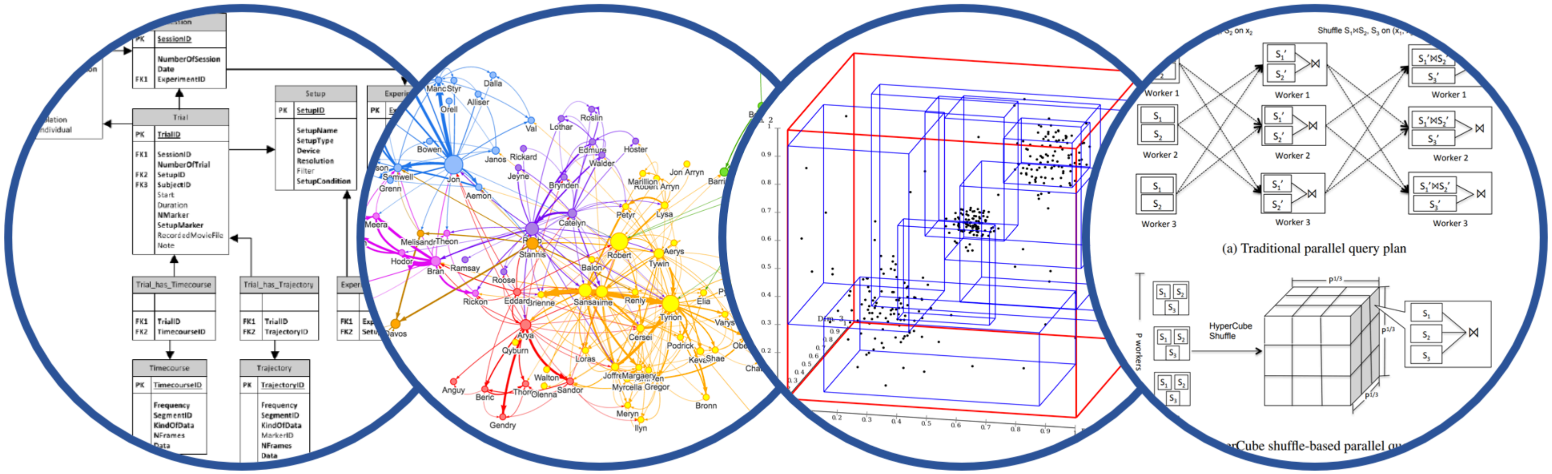


Database System Internals

Concurrency Control - Locking

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

Announcement

We cancel the quiz!

Reason:

- Learning is difficult during lockdown
- This course is intense: 1 hw or lab each week
- The quiz only adds to the stress
- It had a low weight anyway…

… so let's just cancel it.

View Equivalence

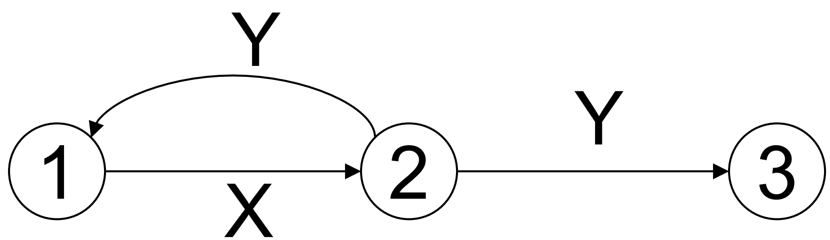
- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$$w_1(X);\ w_2(X);\ w_2(Y);\ w_1(Y);\ w_3(Y);$$

Is this schedule conflict-serializable ?

View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption



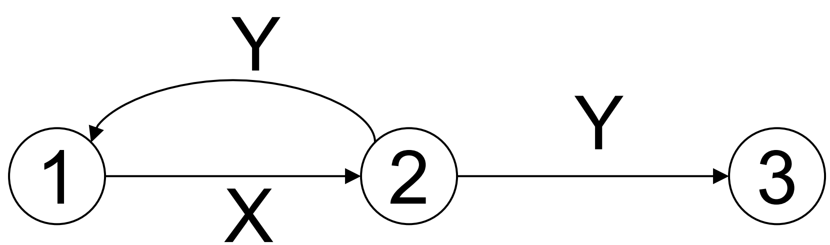
$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Is this schedule conflict-serializable ?

No…

View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption



$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Lost write

$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

Equivalent, but not conflict-equivalent

View Equivalence

T1	T2	T3
W1(X)		
	W2(X)	
	W2(Y)	
	CO2	
W1(Y)		
CO1		
		W3(Y)
		CO3

Lost

T1	T2	T3
W1(X)		
W1(Y)		
CO1		
	W2(X)	
	W2(Y)	
	CO2	
		W3(Y)
		CO3

Serializable, but not conflict serializable

View Equivalence

Two schedules S, S' are *view equivalent* if:

- If T reads an initial value of A in S,
  then T reads the initial value of A in S'

- If T reads a value of A written by T' in S,
  then T reads a value of A written by T' in S'

- If T writes the final value of A in S,
  then T writes the final value of A in S'

View-Serializability

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

Remark:

- If a schedule is *conflict serializable*, then it is also *view serializable*
- But not vice versa

Schedules with Aborted Transactions

- When a transaction aborts, the recovery manager undoes its updates

- But some of its updates may have affected other transactions !

Schedules with Aborted Transactions

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

What's wrong?

Schedules with Aborted Transactions

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

What's wrong?

Cannot abort T1 because cannot undo T2

Recoverable Schedules

A schedule is *recoverable* if:

- It is conflict-serializable, and
- Whenever a transaction T commits, all transactions that have written elements read by T have already committed

Recoverable Schedules

A schedule is *recoverable* if:

- It is conflict-serializable, and
- Whenever a transaction T commits, all transactions that have written elements read by T have already committed

Recoverable Schedules

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

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

Recoverable Schedules

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

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

Recoverable Schedules

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

Nonrecoverable

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

Recoverable Schedules

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

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

Nonrecoverable

Recoverable Schedules

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

Nonrecoverable

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

Recoverable

Recoverable Schedules

T1	T2	T3	T4
R(A)			
W(A)			
	R(A)		
	W(A)		
	R(B)		
	W(B)		
		R(B)	
		W(B)	
		R(C)	
		W(C)	
			R(C)
			W(C)
			R(D)
			W(D)
Abort			

How do we recover ?

Recoverable Schedules

T1	T2	T3	T4
R(A)			
W(A)			
	R(A)		
	W(A)		
	R(B)		
	W(B)		
		R(B)	
		W(B)	
		R(C)	
		W(C)	
			R(C)
			W(C)
			R(D)
			W(D)
Abort			

Abort

How do we recover ?

Recoverable Schedules

T1	T2	T3	T4
R(A)			
W(A)			
	~~R(A)~~		
	~~W(A)~~	Abort	
	~~R(B)~~		
	~~W(B)~~		
		~~R(B)~~	
		~~W(B)~~	Abort
		~~R(C)~~	
		~~W(C)~~	
			R(C)
			W(C)
			R(D)
			W(D)
Abort			

How do we recover ?

Recoverable Schedules

T1	T2	T3	T4	
R(A)				
W(A)				
	~~R(A)~~			
	~~W(A)~~	Abort		
	~~R(B)~~			
	~~W(B)~~			
		~~R(B)~~		
		~~W(B)~~	Abort	
		~~R(C)~~		
		~~W(C)~~		
			~~R(C)~~	
			~~W(C)~~	Abort
			~~R(D)~~	
			~~W(D)~~	
Abort				

How do we recover ?

Cascading Aborts

- If a transaction T aborts, then we need to abort any other transaction T' that has read an element written by T

- A schedule *avoids cascading aborts* if whenever a transaction reads an element, the transaction that has last written it has already committed.

We base our locking scheme on this rule!

Avoiding Cascading Aborts

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

With cascading aborts

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

Without cascading aborts

Serializability

- Serial
- Serializable
- Conflict serializable
- View serializable

Recoverability

- Recoverable
- Avoids cascading deletes

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

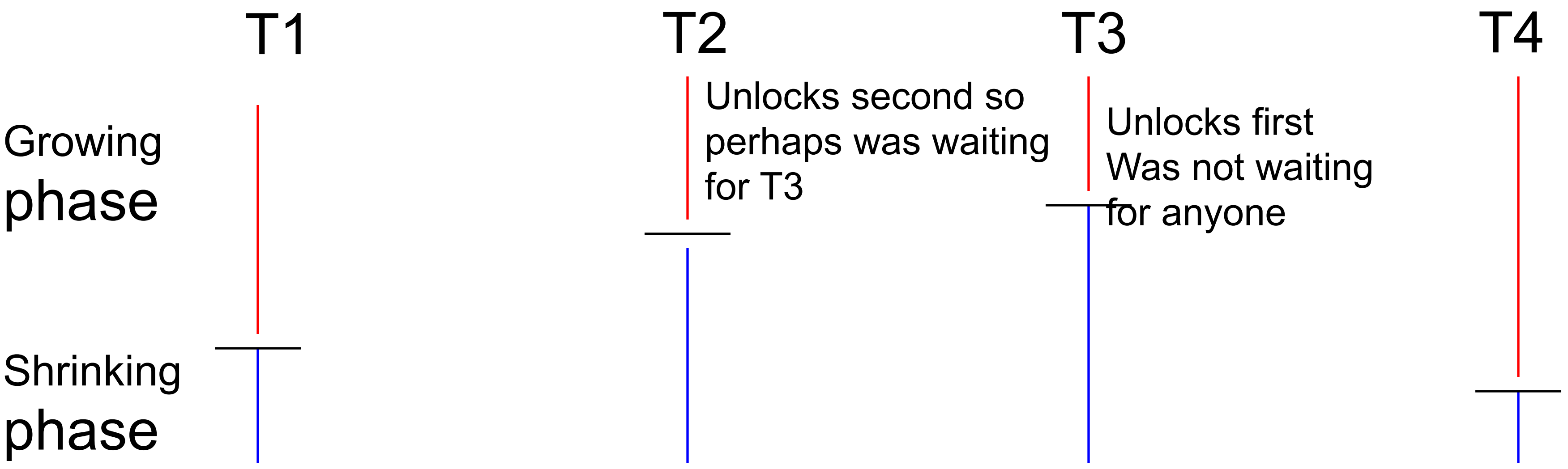
- This ensures conflict serializability ! (will prove this shortly)

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

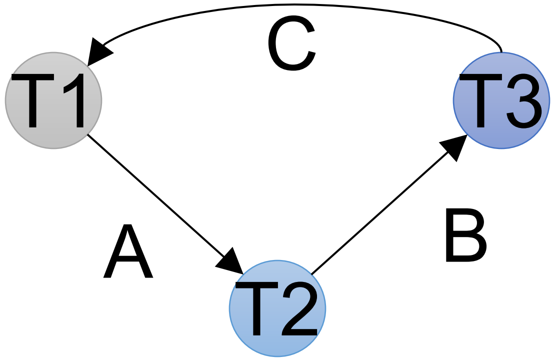
Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

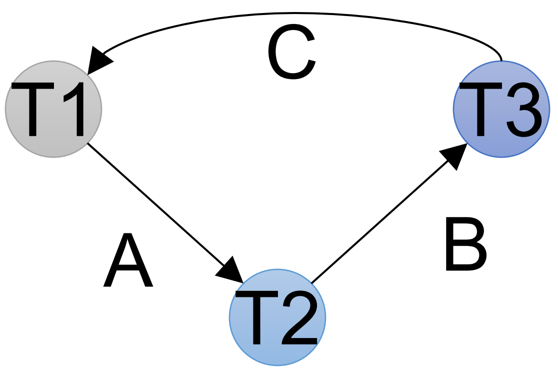
Proof. Suppose not: then there exists a cycle in the precedence graph.



Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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

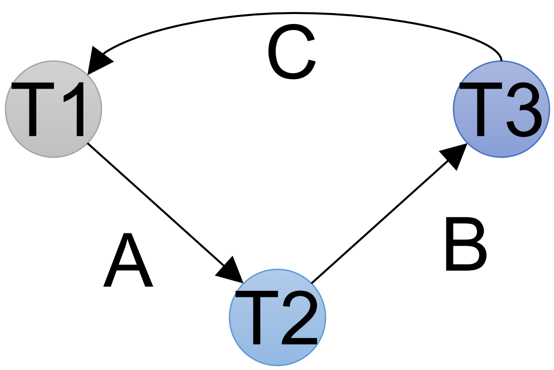


Then there is the following **temporal** cycle in the schedule:

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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



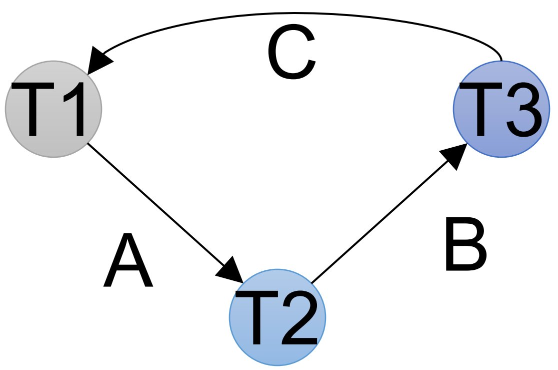
Then there is the following **<u>temporal</u>** cycle in the schedule:

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

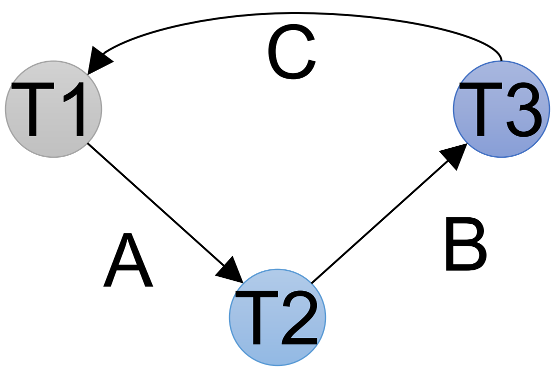
$U_1(A) \rightarrow L_2(A)$

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

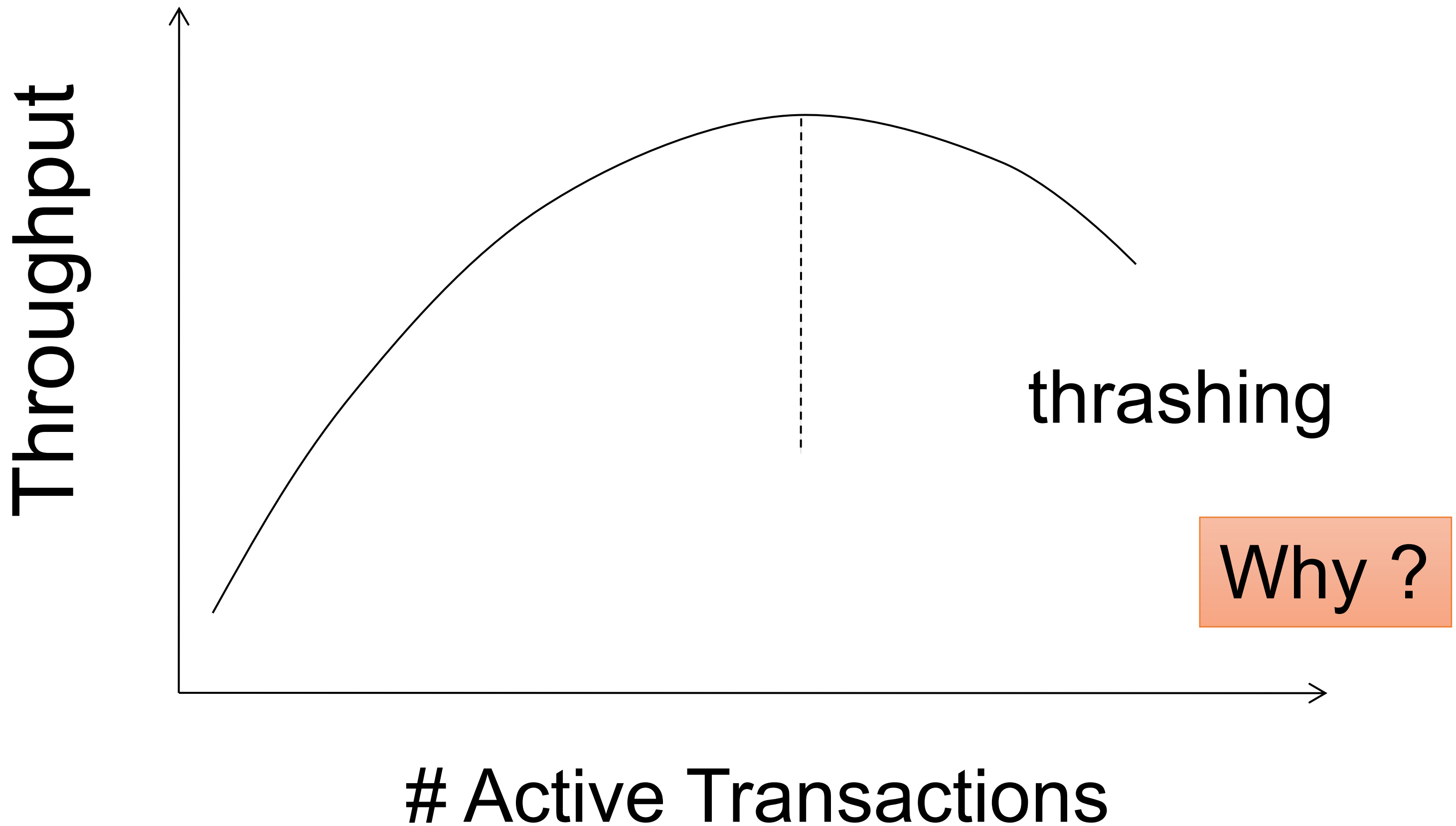
Lock Granularity

- 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

Deadlocks

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

Lock Performance



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 * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

Is this schedule serializable ?

Phantom Problem

Suppose there are two blue products, A1, A2:

T1	T2
SELECT * FROM Product WHERE color='blue'	
	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

Phantom Problem

Suppose there are two blue products, A1, A2:

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘A3’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

$$\boxed{R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)}$$

Phantom Problem

Suppose there are two blue products, A1, A2:

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product 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 * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product 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