CSE 344

AUGUST 13TH

ISOLATION

ADMINISTRIVIA

- **WQ7 due today**

- **HW8 due Wednesday**

- **Final on Friday**
  - strong focus on 2^{nd} half material
    - but first half still fair game (expect some small Qs)
  - more details on Wednesday

FEEDBACK

- **Course evaluations out**
  - these help us out a lot
  - (may help your grade if participation is high)

- **Feedback on Tech Interview talk**
  - https://goo.gl/forms/ZxCGt0ATJ0VU3n8S2

CONFLICT SERIALIZABILITY

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

Every conflict-serializable schedule is serializable

The converse is not true (why?)

LOCKING SCHEDULER

Simple idea:

- **Each element has a unique lock**
- **Each transaction must first acquire the lock before reading/writing that element**
- **The transaction must eventualy release the lock(s)**
- **Until then, another transaction wanting the lock must wait**
    - lock delays the second transaction
    - forces the next operation to come after first txn's release

By using locks scheduler ensures conflict-serializability

MORE NOTATIONS

$L_i(A)$ = 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)	

EXAMPLE

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); $U_1(B)$;	

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	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

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

RECALL...

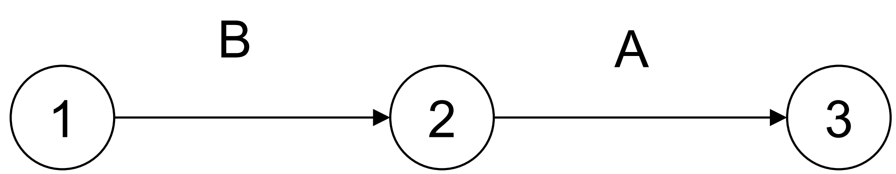
Precedence graph:

- **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**
  - (T_i must come before T_j in any equivalent serial ordering)

The schedule is conflict-serializable iff the precedence graph is acyclic

EXAMPLE 1

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

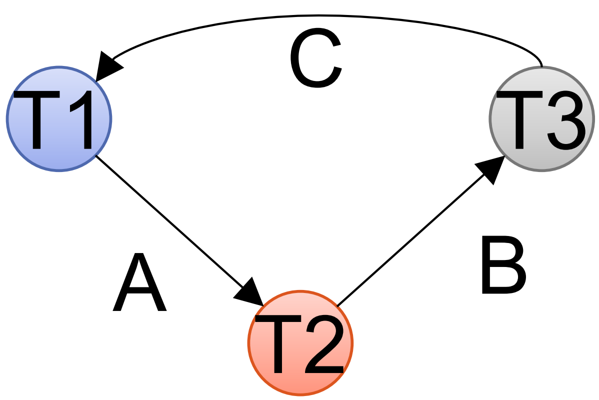


This schedule is conflict-serializable

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

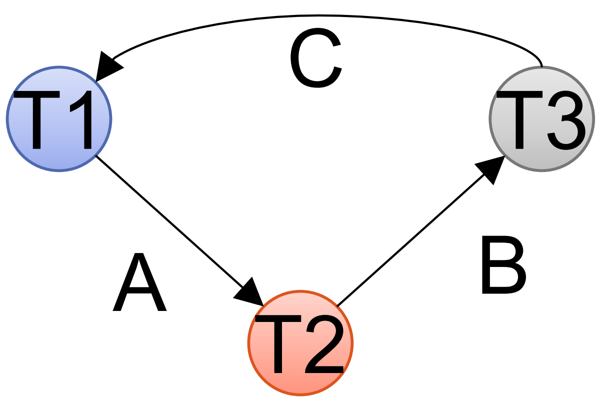
Proof. Suppose not: then there exists a cycle in the precedence graph.



TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

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

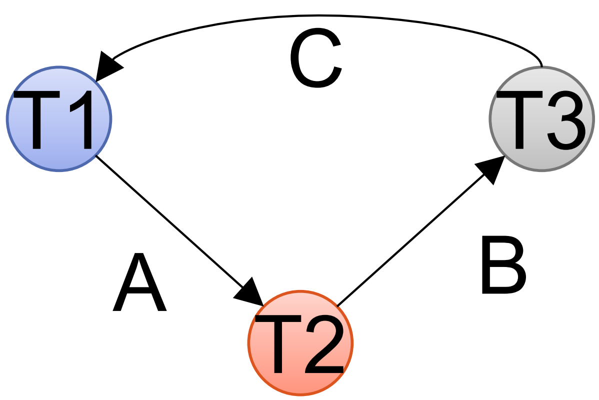


Then there is the following **temporal** cycle in the schedule:

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

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



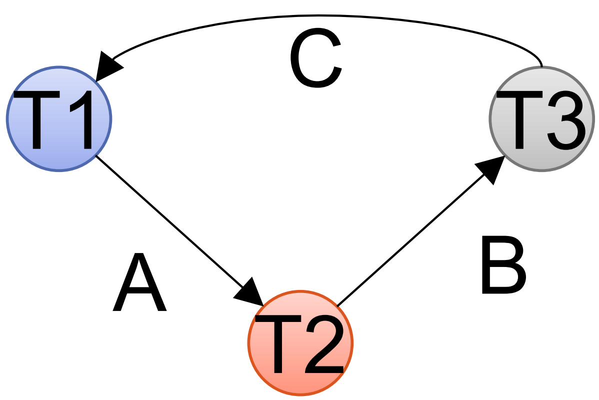
Then there is the following **temporal** cycle in the schedule:

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

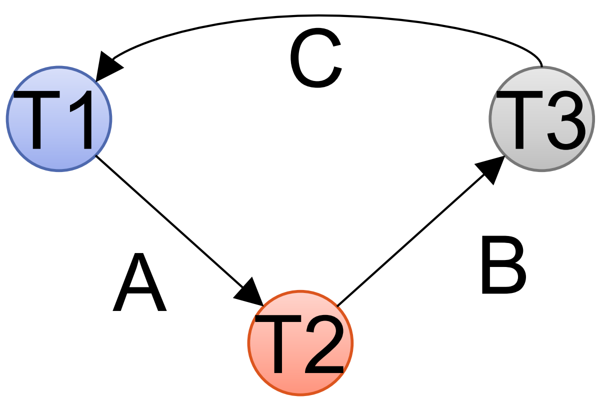
$U_1(A) \rightarrow L_2(A)$ why?

$U_1(A)$ 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:

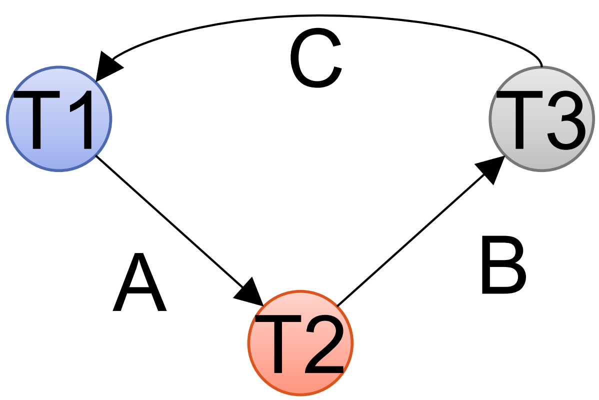
$U_1(A) \rightarrow L_2(A)$

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

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

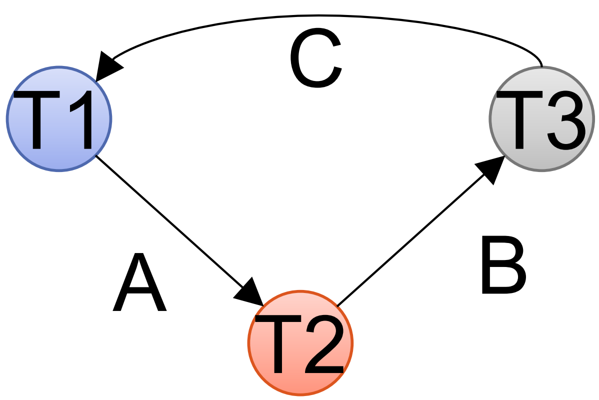
$L_2(A) \rightarrow U_2(B)$ why?

$L_2(A)$ happened strictly *before* $U_2(B)$

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:

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

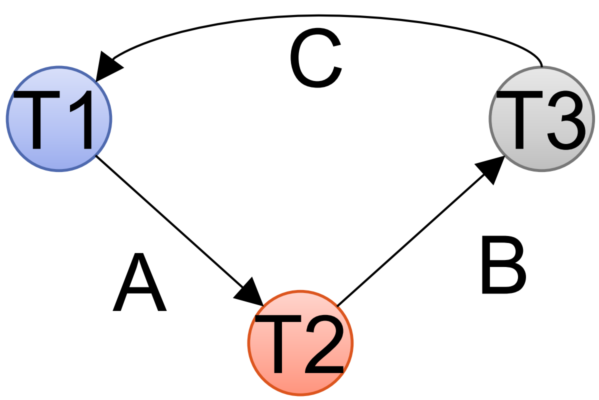
$L_2(A) \rightarrow U_2(B)$

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

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

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

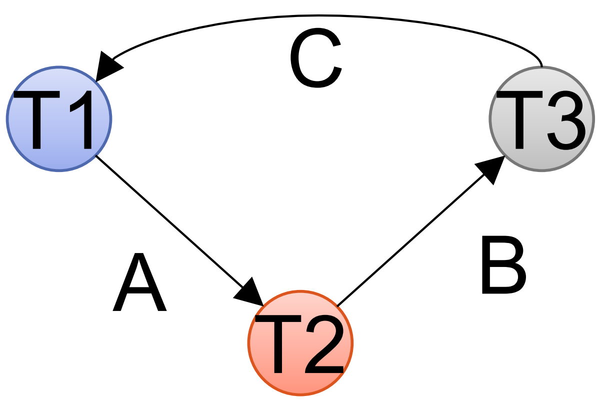
$U_2(B) \rightarrow L_3(B)$

......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	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	

A NEW PROBLEM: 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	

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

A NEW PROBLEM: 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	

Dirty reads of A, B lead to incorrect writes.

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

A NEW PROBLEM: 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	

Dirty reads of A, B lead to incorrect writes.

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

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	T2
$L_1(A)$; READ(A)	
A :=A+100	
WRITE(A);	
	$L_2(A)$; 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);
	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
    - locks accumulate until the end

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

LOCK MODES

S = shared lock (for READ)

X = exclusive lock (for WRITE)

Lock compatibility matrix:

	None	S	X
None			
S			
X			

LOCK MODES

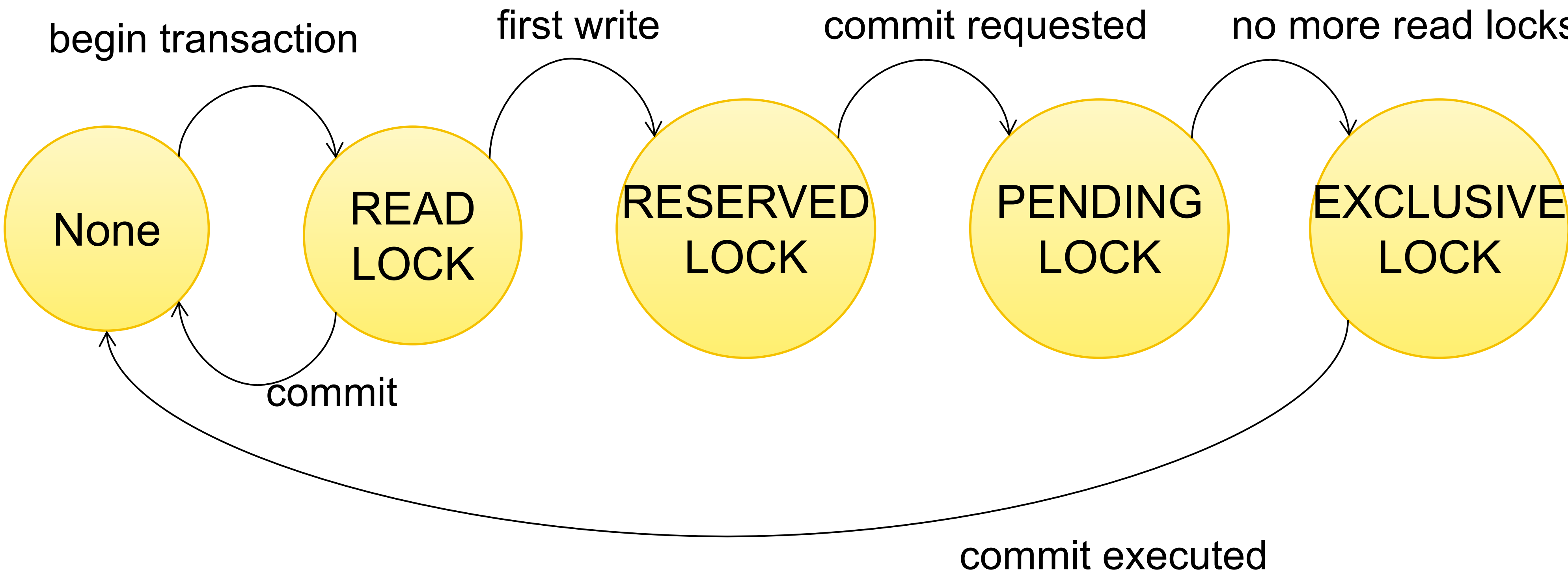
S = shared lock (for READ)

X = exclusive lock (for WRITE)

Lock compatibility matrix:

	None	S	X
None	✔	✔	✔
S	✔	✔	✖
X	✔	✖	✖

SQLITE



Real systems have many types of locks

LOCK GRANULARITY

Fine granularity locking (e.g., tuples)

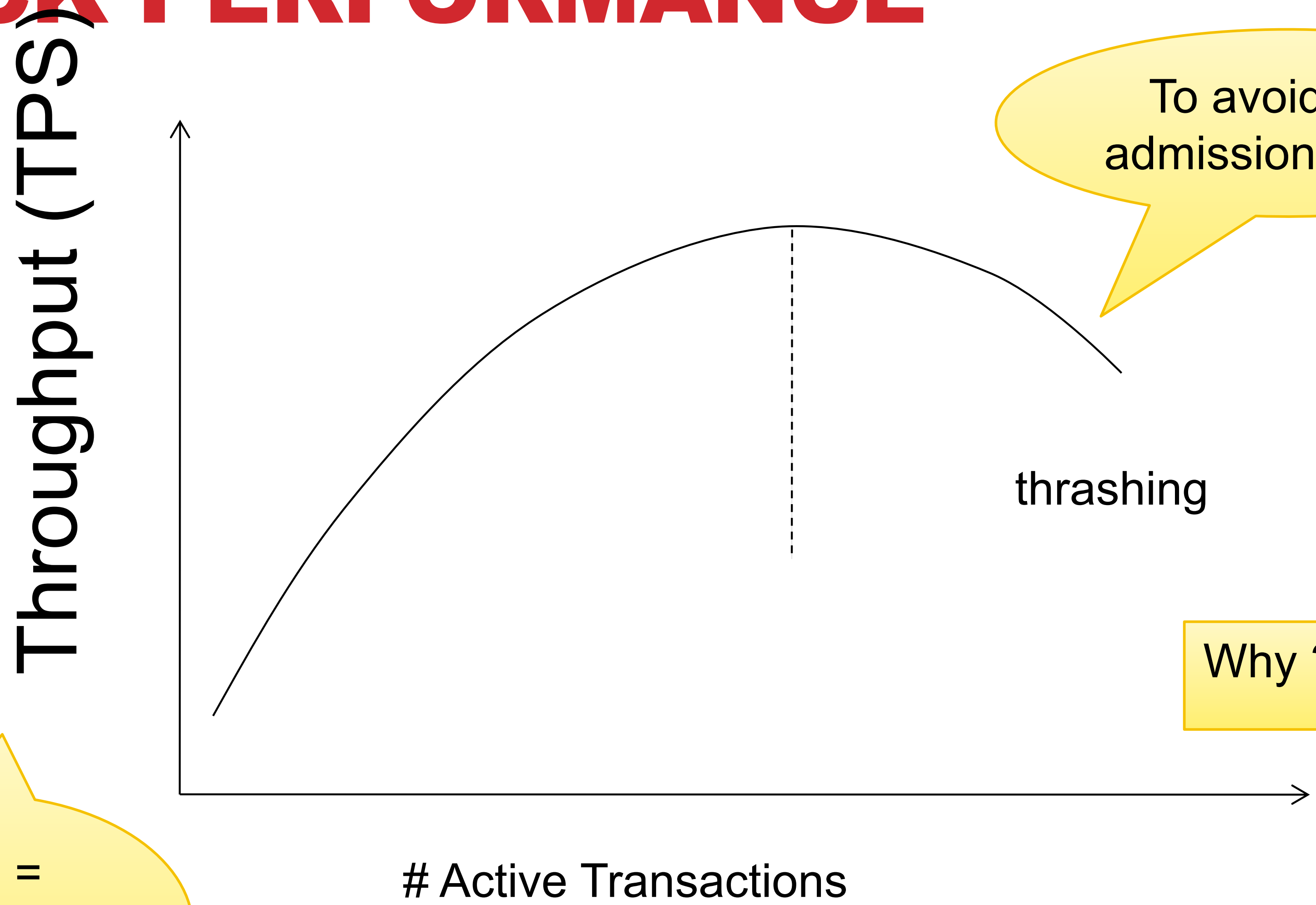
- High concurrency
- High overhead in managing locks
- E.g., SQL Server

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

LOCK PERFORMANCE



To avoid, use admission control

Why ?

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

Suppose there are two blue products, A1, A2:

PHANTOM PROBLEM

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 ?

Suppose there are two blue products, A1, A2:

PHANTOM PROBLEM

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

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

Suppose there are two blue products, A1, A2:

PHANTOM PROBLEM

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

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 !

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

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

ACID

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 problems

Bottom line: Read the doc for your DBMS!