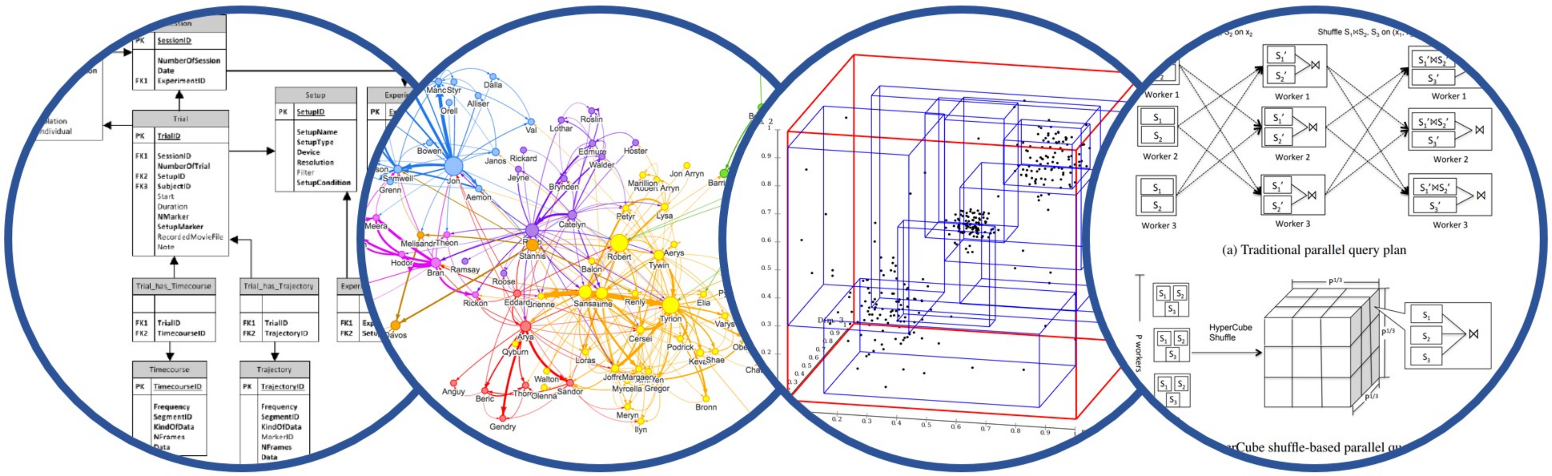


# Database System Internals

# Concurrency Control - Locking

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

# Announcement

- Lab 3 out now – Transaction Scheduler
  - Make sure to go to section for important tips!
    - Synchronized keyword is very important, but making all methods synchronized will lead to problems
  - Part 1 due Friday, May 9
  - Part 2 due Wednesday, May 14

# About Lab 3

- In lab 3, we implement transactions
- Focus on concurrency control
  - Want to run many transactions at the same time
  - Transactions want to read and write same pages
  - Will use locks to ensure conflict serializable execution
  - Use strict 2PL
- Build your own lock manager
  - Understand how locking works in depth
  - Ensure transactions rather than threads hold locks
    - Many threads can execute different pieces of the same transaction
    - Need to detect deadlocks and resolve them by aborting a transaction
  - But use Java synchronization to protect your data structures

# Scheduler

- The scheduler:
- Module that schedules the transaction's actions, ensuring serializability

- Two main approaches
- Pessimistic: locks
- Optimistic: timestamps, multi-version, validation

# Pessimistic 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)

# Notation

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

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

# A Non-Serializable Schedule

| T1 | T2 |
|---|---|
| READ(A, t) | |
| t := t+100 | |
| WRITE(A, t) | |
| | 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) | |

# Example

| T1 | T2 |
|---|---|
| $L_1(A)$; READ(A, t) | |
| t := t+100 | |
| WRITE(A, t); $U_1(A)$; $L_1(B)$ | |
| | $L_2(A)$; READ(A,s) |
| | s := s*2 |
| | WRITE(A,s); $U_2(A)$; |
| | $L_2(B)$; DENIED… |
| READ(B, t) | |
| t := t+100 | |
| WRITE(B,t); $U_1(B)$; | |
| | …GRANTED; READ(B,s) |
| | s := s*2 |
| | WRITE(B,s); $U_2(B)$; |

Scheduler has ensured a conflict-serializable schedule

# But…

| T1 | T2 |
|---|---|
| $L_1(A)$; READ(A, t) | |
| t := t+100 | |
| WRITE(A, t); $U_1(A)$; | |
| | $L_2(A)$; READ(A,s) |
| | s := s*2 |
| | WRITE(A,s); $U_2(A)$; |
| | $L_2(B)$; READ(B,s) |
| | s := s*2 |
| | WRITE(B,s); $U_2(B)$; |
| $L_1(B)$; READ(B, t) | |
| t := t+100 | |
| WRITE(B,t); $U_1(B)$; | |

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

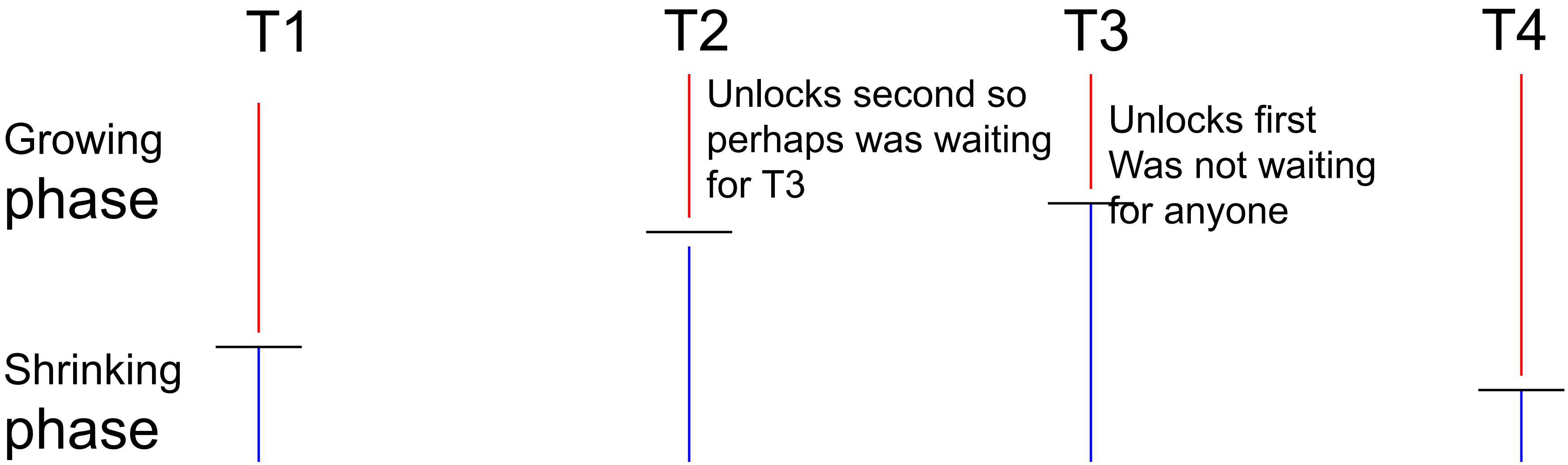
- This ensures conflict serializability ! (will prove this shortly)

# Example: 2PL transactions

| T1 | T2 |
| --- | --- |
| $L_1(A)$; $L_1(B)$; READ(A, t) | |
| t := t+100 | |
| WRITE(A, t); $U_1(A)$ | |
| | $L_2(A)$; READ(A,s) |
| | s := s*2 |
| | WRITE(A,s); |
| | $L_2(B)$; DENIED… |
| READ(B, t) | |
| t := t+100 | |
| WRITE(B,t); $U_1(B)$; | |
| | …GRANTED; READ(B,s) |
| | s := s*2 |
| | WRITE(B,s); $U_2(A)$; $U_2(B)$; |

Now it is conflict-serializable

# Example with Multiple Transactions




**Equivalent to each transaction executing entirely the moment it enters shrinking phase**

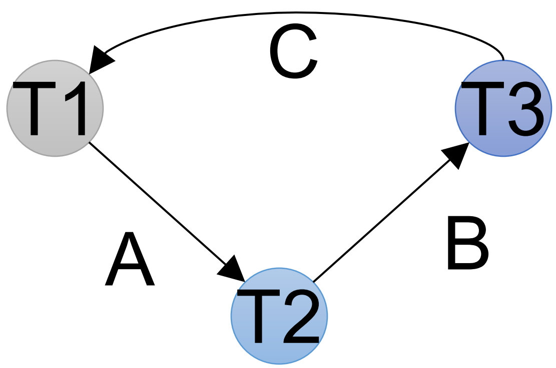
# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

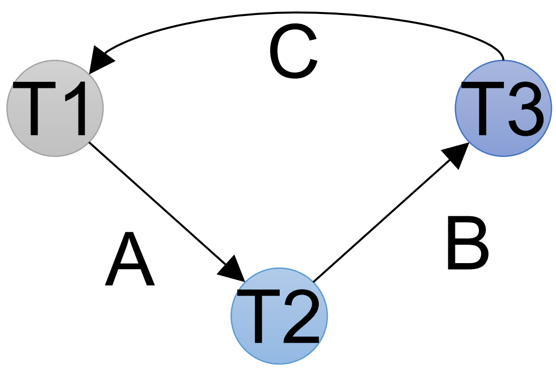
**Proof**. Suppose not: then there exists a cycle in the precedence graph.



# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

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

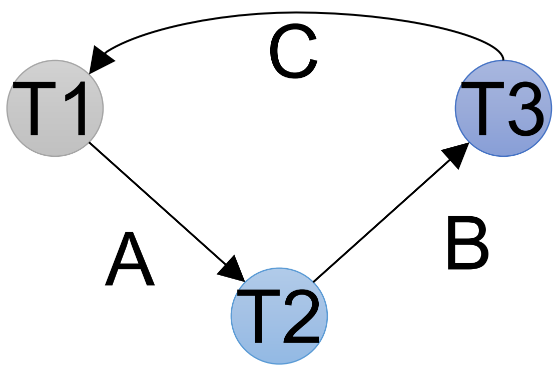



Then there is the following **temporal** cycle in the schedule:

# Two Phase Locking (2PL)

**Theorem**: 2PL ensures conflict serializability

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

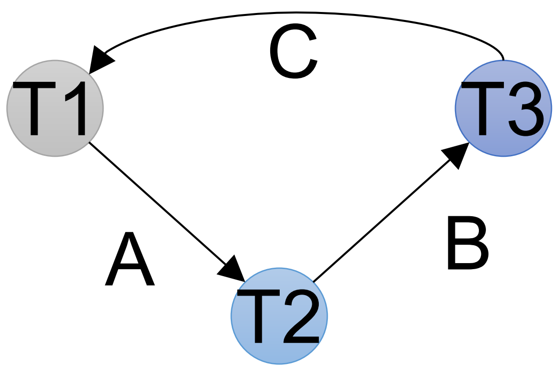



Then there is the following **temporal** cycle in the schedule:
$U_1(A) \rightarrow L_2(A)$ why?

# 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:

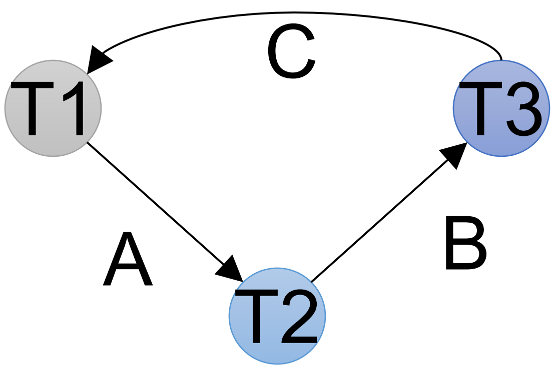
$U_1(A) \rightarrow L_2(A)$

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

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

Contradiction

# Problem: Non-recoverable Schedule

| T1 | T2 |
|---|---|
| $L_1(A)$; $L_1(B)$; READ(A, t) | |
| t := t+100 | |
| WRITE(A, t); $U_1(A)$ | |
| | $L_2(A)$; READ(A,s) |
| | s := s*2 |
| | WRITE(A,s); |
| | $L_2(B)$; DENIED… |
| READ(B, t) | |
| t := t+100 | |
| WRITE(B,t); $U_1(B)$; | |
| | …GRANTED; READ(B,s) |
| | s := s*2 |
| | WRITE(B,s); $U_2(A)$; $U_2(B)$; |
| | **Commit** |
| **Abort** | |

# Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed; release happens at the time of COMMIT or ROLLBACK
- Schedule is recoverable
- Schedule avoids cascading aborts

# Strict 2PL

| T1 | T2 |
|---|---|
| $L_1(A)$; READ(A) | |
| A :=A+100 | |
| WRITE(A); | |
| | $L_2(A)$; DENIED… |
| $L_1(B)$; READ(B) | |
| B :=B+100 | |
| WRITE(B); | |
| $U_1(A)$,$U_1(B)$; Rollback | |
| | …GRANTED; READ(A) |
| | A := A*2 |
| | WRITE(A); |
| | $L_2(B)$;  READ(B) |
| | B := B*2 |
| | WRITE(B); |
| | $U_2(A)$; $U_2(B)$; Commit |

# Summary of Strict 2PL

Ensures:

- Serializability
- Recoverability
- Avoids cascading aborts

# The Locking Scheduler

Task 1: – act on behalf of the transaction
Add lock/unlock requests to transactions

- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- On COMMIT/ROLLBACK release all locks
- Ensures Strict 2PL !

# The Locking Scheduler

Task 2: – act on behalf of the system
Execute the locks accordingly

- Lock table: a big, critical data structure in a DBMS !
- When a lock is requested, check the lock table
  Grant, or add the transaction to the element's wait list
- When lock is released reactivate transaction from its wait list
- When a transaction aborts, release all its locks
- Check for deadlocks occasionally

# Lock Modes

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

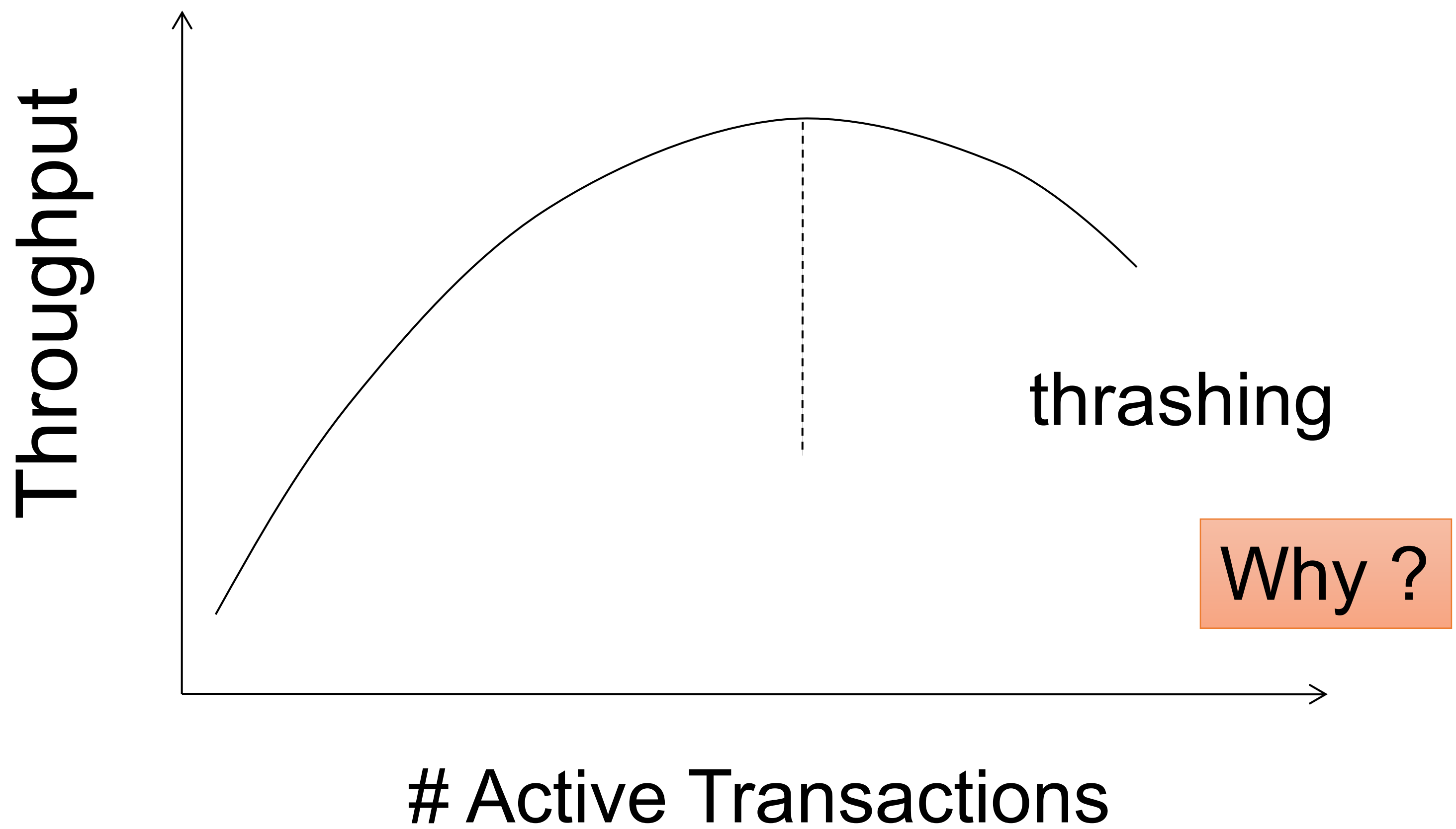
Lock compatibility matrix:

| | None | S | X |
|---|---|---|---|
| None | OK | OK | OK |
| S | OK | OK | Conflict |
| X | OK | Conflict | Conflict |

# Lock Granularity

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
- Coarse grain locking (e.g., tables, predicate locks)
  - Many false conflicts
  - Less overhead in managing locks

# Lock Performance



# 2PL Deadlocks

| T1 (A, B) | T2 (B, C) | T3 (C, D) | T4 (D, A) |
|---|---|---|---|
| L(A) | L(B) | L(C) | L(D) |
| L(B) blocked… | | | |
| | L(C) blocked… | | |
| | | L(D) blocked… | |
| | | | L(A) blocked… |
| … | … | … | … |

# 2PL Deadlocks

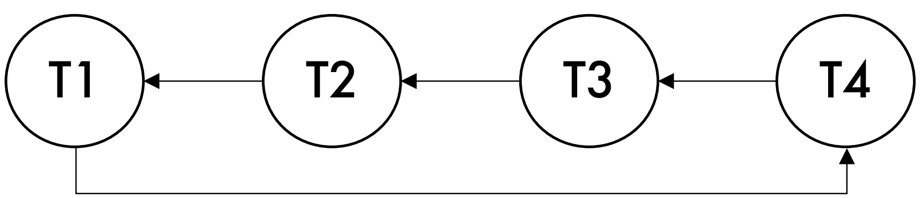
| T1 (A, B) | T2 (B, C) | T3 (C, D) | T4 (D, A) |
|---|---|---|---|
| L(A) | L(B) | L(C) | L(D) |
| L(B) blocked... | | | |
| | L(C) blocked... | | |
| | | L(D) blocked... | |
| | | | L(A) blocked... |
| ... | | ... | ... |

Can't make progress since locking phase is not complete for any txn!

# 2PL Deadlocks

| T1 (A, B) | T2 (B, C) | T3 (C, D) | T4 (D, A) |
|---|---|---|---|
| L(A) | L(B) | L(C) | L(D) |
| L(B) blocked… | | | |
| | L(C) blocked… | | |
| | | L(D) blocked… | |
| | | | L(A) blocked… |
| … | … | … | … |

- Lock requests create a precedence/waits-for graph where deadlock → cycle (2PL is doing its job!).
- Cycle detection over a graph is somewhat expensive, so we check the graph only periodically



# 2PL Deadlocks

| T1 (A, B) | T2 (B, C) | T3 (C, D) | T4 (D, A) |
|---|---|---|---|
| L(A) | L(B) | L(C) | L(D) |
| L(B) blocked… | | | |
| | L(C) blocked… | | |
| | | L(D) blocked… | |
| | | | L(A) blocked… |
| … | … | … | … |

If the DBMS finds a cycle:
- We rollback txns
- (Hopefully) make progress
- Eventually retry the rolledback txns

# 2PL Deadlocks

| T1 (A, B) | T2 (B, C) | T3 (C, D) | T4 (D, A) |
|---|---|---|---|
| L(A) | L(B) | L(C) | L(D) |
| L(B) blocked… | | | |
| | L(C) blocked… | | |
| | | L(D) blocked… | |
| | | | L(A) blocked… |
| | | | |
| | | | |
| | | | |

# 2PL Deadlocks

| T1 (A, B) | T2 (B, C) | T3 (C, D) | T4 (D, A) |
|---|---|---|---|
| L(A) | L(B) | L(C) | L(D) |
| L(B) blocked… | | | |
| | L(C) blocked… | | |
| | | L(D) blocked… | |
| | | | L(A) blocked… |
| | | | Abort, U(D) |
| | | | |
| | | | |

# 2PL Deadlocks

| T1 (A, B) | T2 (B, C) | T3 (C, D) | T4 (D, A) |
|---|---|---|---|
| L(A) | L(B) | L(C) | L(D) |
| L(B) blocked… | | | |
| | L(C) blocked… | | |
| | | L(D) blocked… | |
| | | | L(A) blocked… |
| | | | Abort, U(D) |
| | | L(D) | |
| | | | |

# 2PL Deadlocks

| T1 (A, B) | T2 (B, C) | T3 (C, D) | T4 (D, A) |
|---|---|---|---|
| L(A) | L(B) | L(C) | L(D) |
| L(B) blocked… | | | |
| | L(C) blocked… | | |
| | | L(D) blocked… | |
| | | | L(A) blocked… |
| | | | Abort, U(D) |
| | | L(D) | |
| | | (do operations) | |

# 2PL Deadlocks

| T1 (A, B) | T2 (B, C) | T3 (C, D) | T4 (D, A) |
|---|---|---|---|
| L(A) | L(B) | L(C) | L(D) |
| L(B) blocked... | | | |
| | L(C) blocked... | | |
| | | L(D) blocked... | |
| | | | L(A) blocked... |
| | | | Abort, U(D) |
| | | L(D) | |
| | | (do operations) | |
| | | Commit, U(C), U(D) | |
| | L(C) | | |

# Deadlocks

- Cycle in the wait-for graph:
  - T1 waits for T2
  - T2 waits for T3
  - T3 waits for T4
  - T4 waits for T1
- Deadlock detection
  - Timeouts
  - Wait-for graph
- Deadlock avoidance
  - Acquire locks in pre-defined order
  - Acquire all locks at once before starting

# 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

# Phantom Problem

Suppose there are two blue products, A1, A2:

| T1 | T2 |
|---|---|
| SELECT * <br> FROM Product <br> WHERE color='blue' | |
| | INSERT INTO Product(name, color) <br> VALUES ('A3','blue') |
| SELECT * <br> FROM Product <br> WHERE color='blue' | |

Is this schedule serializable ?

# Phantom Problem

Suppose there are two blue products, A1, A2:

| T1 | T2 |
|---|---|
| SELECT * <br>FROM Product <br>WHERE color='blue' | |
| | INSERT INTO Product(name, color) <br>VALUES ('A3','blue') |
| SELECT * <br>FROM Product <br>WHERE color='blue' | |

Is this schedule serializable ?

No: T1 sees a "phantom" product A3

# Phantom Problem

Suppose there are two blue products, A1, A2:

| T1 | T2 |
|---|---|
| SELECT *<br>FROM Product<br>WHERE color='blue' | |
| | INSERT INTO Product(name, color)<br>VALUES ('A3','blue') |
| SELECT *<br>FROM Product<br>WHERE color='blue' | |

$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$

# Phantom Problem

Suppose there are two blue products, A1, A2:

| T1 | T2 |
|---|---|
| SELECT *<br>FROM Product<br>WHERE color='blue' | |
| | INSERT INTO Product(name, color)<br>VALUES ('A3','blue') |
| SELECT *<br>FROM Product<br>WHERE color='blue' | |

$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$

$W_2(A3);R_1(A1);R_1(A2);R_1(A1);R_1(A2);R_1(A3)$

# Phantom Problem

Suppose there are two blue products, A1, A2:

| T1 | T2 |
|---|---|
| SELECT *<br>FROM Product<br>WHERE color='blue' | |
| | INSERT INTO Product(name, color)<br>VALUES ('A3','blue') |
| SELECT *<br>FROM Product<br>WHERE color='blue' | |

But this is conflict-serializabel

$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$

$W_2(A3);R_1(A1);R_1(A2);R_1(A1);R_1(A2);R_1(A3)$

# 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 !

# 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 !**

# Discussion

We *always* want a serializable schedule

Strict 2PL guarantees conflict serializability

- In a ***static*** database:
  - Conflict serializability implies serializability

- In a ***dynamic*** database:
  - Need both conflict serializability *and* handling of phantoms to ensure serializability