Introduction to Data Management
CSE 344

Lecture 22:
More Transaction Implementations

Review:
Schedules, schedules, schedules

$R_1(A); R_1(B)$

- The DBMS scheduler determines the order of operations from txns are executed

$t = R_1(A);$ if($t > 20$)
$t = R_1(B)$

- A *serial schedule* is one in which transactions are executed one after the other, in some sequential order

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

- A schedule is *conflict serializable* if it has the same conflicts as a serial schedule

- Conflicts: **data dependencies** between two ops that, if swapped, will lead to different program behavior

A Tale of 🔒

Letting threads R/W data freely leads to inconsistencies

Grab locks on element before R/W

Who gets lock first can lead to inconsistencies

2PL: In every transaction, all lock requests must precede all unlock requests

Schedules are conflict-serializable but not recoverable

strict 2PL :

All locks are held until the transaction commits or aborts.

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

Are We Done? No Deadlocks

- T_1 waits for a lock held by T_2;
- T_2 waits for a lock held by T_3;
- T_3 waits for . . . .
- . . .
- T_n waits for a lock held by T_1

SQL Lite: there is only one exclusive lock; thus, never deadlocks

SQL Server: checks periodically for deadlocks and aborts 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)

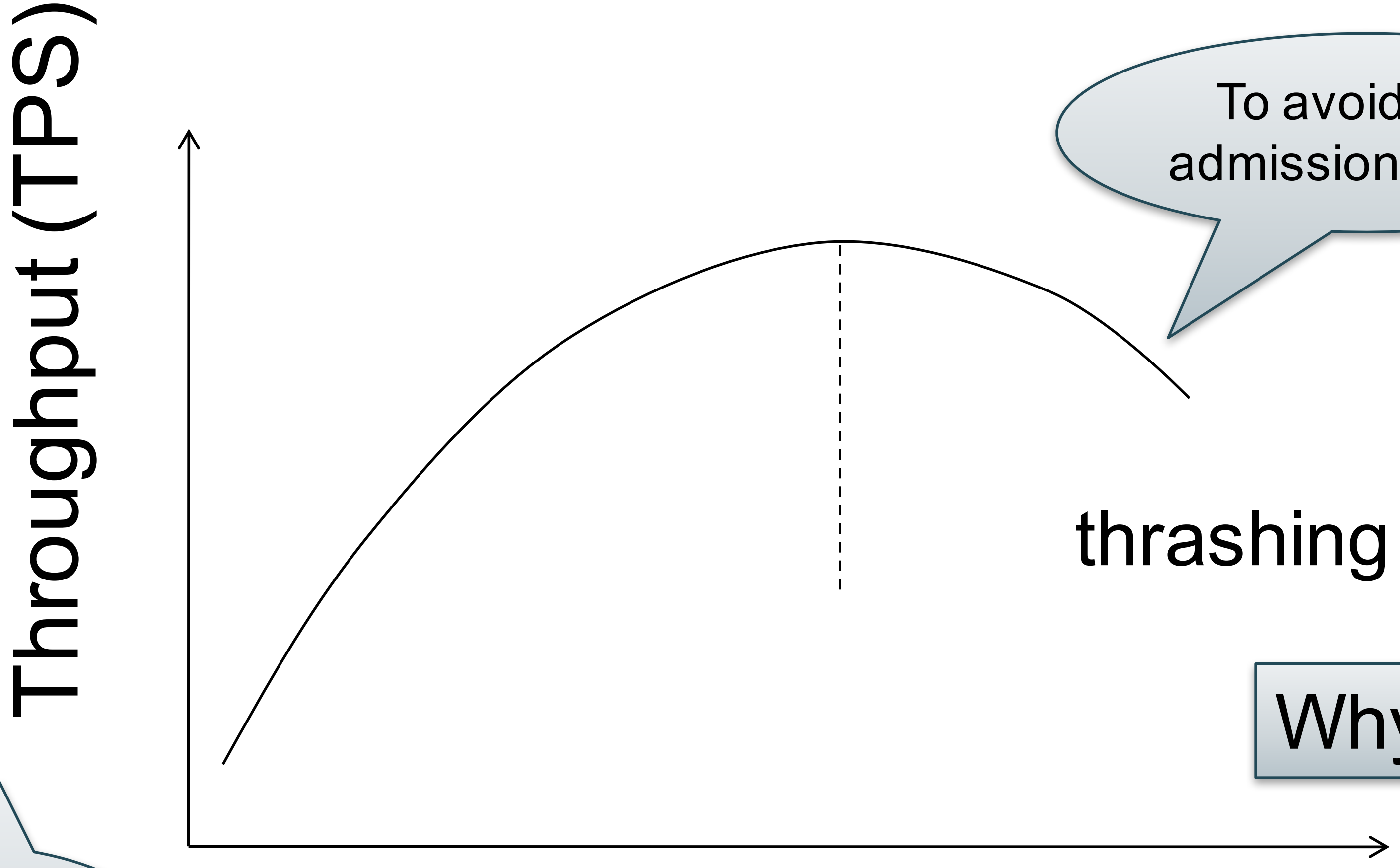
Lock compatibility matrix:

	None	S	X
None	✓	✓	✓
S	✓	✓	✖
X	✓	✖	✖

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

Are We Done? No
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’	

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

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’	

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

$T_2;$

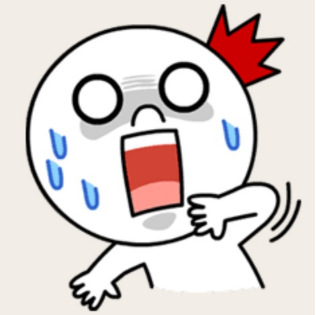
$W_2(A3);R_1(A1);R_1(A2);R_1(A1);R_1(A2);R_1(A3)$

T_1

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 !

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 pbs
- Bottom line: Read the doc for your DBMS!

In-Class Exercise

- Draw the precedence graph for these schedules and the corresponding serial schedules:

$R_2(Y);W_2(Y);R_3(Y);R_1(X);W_1(X);W_3(Y);R_2(X);R_1(Y);W_1(Y)$

$R_3(Y);R_3(Z);R_1(X);W_1(X);W_3(Y);R_2(Z);R_1(Y);R_2(X);W_1(Y);$
$W_2(X)$

In-Class Exercise

$R_2(Y);W_2(Y);R_3(Y);R_1(X);W_1(X);W_3(Y);R_2(X);R_1(Y);W_1(Y)$

$R_3(Y);R_3(Z);R_1(X);W_1(X);W_3(Y);R_2(Z);R_1(Y);R_2(X);W_1(Y);W_2(X)$