DATA 514

Lecture 8: Transactions

Announcements

- WQ6 due Monday, March 12
- HW6 due Monday, March 12

Concurrency

- One of the most important properties of a modern DBMS is its ability to manage multiple client sessions simultaneously
- It is important for a DBA to understand how such concurrency control is managed by the database as it can have a significant impact on the overall performance of the database.

Challenges

- Want to execute many apps concurrently
  - All these apps read and write data to the same DB
- Simple solution: only serve one app at a time
  - What's the problem?
- Better: multiple operations need to be executed *atomically* over the DB

What can go wrong?

- Manager: balance budgets among projects
  - Remove $10k from project A
  - Add $7k to project B
  - Add $3k to project C

- CEO: check company's total balance
  - SELECT SUM(money) FROM budget;

- This is called a dirty / inconsistent read aka WRITE-READ conflict

What can go wrong?

- App 1:
  SELECT inventory FROM products WHERE pid = 1

- App 2:
  UPDATE products SET inventory = 0 WHERE pid = 1

- App 1:
  SELECT inventory * price FROM products
  WHERE pid = 1

- This is known as an unrepeatable read aka READ-WRITE conflict

What can go wrong?

Account 1 = $100
Account 2 = $100
Total = $200

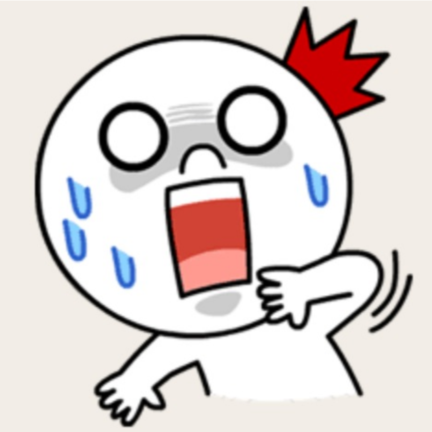
- App 1:
  - Set Account 1 = $200
  - Set Account 2 = $0
- App 2:
  - Set Account 2 = $200
  - Set Account 1 = $0
- At the end:
  - Total = $200

- App 1: Set Account 1 = $200
- App 2: Set Account 2 = $200
- App 1: Set Account 2 = $0
- App 2: Set Account 1 = $0
- At the end:
  - Total = $0

This is called the lost update aka WRITE-WRITE conflict

What can go wrong?

- Buying tickets to the next Bieber concert:
  - Fill up form with your mailing address
  - Put in debit card number
  - Click submit
  - Screen shows money deducted from your account
  - [Your browser crashes]



Changes to the database should be ALL or NOTHING

Transactions

- Collection of statements that are executed atomically (logically speaking)

```
BEGIN TRANSACTION
        [SQL statements]
COMMIT    or
ROLLBACK (=ABORT)
```

```
[single SQL statement]
```

If BEGIN… missing, then TXN consists of a single instruction

Rollback a Transaction

- Initiated by the applications or by the system

- The DB returns to the state prior to the transaction

ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (i.e., looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

Atomic

- **Definition**: A transaction is ATOMIC if all its updates must happen or not at all.
- **Example**: move $100 from A to B

```
BEGIN TRANSACTION;
UPDATE accounts SET bal = bal – 100 WHERE acct = A;
UPDATE accounts SET bal = bal + 100 WHERE acct = B;
COMMIT;
```

The atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

Isolated

- **Definition** An execution ensures that txns are isolated, if the effect of each txn is as if it were the only txn running on the system.

- **Example**: in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Consistent

- Data is in a consistent state when a transaction starts and when it ends.
  - Can be enforced by the DBMS, or ensured by the app
- Example: in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.
- How consistency is achieved by the app:
  - App programmer ensures that txns only takes a consistent DB state to another consistent state
  - DB makes sure that txns are executed atomically

Durable

- A transaction is durable if its effects continue to exist after the transaction and even after the program has terminated
- How? By writing to disk

Isolation: The Problem

- Multiple transactions are running concurrently T_1, T_2, …
- They read/write some common elements A_1, A_2, …
- How can we prevent unwanted interference ?
- The SCHEDULER is responsible for that

Schedules

A *schedule* is a sequence of interleaved actions from all transactions

Serial Schedule

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

- Fact: nothing can go wrong if the system executes transactions serially
  – But database systems don’t do that because we need better performance

A and B are elements in the database
t and s are variables in txn source code

Example

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A Serial Schedule

Time

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

Another Serial Schedule

Time

T1	T2
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	

Serializable Schedule

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

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

This is a serializable schedule.
This is NOT a serial schedule

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)	

How do We Know if a Schedule is Serializable?

Notation

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Key Idea: Focus on *conflicting* operations

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

Conflict Serializability

Conflicts: (it means: cannot be swapped)

Two actions by same transaction T_i: $\boxed{r_i(X); w_i(Y)}$

Two writes by T_i, T_j to same element $\boxed{w_i(X); w_j(X)}$

Read/write by T_i, T_j to same element $\boxed{w_i(X); r_j(X)}$ $\boxed{r_i(X); w_j(X)}$

Conflict Serializability

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

- Every conflict-serializable schedule is serializable
- A serializable schedule may not necessarily be conflict-serializable

Conflict Serializability

Example:

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

Conflict Serializability

Example:

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

⇩

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

Conflict Serializability

Example:

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

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

Conflict Serializability

Example:

$$r_1(A); w_1(A); r_2(A); \boxed{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B)$$

$$r_1(A); w_1(A); \boxed{r_2(A); r_1(B);} w_2(A); w_1(B); r_2(B); w_2(B)$$

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

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); \boxed{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B)$

$\Downarrow$

$r_1(A); w_1(A); \boxed{r_2(A); r_1(B);} w_2(A); w_1(B); r_2(B); w_2(B)$

$\Downarrow$

$r_1(A); w_1(A); r_1(B); r_2(A); \boxed{w_2(A); w_1(B)}; r_2(B); w_2(B)$

$\Downarrow$

….

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

Testing for Conflict-Serializability

Precedence graph:

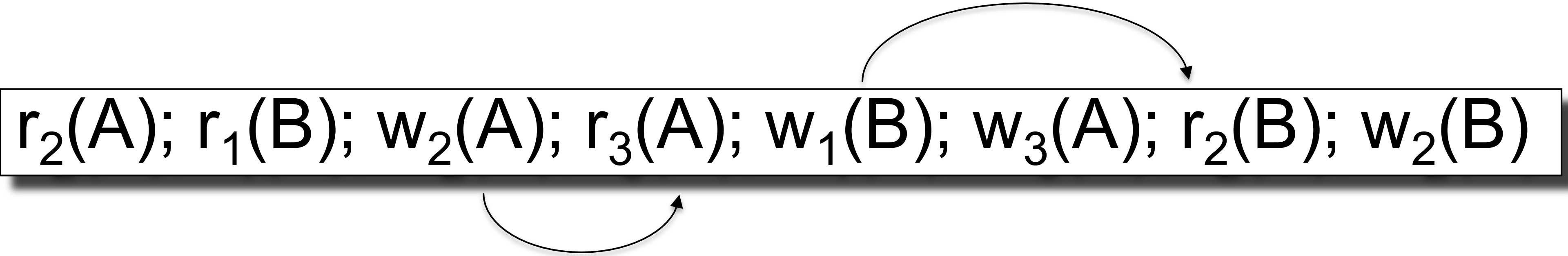
- A node for each transaction T_i,
- An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j

- The schedule is serializable iff the precedence graph is acyclic

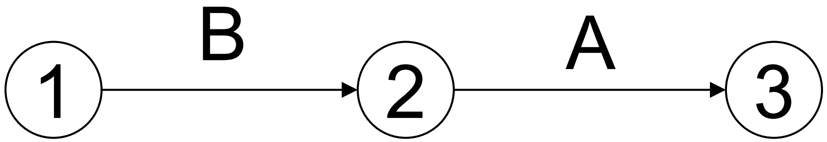
Example 1

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

Example 1



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

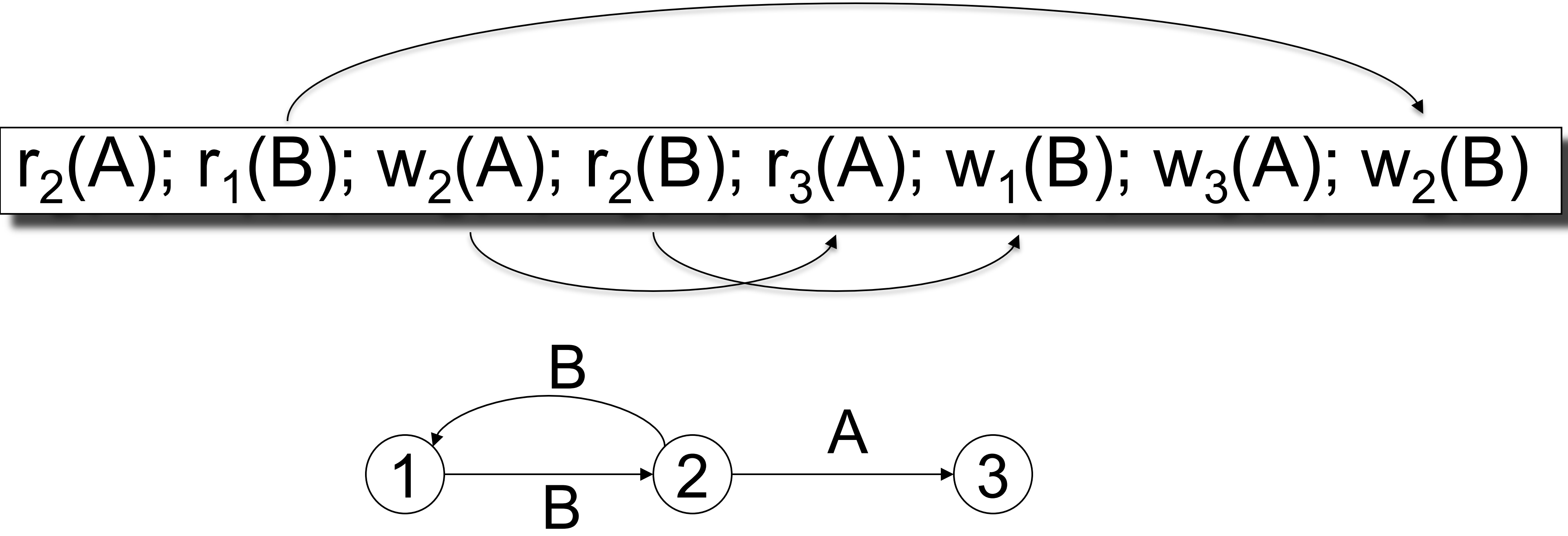


This schedule is conflict-serializable

Example 2

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

Example 2



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

This schedule is NOT conflict-serializable

Scheduler

- Scheduler = is the module that schedules the transaction's actions, ensuring serializability
- Also called Concurrency Control Manager
- We discuss next how a scheduler may be implemented

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

Locking 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)

By using locks scheduler ensures 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

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)	

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

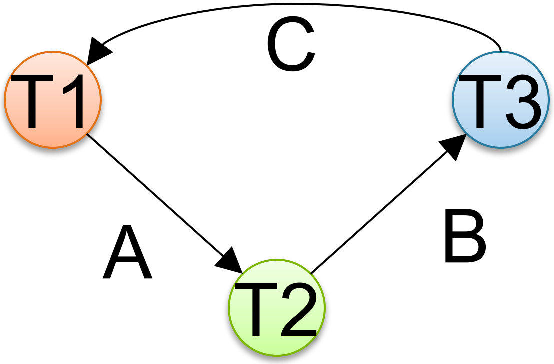
Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

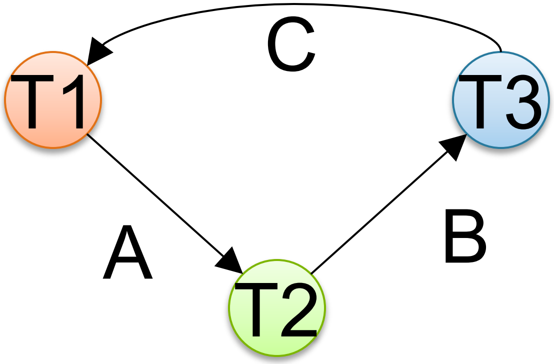
Proof. Suppose not: then there exists a cycle in the precedence graph.



Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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

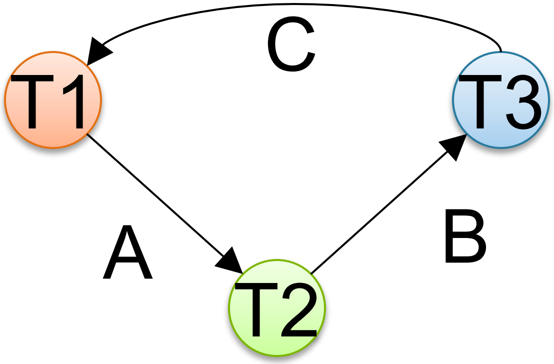


Then there is the following **temporal** cycle in the schedule:

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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



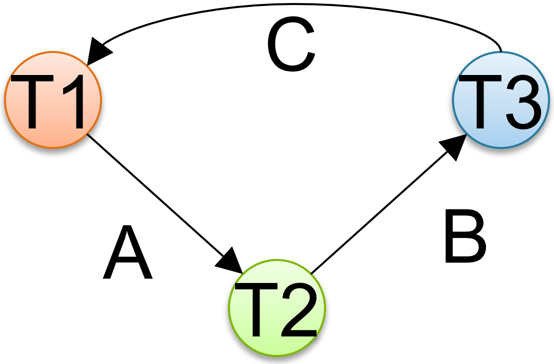
Then there is the following **temporal** cycle in the schedule:

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

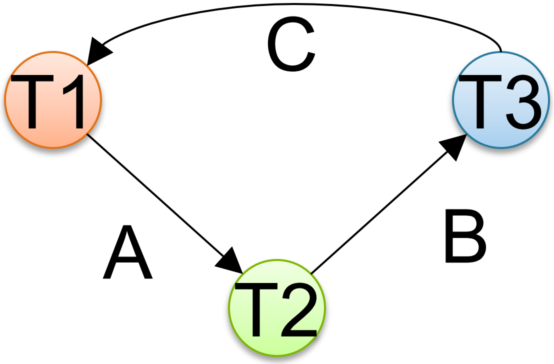
$U_1(A) \rightarrow L_2(A)$

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

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);	
$U_1(A)$,$U_1(B)$; 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

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:

	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

Sqlite

- SQLite is very simple
- More info: http://www.sqlite.org/atomiccommit.html

- Lock types
  - READ LOCK (to read)
  - RESERVED LOCK (to write)
  - PENDING LOCK (wants to commit)
  - EXCLUSIVE LOCK (to commit)

Sqlite

Step 1: when a transaction begins

- Acquire a READ LOCK (aka "SHARED" lock)
- All these transactions may read happily
- They all read data from the database file
- If the transaction commits without writing anything, then it simply releases the lock

Sqlite

Step 2: when one transaction wants to write

- Acquire a RESERVED LOCK
- May coexists with many READ LOCKs
- Writer TXN may write; these updates are only in main memory; others don't see the updates
- Reader TXN continue to read from the file
- New readers accepted
- No other TXN is allowed a RESERVED LOCK

Sqlite

Step 3: when writer transaction wants to commit, it needs *exclusive lock*, which can't coexists with *read locks*

- Acquire a PENDING LOCK
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released

Why not write to disk right now?

Sqlite

Step 4: when all read locks have been released

- Acquire the EXCLUSIVE LOCK
- Nobody can touch the database now
- All updates are written permanently to the database file

- Release the lock and COMMIT

Sqlite Demo

```
create table r(a int, b int);
insert into r values (1,10);
insert into r values (2,20);
insert into r values (3,30);
```

Demonstrating Locking in SQLite

```
T1:
  begin transaction;
  select * from r;
  -- T1 has a READ LOCK
T2:
  begin transaction;
  select * from r;
  -- T2 has a READ LOCK
```

Demonstrating Locking in SQLite

T1:

```
update r set b=11 where a=1;
-- T1 has a RESERVED LOCK
```

T2:

```
update r set b=21 where a=2;
-- T2 asked for a RESERVED LOCK:  DENIED
```

Demonstrating Locking in SQLite

T3:

```
begin transaction;
select * from r;
commit;
-- everything works fine, could obtain READ LOCK
```

Demonstrating Locking in SQLite

T1:

```
commit;
-- SQL error: database is locked
-- T1 asked for PENDING LOCK -- GRANTED
-- T1 asked for EXCLUSIVE LOCK -- DENIED
```

Demonstrating Locking in SQLite

T3':

```
  begin transaction;
  select * from r;
  -- T3 asked for READ LOCK-- DENIED (due to
T1)
```

T2:

```
  commit;
  -- releases the last READ LOCK; T1 can commit
```

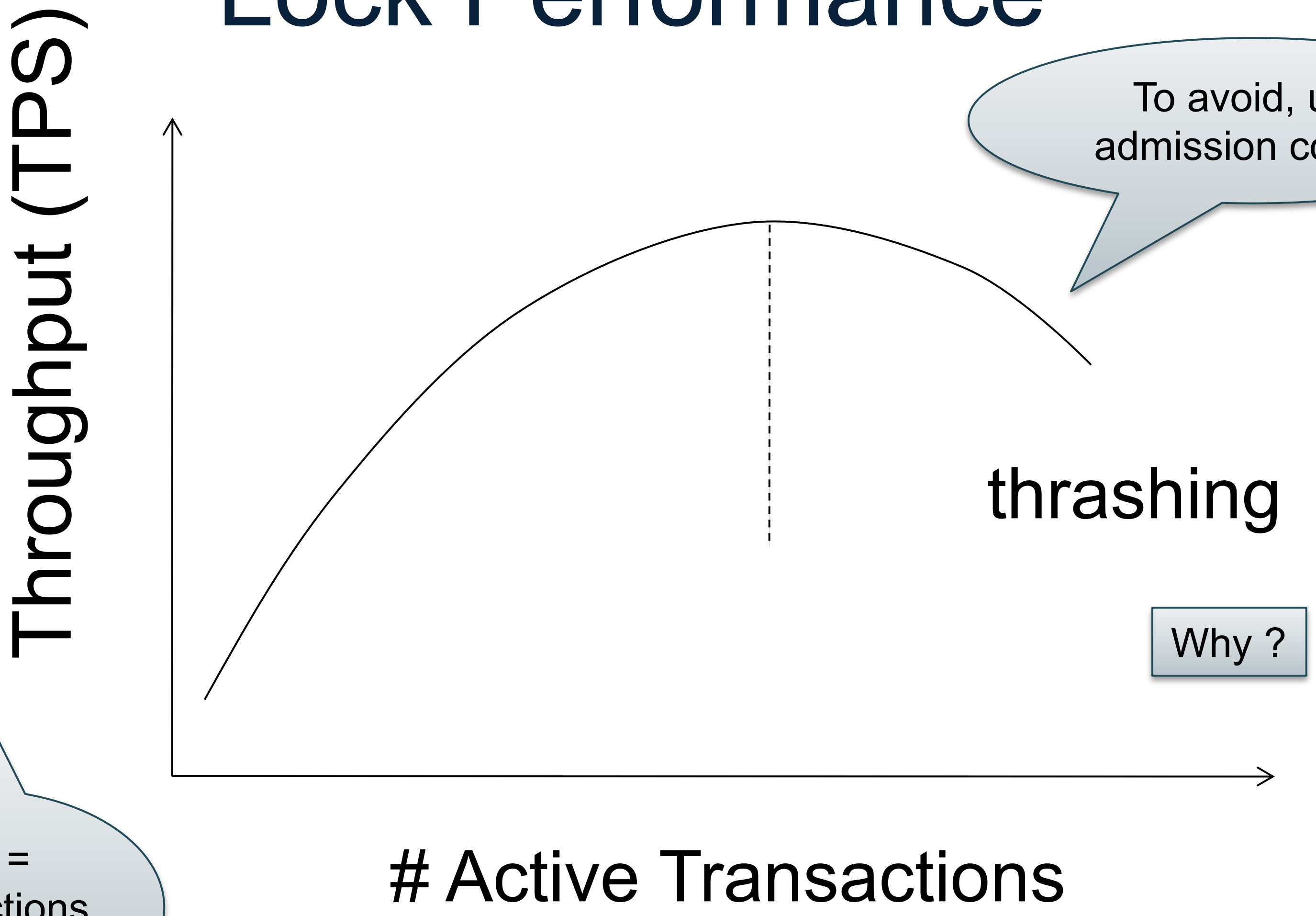
Recap

- What are transactions
  - And why do we need them

- How to maintain ACID properties via schedules
  - We focus on the **isolation** property
  - We do not discuss **atomicity**

- How to ensure conflict-serializable schedules with locks

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’	

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

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 !

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
- Also, some DBMSs do NOT use locking and different isolation levels can lead to different pbs
- 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
```

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
```

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
```