

# Fundamentals of Database Systems

## Unit 6: Transactions

Schedules

Implementation

Two-phase Locking

(3-4 lectures)  
174A - 2021Wi

# Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: Non-relational data
- Unit 4: DBMS usability, conceptual design
- Unit 5: RDMBS internals and query optimization
- Unit 6: Transactions
  - Writing DB applications
  - Locking and schedules

# Logistics

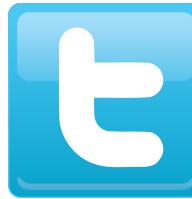
- HW4 due today
- HW5 is out
  - We put what we learned in class in to practice
  - Coding could be non-trivial so start early

# Data Management Pipeline

## Transactions

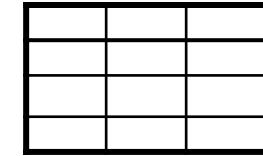
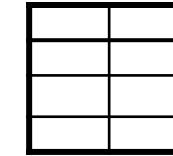
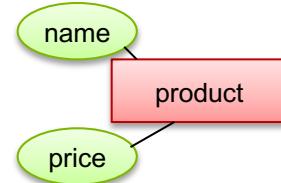


Application  
programmer



Schema  
designer

## Conceptual Schema

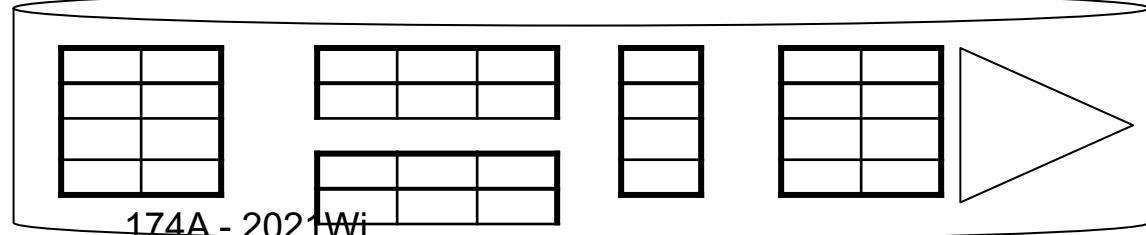


## Transactions



Database  
administrator

## Physical Schema



# Transactions

- We use database transactions everyday
  - Bank \$\$\$ transfers
  - Online shopping
  - Signing up for classes
- Applications that talk to a DB **must** use transactions in order to keep the database consistent.

# What's the big deal?

# Challenges

- Suppose we only serve one app at a time
  - What's the problem?
- Suppose we execute apps concurrently
  - What's the problem?
- Want: multiple operations to be executed *atomically* over the same DBMS

# What can go wrong?

- Manager: balance budgets among projects
  - Remove \$10k from project A
  - Add \$7k to project B
  - Add \$3k to project C
- CEO: check company's total balance
  - `SELECT SUM(money) FROM budget;`
- This is called a dirty / inconsistent read  
aka a **WRITE-READ** conflict

# What can go wrong?

- App 1: 

```
SELECT inventory FROM products  
WHERE pid = 1
```
- App 2: 

```
UPDATE products SET inventory = 0  
WHERE pid = 1
```
- App 1: 

```
SELECT inventory * price FROM products  
WHERE pid = 1
```
- This is known as an unrepeatable read  
aka **READ-WRITE** conflict

# What can go wrong?

Account 1 = \$100

Account 2 = \$100

Total = \$200

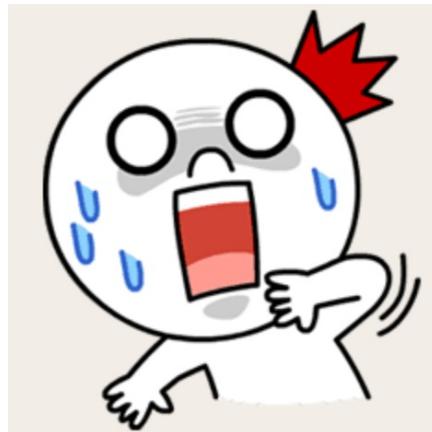
- App 1:
  - Set Account 1 = \$200
  - Set Account 2 = \$0
- App 2:
  - Set Account 2 = \$200
  - Set Account 1 = \$0
- At the end:
  - Total = \$200

- App 1: Set Account 1 = \$200
  - .
- App 2: Set Account 2 = \$200
  - .
- App 1: Set Account 2 = \$0
  - .
- App 2: Set Account 1 = \$0
  - .
- At the end:
  - Total = \$0

This is called the lost update aka **WRITE-WRITE** conflict

# What can go wrong?

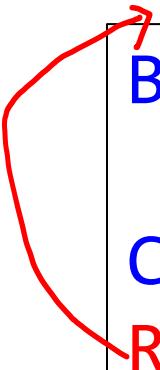
- Buying tickets to the next Bieber concert:
  - Fill up form with your mailing address
  - Put in debit card number
  - Click submit
  - Screen shows money deducted from your account
  - [Your browser crashes]



Lesson:  
Changes to the database  
should be **ALL or NOTHING**

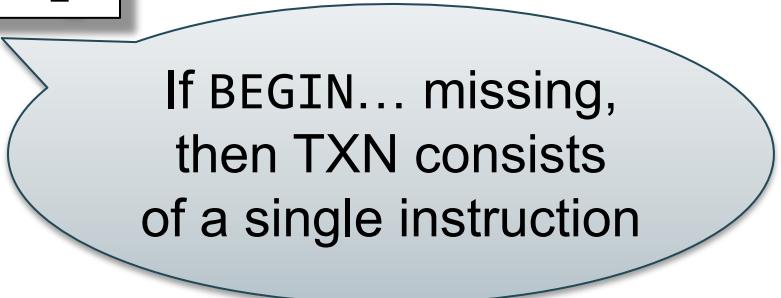
# Transactions

- Collection of statements that are executed atomically (logically speaking)



```
BEGIN TRANSACTION  
[SQL statements]  
COMMIT      or  
ROLLBACK (=ABORT)
```

```
[single SQL statement]
```

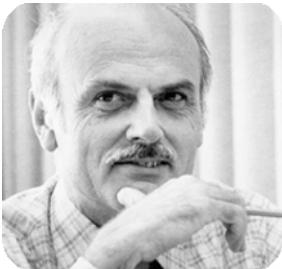


If BEGIN... missing,  
then TXN consists  
of a single instruction

# Turing Awards in Data Management



Charles Bachman, 1973  
*IDS and CODASYL*



Ted Codd, 1981  
*Relational model*



Jim Gray, 1998  
*Transaction processing*



Michael Stonebraker, 2014  
*INGRES and Postgres*

# Know your ~~chemistry~~ transactions: ACID

- **Atomic**
  - State shows either all the effects of txn, or none of them
- **Consistent**
  - Txn moves from a DBMS state where integrity holds, to another where integrity holds
    - remember integrity constraints?
- **Isolated**
  - Effect of txns is the same as txns running one after another (i.e., looks like batch mode)
- **Durable**
  - Once a txn has committed, its effects remain in the database

# Atomic

- **Definition:** A transaction is ATOMIC if all its updates must happen or not at all.

Transactions are often composed of multiple statements. Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds or fails as a whole.

-- Example: move \$100 from A to B:

BEGIN TRANSACTION;

→ UPDATE accounts SET bal = bal - 100 WHERE acct = A;  
→ UPDATE accounts SET bal = bal + 100 WHERE acct = B;  
COMMIT;

# Isolated

Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time). Isolation ensures that each transaction sees the world as if it were the only transaction running.

- **Definition** An execution ensures that txns are isolated, if the effect of each txn is as if it were the only txn running on the system.

-- App 1:

```
BEGIN TRANSACTION;
```

```
    SELECT inventory  
    FROM products  
    WHERE pid = 1;
```

```
    SELECT inventory * price  
    FROM products  
    WHERE pid = 1;
```

```
COMMIT
```

-- App 2:

```
BEGIN TRANSACTION;  
    UPDATE products  
    SET inventory = 0  
    WHERE pid = 1;  
    COMMIT;
```

# Consistent

- Recall: **integrity constraints** govern how values in tables are related to each other
  - Can be enforced by the DBMS, or ensured by the app

Consistency ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants.

- How **consistency** is achieved by the app:
  - App programmer ensures that txns only takes a consistent DB state to another consistent state
  - DB makes sure that txns are executed atomically
- Can defer checking the validity of constraints until the end of a transaction

# Durable

- A transaction is durable if its effects continue to exist after the transaction and even after the program has terminated
- How?
  - By writing to disk!
  - More in 444

# Rollback transactions

- If the app gets to a state where it cannot complete the transaction successfully, execute ROLLBACK
- The DB returns to the state prior to the transaction
- What are examples of such program states?

# ACID

- Atomic
- Consistent
- Isolated
- Durable
- Enjoy this in HW5!
- Again: by default each statement is its own txn
  - Unless auto-commit is off then each statement starts a new txn

# Implementing Transactions

Need to address two problems:

- “**I**” – Isolation:

- Means concurrency control
  - We will discuss this

parallel same as sequential

- “**A**” – Atomicity:

- Means recover from crash
  - We will not discuss this

all or none.

# Transaction Schedules

# Modeling a Transaction

- Database = a collection of elements
  - An element can be a record (logical elements)
  - Or can be a disc block (physical element)

Database: A    B    C    D    ...

- Transaction = sequence of read/writes of elements

# Schedules

A **schedule** is a sequence  
of interleaved actions  
from all transactions

# Serial Schedule

- A serial schedule is one in which transactions are executed one after the other, in some sequential order
- **Fact:** nothing can go wrong if the system executes transactions serially
- But DBMS don't do that because we want better overall system performance

A and B are elements  
in the database  
t and s are variables  
in txn source code

## Example

T1	T2
READ(A, t)	READ(A, s)
$t := t + 100$	$s := s^2$
WRITE(A, t)	WRITE(A, s)
READ(B, t)	READ(B, s)
$t := t + 100$	$s := s^2$
WRITE(B, t)	WRITE(B, s)

# Example of a (Serial) Schedule

T1	T2
READ(A, t)	
$t := t + 100$	
WRITE(A, t)	
READ(B, t)	Schedules in which the transactions are executed non-interleaved, i.e.
$t := t + 100$	
WRITE(B, t)	
	READ(A, s)
	$s := s^* 2$
	WRITE(A, s)
	READ(B, s)
	$s := s^* 2$
	WRITE(B, s)

# Another Serial Schedule

T1	T2
	READ(A,s)
	$s := s^*2$
	WRITE(A,s)
	READ(B,s)
	$s := s^*2$
	WRITE(B,s)
READ(A, t)	
$t := t+100$	
WRITE(A, t)	
READ(B, t)	
$t := t+100$	
WRITE(B,t)	

# Review: Serializable Schedule

A schedule is **serializable** if it is equivalent to a serial schedule

A serial schedule is always a serializable schedule.

# A Serializable Schedule

T1

READ(A, t)

t := t+100

WRITE(A, t)

T2

READ(A,s)

s := s\*2

WRITE(A,s)

READ(B, t)

t := t+100

WRITE(B,t)

READ(B,s)

s := s\*2

WRITE(B,s)

This is a **serializable** schedule.

This is **NOT** a serial schedule

# A Non-S Serializable Schedule

T1

READ(A, t)

$t := t + 100$

WRITE(A, t)

T2

READ(A,s)

$s := s^*2$

WRITE(A,s)

READ(B,s)

$s := s^*2$

WRITE(B,s)

READ(B, t)

$t := t + 100$

WRITE(B,t)

# How do We Know if a Schedule is Serializable?

Notation:

$$T_1: r_1(A); w_1(A); r_1(B); w_1(B)$$
$$T_2: r_2(A); w_2(A); r_2(B); w_2(B)$$

Key Idea: Focus on *conflicting* operations

# Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

swapping will change program behavior

# Conflict Serializability

**Conflicts:** (i.e., swapping will change program behavior)

Two actions by same transaction  $T_i$ :

same transaction, diff element, read-write

$r_i(X); w_i(Y)$

Two writes by  $T_i, T_j$  to same element

write-write diff transaction to same element

$w_i(X); w_j(X)$

Read/write by  $T_i, T_j$  to same element

read write diff transaction same

$w_i(X); r_j(X)$

$r_i(X); w_j(X)$

# Conflict Serializability

- A schedule is *conflict serializable* if it can be transformed into **a serial schedule** by a series of swappings of adjacent non-conflicting actions
- Every conflict-serializable schedule is serializable
- The converse is not true (why?)

conflict serializable is a subset of serializable. If a schedule is serializable, it is conflict serializable.

# Conflict Serializability

Example:

```
r1(A); w1(A); r2(A); w2(A); r1(B); w1(B); r2(B); w2(B)
```

# Conflict Serializability

Example:

```
r1(A); w1(A); r2(A); w2(A); r1(B); w1(B); r2(B); w2(B)
```

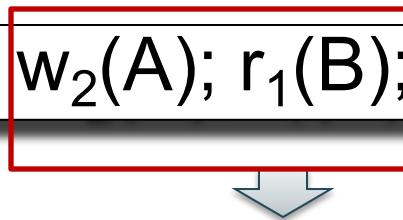


```
r1(A); w1(A); r1(B); w1(B); r2(A); w2(A); r2(B); w2(B)
```

# Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$



OK, diff trans, diff element, read-write

SWAP

"by a series ofswappings of adjacent non-co

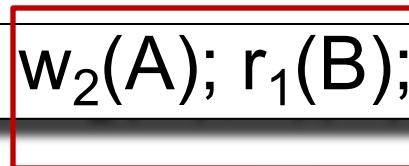
$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

# Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

read-write conflict



$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

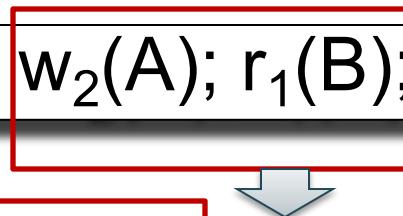


$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

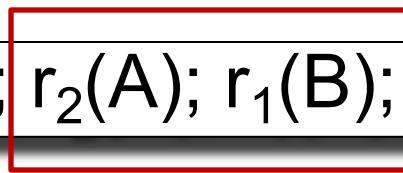
# Conflict Serializability

Example:

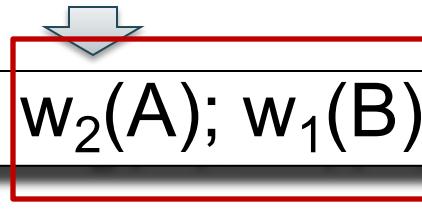
$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$



$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B)$



....

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

# Serializable, Not Conflict-Serializable

T1	T2
READ(A, t)	
$t := t + 100$	
WRITE(A, t)	
	READ(A,s)
	$s := s + 200$
	WRITE(A,s)
	READ(B,s)
	$s := s + 200$
	WRITE(B,s)
READ(B, t)	
$t := t + 100$	
WRITE(B,t)	

A schedule is conflict serializable if it can be transformed into a

# Testing for Conflict-Serializability

## Precedence graph:

- A node for each transaction  $T_i$ ,
- An edge from  $T_i$  to  $T_j$  whenever an action in  $T_i$  conflicts with, and comes before an action in  $T_j$
- The schedule is conflict-serializable iff the precedence graph is acyclic

# Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

1

2

3

# Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



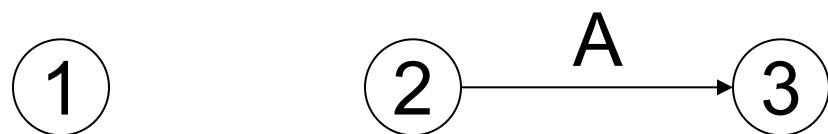
1

2

3

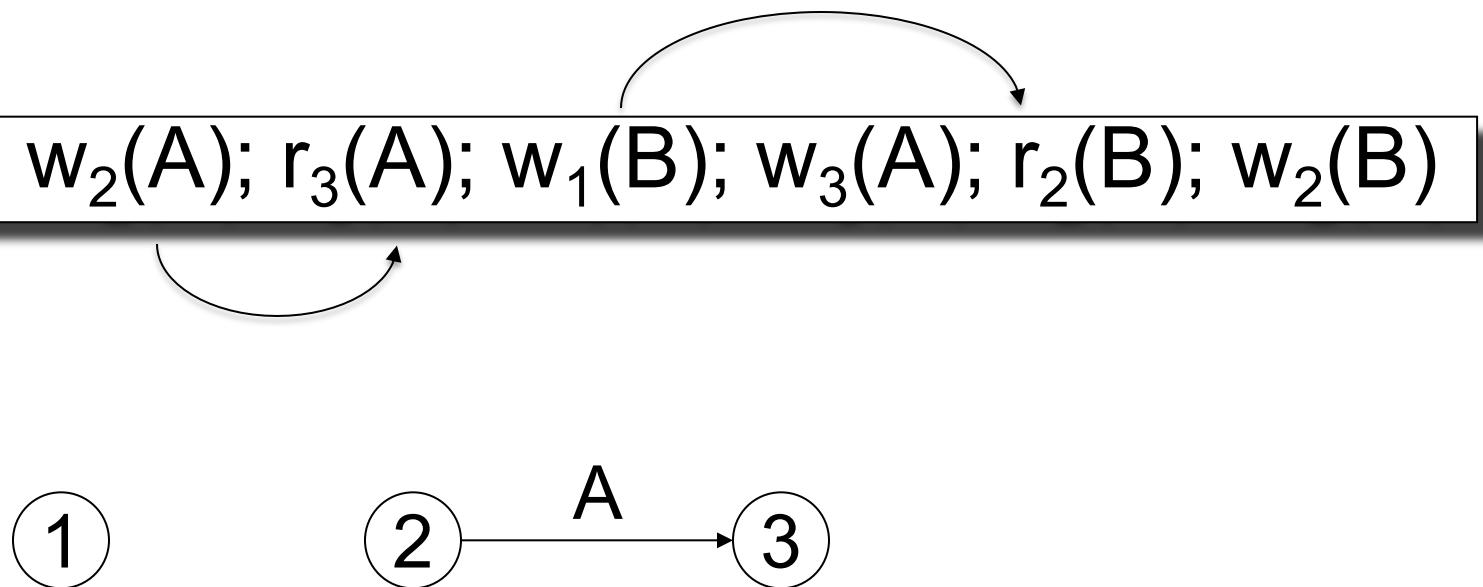
# Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



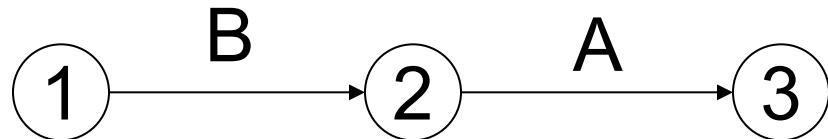
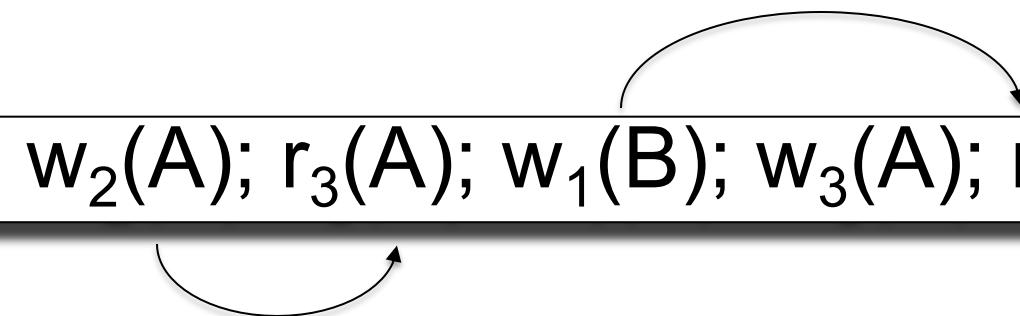
# Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



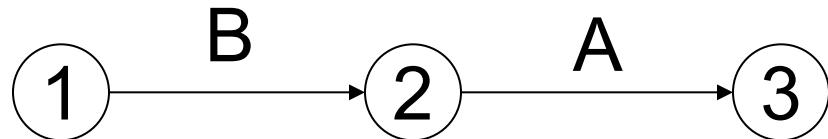
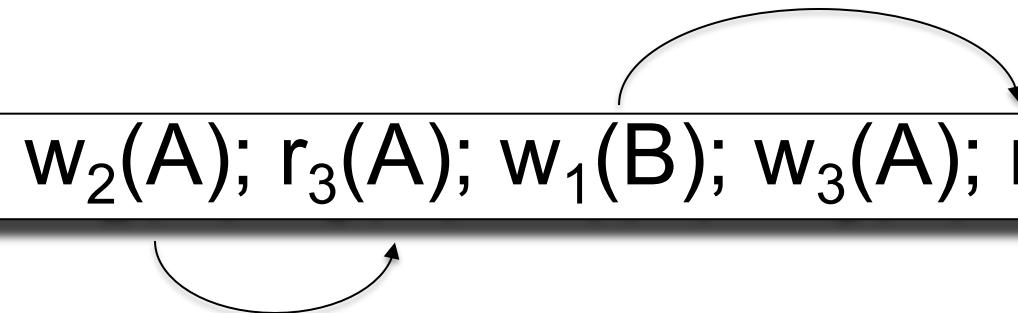
# Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



# Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



This schedule is conflict-serializable

# Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

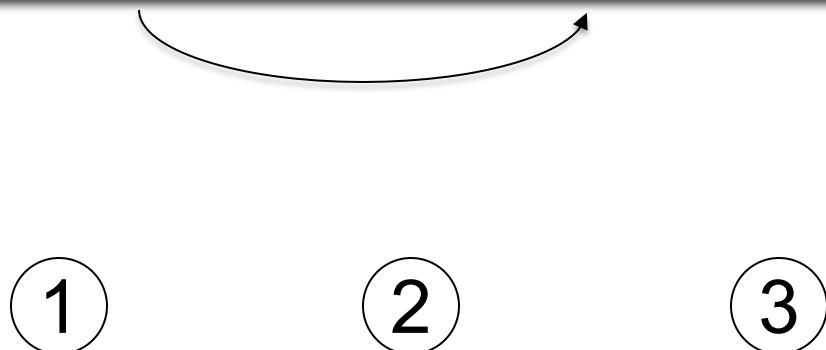
1

2

3

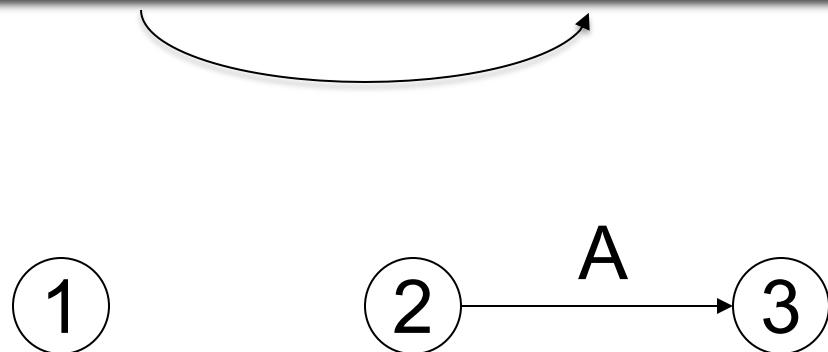
# Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



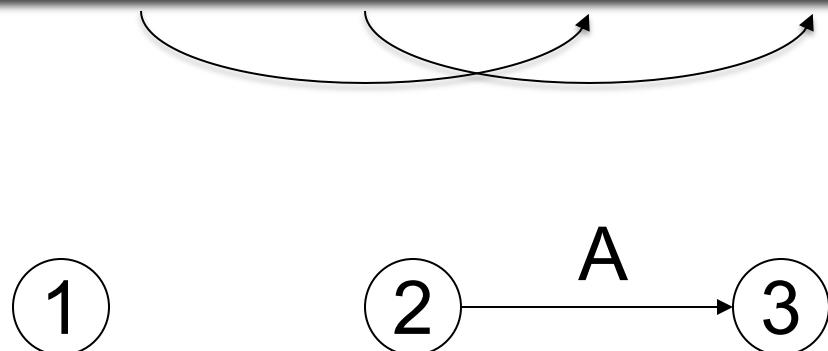
## Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



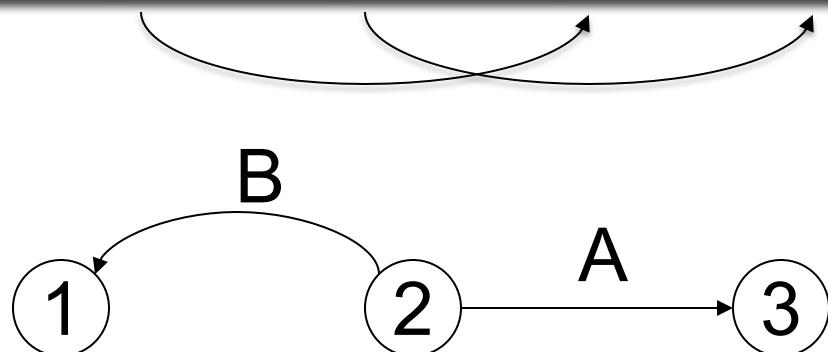
# Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



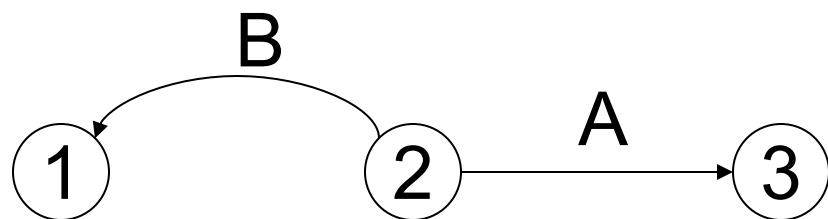
## Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



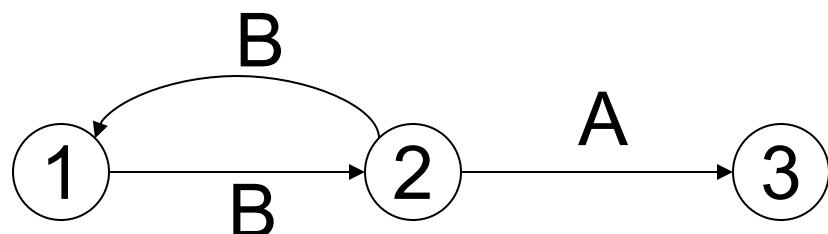
## Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



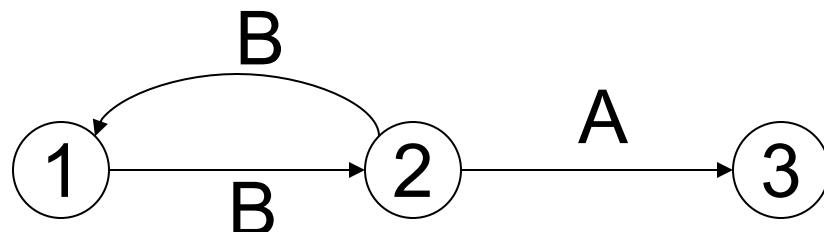
## Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



## Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



If you find cycles in the conflict diagram, then the schedule is not conflict-serializable.

This schedule **is NOT** conflict-serializable

# Implementing Transactions

# Scheduler

- Scheduler a.k.a. Concurrency Control Manager
  - The module that schedules the transaction's actions
  - Goal: ensure the schedule is serializable
- We discuss next how a scheduler may be implemented

# Implementing a Scheduler

Two major approaches:

- **Locking Scheduler**
  - Aka “pessimistic concurrency control”
  - SQLite, SQL Server, DB2
- **Multiversion Concurrency Control (MVCC)**
  - Aka “optimistic concurrency control”, *SQLite*
  - Postgres, Oracle: Snapshot Isolation (SI)

WAL

We discuss only locking schedulers in this class

# Lock-based Implementation of Transactions

# Locking Scheduler

Simple idea:

- Each element has a unique **lock**
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must **release** the lock(s)

By using locks scheduler ensures conflict-serializability

# What Data Elements are Locked?

Major differences between vendors:

- Lock on the entire database
  - SQLite
- Lock on individual records (“elements”)
  - SQL Server, DB2, etc

# Actions on Locks

$L_i(A)$  = transaction  $T_i$  acquires lock for element A

$U_i(A)$  = transaction  $T_i$  releases lock for element A

Let's see this in action...

# A Non-S Serializable Schedule

T1	T2
READ(A)	
$A := A + 100$	
WRITE(A)	
r1(A)w1(A)r2(A)w2(A)r2(B)w2(B)r1(B)w1(B)	
	READ(A)
	$A := A^*2$
	WRITE(A)
	READ(B)
	$B := B^*2$
	WRITE(B)
READ(B)	
$B := B + 100$	
WRITE(B)	

# Example

T1

$L_1(A)$ ; READ(A)

$A := A + 100$

WRITE(A);  $U_1(A)$ ;  $L_1(B)$

READ(B)

$B := B + 100$

WRITE(B);  $U_1(B)$ ;

T2

$L_2(A)$ ; READ(A)

$A := A * 2$

WRITE(A);  $U_2(A)$ ;

$L_2(B)$ ; BLOCKED...

r1(A)w1(A)r2(A)w2(A)r2(B)w2(B)r1(B)w1(B)

...GRANTED; READ(B)

$B := B * 2$

WRITE(B);  $U_2(B)$ ;

Scheduler has ensured a conflict-serializable schedule

# But...

T1

---

$L_1(A)$ ; READ(A)  
A := A+100  
WRITE(A);  $U_1(A)$ ;

T2

$L_2(A)$ ; READ(A)  
A := A\*2  
WRITE(A);  $U_2(A)$ ;  
 $L_2(B)$ ; READ(B)  
B := B\*2  
WRITE(B);  $U_2(B)$ ;

$L_1(B)$ ; READ(B)  
B := B+100  
WRITE(B);  $U_1(B)$ ;

Locks did not enforce conflict-serializability!!! What's wrong ?

# Two Phase Locking (2PL)

The 2PL rule:

In every transaction, all lock requests must precede all unlock requests

# Example: 2PL transactions

T1

L<sub>1</sub>(A); L<sub>1</sub>(B); READ(A)

A := A+100

WRITE(A); U<sub>1</sub>(A)

READ(B)

B := B+100

WRITE(B); U<sub>1</sub>(B);

T2

L<sub>2</sub>(A); READ(A)

A := A\*2

WRITE(A);

L<sub>2</sub>(B); BLOCKED...

...GRANTED; READ(B)

B := B\*2

WRITE(B); U<sub>2</sub>(A); U<sub>2</sub>(B);

Now it is conflict-serializable

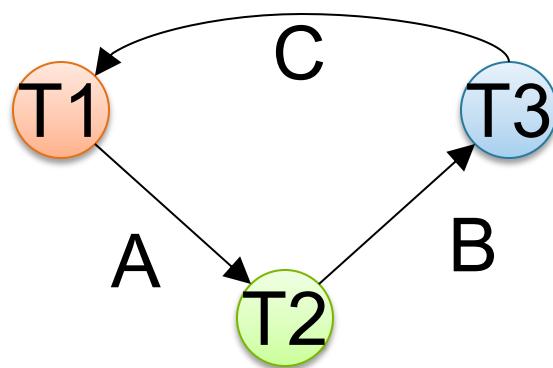
# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

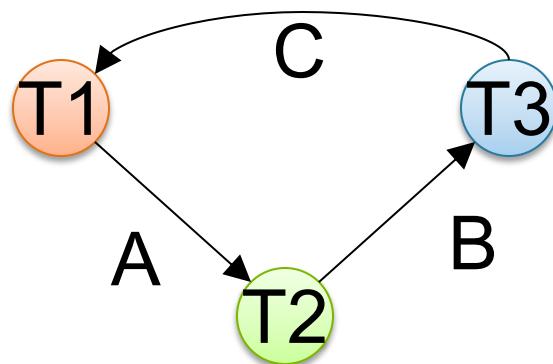
**Proof.** Suppose not: then there exists a cycle in the precedence graph.



# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

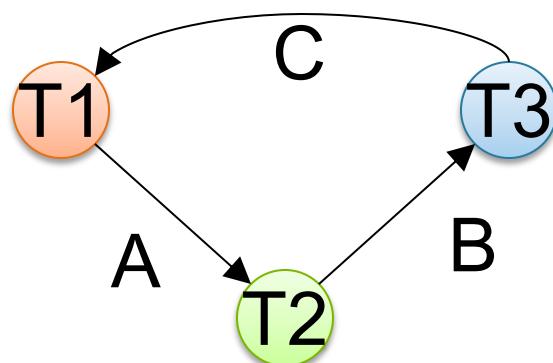


Then there is the following temporal cycle in the schedule:

# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.



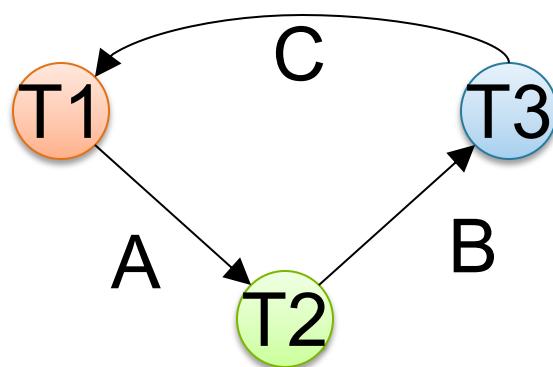
Then there is the following temporal cycle in the schedule:  
 $U_1(A) \rightarrow L_2(A)$  why?

$U_1(A)$  happened strictly before  $L_2(A)$

# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

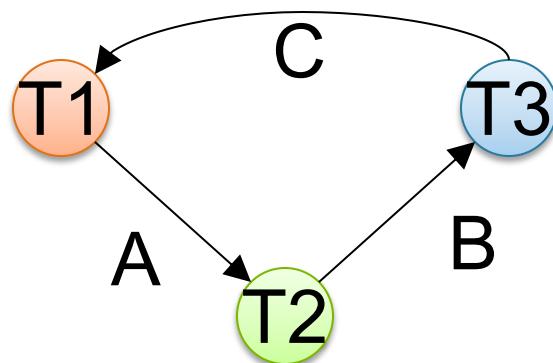


Then there is the following temporal cycle in the schedule:  
 $U_1(A) \rightarrow L_2(A)$  why?

# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.



Then there is the following temporal cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$

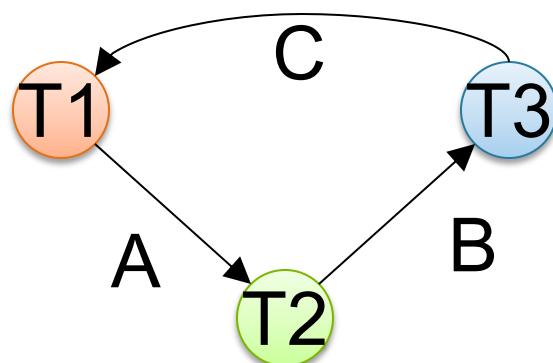
why?

$L_2(A)$  happened strictly before  $U_1(A)$

# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.

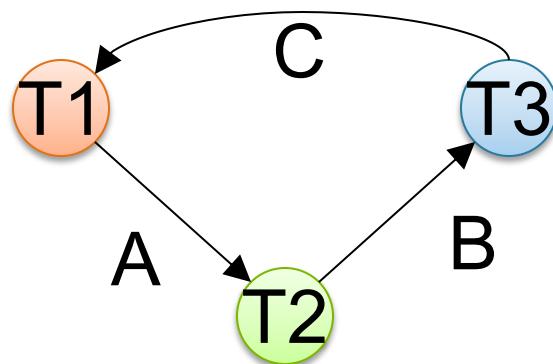


Then there is the following temporal cycle in the schedule:  
 $U_1(A) \rightarrow L_2(A)$   
 $L_2(A) \rightarrow U_2(B)$       why?

# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.



Then there is the following temporal cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$

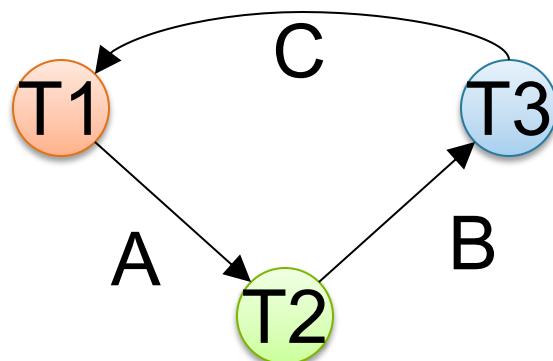
$U_2(B) \rightarrow L_3(B)$

why?

# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.



Then there is the following temporal cycle in the schedule:

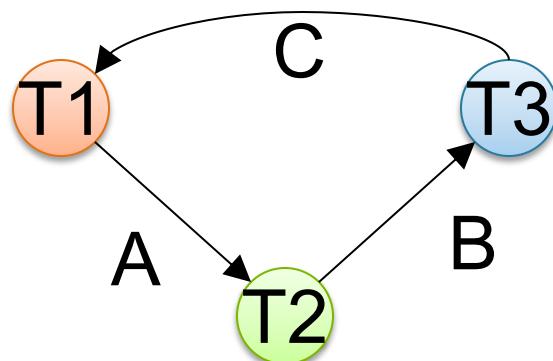
$$\begin{aligned} U_1(A) &\rightarrow L_2(A) \\ L_2(A) &\rightarrow U_2(B) \\ U_2(B) &\rightarrow L_3(B) \end{aligned}$$

.....etc.....

# Two Phase Locking (2PL)

**Theorem:** 2PL ensures conflict serializability

**Proof.** Suppose not: then there exists a cycle in the precedence graph.



Then there is the following temporal cycle in the schedule:

- $U_1(A) \rightarrow L_2(A)$
- $L_2(A) \rightarrow U_2(B)$
- $U_2(B) \rightarrow L_3(B)$
- $L_3(B) \rightarrow U_3(C)$
- $U_3(C) \rightarrow L_1(C)$
- $L_1(C) \rightarrow U_1(A)$

Cycle in time:  
Contradiction

# A New Problem: Non-recoverable Schedule

T1

$L_1(A); L_1(B);$  READ(A)

$A := A + 100$

WRITE(A);  $U_1(A)$

READ(B)

$B := B + 100$

WRITE(B);  $U_1(B)$

Rollback

T2

$L_2(A);$  READ(A)

$A := A * 2$

WRITE(A);

$L_2(B);$  BLOCKED...

...GRANTED; READ(B)

$B := B * 2$

WRITE(B);  $U_2(A); U_2(B);$

Commit

# A New Problem: Non-recoverable Schedule

T1

$L_1(A)$ ;  $L_1(B)$ ; READ(A)

$A := A + 100$

WRITE(A);  $U_1(A)$

READ(B)

$B := B + 100$

WRITE(B);  $U_1(B)$

T2

$L_2(A)$ ; READ(A)

$A := A * 2$

WRITE(A);

$L_2(B)$ ; BLOCKED...

...GRANTED; READ(B)

$B := B * 2$

WRITE(B);  $U_2(A)$ ;  $U_2(B)$

Commit

Rollback

Elements A, B written  
by T1 are restored  
to their original value.

# A New Problem: Non-recoverable Schedule

T1

$L_1(A)$ ;  $L_1(B)$ ; READ(A)

$A := A + 100$

WRITE(A);  $U_1(A)$

READ(B)

$B := B + 100$

WRITE(B);  $U_1(B)$

T2

$L_2(A)$ ; READ(A)

$A := A * 2$

WRITE(A);

$L_2(B)$ ; BLOCKED...

Dirty reads of  
A, B lead to  
incorrect writes.

...GRANTED; READ(B)

$B := B * 2$

WRITE(B);  $U_2(A)$ ;  $U_2(B)$ ;

Commit

Rollback

Elements A, B written  
by T1 are restored  
to their original value.

# A New Problem: Non-recoverable Schedule

T1

L<sub>1</sub>(A); L<sub>1</sub>(B); READ(A)

A := A+100

WRITE(A); U<sub>1</sub>(A)

READ(B)

B := B+100

WRITE(B); U<sub>1</sub>(B);

T2

All the lock operations precedes all the unlock operations

L<sub>2</sub>(A); READ(A)

A := A\*2

WRITE(A);

L<sub>2</sub>(B); BLOCKED...

Dirty reads of  
A, B lead to  
incorrect writes.

...GRANTED: READ(B)

B := B\*2

WRITE(B); U<sub>2</sub>(A); U<sub>2</sub>(B);

Commit

Rollback

Elements A, B written  
by T1 are restored  
to their original value.

2021Wi

Can no longer undo!

# Strict 2PL

The Strict 2PL rule:

All locks are held until commit/abort:

All unlocks are done together with commit/abort.

With strict 2PL, we will get schedules that are both conflict-serializable and recoverable

# Strict 2PL

T1

$L_1(A)$ ; READ(A)

$A := A + 100$

WRITE(A);

T2

$L_2(A)$ ; BLOCKED...

$L_1(B)$ ; READ(B)

$B := B + 100$

WRITE(B);

Rollback &  $U_1(A); U_1(B)$ ;

...GRANTED; READ(A)

$A := A * 2$

WRITE(A);

$L_2(B)$ ; READ(B)

$B := B * 2$

WRITE(B);

When the transaction commits/aborts, then release

Commit &  $U_2(A); U_2(B)$ ;

# Strict 2PL

- Lock-based systems always use strict 2PL
- Easy to implement:
  - Before a transaction reads or writes an element A, insert an L(A)
  - When the transaction commits/aborts, then release all locks
- Ensures both conflict serializability and recoverability

# Another problem: Deadlocks

- $T_1$ : R(A), W(B)
  - $T_2$ : R(B), W(A)
- 
- $T_1$  holds the lock on A, waits for B
  - $T_2$  holds the lock on B, waits for A

This is a deadlock!

# Another problem: Deadlocks

To detect a deadlocks, search for a cycle in the waits-for graph:

- $T_1$  waits for a lock held by  $T_2$ ;
- $T_2$  waits for a lock held by  $T_3$ ;
- . . .
- $T_n$  waits for a lock held by  $T_1$

Relatively expensive: check periodically, if deadlock is found, then abort one TXN;  
re-check for deadlock more often (why?)

# Lock Modes

- $S$  = shared lock (for READ)
- $X$  = exclusive lock (for WRITE)

Lock compatibility matrix:

	None	$S$	$X$
None			
$S$			
$X$			

# Lock Modes

- $S$  = shared lock (for READ)
- $X$  = exclusive lock (for WRITE)

Lock compatibility matrix:

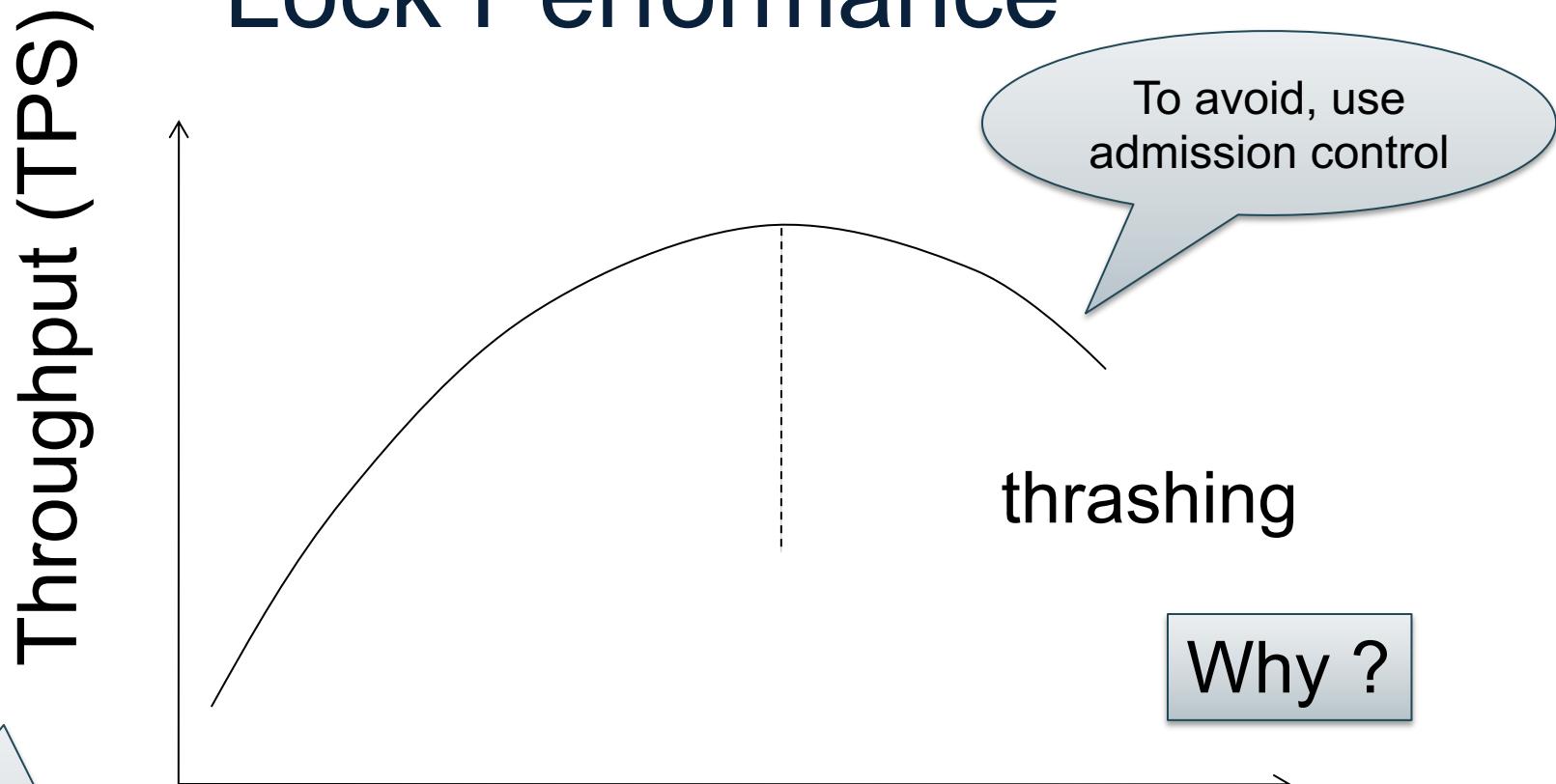
	None	$S$	$X$
None	✓	✓	✓
$S$	✓	✓	✗
$X$	✓	✗	✗

# Lock Granularity

- **Fine granularity locking** (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
  - E.g., SQL Server
- **Coarse grain locking** (e.g., tables, entire database)
  - Many false conflicts
  - Less overhead in managing locks
  - E.g., SQL Lite
- **Solution: lock escalation changes granularity as needed**

fine granularity -> tends to lock more objects

# Lock Performance



TPS =  
Transactions  
per second

# Phantom Problem

- So far we have assumed the database to be a *static collection of elements (=tuples)*
- If tuples are inserted/deleted then the *phantom problem* appears

The columns being returned have different number of row

Suppose there are two blue products, A1, A2:

## Phantom Problem

T1

```
SELECT *  
FROM Product  
WHERE color='blue'
```

T2

```
INSERT INTO Product(name, color)  
VALUES ('A3','blue')
```

```
SELECT *  
FROM Product  
WHERE color='blue'
```

Is this schedule serializable ?

Suppose there are two blue products, A1, A2:

## Phantom Problem

T1

```
SELECT *  
FROM Product  
WHERE color='blue'
```

T2

```
INSERT INTO Product(name, color)  
VALUES ('A3','blue')
```

```
SELECT *  
FROM Product  
WHERE color='blue'
```

Is this schedule serializable ?

No: T1 sees a “phantom” product A3

Suppose there are two blue products, A1, A2:

## Phantom Problem

T1

```
SELECT *
FROM Product
WHERE color='blue'
```

T2

```
INSERT INTO Product(name, color)
VALUES ('A3','blue')
```

```
SELECT *
FROM Product
WHERE color='blue'
```

R<sub>1</sub>(A1);R<sub>1</sub>(A2);W<sub>2</sub>(A3);R<sub>1</sub>(A1);R<sub>1</sub>(A2);R<sub>1</sub>(A3)

Suppose there are two blue products, A1, A2:

## Phantom Problem

T1

```
SELECT *
FROM Product
WHERE color='blue'
```

T2

```
INSERT INTO Product(name, color)
VALUES ('A3','blue')
```

```
SELECT *
FROM Product
WHERE color='blue'
```

```
R1(A1);R1(A2);W2(A3);R1(A1);R1(A2);R1(A3)
```

```
W2(A3);R1(A1);R1(A2);R1(A1);R1(A2);R1(A3)96
```

Suppose there are two blue products, A1, A2:

## Phantom Problem

T1

```
SELECT *  
FROM Product  
WHERE color='blue'
```

T2

```
INSERT INTO Product(name, color)  
VALUES ('A3','blue')
```

```
SELECT *  
FROM Product  
WHERE color='blue'
```

But this is conflict-serializable!

```
R1(A1);R1(A2);W2(A3);R1(A1);R1(A2);R1(A3)
```

```
W2(A3);R1(A1);R1(A2);R1(A1);R1(A2);R1(A3)97
```

# Phantom Problem

- A “phantom” is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution
- In our example:
  - T1: reads list of products
  - T2: inserts a new product
  - T1: re-reads: a new product appears !
- Conflict-serializability assumes DB is static
- When DB is dynamic then c-s is not serializable.



# Dealing With Phantoms

- Lock the entire table
- Lock the index entry for ‘blue’
  - If index is available
- Or use predicate locks
  - A lock on an arbitrary predicate

Dealing with phantoms is expensive !

# Summary of Serializability

- Serializable schedule = equivalent to a serial schedule
- (strict) 2PL guarantees *conflict serializability*
  - What is the difference?
- **Static database:**
  - *Conflict serializability* implies serializability
- **Dynamic database:**
  - This no longer holds

In a dynamic database because we have the phantom problem, conflict serializable

# Weaker Isolation Levels

- Serializable are expensive to implement
- SQL allows more efficient implementations, which are not serializable: *weak isolation levels*
- Certain conflicts may happen:
  - Dirty reads
  - Inconsistent reads
  - Unrepeatable reads
  - Lost updates

# Dirty Reads

Write-Read Conflict

T<sub>1</sub>: WRITE(A)

T<sub>2</sub>: READ(A)

T<sub>1</sub>: ABORT

# Inconsistent Read

## Write-Read Conflict

$T_1$ :  $A := 20; B := 20;$

$T_1$ : WRITE(A)

$T_1$ : WRITE(B)

$T_2$ : READ(A);  
 $T_2$ : READ(B);

# Unrepeatable Read

**Read-Write Conflict**

T<sub>1</sub>: WRITE(A)

T<sub>2</sub>: READ(A);

T<sub>2</sub>: READ(A);

# Lost Update

## Write-Write Conflict

$T_1$ : READ(A)

$T_1$ :  $A := A + 5$

$T_1$ : WRITE(A)

$T_2$ : READ(A);

$T_2$ :  $A := A * 1.3$

$T_2$ : WRITE(A);

# Isolation Levels in SQL

1. “Dirty reads”

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE



# 1. Isolation Level: Dirty Reads

- “Long duration” WRITE locks
  - Strict 2PL
- No READ locks
  - Read-only transactions are never delayed

Possible problems: dirty and inconsistent reads

## 2. Isolation Level: Read Committed

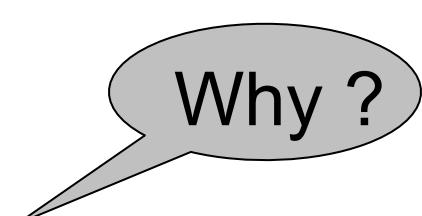
- “Long duration” WRITE locks
  - Strict 2PL
- “Short duration” READ locks
  - Only acquire lock while reading (not 2PL)

Unrepeatable reads:

When reading same element twice,  
may get two different values

### 3. Isolation Level: Repeatable Read

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL



Why ?

This is not serializable yet !!!

# 4. Isolation Level Serializable

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL
- Predicate locking
  - To deal with phantoms

# Beware!

In commercial DBMSs:

- Default level is often NOT serializable
- Default level differs between DBMSs
- Some engines support subset of levels!
- Serializable may not be exactly ACID
  - Locking ensures isolation, not atomicity
- Also, some DBMSs do NOT use locking and different isolation levels can lead to different pbs
- **Bottom line: Read the doc for your DBMS!**

# Case Study: SQLite

- SQLite is very simple
- More info: <http://www.sqlite.org/atomiccommit.html>
- Lock types
  - READ LOCK (to read)
  - RESERVED LOCK (to write)
  - PENDING LOCK (wants to commit)
  - EXCLUSIVE LOCK (to commit)

# SQLite

## Step 1: when a transaction begins

- Acquire a **READ LOCK** (aka "SHARED" lock)
- All these transactions may read happily
- They all read data from the database file
- If the transaction commits without writing anything, then it simply releases the lock

# SQLite

**Step 2:** when one transaction wants to write

- Acquire a **RESERVED LOCK**
- May coexists with many READ LOCKs
- Writer TXN may write; these updates are only in main memory; others don't see the updates
- Reader TXN continue to read from the file
- New readers accepted
- No other TXN is allowed a RESERVED LOCK

# SQLite

**Step 3:** when writer transaction wants to commit, it needs *exclusive lock*, which can't coexists with *read locks*

- Acquire a **PENDING LOCK**
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released



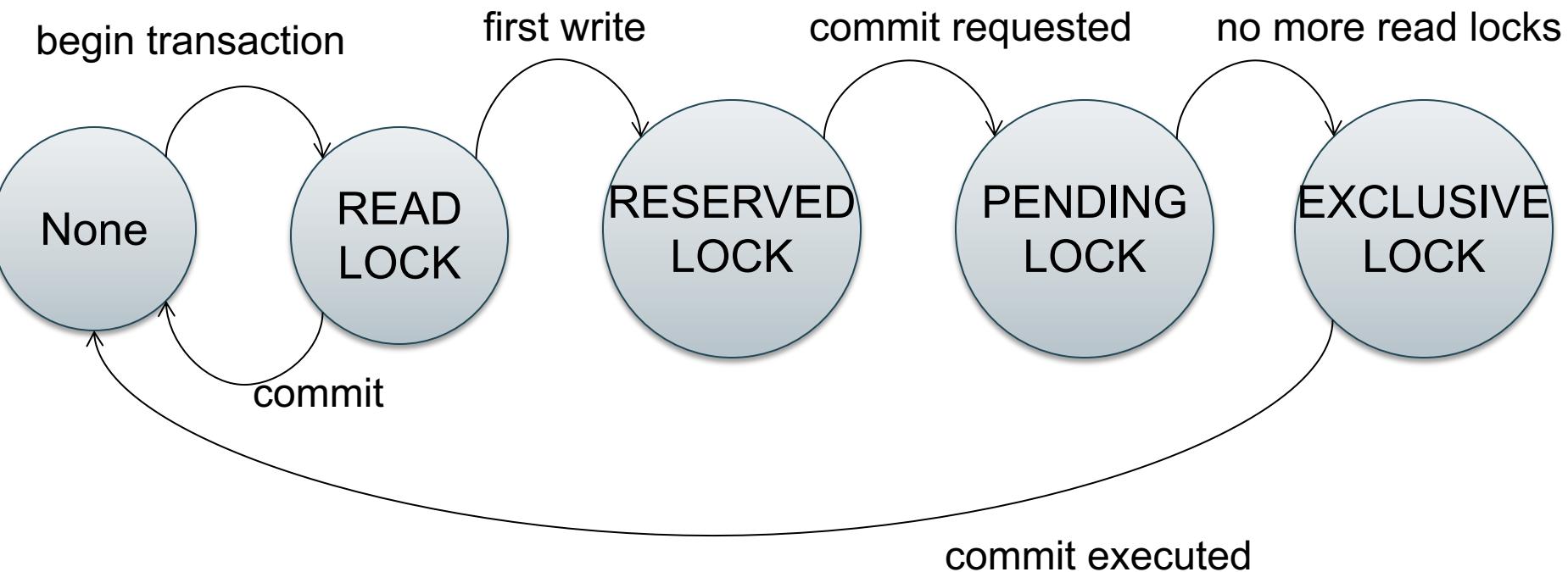
Why not write to disk right now?

# SQLite

**Step 4:** when all read locks have been released

- Acquire the **EXCLUSIVE LOCK**
- Nobody can touch the database now
- All updates are written permanently to the database file
- Release the lock and **COMMIT**

# SQLite



# SQLite Demo

```
create table r(a int, b int);
insert into r values (1,10);
insert into r values (2,20);
insert into r values (3,30);
```

# Demonstrating Locking in SQLite

T1:

```
begin transaction;  
select * from r;  
-- T1 has a READ LOCK
```

T2:

```
begin transaction;  
select * from r;  
-- T2 has a READ LOCK
```

# Demonstrating Locking in SQLite

T1:

```
update r set b=11 where a=1;
```

-- T1 has a RESERVED LOCK

T2:

```
update r set b=21 where a=2;
```

-- T2 asked for a RESERVED LOCK: DENIED

# Demonstrating Locking in SQLite

T3:

```
begin transaction;
```

```
select * from r;
```

```
commit;
```

-- everything works fine, could obtain READ LOCK

# Demonstrating Locking in SQLite

T1:

commit;

-- SQL error: database is locked

-- T1 asked for PENDING LOCK -- GRANTED

-- T1 asked for EXCLUSIVE LOCK -- DENIED

# Demonstrating Locking in SQLite

T3':

```
begin transaction;
```

```
select * from r;
```

-- T3 asked for READ LOCK-- DENIED (due to  
T1)

T2:

```
commit;
```

-- releases the last READ LOCK; T1 can commit

# How do anomalies show up in schedules?

- What could go wrong if we didn't have concurrency control:
  - Dirty reads (including inconsistent reads)
  - Unrepeatable reads
  - Lost updates

Many other things can go wrong too

# Demonstration with SQL Server

## **Application 1:**

```
create table R(a int);
insert into R values(1);
set transaction isolation level serializable;
begin transaction;
select * from R; -- get a shared lock
```

## **Application 2:**

```
set transaction isolation level serializable;
begin transaction;
select * from R; -- get a shared lock
insert into R values(2); -- blocked waiting on exclusive lock
    -- App 2 unblocks and executes insert after app 1
commits/aborts
```

# Demonstration with SQL Server

## Application 1:

```
create table R(a int);
insert into R values(1);
set transaction isolation level repeatable read;
begin transaction;
select * from R; -- get a shared lock
```

## Application 2:

```
set transaction isolation level repeatable read;
begin transaction;
select * from R; -- get a shared lock
insert into R values(3); -- gets an exclusive lock on new tuple
    -- If app 1 reads now, it blocks because read dirty
    -- If app 1 reads after app 2 commits, app 1 sees new value
```