Database Systems
CSE 344

Lecture 20:
Transaction Implementations

Announcements

- WQ7 (last!) due Monday Aug 7

- HW7:
  - due on Thursday, Aug 10
  - using JDBC to execute SQL from Java
  - using SQL Server via Azure

- Late Days!
  - After 4 late days ($100 * 0.6^n$)

Recap

- What are transactions?
    - Why do we need them?

- Maintain ACID properties via schedules
    - We focus on the **isolation** property
    - We briefly discussed **consistency** & **durability**
    - We do not discuss **atomicity**

- Ensure conflict-serializable schedules with locks

Implementing a Scheduler

Major differences between database vendors

- Locking Scheduler
  - Aka "pessimistic concurrency control"
  - SQLite, SQL Server, DB2

- Multiversion Concurrency Control (MVCC)
  - Aka "optimistic concurrency control"
  - Postgres, Oracle

We discuss only locking in 344

Locking Scheduler

Simple idea:

- Each element has a unique lock

- Each transaction must first acquire the lock before reading/writing that element

- If lock is taken by another transaction, then wait

- The transaction must release the lock(s)

By using locks, scheduler **can** ensure conflict-serializability

What Data Elements are Locked?

Major differences between vendors:

- Lock on the entire database
  - SQLite

- Lock on individual records
  - SQL Server, DB2, etc.
  - can be even more fine-grained by having **different types** of locks (allows more txns to run simultaneously)

Locks in the Abstract

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)	

$r_1(A); w_1(A); r_2(A); w_2(A); r_2(B); w_2(B); r_1(B); w_1(B);$

Notation

$L_i(A)$ = transaction T_i acquires lock for element A

$U_i(A)$ = transaction T_i releases lock for element A

$r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$; $r_1(B)$; $w_1(B)$;

$L_1(A)$; $r_1(A)$; $w_1(A)$; $U_1(A)$; $L_1(B)L_2(A)$;
$r_2(A)$; $w_2(A)$; $U_2(A)$; $L_2(B)$; $r_2(B)$; $w_2(B)$; $U_2(B)$;
$r_1(B)$; $w_1(B)$; $U_1(B)$;

Example

$L_1(A); r_1(A); w_1(A); U_1(A); L_1(B)L_2(A);$
$r_2(A); w_2(A); U_2(A); L_2(B); r_2(B); w_2(B); U_2(B);$
$r_1(B); w_1(B); U_1(B);$

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); $U_1(B)$;	release
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Scheduler has ensured a conflict-serializable schedule

10

But….

$L_1(A);r_1(A);w_1(A);U_1(A);L_1(B)L_2(A);$
$r_2(A);w_2(A);U_2(A);L_2(B);r_2(B);w_2(B);U_2(B);$
$r_1(B);w_1(B);U_1(B);$

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

2PL approach developed by Jim Gray

Example: 2PL transactions

T1

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

READ(B)
B := B+100
WRITE(B); $U_1(B)$;

Now it is conflict-serializable

T2

$L_2(A);$ READ(A)
A := A*2
WRITE(A);
$L_2(B);$ BLOCKED…

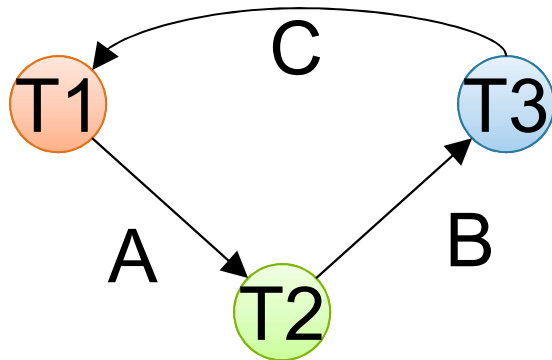…GRANTED; READ(B)
B := B*2
WRITE(B); $U_2(A); U_2(B)$;

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Two Phase Locking (2PL)

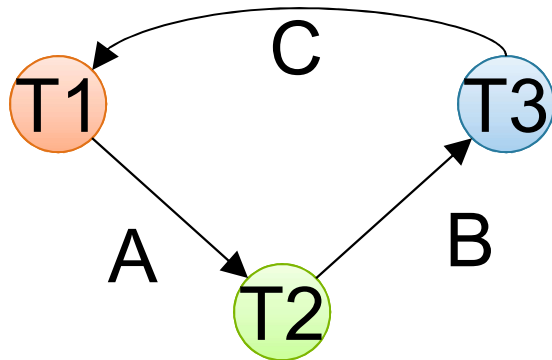**Theorem**: 2PL ensures conflict serializability

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:

C

T1 → T3

A → T2 → B

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

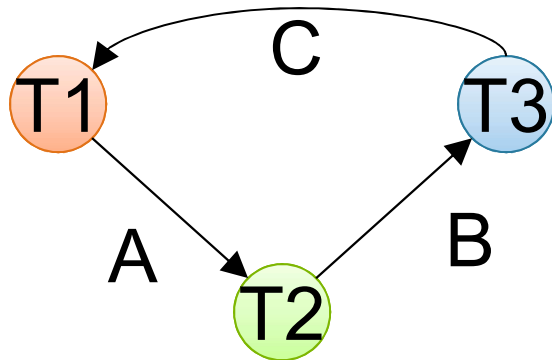**Proof**. Suppose not: then there exists a cycle in the precedence graph.

Then there is the following **temporal** cycle in the schedule:
$U_1(A) \rightarrow L_2(A)$ why? def of lock

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

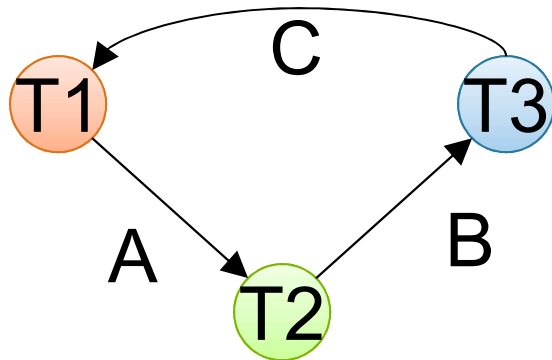**Proof.** Suppose not: then there exists a cycle in the precedence graph.

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

$U_1(A) \rightarrow L_2(A)$ 2PL

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

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

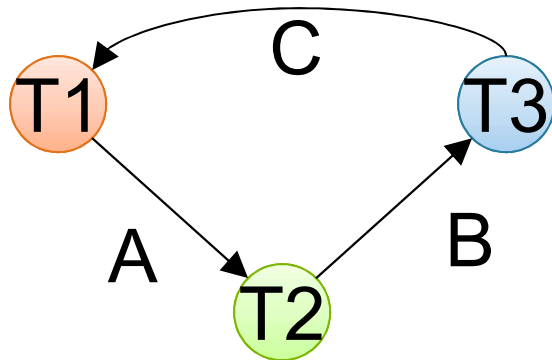**Proof.** Suppose not: then there exists a cycle in the precedence graph.

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

A New Problem:
Non-recoverable Schedule

T1	T2	$A = 50$ $B = 50$

T1

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

READ(B)
B := B+100
WRITE(B); $U_1(B)$;

Abort

Rollback

T2

$L_2(A);$ READ(A) X: 150
A := A*2   A: 300
WRITE(A);
$L_2(B);$ BLOCKED…

…GRANTED; READ(B)
B := B*2
WRITE(B); $U_2(A); U_2(B);$
Commit

Strict 2PL

The Strict 2PL rule:

> All locks are held until the transaction commits or aborts.

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; $U_2(A)$; $U_2(B)$

Another problem: 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:

Lock Request

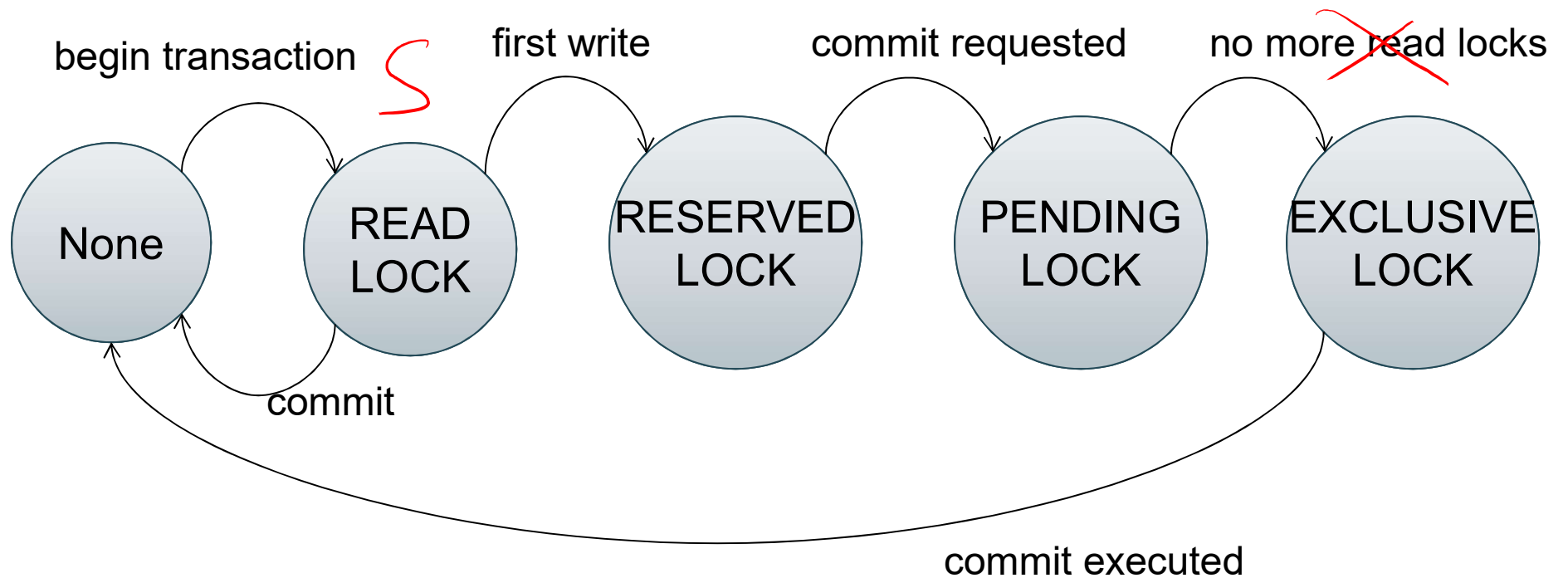| Lock Held	None	S	X
None	✓	✓	✓
S	✓	✓	✗
X	✓	✗	✗

Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)

Lock compatibility matrix:

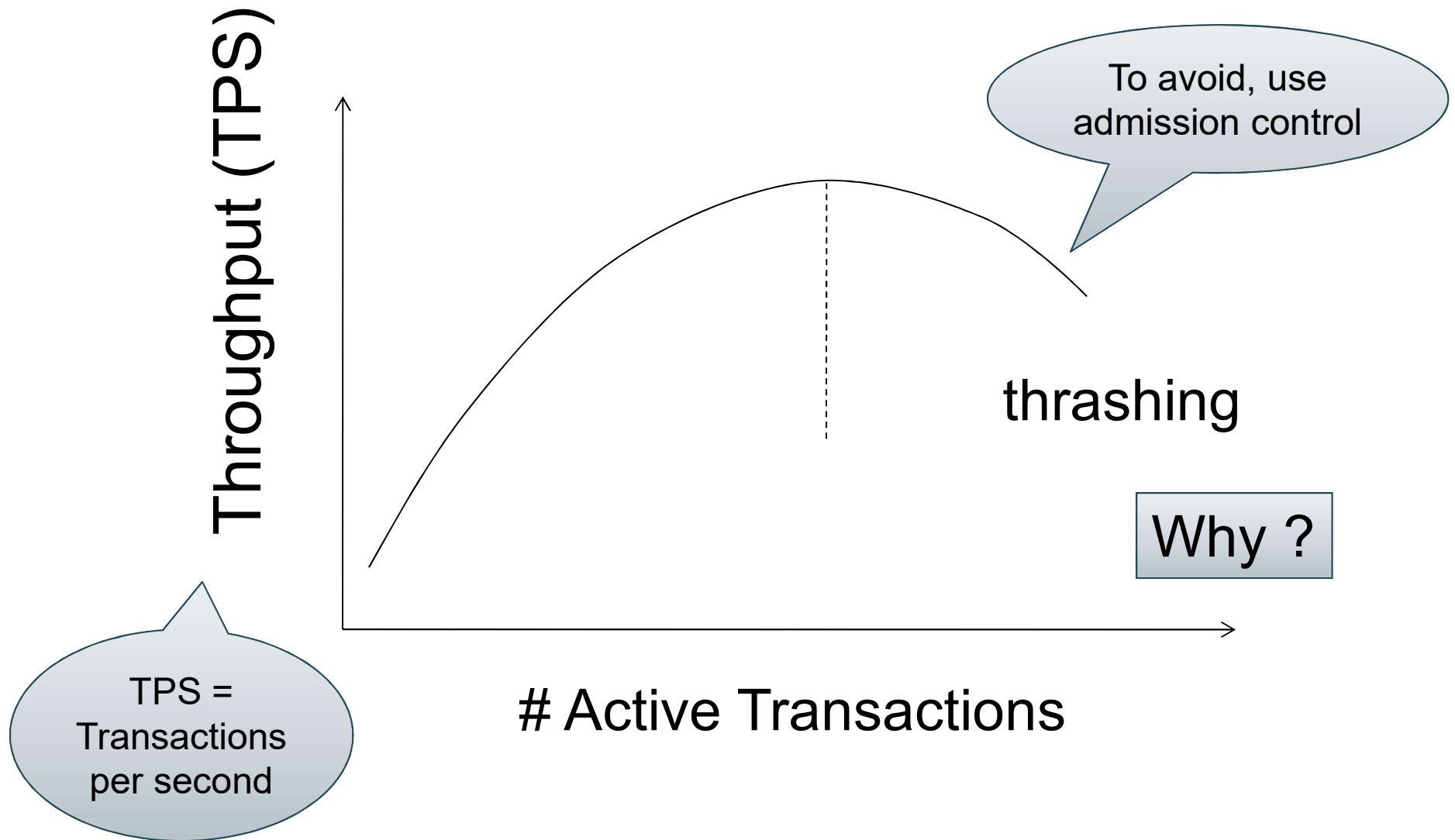|	None	S	X
None	✓	✓	✓
S	✓	✓	✗
X	✓	✗	✗

SQLite

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

Throughput (TPS)

Active Transactions

To avoid, use admission control

thrashing

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'

R1(A1),R1(A2),W2(A3),R1(A1),R1(A2),R1(A3)

Phantom Problem

T1   T2
_____

SELECT *
FROM Product
WHERE color='blue'

                                        INSERT INTO Product(name, color)
                                        VALUES ('A3','blue')

SELECT *
FROM Product
WHERE color='blue'

R1(A1),R1(A2),W2(A3),R1(A1),R1(A2),R1(A3)

W2(A3),R1(A1),R1(A2),R1(A1),R1(A2),R1(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 !

Exception to serializablity of conflict-serializable schedule

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 !

Locking & SQL

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

Dirty Reads

Write-Read Conflict

T_1: WRITE(A)

T_1: ABORT

T_2: READ(A)

Inconsistent Read

> Write-Read Conflict

T_1: A := 20; B := 20;
T_1: WRITE(A)

T_1: WRITE(B)

T_2: READ(A);
T_2: READ(B);

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

Unrepeatable Read

Read-Write Conflict

$L_2(A)$

T$_2$: READ(A);

$U_2(A)$

T$_1$: WRITE(A)

$L_2(A)$

T$_2$: READ(A);

$U_2(A)$

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

ACID

Beware!

In commercial DBMSs:

- Default level is often **NOT** serializable (SQL Server!)
- 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 probs
- Bottom line: Read the doc for your DBMS!

Next two slides: try them on Azure

Demonstration with SQL Server

Application 1:

create table R(a int);

insert into R values(1);

set transaction isolation level serializable;

begin transaction;

select * from R; -- get a shared lock

waitfor delay '00:01'; -- wait for one minute

Application 2:

set transaction isolation level serializable;

begin transaction;

select * from R; -- get a shared lock

insert into R values(2); -- blocked waiting on exclusive lock

     -- App 2 unblocks and executes insert after app 1 commits/aborts

Demonstration with SQL Server

Application 1:

create table R(a int);

insert into R values(1);

set transaction isolation level repeatable read;

begin transaction;

select * from R; -- get a shared lock

waitfor delay '00:01'; -- wait for one minute

Application 2:

set transaction isolation level repeatable read;

begin transaction;

select * from R; -- get a shared lock

insert into R values(3); -- gets an exclusive lock on new tuple

    -- If app 1 reads now, it blocks because read dirty

    -- If app 1 reads after app 2 commits, app 1 sees new value