CSE 344

MARCH 7TH – TRANSACTIONS

ADMINISTRIVIA

- **HW7 Due Tonight 11:30**
- **HW8 Due Monday**
  - Max Two Late days
- **Exam Review**
  - Sunday: 5pm EEB 045
- **Section tomorrow**
  - Fair game for final – slides posted

TRANSACTIONS

We use database transactions everyday

- Bank $$$ transfers
- Online shopping
- Signing up for classes

For this class, a transaction is a series of DB queries

- Read / Write / Update / Delete / Insert
- Unit of work issued by a user that is independent from others

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?

Want: multiple operations to be executed *atomically* over the same DBMS

ACID

Atomic

Consistent

Isolated

Durable

Again: by default each statement is its own txn

- Unless auto-commit is off then each statement starts a new txn

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

- (up to what we have learned so far)
- But DBMS don’t do that because we want better overall system performance

EXAMPLE

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

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)

EXAMPLE OF 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)	

REVIEW: 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

Read-Read?

CONFLICT SERIALIZABILITY

Conflicts: (i.e., swapping will change program behavior)

Two actions by same transaction T_i:

$r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element

$w_i(X); w_j(X)$

Read/write by T_i, T_j to same element

$w_i(X); r_j(X)$

$r_i(X); w_j(X)$

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

The converse is not true (why?)

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:

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

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

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

….

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

TESTING FOR CONFLICT-SERIALIZABILITY

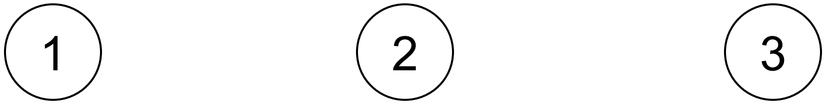
Precedence graph:

- **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**

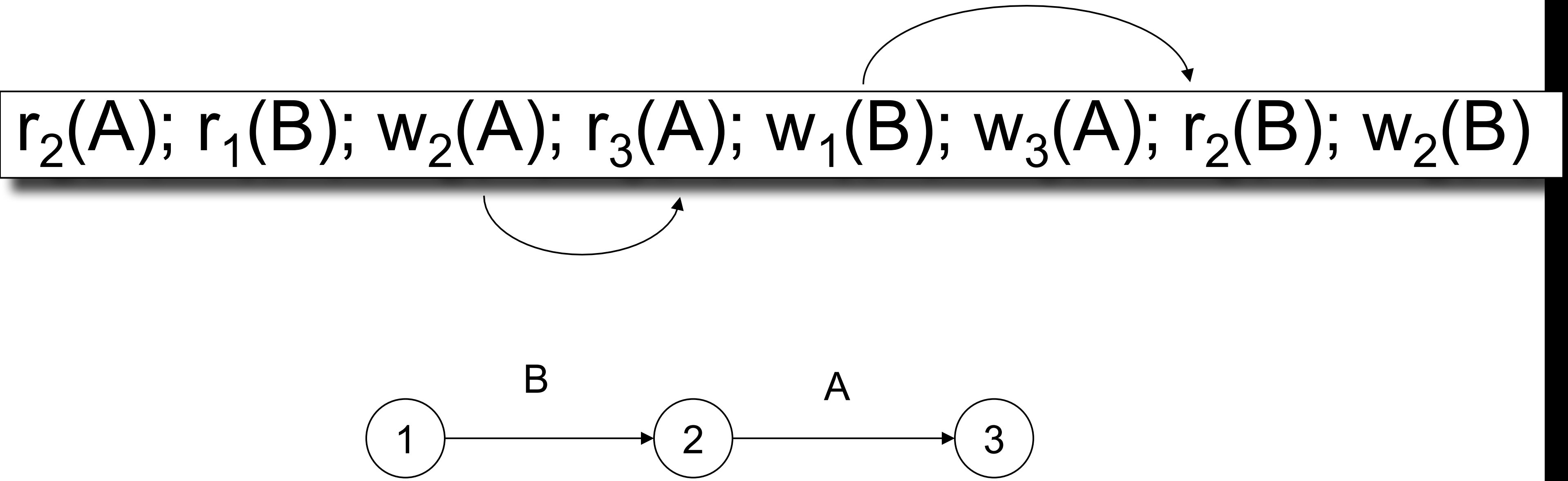
The schedule is conflict-serializable iff the precedence graph is acyclic

EXAMPLE 1

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



EXAMPLE 1

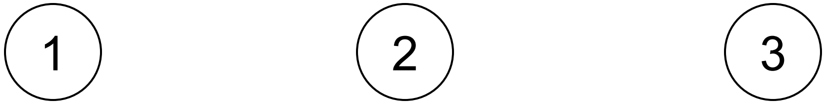


$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

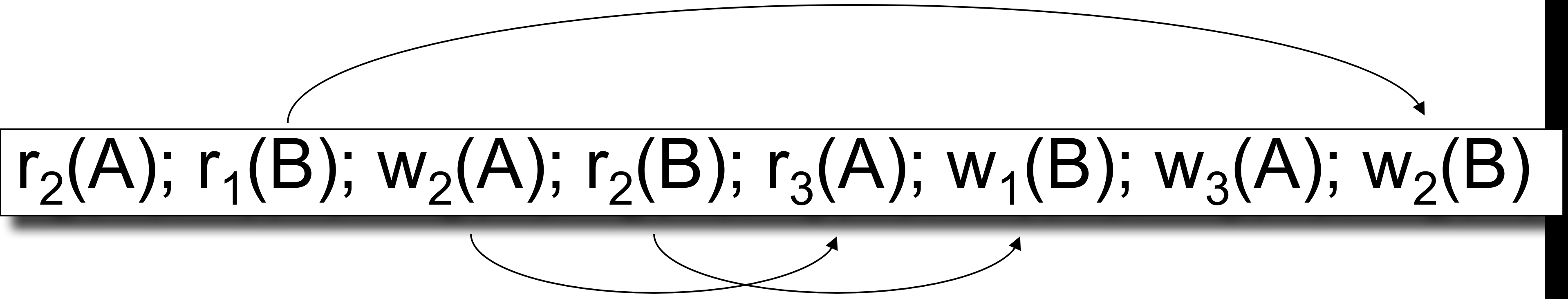
This schedule is conflict-serializable

EXAMPLE 2

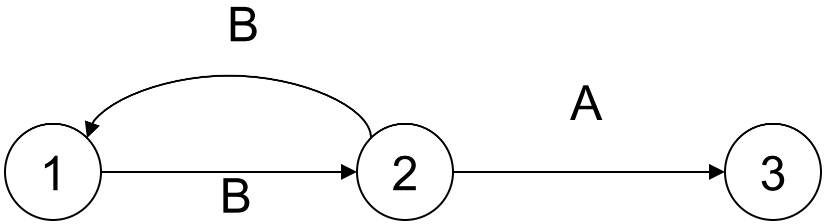
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



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 = 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: Snapshot Isolation (SI)

We discuss only locking schedulers in this class

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

CASE STUDY: 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

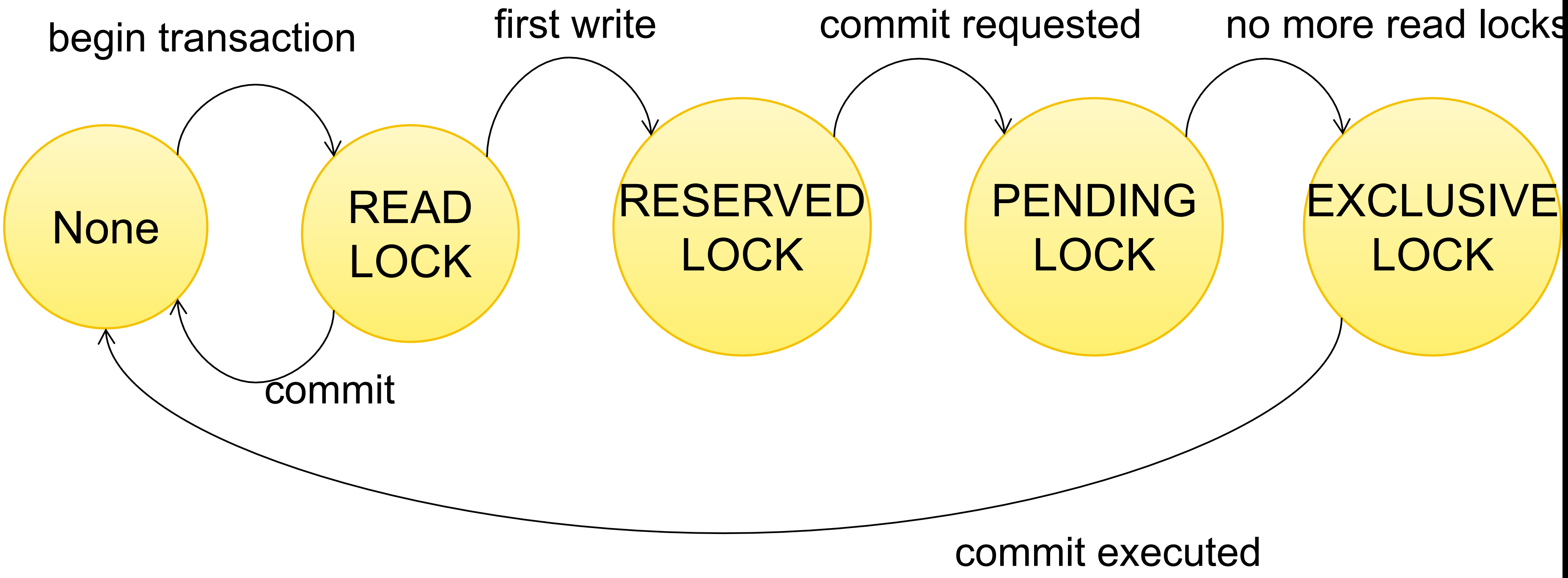
Acquire the EXCLUSIVE LOCK

Nobody can touch the database now

All updates are written permanently to the database file

Release the lock and COMMIT

SQLITE



begin transaction

first write

commit requested

no more read locks

None

READ LOCK

RESERVED LOCK

PENDING LOCK

EXCLUSIVE LOCK

commit

commit executed

SCHEDULE ANOMALIES

What could go wrong if we didn’t have concurrency control:

- Dirty reads (including inconsistent reads)
- Unrepeatable reads
- Lost updates

Many other things can go wrong too

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);

UNREPEATABLE READ

Read-Write Conflict

T_2: READ(A);

T_1: WRITE(A)

T_2: READ(A);

LOST UPDATE

Write-Write Conflict

T_1: READ(A)

T_1: A := A+5

T_1: WRITE(A)

T_2: READ(A);

T_2: A := A*1.3

T_2: WRITE(A);

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

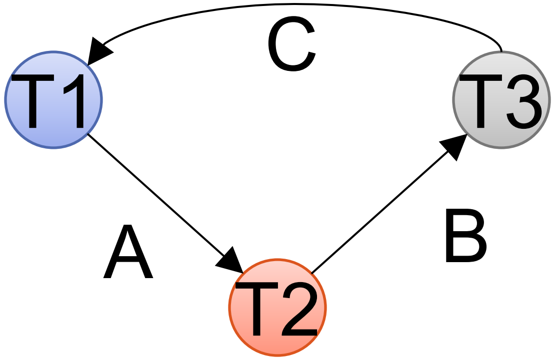
TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

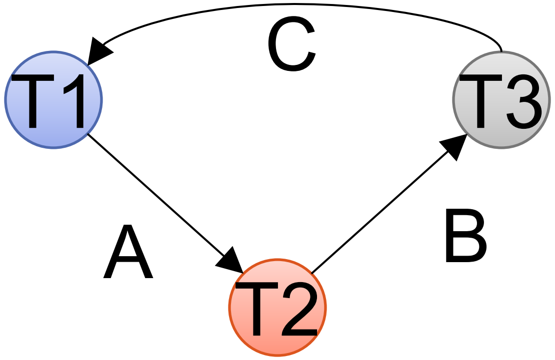
Proof. Suppose not: then there exists a cycle in the precedence graph.



TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

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

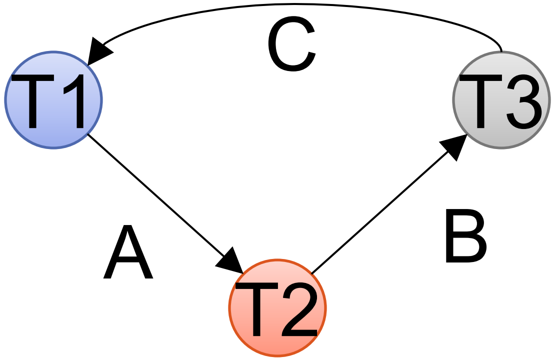


Then there is the following **temporal** cycle in the schedule:

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:

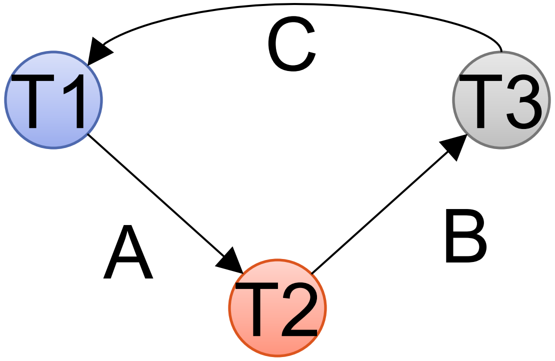
$U_1(A) \rightarrow L_2(A)$ why?

$U_1(A)$ 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.



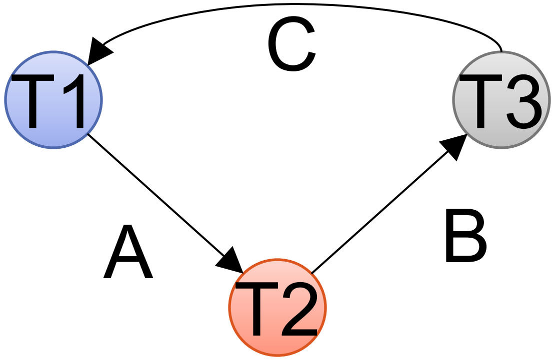
Then there is the following **temporal** cycle in the schedule:

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

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

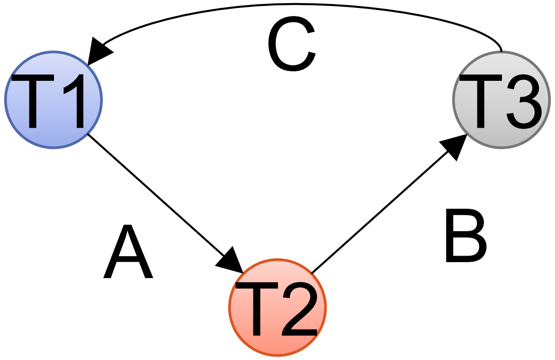
$L_2(A) \rightarrow U_2(B)$ why?

$L_2(A)$ happened strictly *before* $U_1(A)$

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:

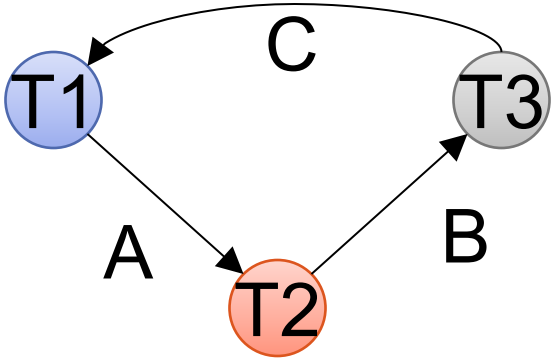
$U_1(A) \rightarrow L_2(A)$

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

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

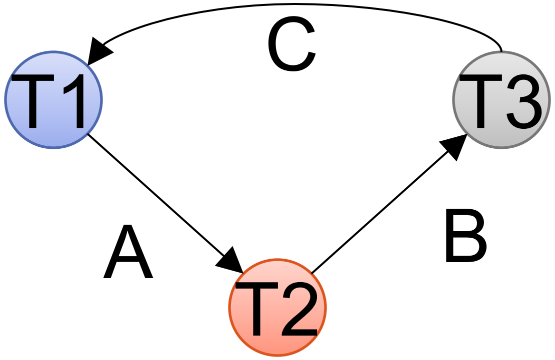
$L_2(A) \rightarrow U_2(B)$

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

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

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

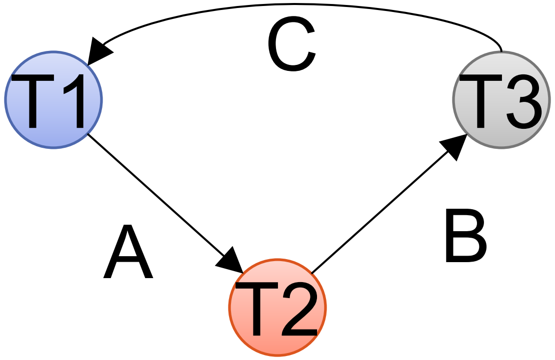
$U_2(B) \rightarrow L_3(B)$

……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

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;
re-check for deadlock more often (why?)

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)

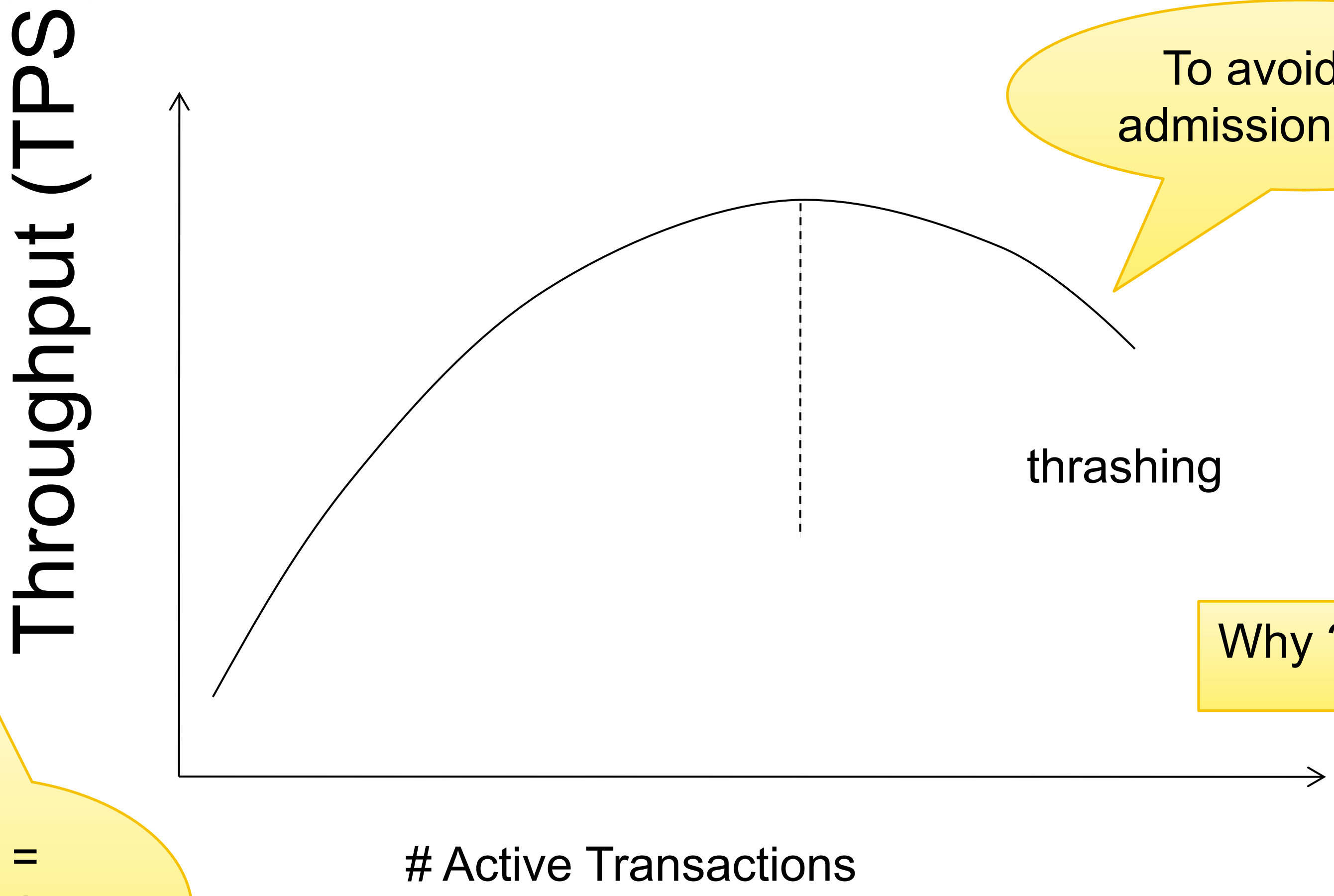
- High concurrency
- High overhead in managing locks
- E.g., SQL Server

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

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

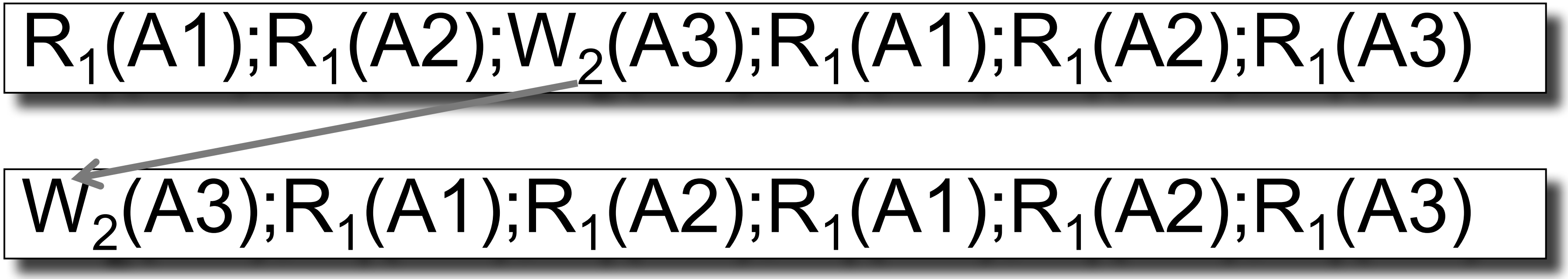
T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘A3’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

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

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**

   ACID

   ```
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
   ```

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!