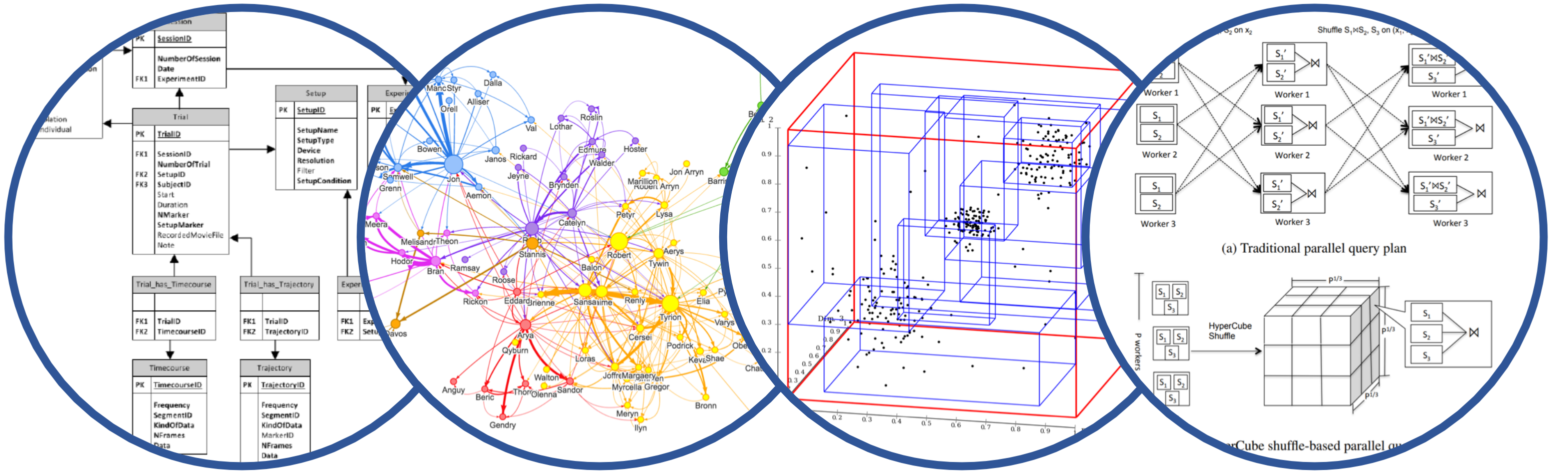


Introduction to Data Management

Transactions with Locking

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

Transactions

- Execute all parts of a transaction as a single action
- **Transactions are atomic**

BEGIN TRANSACTION
 [SQL Statements]
COMMIT – finalizes execution

BEGIN TRANSACTION
 [SQL Statements]
ROLLBACK – undo everything

Conflict Order Rules

- Observation: Reordering operation of the same element around writes will cause different program behavior
- **Inter-transaction conflicts**
  - WW conflicts → $W_1(X)$, $W_2(X)$
    - Not always the same as $W_2(X)$, $W_1(X)$
  - WR conflicts → $W_1(X)$, $R_2(X)$
    - Not always the same as $R_2(X)$, $W_1(X)$
  - RW conflicts → $R_1(X)$, $W_2(X)$
    - Not always the same as $W_2(X)$, $R_1(X)$

Equivalent Behavior Schedules

- A reordered schedule of operations is **guaranteed to be equivalent** when WR, RW, and WW conflicts are preserved

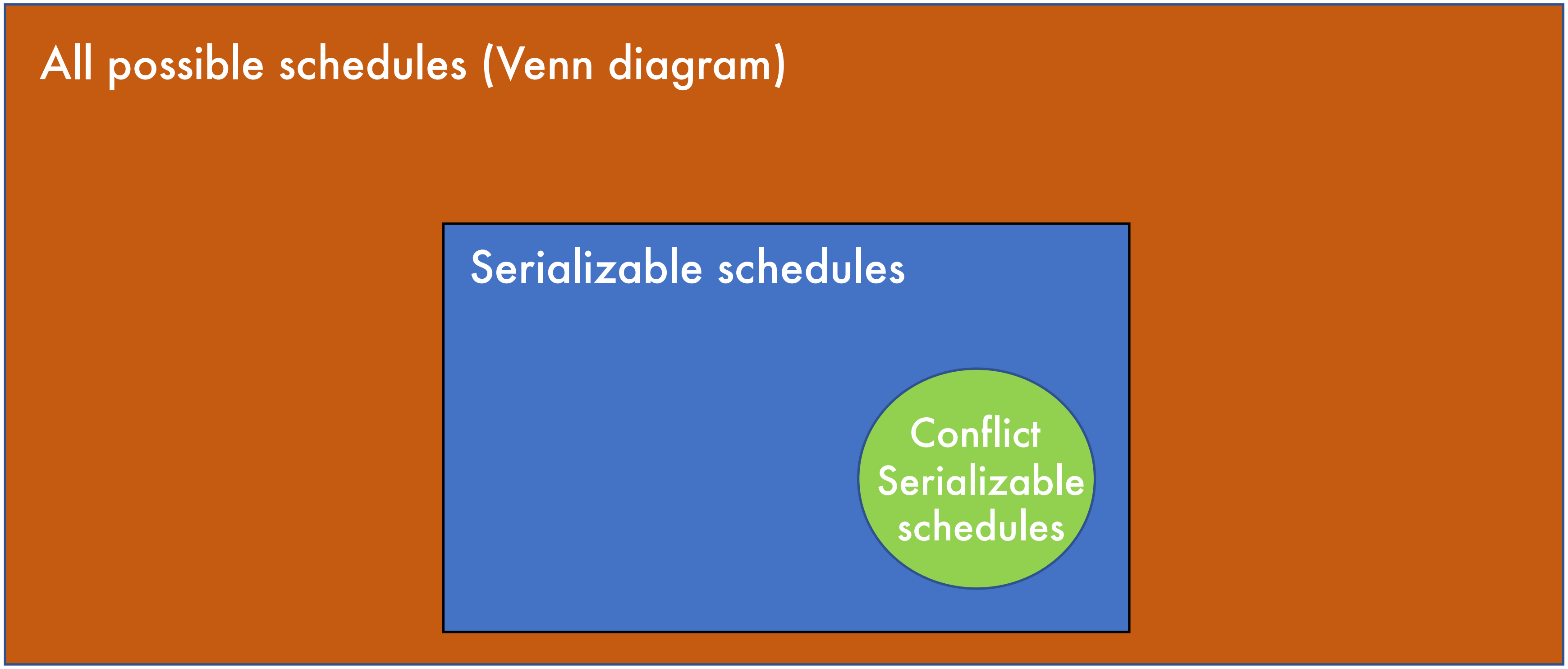
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

→

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Conflict Serializability

- **Conflict serializability means we can reorder the schedule into an actual serial schedule**
- Conflict serializability implies serializability



Conflict Serializable Schedule Example

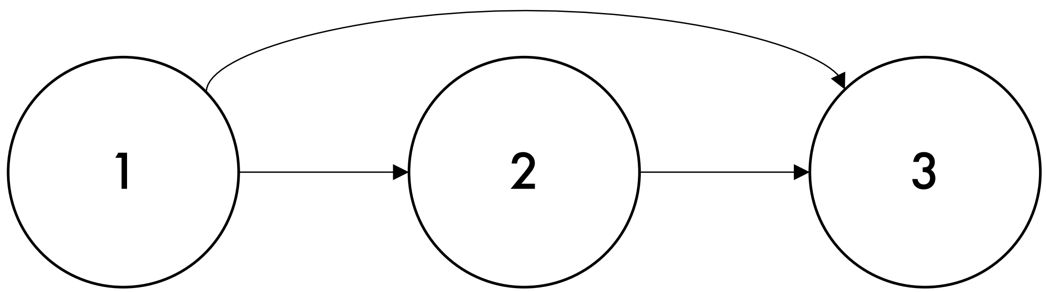
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

→

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Example of Checking Conflict Serializability

$R_1(X), W_2(X), W_1(Y), W_2(Y), W_3(Y), R_3(X), W_3(X)$



DAG → Conflict serializable to T1, T2, T3 → Serializable to T1, T2, T3

Question for Today

The goal of concurrency control is to insure Isolation (the appearance of serial schedules)

What mechanisms do we use to enforce (conflict) serializable schedules?

Scheduling

- **Scheduler** a.k.a. **concurrency control manager**
  - Impractical (slow and space inefficient) to issue R, W, … from a literal schedule
  - Use mechanisms like logs and locks to force ACID properties
- Why do we care? Application considerations!
  - Indexes should be created based on expected workload. So should your choice of transaction management.
  - **Pessimistic Concurrency Control** (this class) good for **high-contention workloads**
  - **Optimistic Concurrency Control** (CSE 444) good for **low-contention workloads**

Optimistic Scheduler

- Commonly interchangeable with **Multi Version Concurrency Control**
- "Optimistic" → Assumes transaction executions will not create conflicts
- Main Idea:
  - Execute first, check later
  - Cheap overhead cost but expensive aborting process

Pessimistic Scheduler

- Commonly interchangeable with **Locking Scheduler**
- "Pessimistic" → Assumes transaction executions will conflict
- Main Idea:
  - Prevent executions that would create conflicts
  - Expensive overhead cost but cheap aborting process

Outline

- Locks
- 2PL and conflict serializability
- Deadlocks
- Strict 2PL and recoverability

Locks

- Pessimistic CC involves locks
- **Binary lock** mechanisms:
  - We have locks on objects that specify which transaction can do operations
  - A txn must **acquire** a lock before reading or writing
    - Notation: txn i acquires lock on element X → $\mathbf{L_i(X)}$
  - A txn must eventually **release** locks (unlock)
    - Notation: txn i releases lock on element X → $\mathbf{U_i(X)}$
  - If txn finds another txn holds a lock for the desired element, **wait for the unlock signal**

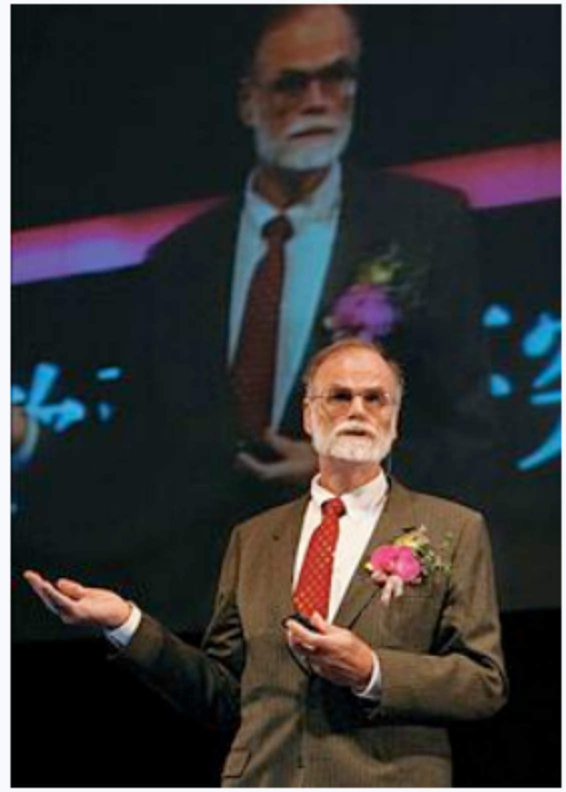
Element Granularity

- A DBMS (and sometimes user) may specify what granularity of elements are locked
  - Dramatically qualifies expected contention
- SQLite → Database locking only
- MySQL, SQL Server, Oracle, … → Row locking, table locking
- SQL syntax varies or may not exist explicitly

2-Phase Locking (2PL)

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

Jim Gray



2006

Born	James Nicholas Gray January 12, 1944[1] San Francisco, California[2]
Disappeared	January 28, 2007 (aged 63) Waters near San Francisco

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*2
	RITE(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 ?

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); $U_1(B)$;	

T1 unlocked A too soon…
T2 was then able to run in full

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

Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.

Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

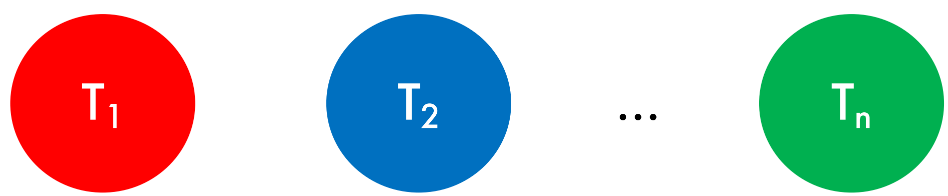
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.

Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as T_1, …, T_n where:

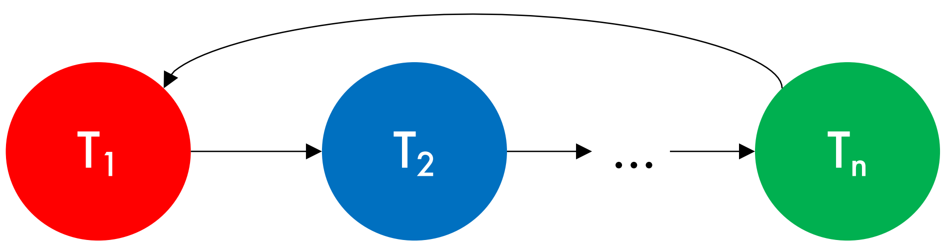


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as T_1, …, T_n where:
  - An edge exists from T_i to T_{i+1} for $i < n$
  - An edge exists from T_n to T_1

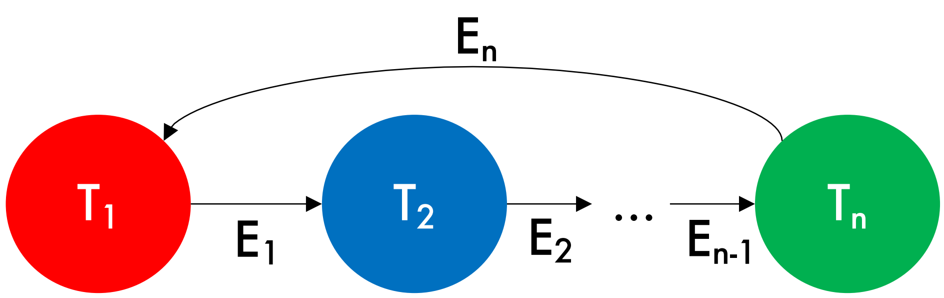


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as T_1, …, T_n where:
  - An edge exists from T_i to T_{i+1} for i < n
  - An edge exists from T_n to T_1
- (An edge means there is a conflict on some element, call it E_i)

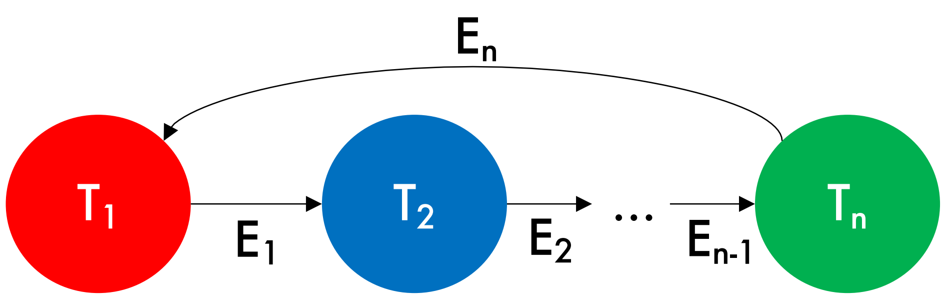


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from T_i to T_{i+1} for $i < n$
  - An edge exists from T_n to T_1
- (An edge means there is a conflict on some element, call it E_i)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$

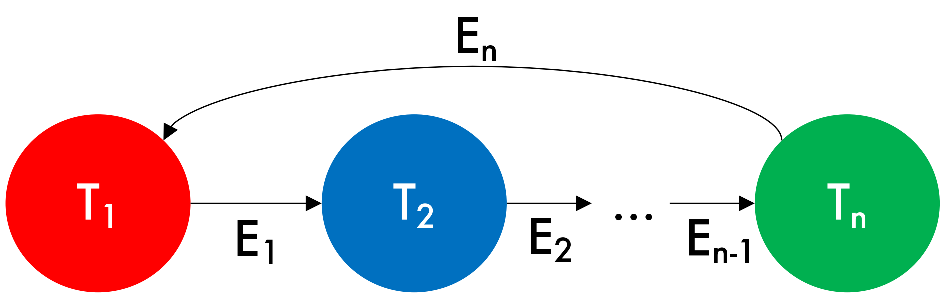


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as T_1, …, T_n where:
    - An edge exists from T_i to T_{i+1} for $i < n$
    - An edge exists from T_n to T_1
- (An edge means there is a conflict on some element, call it E_i)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
    - $U_1(E_1)$ then $L_2(E_1)$
    - $L_2(E_1)$ then $U_2(E_2)$

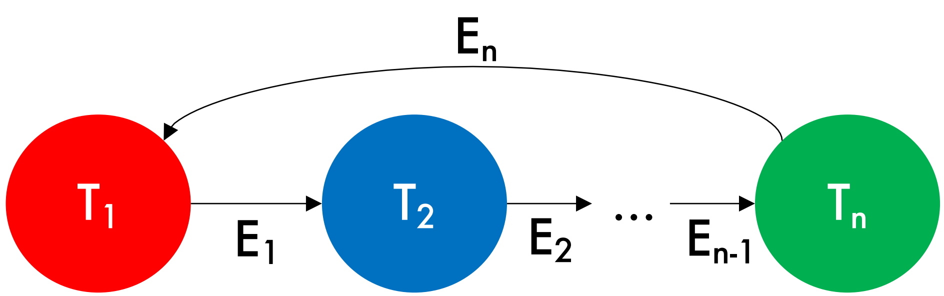


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from T_i to T_{i+1} for $i < n$
  - An edge exists from T_n to T_1
- (An edge means there is a conflict on some element, call it E_i)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - …

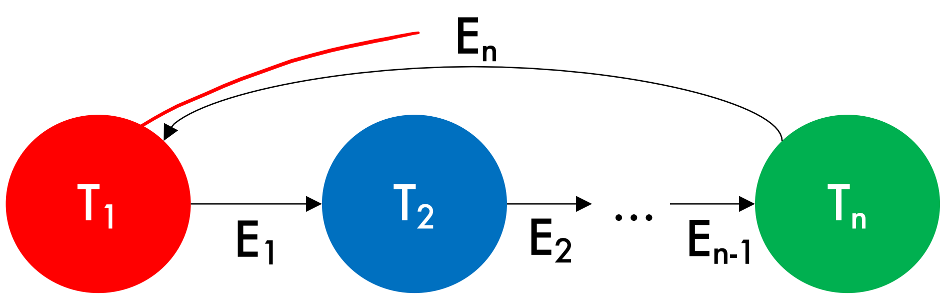


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from T_i to T_{i+1} for $i < n$
  - An edge exists from T_n to T_1
- (An edge means there is a conflict on some element, call it E_i)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - …
  - $U_n(E_n)$ then $L_1(E_n)$

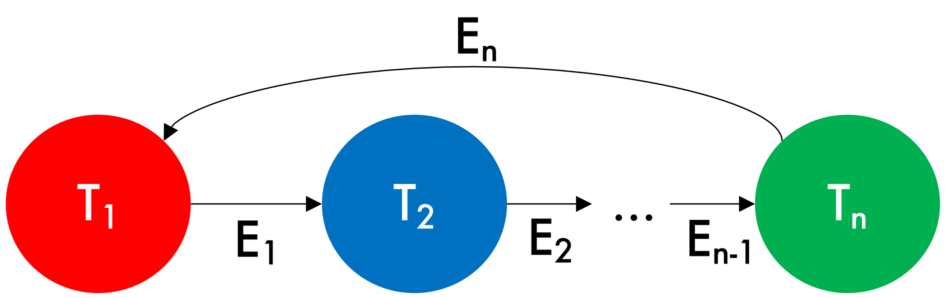


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as T_1, …, T_n where:
  - An edge exists from T_i to T_{i+1} for $i < n$
  - An edge exists from T_n to T_1
- (An edge means there is a conflict on some element, call it E_i)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - …
  - $U_n(E_n)$ then $L_1(E_n)$
  - $L_1(E_n)$ then $U_1(E_1)$

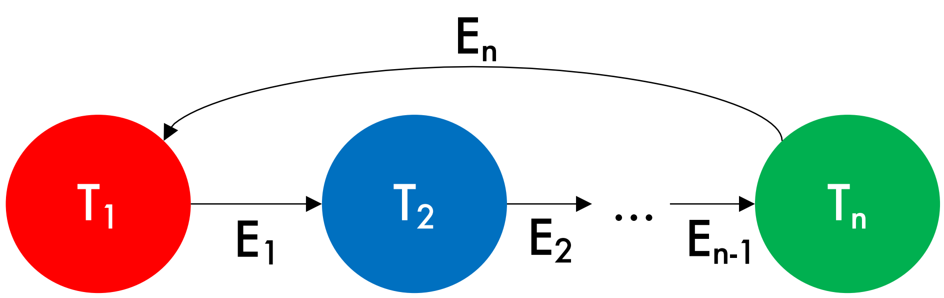


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as T_1, …, T_n where:
  - An edge exists from T_i to T_{i+1} for $i < n$
  - An edge exists from T_n to T_1
- (An edge means there is a conflict on some element, call it E_i)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - …
  - $U_n(E_n)$ then $L_1(E_n)$
  - $L_1(E_n)$ then $U_1(E_1)$
- There is a **cycle in time** which is a contradiction

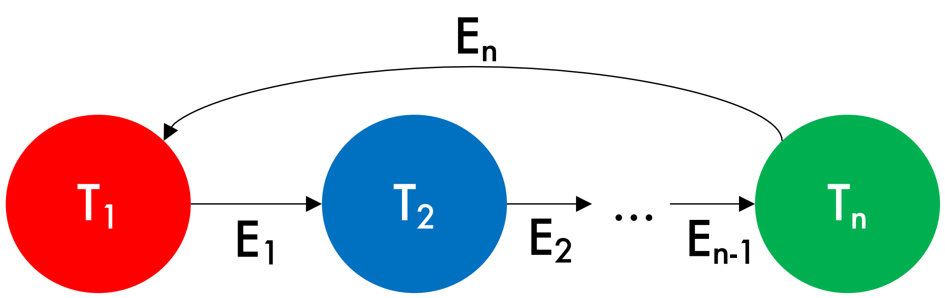


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as $T_1, \ldots, T_n$ where:
  - An edge exists from T_i to T_{i+1} for $i < n$
  - An edge exists from T_n to T_1
- (An edge means there is a conflict on some element, call it E_i)
- Under 2PL, we can guarantee the series of locks and unlocks in time:
  - $U_1(E_1)$ then $L_2(E_1)$
  - $L_2(E_1)$ then $U_2(E_2)$
  - $U_2(E_2)$ then $L_3(E_2)$
  - $L_3(E_2)$ then $U_3(E_3)$
  - …
  - $U_n(E_n)$ then $L_1(E_n)$
  - $L_1(E_n)$ then $U_1(E_1)$
- There is a **cycle in time** which is a contradiction



2PL Problems

- **Deadlocks!**
  - Byproduct of dealing with groups of locks
- **Recoverability**
  - Transactions might want an abort feature if not all consistency constraints are enforced by the DB
  - Can't abort txns under vanilla 2PL

2PL Deadlocks

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

2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A) L(B) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…
R(A) W(A)	R(B) W(B)	R(C) W(C)	R(D) W(D)

2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A) L(B) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…
R(A) W(A)	R(B) W(B)	R(C) W(C)	R(D) W(D)
(can't unlock A)	(can't unlock B)	(can't unlock C)	(can't unlock D)

2PL Deadlocks

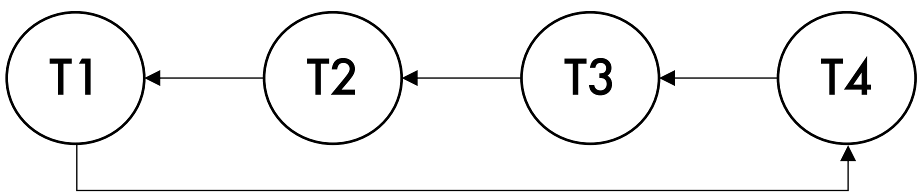
T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A) L(B) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…
R(A) W(A)	R(B) W(B)	R(C) W(C)	R(D) W(D)
(can't unlock A)	(can't unlock B)	(can't unlock C)	(can't unlock D)

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) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…

- Lock requests create a precedence/waits-for graph where deadlock → cycle (2PL is doing its job!).
- Cycle detection is somewhat expensive O(V+E), 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) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…
R(A) W(A)	R(B) W(B)	R(C) W(C)	R(D) W(D)

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) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…
R(A) W(A)	R(B) W(B)	R(C) W(C)	R(D) W(D)
		…granted L(D)	abort U(D)

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) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…
R(A) W(A)	R(B) W(B)	R(C) W(C)	R(D) W(D)
		…granted L(D)	abort U(D)
		R(D) W(D)	N/A

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) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…
R(A) W(A)	R(B) W(B)	R(C) W(C)	R(D) W(D)
		…granted L(D)	abort U(D)
		R(D) W(D)	N/A
	…granted L(C)	U(D) U(C)	N/A

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) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…
R(A) W(A)	R(B) W(B)	R(C) W(C)	R(D) W(D)
		…granted L(D)	abort U(D)
		R(D) W(D)	N/A
	…granted L(C)	U(D) U(C)	N/A
…	…	…	retry

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

2PL Problems

- **Deadlocks!**
  - Byproduct of dealing with groups of locks
- **Recoverability**
  - Transactions might want an abort feature if not all consistency constraints are enforced by the DB
  - Can't always abort txns under vanilla 2PL

Non-Recoverable Schedule

T1	T2
L(A) L(B)	L(A) blocked… L(B) blocked…
R(A) W(A)	
U(A)	…granted L(A)
R(B) W(B)	R(A) W(A)
U(B)	…granted L(B) U(A)
	R(B) W(B)
	U(B)
	COMMIT
ROLLBACK	

ROLLBACK will try to signal the DBMS to revert to original values

Non-Recoverable Schedule

T1	T2
L(A) L(B)	L(A) blocked… L(B) blocked…
R(A) W(A)	
U(A)	…granted L(A)
R(B) W(B)	R(A) W(A)
U(B)	…granted L(B) U(A)
	R(B) W(B)
	U(B)
	COMMIT
ROLLBACK	

T2 already executed under modified A and B values (dirty read)

ROLLBACK will try to signal the DBMS to revert to original values

Non-Recoverable Schedule

T1	T2
L(A) L(B)	L(A) blocked… L(B) blocked…
R(A) W(A)	
U(A)	…granted L(A)
R(B) W(B)	R(A) W(A)
U(B)	…granted L(B) U(A)
	R(B) W(B)
	U(B)
	COMMIT
ROLLBACK	

T2 already executed under modified A and B values (dirty read)

ROLLBACK would break the COMMIT promise that T2's execution was valid

ROLLBACK will try to signal the DBMS to revert to original values

Strict 2PL

Protocol: All unlocks are done together with the COMMIT or ROLLBACK

Strict 2PL guarantees schedule conflict serializability and recoverability

Recoverable Schedule

T1	T2
L(A) L(B)	L(A) blocked… L(B) blocked…
R(A) W(A)	
R(B) W(B)	
ROLLBACK U(A) U(B)	…granted L(A) …granted L(B)
	R(A) W(A)
	R(B) W(B)
	COMMIT U(A) U(B)

“Do I need to implement any of this?”

“Do I need to implement any of this?”

Short Answer: **No**

“Do I need to implement any of this?”

Long Answer:

These mechanisms are internal to the DBMS.

The DBMS manages locks with strict 2PL. The DBMS creates the precedence graph. The DBMS does the deadlock retry.

As an application programmer / database user you only need to (and should only need to) specify transactions and think about application-level consistency.

Next Time (Recording)

- Phantom reads
- Isolation levels
- Hierarchical locking