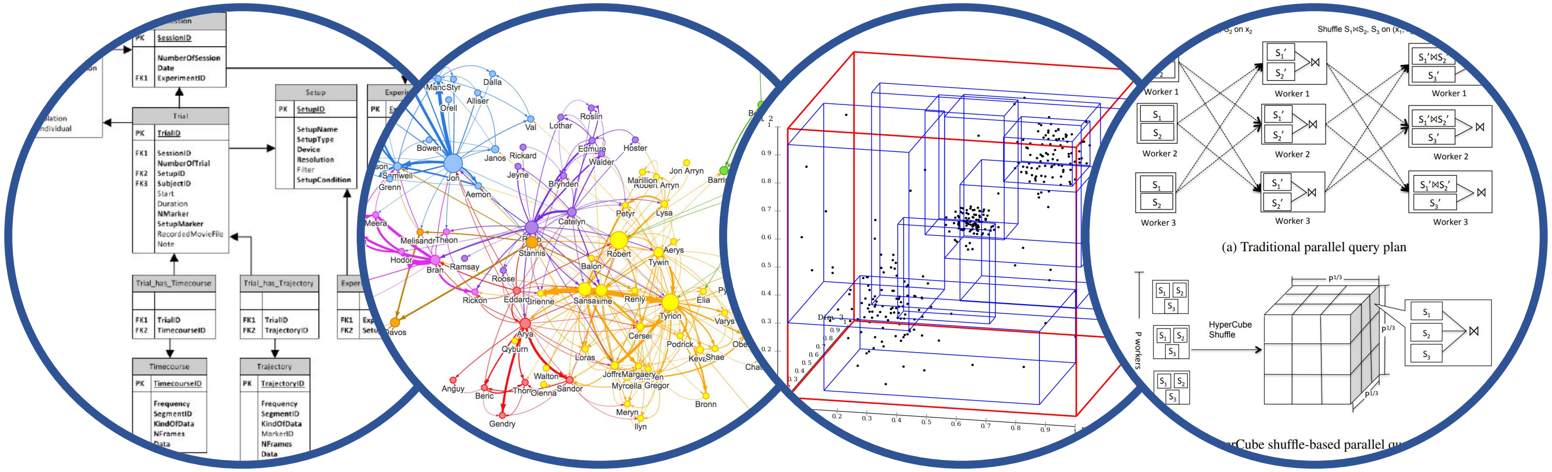


Introduction to Data Management

Transactions: Locking

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

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

Announcements

- Midterm next Monday, 2/10, in class
  - Covers material through today's lecture
    - e.g. can expect transactions but not isolation levels
  - Notes: 1 sheet, both sides, handwritten
- TA-led midterm review
  - Sunday 10 am, room TBA
- Engineering Teaching & Learning assessment during next lecture

Recap: 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

Recap: Serializable Schedule

- Serializable to T1 then T2

$R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B), R_2(B), W_2(B)$

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

Looks like T2 finished after T1 for each element

Recap: Serializable Schedule

- Not serializable to either order

$R_1(A), W_1(A), R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B)$

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

Looks like T2 finished last looking at A

Looks like T1 finished last looking at B

Recap: 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 $W_2(X)$, $R_1(X)$

Recap: Equivalent Behavior Schedules

- A **conflict serializable schedule** is a schedule that can be transformed into a serial schedule by performing a series of swaps of adjacent non-conflicting actions

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)

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

Recap: Non Conflict Serializable 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(A)
	W(A)
	R(B)
R(B)	
	W(B)
W(B)	

Conflict rule broken!

Outline

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

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

Scheduling

- Scheduling matters to us because it affects our application behavior and performance!
  - Your choice of transaction management should be based on expected workload.
  - **Pessimistic Concurrency Control** (this class) good for **high-contention workloads**
  - **Optimistic Concurrency Control** (CSE 444) good for **low-contention workloads**

Optimistic Scheduler

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

Question for Today

The goal of concurrency control is to ensure isolation (the appearance of serial schedules) and atomicity.

What mechanisms does the DBMS use to make (conflict) serializable schedules?

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 a txn wants an element for which another txn holds the lock, **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

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)	
A := A+100	
WRITE(A)	
	READ(A)
	A := A*2
	WRITE(A)
	READ(B)
	B := B*2
	WRITE(B)
READ(B)	
B := B+100	
WRITE(B)	

Add locking....

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Scheduler has ensured a conflict serializable schedule

But...

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$;	
	$L_2(A)$; 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); $U_1(B)$;	

The locks didn't enforce conflict serializability! What happened?

But…

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$;	
	$L_2(A)$; 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); $U_1(B)$;	

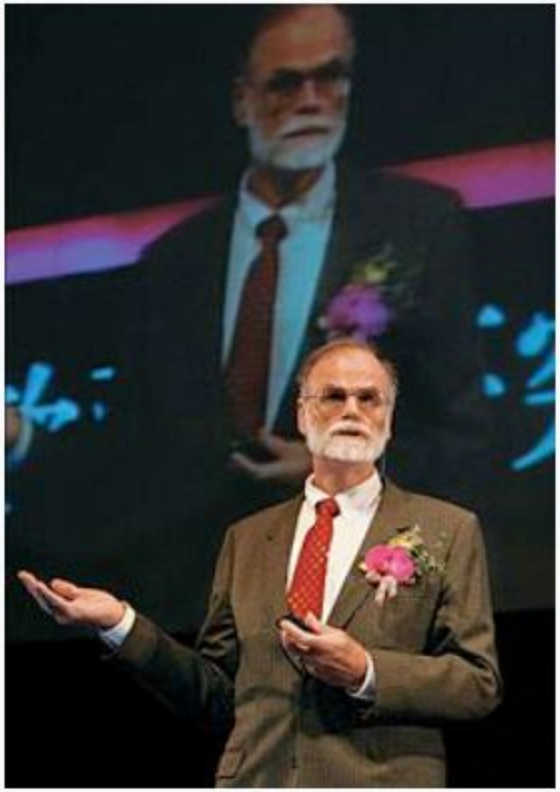
T1 unlocked A too soon…T2 was able to run in full

The locks didn't enforce conflict serializability! What happened?

2-Phase Locking (2PL)

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

Jim Gray



2006

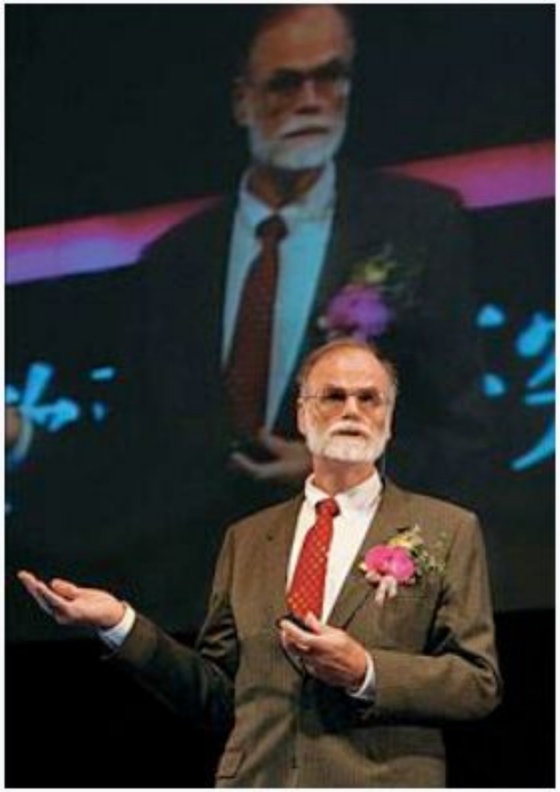
Born	James Nicholas Gray January 12, 1944[1] San Francisco, California[2]
Disappeared	January 28, 2007 (aged 63) Waters near San Francisco

2-Phase Locking (2PL)

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

This will ensure conflict serializability

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

2PL Example

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;

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:

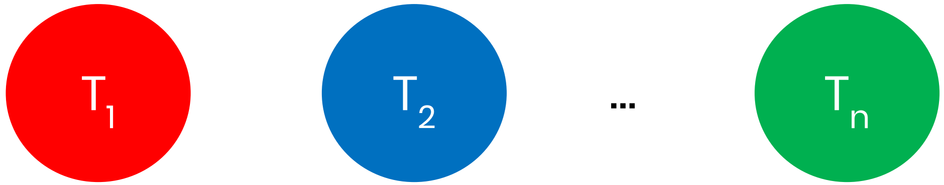
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.

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 $T_1, \ldots, T_n$ where:

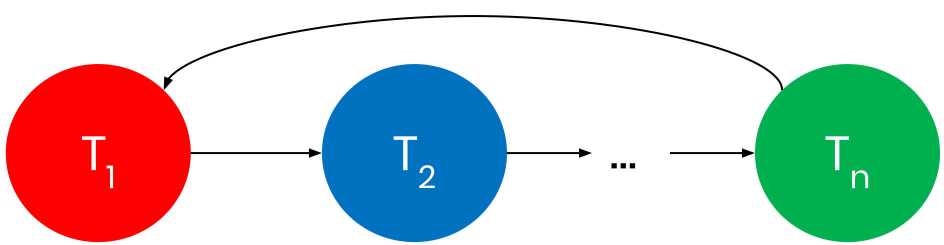


Conflict Serializability through 2PL

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 $T_1, \ldots, T_n$ where:
  - An edge exists from T_i to T_{i+1} for $i < n$
  - An edge exists from T_n to T_1

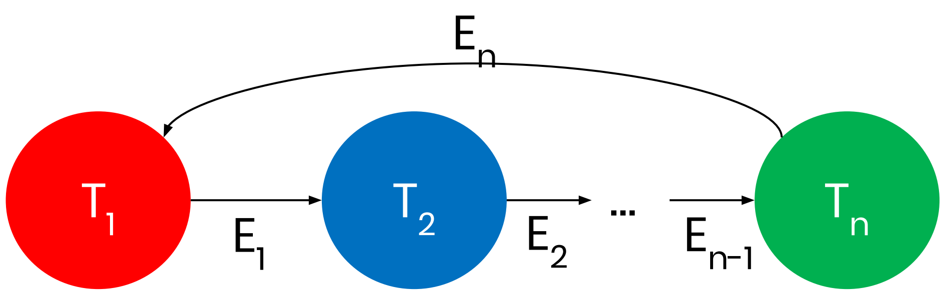


Conflict Serializability through 2PL

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)

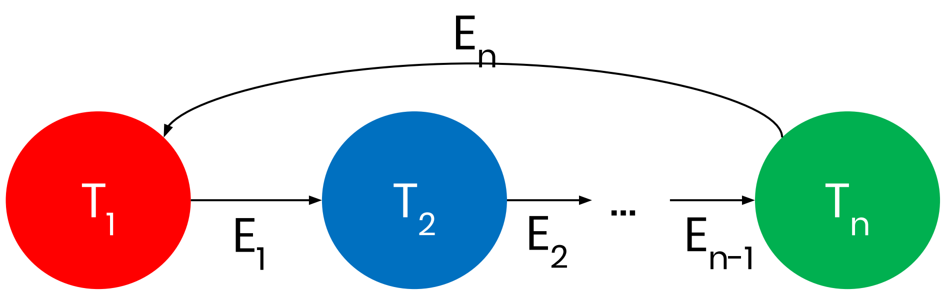


Conflict Serializability through 2PL

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 $L_2(E_1)$

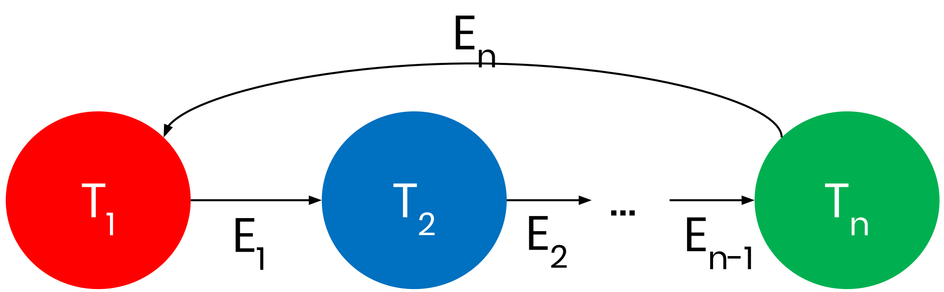


Conflict Serializability through 2PL

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 $T_1, \ldots, T_n$ 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 $U_2(E_2)$

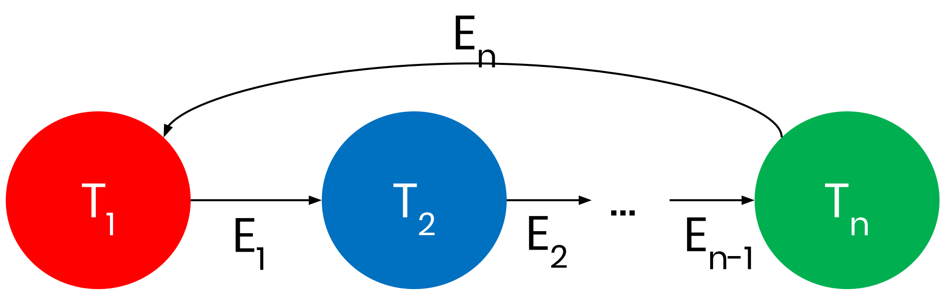


Conflict Serializability through 2PL

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 $U_3(E_3)$
  - …

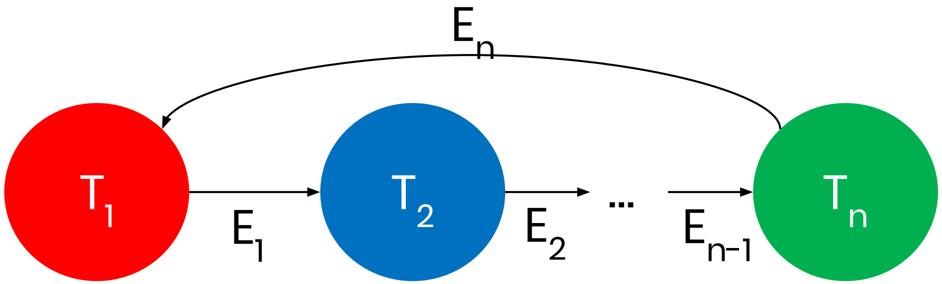


Conflict Serializability through 2PL

Theorem: **2PL ensures conflict serializability**

Proof by contradiction:

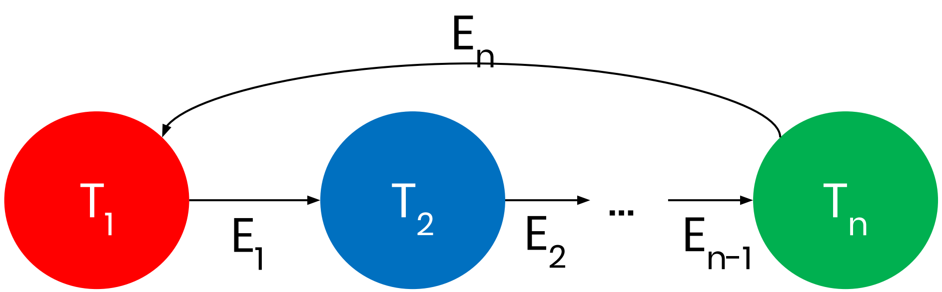
- Suppose a schedule was executed under 2PL that was not conflict serializable.
- Then that schedule must have a **precedence graph** with a **cycle**.
- Name the transactions in the cycle as T_1, …, T_n where:
  - An edge exists from T_i to T_{i+1} for $i < n$
  - An edge exists from T_n to T_1
- (An 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 $L_1(E_n)$



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 $U_1(E_1)$

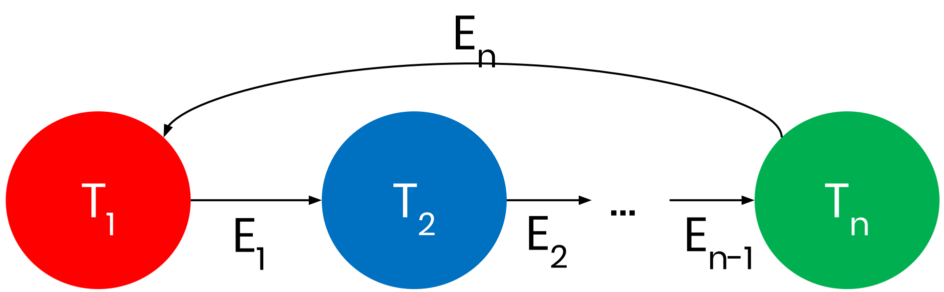


Conflict Serializability through 2PL

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

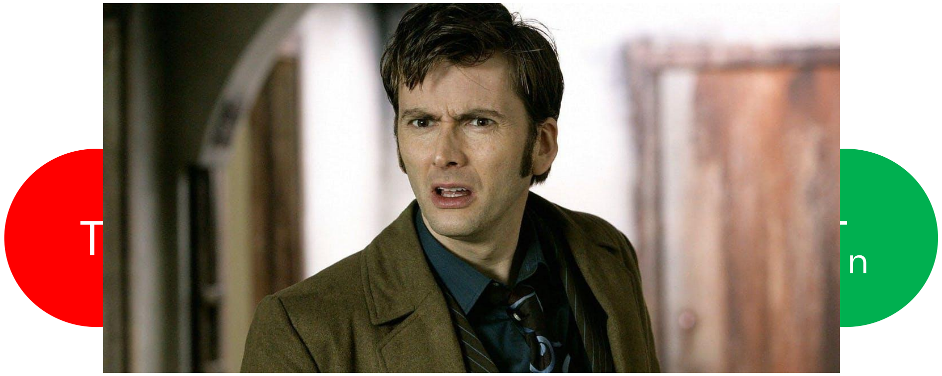


Conflict Serializability through 2PL

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 Non-Recoverable Schedule

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

2PL Non-Recoverable Schedule

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

ROLLBACK will signal the DBMS to revert to original values

2PL Non-Recoverable Schedule

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

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

ROLLBACK will signal the DBMS to revert to original values

2PL Non-Recoverable Schedule

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

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

T1's ROLLBACK would break the COMMIT promise that T2's execution was valid

ROLLBACK will signal the DBMS to revert to original values

Strict 2PL

- Protocol:
  - All locks are held until commit/abort
  - All unlocks are done together with commit/abort.

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

Strict 2PL

- Protocol:
  - All locks are held until commit/abort
  - All unlocks are done together with commit/abort.

This is what SQL Server uses!

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

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

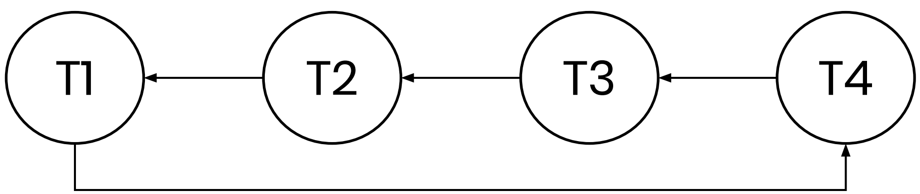
T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A) L(B) blocked…	L(B) L(C) blocked…	L(C) L(D) blocked…	L(D) L(A) blocked…
R(A) W(A)	R(B) W(B)	R(C) W(C)	R(D) W(D)

Can't 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:
- It aborts txns (rollback)
- (Hopefully) makes progress
- Eventually retries 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:
- It aborts txns (rollback)
- (Hopefully) makes progress
- Eventually retries 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:
- It aborts txns (rollback)
- (Hopefully) makes progress
- Eventually retries 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:
- It aborts txns (rollback)
- (Hopefully) makes progress
- Eventually retries 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:
- It aborts txns (rollback)
- (Hopefully) mak
- Eventually r

On the application level:
Can lock resources in a defined order
Can retry transactions that were aborted due to deadlock

Conservative 2PL

- Protocol: All locks are acquired before the transaction begins

“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 a locking protocol. The DBMS creates the precedence graph. The DBMS checks for deadlocks.

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

- Phantom reads
- Isolation levels
- Hierarchical locking