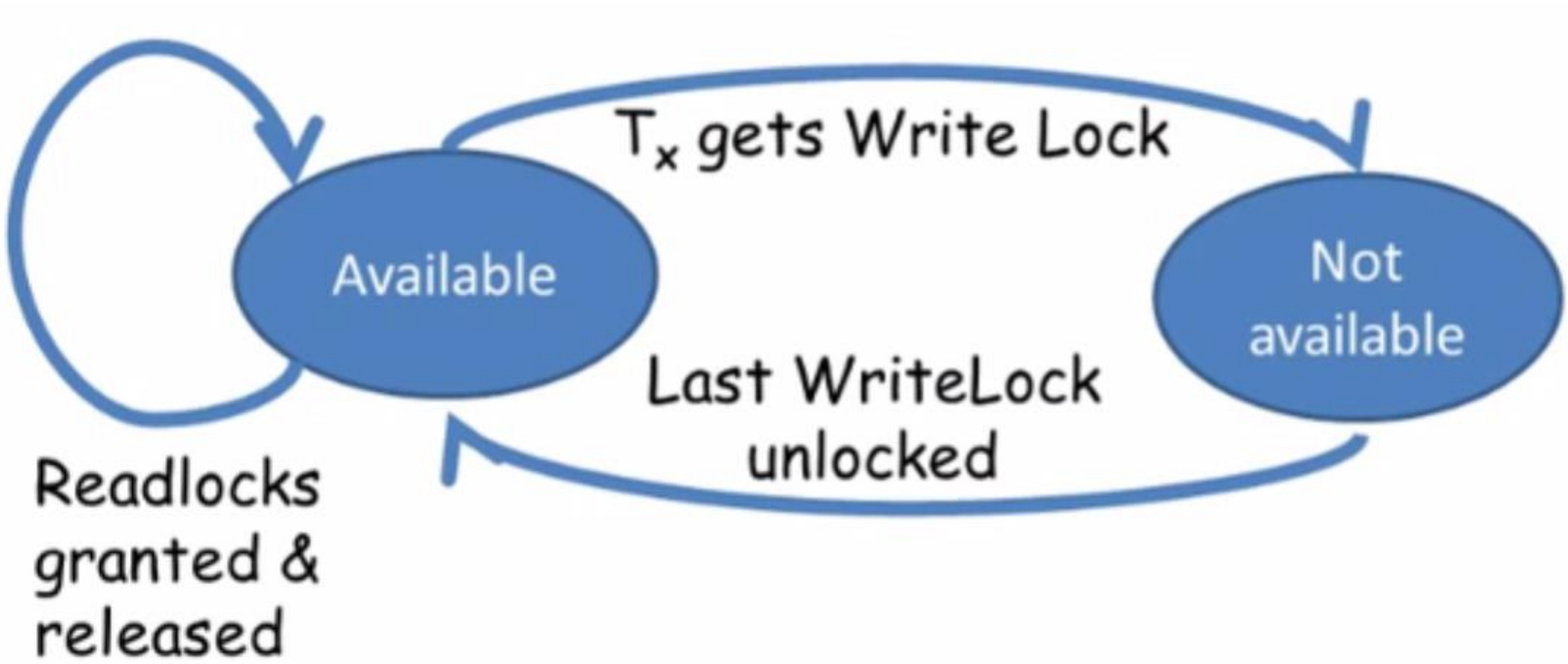
Lock State Diagram



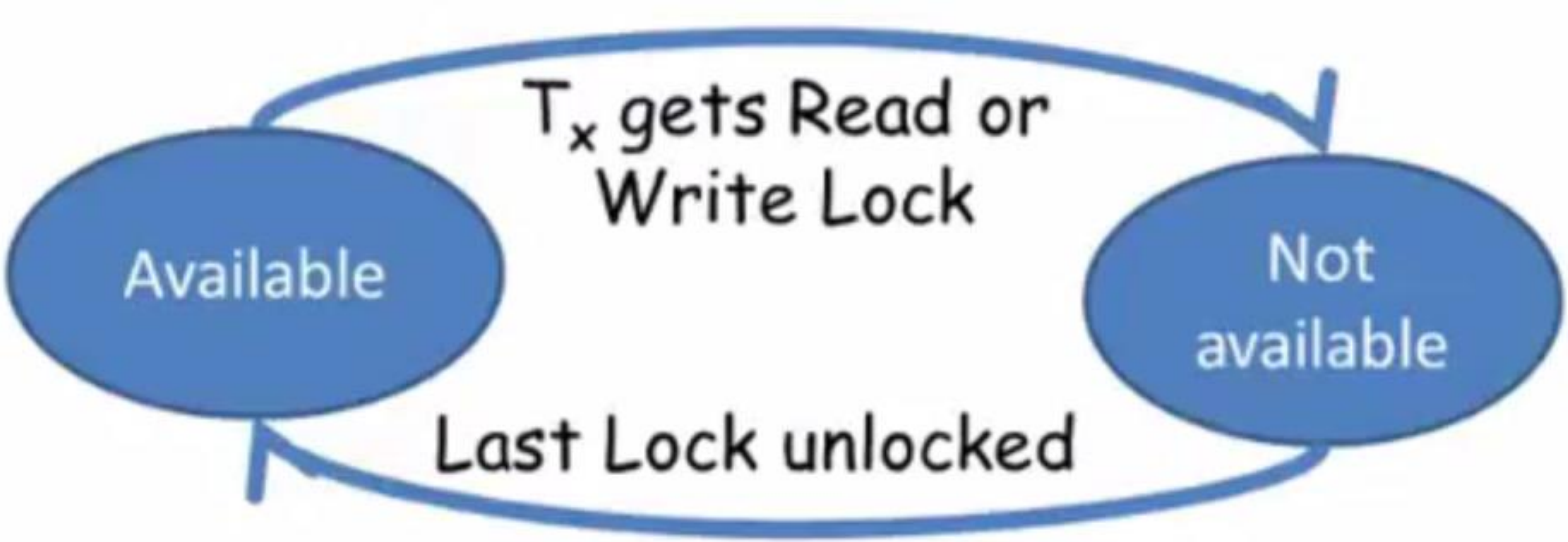
Multi-transaction Rules

- Multiple Read Locks OK
  - Called Shared Lock
  - Several transactions can read same info
- Write Lock must be only lock on item
  - 2 Write Locks lead to race condition: lost update
  - Sharing with Read lock leads to dirty read or non-repeatable read.

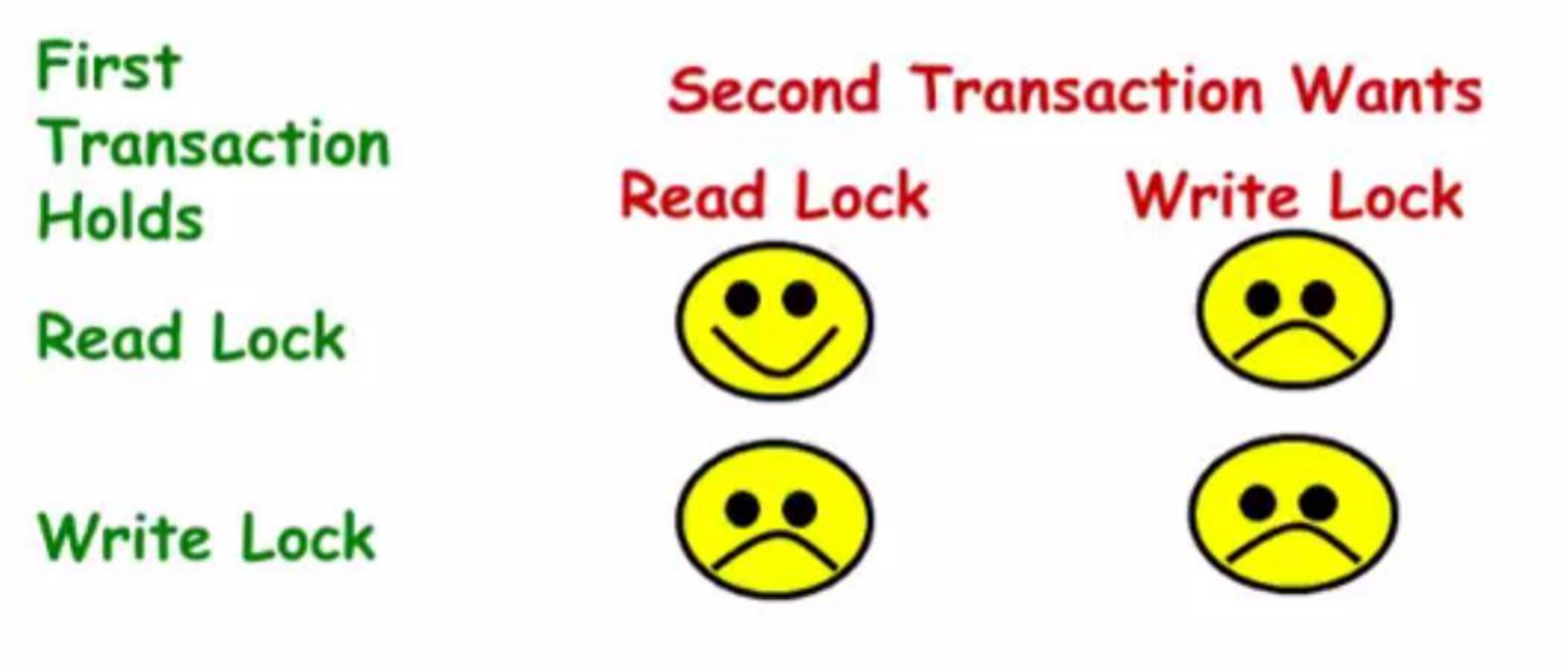
Read Lock State Diagram



Write Lock State Diagram



ReadLock WriteLock Compatibility Matrix



Locking protocol

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks
- Most popular protocol:

- Two-phase locking (2PL), a transaction is not allowed to acquire any new locks after it has released a lock

- Strict 2 Phase Locking (S2PL) = a transaction must hold all its exclusive locks till it commits/aborts

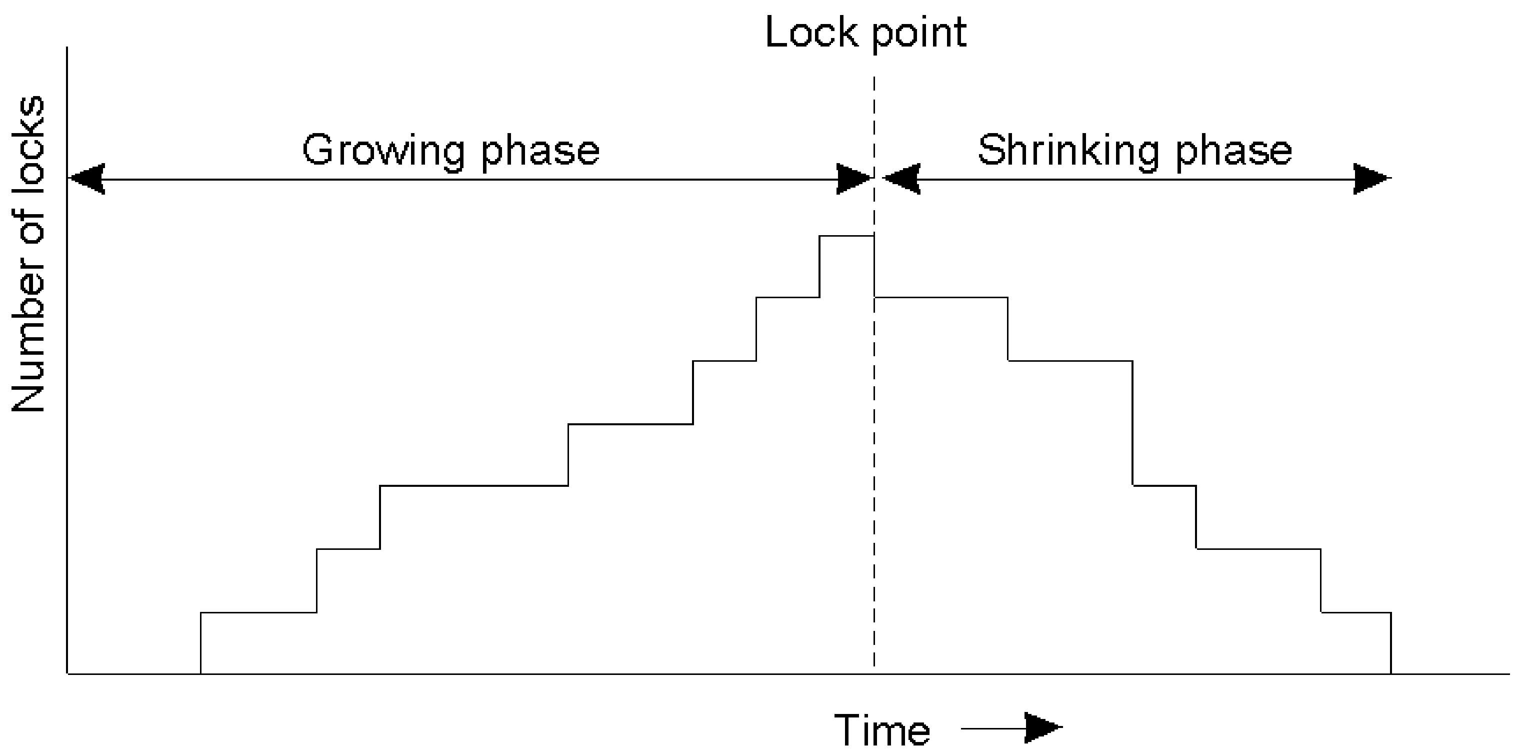
- Rigorous two-phase locking (R2PL) is even stricter: here *all* locks are held till commit/abort

- **THEOREM**: if all transactions obey 2PL -> all schedules are serializable

2PL

- Phase 1: **Growing Phase**
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: **Shrinking Phase**
  - transactions issue no lock/upgrade request, after the first unlock/downgrade
  - transaction may release locks
  - transaction may not obtain locks
- 2PL assures serializability

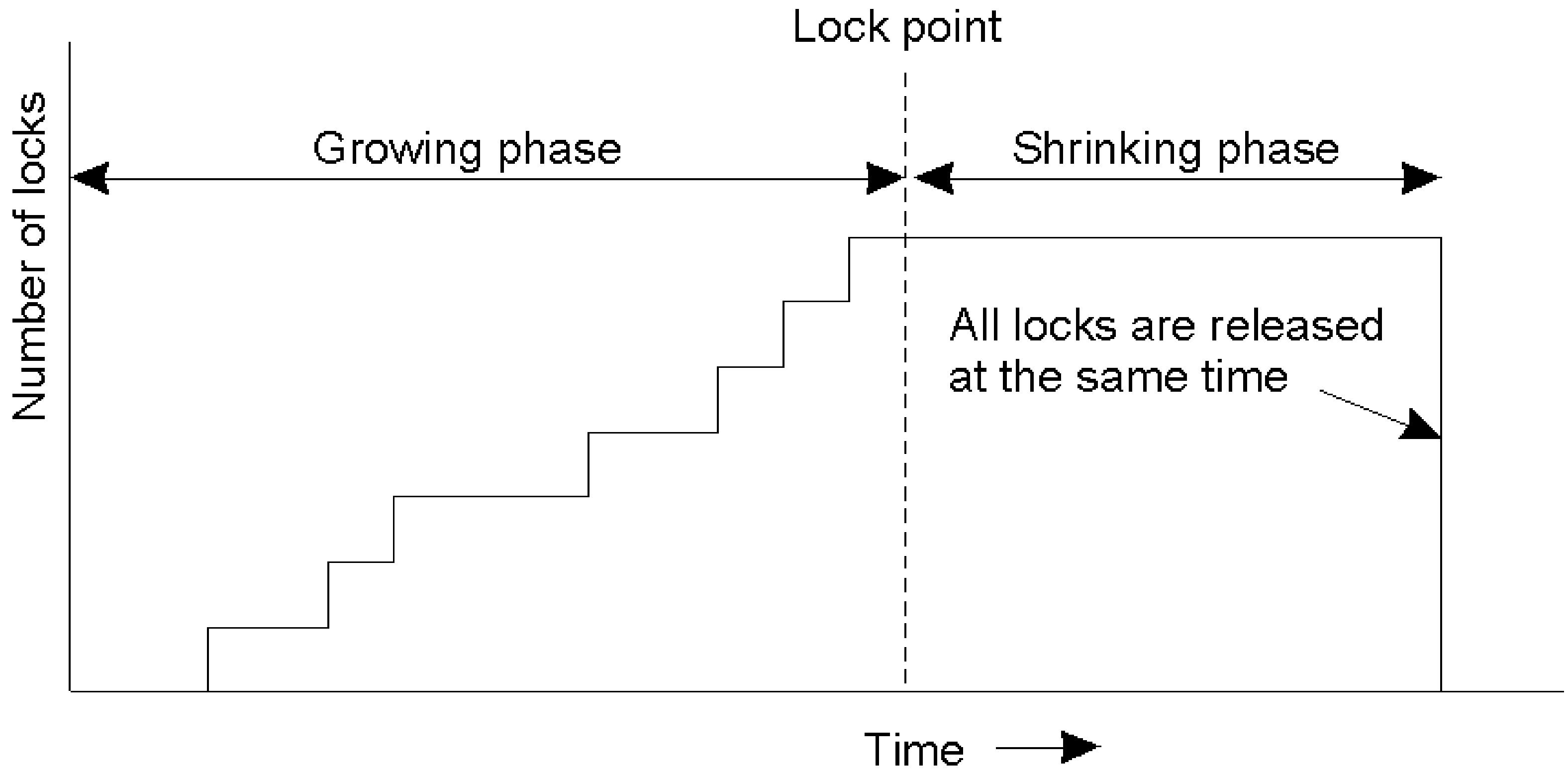
Two-Phase Locking (1)



In two-phase locking, a transaction is not allowed to acquire any new locks after it has released a lock

Strict Two-Phase Locking (2)

- Strict two-phase locking.



Strict 2PL	
T1	**T2**
s-lock(A)	
read(A)	
	s-lock(A)
x-lock(B)	
unlock(A)	
read(B)	
write(B)	
	read(A)
	unlock(A)
commit	
unlock(B)	
	s-lock(B)
	read(B)
	unlock(B)
	commit

Vs.

Rigorous 2PL	
T1	**T2**
s-lock(A)	
read(A)	
	s-lock(A)
x-lock(B)	
	read(A)
read(B)	
write(B)	
commit	
unlock(B)	
	s-lock(B)
	read(B)
unlock(A)	
	commit
	unlock(A)
	unlock(B)

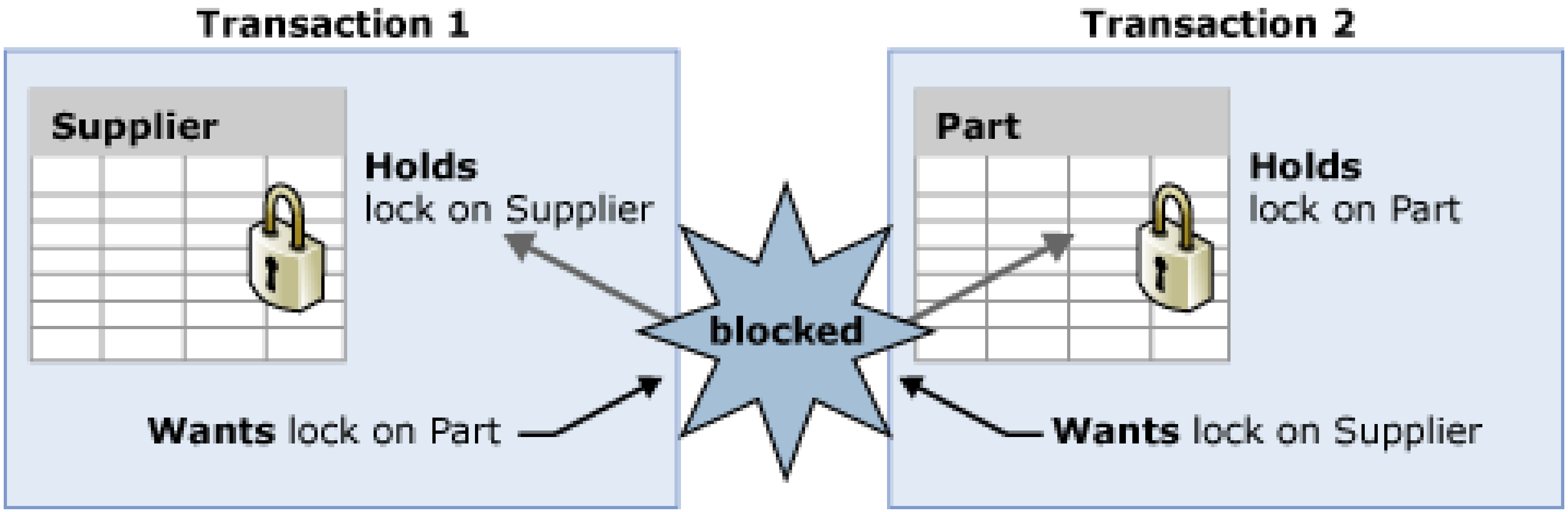
Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

Deadlocks

- The use of locks may lead to deadlocks
- **Definition**
  - A deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock
- The scheduler is responsible for detecting and breaking the deadlock
  - Deadlocks may be broken by simply aborting one of the transactions involved
- How do you choose which transaction to abort?
  - Abort the oldest
  - Abort depending on the complexity of the transactions

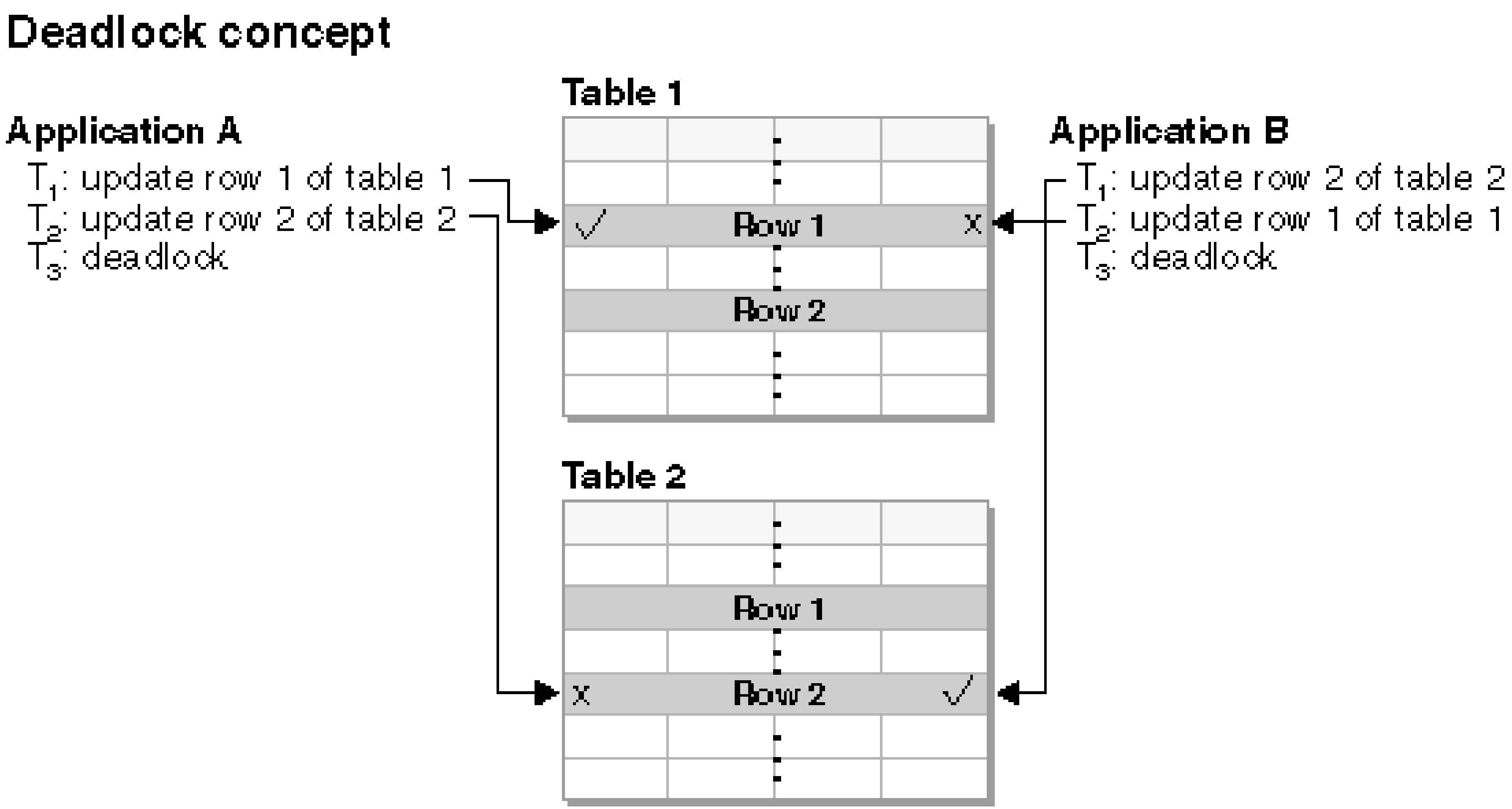
Deadlocks



- 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
```

Deadlock Example



Handling Deadlocks

- Deadlock Detection
  - Use the Precedence graph to identify cycles and select transactions to be aborted
    - Select the transaction involved in most of the cycles
    - Select the oldest transactions
    - Select the one that did the least amount of work
- Deadlock Prevention
  - Lock all objects at the very beginning of a transaction in one atomic action
  - Problem
    - Reduced concurrency: unnecessary access restriction to shared resources

Handling Deadlocks using Timeout

- Rather than detecting deadlock (an overhead), you could just use timeouts, but how long should the timeout be?
- Timeouts
  - Each lock is given a period of time where it is invulnerable
  - After this timeout, it becomes vulnerable
  - If transaction X holds a lock that becomes vulnerable and transaction Y is waiting for X, then X is aborted
  - Problem
    - Hard to decide on an appropriate length of timeout
    - Transactions may be aborted, when the lock becomes vulnerable and another transaction waits, but there is no deadlock

Transaction Isolation Levels

Transaction Isolation Levels

- Determine how much one transaction impacts another
- 4 standard levels
  - Read Uncommitted
  - Read Committed (default in many DMBS)
  - Repeatable Read
  - Serializable
- The *isolation level* of a transaction defines what data *that* transaction may see

Read Uncommitted

- Does not take or 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

- 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
- 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
- If the SELECT statement contains any ranged WHERE clauses, phantom reads can occur
- Reduce concurrency and degrade performance

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

Time-based Concurrency control

< Back

Timestamp Ordering

- The timestamp of a transaction T is the time at which that transaction was initiated in the DBMS: TS(T)
- We can use clock time or an incremental identifier (counter) for TS(T)
- Two timestamps are also associated with each data item x.

  1. read_TS(x) is the TS(T) of the last transaction T to read from x.
  2. write_TS(x) is the TS(T) of the last transaction T to write to x.

Protocol Rules

- Two simple rules to follow:
  1. Before T issues a write(x), check to see if
     - TS(T) < read_TS(x) or if
     - TS(T) < write_TS(x)
     - If so, then <u>abort transaction </u>T
     - If not, then perform write(x) and set write_TS(x) = TS(T)

  2. Before T issues a read(x), check to see if
     - TS(T) < write_TS(x) Then <u>**abort**</u> transaction T.
     - if TS(T) >= write_TS(x) then <u>**execute**</u> read(x) and set read_TS(x) = TS(T) only if TS(T) **>** the current read_TS(x)

Advantages and Limitation

- Note that with timestamp ordering, **deadlock can not occur**
- However, starvation is possible i.e., a transaction keeps getting aborted over and over
  - When a transaction is aborted, it is the restarted and issued a new TS(T)

Cascading rollbacks

- Timestamp ordering can also produce cascading rollbacks:
  - Assume transaction T begins executing and performs some read and write operations on data items ***a, b*** *and* ***c***
  - However, T then reaches a data item it can not read or write and T must then be aborted.
  - Any effects of transaction T must then be rolled back.
  - Before T aborts, however, other transactions (T1, T2 and T3) have read and written data items ***a, b*** *and* ***c*** *so these other transactions must also be rolled* back.
  - There may be other transactions (T4 and T5) that worked with data items read or written by T1, T2 and T3, etc.

Optimistic Concurrency Control

< Back

Pessimistic Concurrency Control

- Two Phase Locking (2PL) and Timestamp Ordering (TO) are *pessimistic concurrency* control protocols:
  - They assume transactions will conflict and take steps to avoid it. i.e., they address the concurrency issues ***while*** *the transaction is executing and* before the transaction commits

Optimistic Concurrency Control

- In an *optimistic concurrency control protocol, we assume that most of the time,* transactions will not conflict thus all of the locking and timestamp checking are not necessary while the transaction is executing
- During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction
- During a validation phase the transactions updates are check to see if they violate serializability

Optimistic Concurrency Control in three stages

1. **Working Stage:** Transactions can read any data item. Writes are done to a local copy of the data item e.g., recorded in a log.
2. **Validation stage:** Transactions containing Write operations that are about to commit are *validated to see if the schedule meets the serializability* requirements.
3. **Write stage:** If the transaction will not conflict with other transactions, then it will be committed (writes to local copy applied to the database). Otherwise, the transaction will be rolled back.

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
- **Transactions manager** ensures that the database remains in a consistent (correct) state despite transaction failures and system failures (e.g., power failures and operating system crashes)
- **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