CSE544
Data Management

Lectures 16-18

Transactions: Concurrency Control

Announcmenets

- Poster presentations on Friday!
- Please arrive around 9:30 to set up
- There will be easels, and power cords for laptops
- Pizza around 12pm

Transactions

- We use database transactions everyday
  - Bank $$$ transfers
  - Online shopping
  - Signing up for classes

- Applications that talk to a DB ***must*** use transactions in order to keep the database consistent.

What’s the big deal?

Challenges

- Suppose we only serve one app at a time
  - No problem…
- Suppose we execute apps concurrently
  - What's the problem?
- Want: multiple operations to be executed *atomically* over the same DBMS

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 a 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

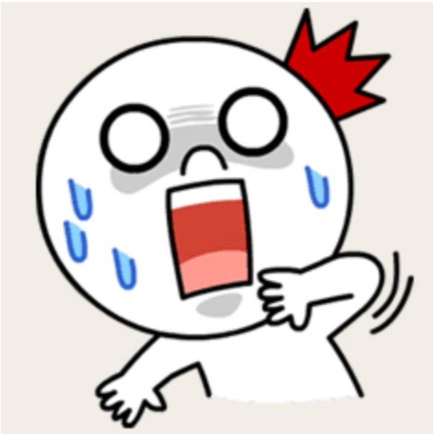
- App 1:
  - Set Account 1 = $200
  - Set Account 2 = $0
- App 2:
  - Set Account 2 = $200
  - Set Account 1 = $0
- At the end:
  - Total = $200

- 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]



Lesson:

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

Know your ~~chemistry~~ transactions: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a DBMS state where integrity holds, to another where integrity holds
    - remember integrity constraints?
- 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;
```

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.

```
-- App 1:
BEGIN TRANSACTION;

  SELECT inventory
  FROM products
  WHERE pid = 1;

  SELECT inventory * price
  FROM products
  WHERE pid = 1;

COMMIT
```

```
-- App 2:
BEGIN TRANSACTION;
  UPDATE products
  SET inventory = 0
  WHERE pid = 1;
COMMIT;
```

Consistent

- Recall: integrity constraints govern how values in tables are related to each other
  - Can be enforced by the DBMS, or ensured by the app
- How consistency is achieved by the app:
  - App programmer ensures txns takes consistent state to consistent state
  - DB makes sure that txns are atomic+isolated

Durable

- A transaction is durable if its effects continue to exist after the transaction and even after the program has terminated

Rollback transactions

- If the app gets to a state where it cannot complete the transaction successfully, execute ROLLBACK
- The DB returns to the state prior to the transaction

Implementing Transactions

Need to address two problems:

- ”I” – Isolation:
  – Means concurrency control
- “A” – Atomicity:
  – Means recover from crash

Transaction Schedules

Modeling a Transaction

- Database = a collection of *elements*
  - An element can be a record (logical elements)
  - Or can be a disc block (physical element)

Database:	A	B	C	D	…

- Transaction = sequence of read/writes of elements

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 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: 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

Example

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: (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:

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

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

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:

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

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

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

….

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

Serializable, Not Conflict-Serializable

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s + 200
	WRITE(A,s)
	READ(B,s)
	s := s + 200
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

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

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

1 2 3

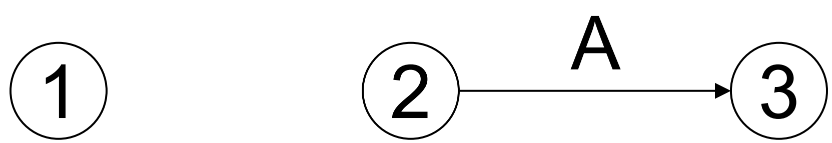
Example 1

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

1 2 3

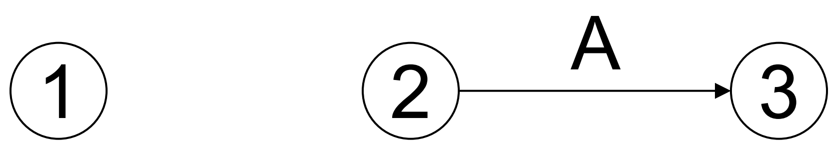
Example 1

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



Example 1

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



Example 1

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

$$1 \xrightarrow{B} 2 \xrightarrow{A} 3$$

Example 1

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

$$1 \xrightarrow{B} 2 \xrightarrow{A} 3$$

This schedule is conflict-serializable

Example 2

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

1 2 3

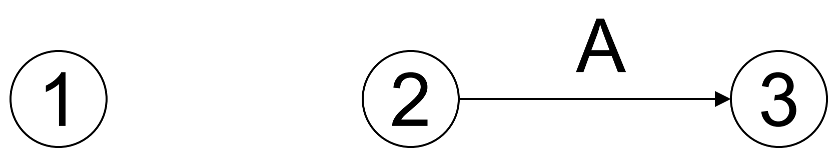
Example 2

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

1 2 3

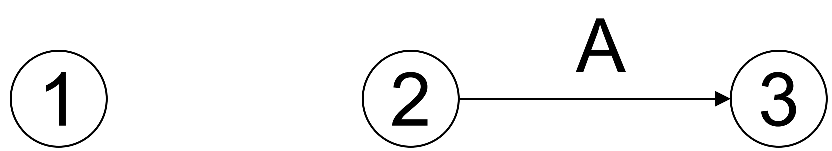
Example 2

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



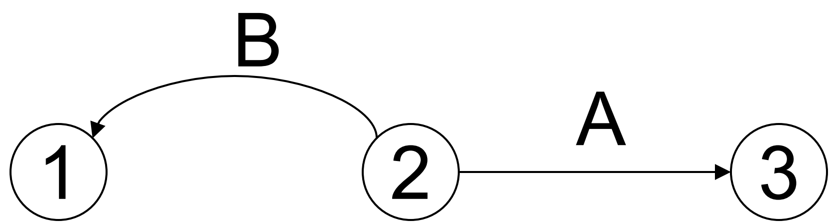
Example 2

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



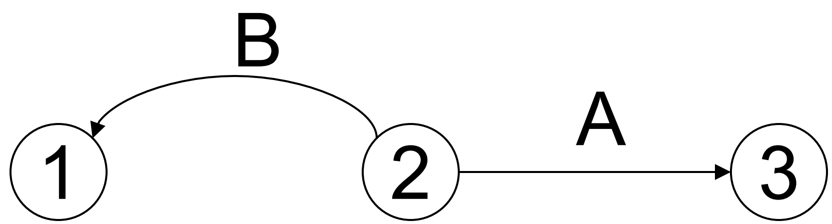
Example 2

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



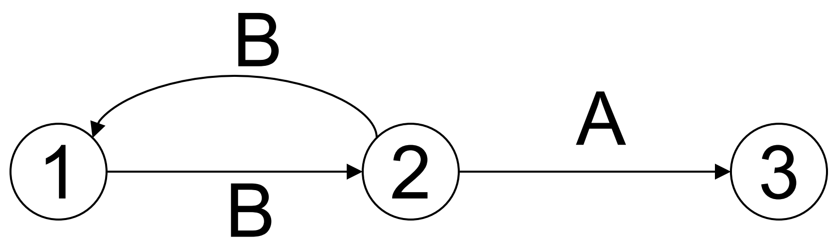
Example 2

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



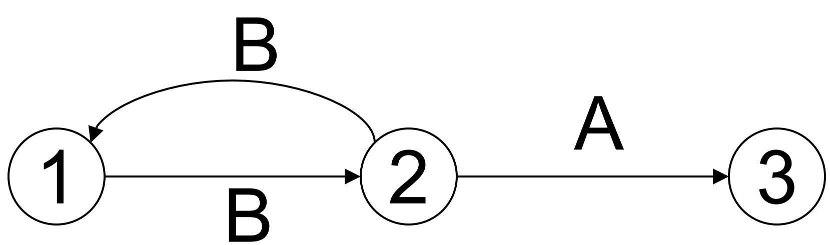
Example 2

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



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

Implementing Transactions

Scheduler

- Scheduler a.k.a. Concurrency Control Manager
  - The module that schedules the transaction's actions
  - Goal: ensure the schedule is serializable

- We discuss next how a scheduler may be implemented

Implementing a Scheduler

Two major approaches:

- Locking Scheduler
  - Aka “pessimistic concurrency control”
  - SQLite, SQL Server, DB2
- Multiversion Concurrency Control (MVCC)
  - Aka “optimistic concurrency control”
  - Postgres, Oracle: Snapshot Isolation (SI)

Lock-based Implementation of Transactions

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, then wait
- The transaction must release the lock(s)

Actions on Locks

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

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

Let's see this in action…

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

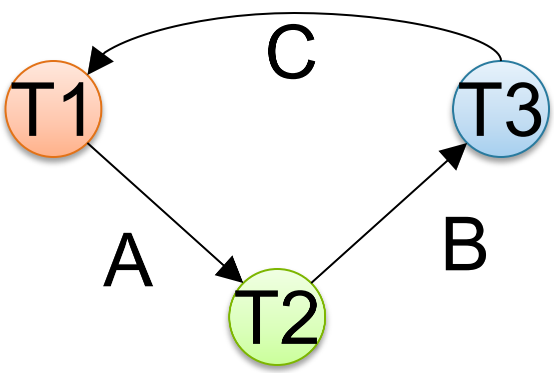
Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

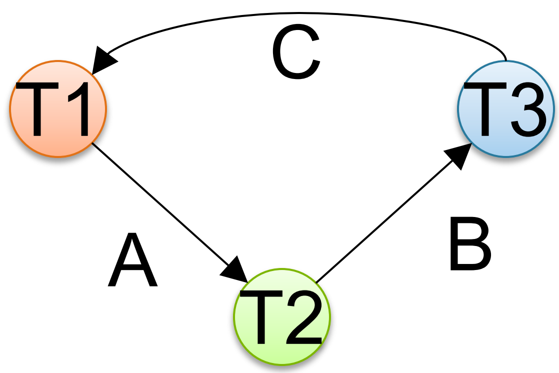
Proof. Suppose not: then there exists a cycle in the precedence graph.



Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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

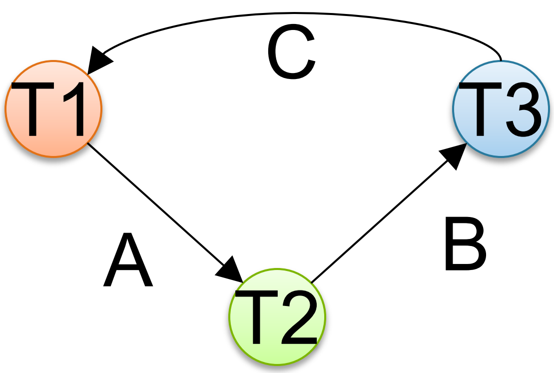


Then there is the following **temporal** cycle in the schedule:

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:

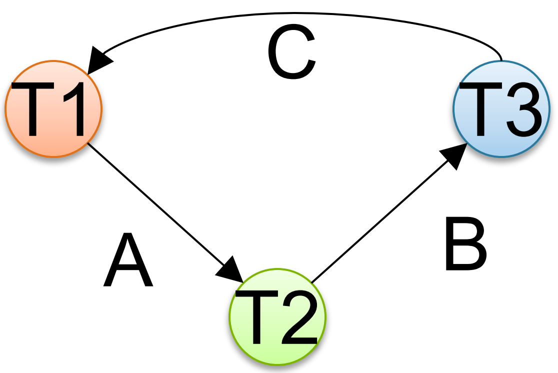
$U_1(A) \rightarrow L_2(A)$ why?

$U_1(A)$ 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.



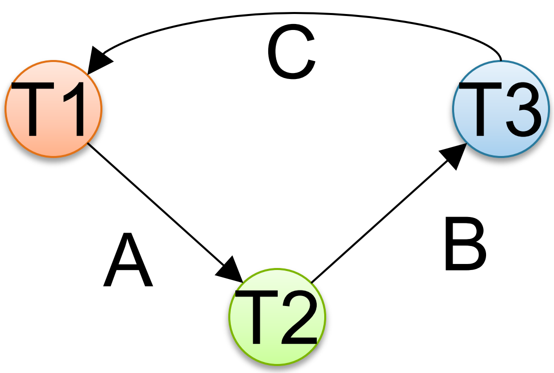
Then there is the following **temporal** cycle in the schedule:

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

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

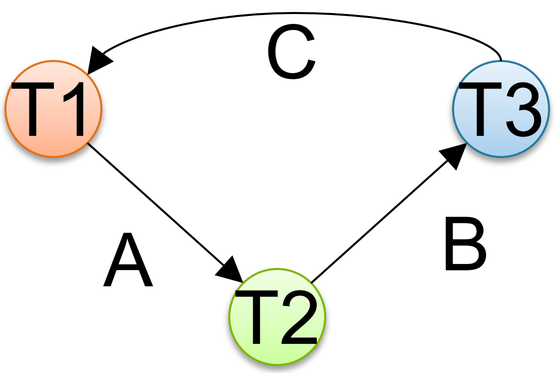
$L_2(A) \rightarrow U_2(B)$ why?

$L_2(A)$ 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:

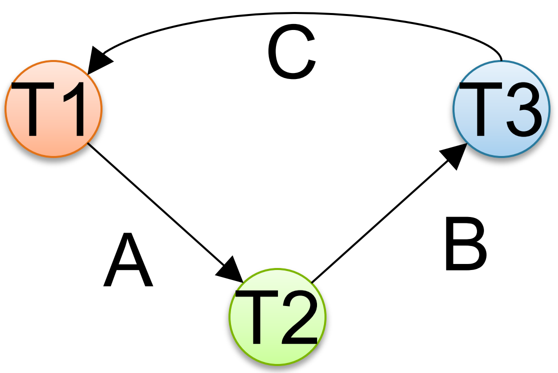
$U_1(A) \rightarrow L_2(A)$

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

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

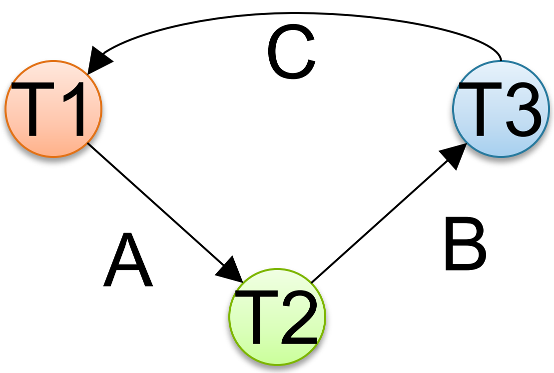
$L_2(A) \rightarrow U_2(B)$

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

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

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

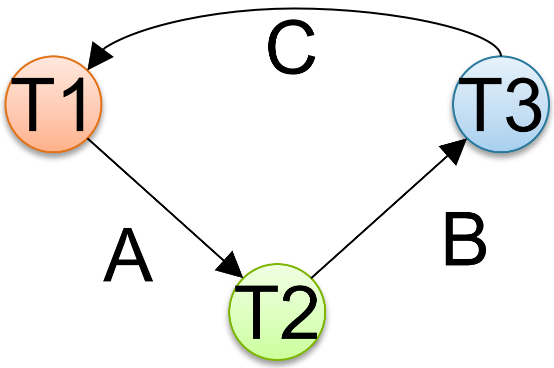
$U_2(B) \rightarrow L_3(B)$

......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.

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

Recoverable Schedule

- A schedule is *recoverable* if, whenever a transaction commits, then all transactions whose values it read have already committed
- A schedule *avoids cascading aborts*, whenever a transaction reads an element, then the transaction that wrote it must have already committed
- Avoiding cascading aborts implies recoverable (why?)

Strict Schedules

- A schedule is *strict* if every value written by a transaction T is not read or overwritten by another transaction until after T commits or aborts

Strict 2PL

- Every scheduled produced by Strict 2PL is conflict-serializable, avoids cascading aborts, and is strict.

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)

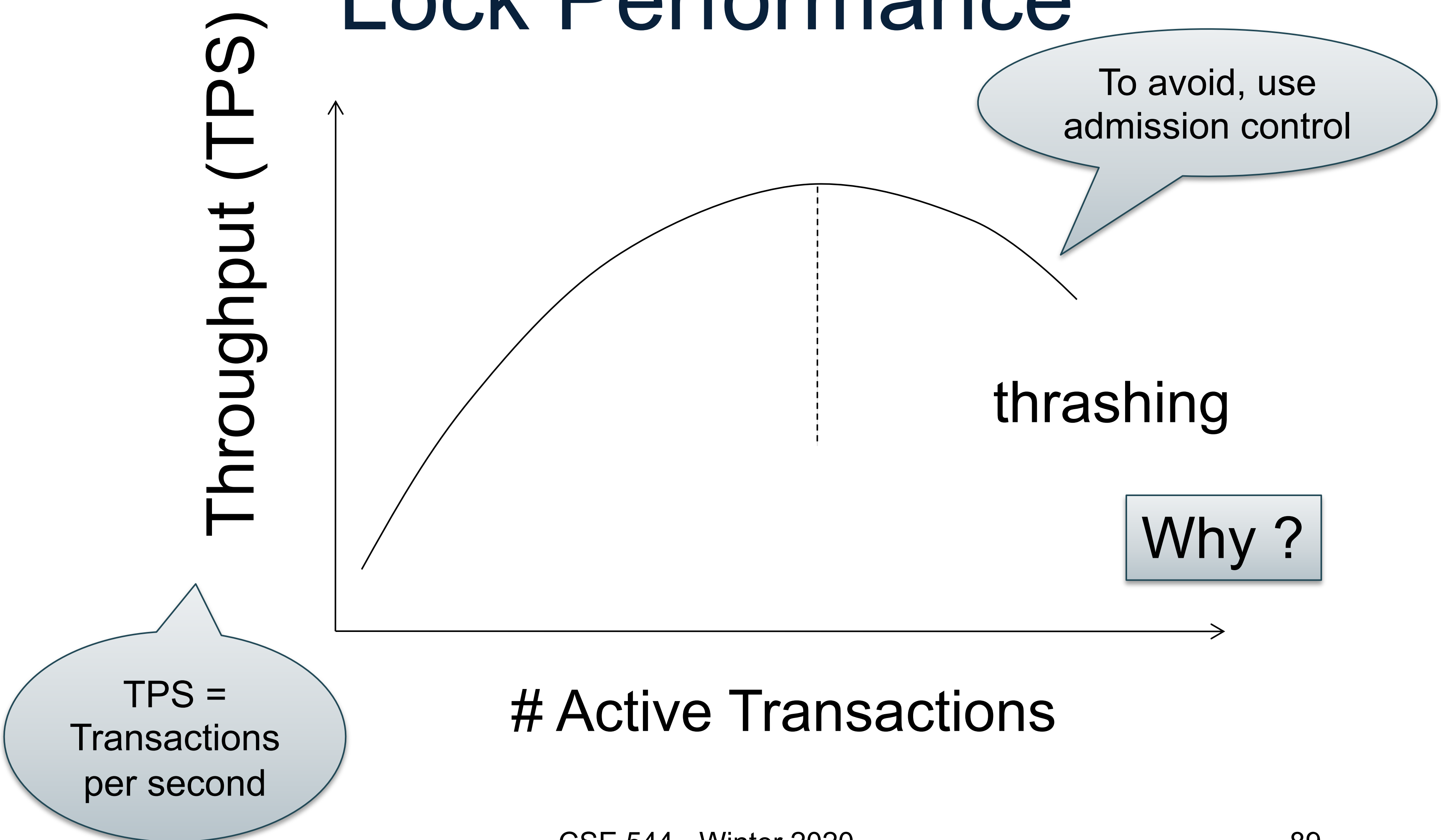
Lock compatibility matrix:

	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



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'	

Is this schedule serializable ?

No: T1 sees a "phantom" product 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'	

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

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'	

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

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

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'	

But this is conflict-serializable!

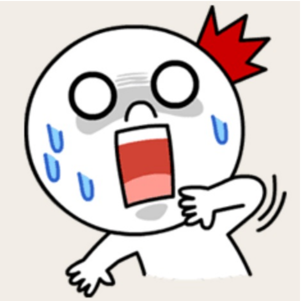
$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$

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

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 !



- Conflict-serializability assumes DB is
- When DB is *dynamic* then c-s is not serializable.

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**:
  – *Conflict serializability* plus *phantom management* implies serializability

Weaker Isolation Levels

- Serializable are expensive to implement

- SQL allows the application to choose a more efficient implementation, which is not always serializable: *weak isolation levels*

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

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);

Never allowed at any level

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

1. Isolation Level: Dirty Reads

Write-Read Conflict

T_1: WRITE(A)

T_1: ABORT

T_2: READ(A)

1. Isolation Level: Dirty Reads

Write-Read Conflict

T_1: A := 20; B := 20;	
T_1: WRITE(A)	
	T_2: READ(A);
	T_2: READ(B);
T_1: WRITE(B)	

Inconsistent read

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

2. Isolation Level: Read Committed

Read-Write Conflict

T_2: READ(A);

T_1: WRITE(A)
COMMIT

T_2: READ(A);

Unrepeatable read

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 may not be serializable
- Default level differs between DBMSs
- Some engines support subset of levels!
- Also, some DBMSs do NOT use locking and different isolation levels can lead to different pbs

Bottom line: Read the doc for your DBMS!

Optimistic concurrency control

Locking vs Optimistic

- Locking prevents unserializable behavior from occurring. It causes transactions to wait for locks
- Optimistic methods assume no unserializable behavior will occur. They abort transactions if it does
- Locking typically better in case of high levels of contention; optimistic better otherwise

Timestamps

- Each transaction receives a unique timestamp TS(T)

Could be:

- The system's clock
- A unique counter, incremented by the scheduler

Timestamps

Main invariant:

The timestamp order defines
the serialization order of the transaction

Will generate a schedule that is view-equivalent to a serial schedule, and recoverable

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$$w_U(X)\ .\ .\ .\ r_T(X)$$
$$r_U(X)\ .\ .\ .\ w_T(X)$$
$$w_U(X)\ .\ .\ .\ w_T(X)$$

Should we allow the OP?

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$$w_U(X) \; . \; . \; . \; r_T(X)$$
$$r_U(X) \; . \; . \; . \; w_T(X)$$
$$w_U(X) \; . \; . \; . \; w_T(X)$$

Should we allow the OP?

$$\boxed{START(U), ...,START(T), ..., w_U(X), ..., r_T(X)}$$

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$w_U(X) \ldots r_T(X)$
$r_U(X) \ldots w_T(X)$
$w_U(X) \ldots w_T(X)$

Should we allow the OP?

OK

START(U), ...,START(T), ..., $w_U(X)$, ..., $r_T(X)$

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$$w_U(X) \ldots r_T(X)$$
$$r_U(X) \ldots w_T(X)$$
$$w_U(X) \ldots w_T(X)$$

Should we allow the OP?

OK

START(U), ...,START(T), ..., $w_U(X)$, ..., $r_T(X)$

START(T), ...,START(U), ..., $w_U(X)$, ..., $r_T(X)$

Main Idea

- Scheduler receives a request, $r_T(X)$ or $w_T(X)$
- Should it allow it to proceed? Wait? Abort?
- Consider these cases:

$$w_U(X) \ldots r_T(X)$$
$$r_U(X) \ldots w_T(X)$$
$$w_U(X) \ldots w_T(X)$$

Should we allow the OP?

OK

START(U), ...,START(T), ..., $w_U(X)$, ..., $r_T(X)$

Too late

START(T), ...,START(U), ..., $w_U(X)$, ..., $r_T(X)$

Timestamps

With each element X, associate

- RT(X) = the highest timestamp of any transaction U that read X
- WT(X) = the highest timestamp of any transaction U that wrote X
- C(X) = the commit bit: true when transaction with highest timestamp that wrote X committed

If element = page, then these are associated with each page X in the buffer pool

Simplified Timestamp-based Scheduling

$w_U(X) \ldots r_T(X)$
$r_U(X) \ldots w_T(X)$
$w_U(X) \ldots w_T(X)$

Only for transactions that do not abort

Otherwise, may result in non-recoverable schedule

Request is $r_T(X)$
?

Request is $w_T(X)$
?

Simplified Timestamp-based Scheduling

$w_U(X) \ldots r_T(X)$
$r_U(X) \ldots w_T(X)$
$w_U(X) \ldots w_T(X)$

Only for transactions that do not abort

Otherwise, may result in non-recoverable schedule

Request is $r_T(X)$
If TS(T) < WT(X) then ROLLBACK
Else READ and update RT(X) to larger of TS(T) or RT(X)

Request is $w_T(X)$
?

Simplified Timestamp-based Scheduling

$w_U(X) \ldots r_T(X)$
$r_U(X) \ldots w_T(X)$
$w_U(X) \ldots w_T(X)$

Only for transactions that do not abort

Otherwise, may result in non-recoverable schedule

Request is $r_T(X)$
If TS(T) < WT(X) then ROLLBACK
Else READ and update RT(X) to larger of TS(T) or RT(X)

Request is $w_T(X)$
If TS(T) < RT(X) then ROLLBACK
Else if TS(T) < WT(X) ignore write & continue (Thomas Write Rule)
Otherwise, WRITE and update WT(X) =TS(T)

Details

Read too late:

- T wants to read X, and TS(T) < WT(X)

START(T) … START(U) … $w_U(X)$ . . . $r_T(X)$

Need to rollback T !

Details

Write too late:

- T wants to write X, and $TS(T) < RT(X)$

START(T) … START(U) … $r_U(X)$ . . . $w_T(X)$

Need to rollback T !

Details

Write too late, but we can still handle it:

- T wants to write X, and
  $TS(T) >= RT(X)$ but $WT(X) > TS(T)$

$\boxed{START(T) \ldots START(V) \ldots w_V(X) \ldots w_T(X)}$

Don't write X at all !
(Thomas' rule)

View-Serializability

- By using Thomas' rule we do not obtain a conflict-serializable schedule
- Instead, we obtain a *view-serializable schedule*
- Will define view-serializability next…

View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$$w_1(X);\ w_2(X);\ w_2(Y);\ w_1(Y);\ w_3(Y);$$

Is this schedule conflict-serializable ?

View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

$$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$$

Is this schedule conflict-serializable ?

No…

View Equivalence

- A serializable schedule need not be conflict serializable, even under the "worst case update" assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Lost write

$\not\Downarrow$

$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

Equivalent, but not conflict-equivalent

View Equivalence

T1	T2	T3
W1(X)		
	W2(X)	
	W2(Y)	
	CO2	
W1(Y)		
CO1		
		W3(Y)
		CO3

Lost

T1	T2	T3
W1(X)		
W1(Y)		
CO1		
	W2(X)	
	W2(Y)	
	CO2	
		W3(Y)
		CO3

Serializable, but not conflict serializable

View-Equivalent Schedules

Two schedules S1, S2 are view-equivalent if:

- If $R_i(X)$ reads an initial value in S1 it also reads an initial value in S2
- If $R_i(X)$ reads the value written by $W_j(X)$ in S1, then it does the same in S2
- If the final value of X in S1 is $W_j(X)$ then so is in S2

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

Connections

- Every conflict-serializable schedule is also view-serializable: CS → VS (why?)
- Every view-serializable schedule is also serializable: VS → S (why?)
- The converse does not necessarily hold

Simplified Timestamp-Based Scheduling

- Fact: the simplified timestamp-based scheduling with Thomas' rule ensures that the schedule is view-serializable

Ensuring Recoverable Schedules

- Use the commit bit C(X) to keep track if the transaction that last wrote X has committed

Ensuring Recoverable Schedules

Read dirty data:

- T wants to read X, and WT(X) < TS(T)
- Seems OK, but…

START(U) … START(T) … w_U(X). . . r_T(X)... ABORT(U)

If C(X)=false, T needs to wait for it to become true

Ensuring Recoverable Schedules

Thomas' rule needs to be revised:

- T wants to write X, and WT(X) > TS(T)
- Seems OK not to write at all, but …

START(T) … START(U)… w_U(X). . . w_T(X)… ABORT(U)

If C(X)=false, T needs to wait for it to become true

Timestamp-based Scheduling

Request is $r_T(X)$
If TS(T) < WT(X) then ROLLBACK
Else If C(X) = false, then WAIT
Else READ and update RT(X) to larger of TS(T) or RT(X)

Request is $w_T(X)$
If TS(T) < RT(X) then ROLLBACK
Else if TS(T) < WT(X)
Then If C(X) = false then WAIT
else IGNORE write (Thomas Write Rule)
Otherwise, WRITE, and update WT(X)=TS(T), C(X)=false

Summary of Timestamp-based Scheduling

- Conflict-serializable
- Recoverable
  - Even avoids cascading aborts
- Does NOT handle phantoms

Multiversion Timestamp

- When transaction T requests r(X) but WT(X) > TS(T), then T must rollback
- Idea: keep multiple versions of X: $X_t, X_{t-1}, X_{t-2}, \ldots$

  $\boxed{TS(X_t) > TS(X_{t-1}) > TS(X_{t-2}) > \ldots}$

- Let T read an older version, with appropriate timestamp

Details

- When $w_T(X)$ occurs,
  create a new version, denoted X_t where $t = TS(T)$

- When $r_T(X)$ occurs,
  find most recent version X_t such that $t < TS(T)$
  Notes:
  - $WT(X_t) = t$ and it never changes
  - $RT(X_t)$ must still be maintained to check legality of writes

- Can delete X_t if we have a later version X_{t1} and all active transactions T have $TS(T) > t1$

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{18}

$R_6(X)$ -- what happens?

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{18}

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{18}

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens?

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens?
$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens? Return X_{14}

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens? Return X_{14}

$W_5(X)$ – what happens?

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens? Return X_{14}
$W_5(X)$ – what happens? ABORT

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3
$W_{14}(X)$ – what happens?
$R_{15}(X)$ – what happens? Return X_{14}
$W_5(X)$ – what happens? ABORT

When can we delete X_3?

Example (in class)

TS(T)=6

X_3 X_9 X_{12} X_{14} X_{18}

$R_6(X)$ -- what happens? Return X_3

$W_{14}(X)$ – what happens?

$R_{15}(X)$ – what happens? Return X_{14}

$W_5(X)$ – what happens? ABORT

When can we delete X_3? When min $TS(T) \geq 9$

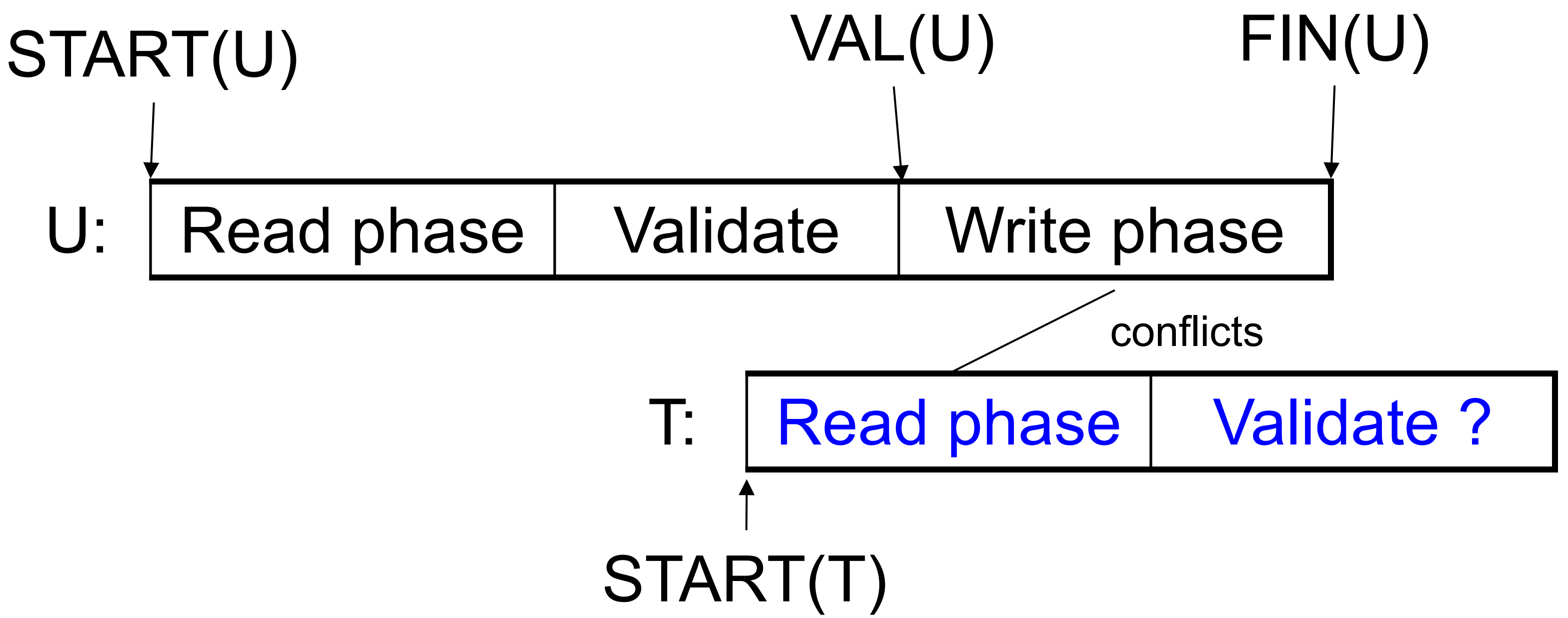
Concurrency Control by Validation

Even more optimistic than timestamp validation

- Each transaction T defines a *read set* RS(T) and a *write set* WS(T)
- Each transaction proceeds in three phases:
  - Read all elements in RS(T). Time = START(T)
  - Validate (may need to rollback). Time = VAL(T)
  - Write all elements in WS(T). Time = FIN(T)

Main invariant: the serialization order is VAL(T)

Avoid $w_U(X)$ - $r_T(X)$ Conflicts

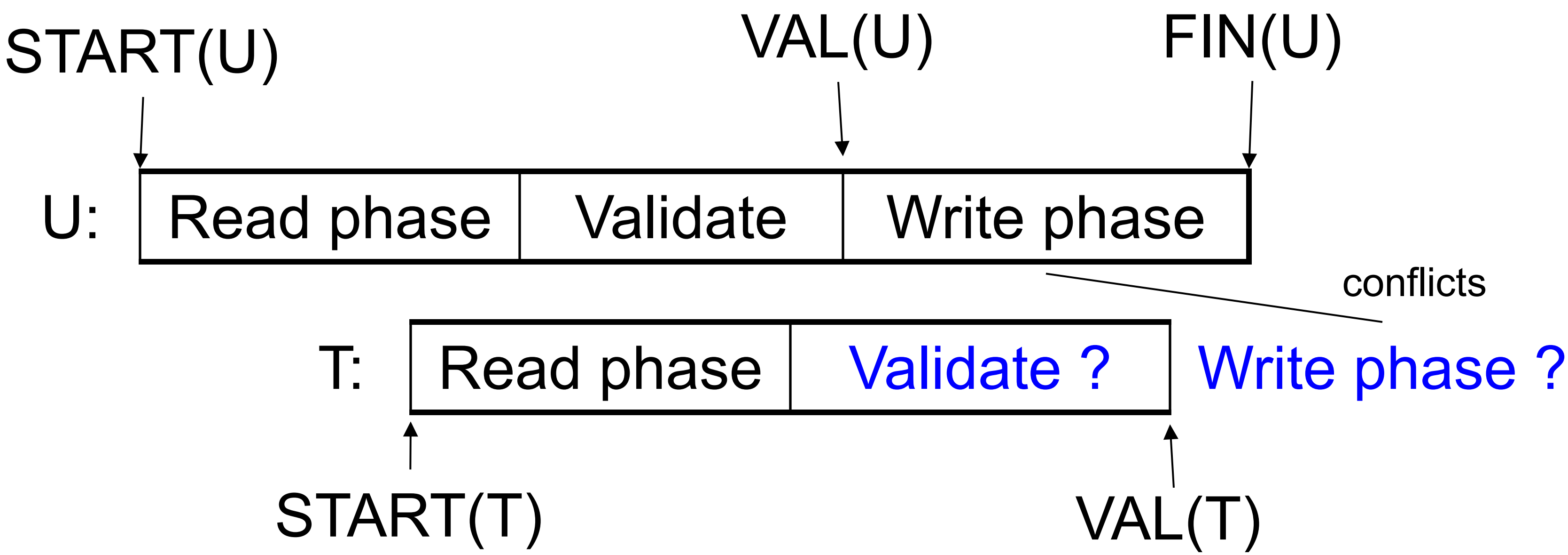


IF RS(T) $\cap$ WS(U) and FIN(U) > START(T)

(U has validated and U has not finished before T begun)

Then ROLLBACK(T)

Avoid $w_U(X)$ - $w_T(X)$ Conflicts



IF WS(T) ∩ WS(U) and FIN(U) > VAL(T)

(U has validated and U has not finished before T validates)

Then ROLLBACK(T)

Snapshot Isolation (SI)

A variant of multiversion/validation

- Very efficient, and very popular
- Oracle, PostgreSQL, SQL Server 2005

Warning: not serializable

- Earlier versions of postgres implemented SI for the SERIALIZABLE isolation level
- Extension of SI to serializable has been implemented recently
- Will discuss only the standard SI (non-serializable)

Snapshot Isolation Rules

- Each transactions receives a timestamp TS(T)
- Transaction T sees snapshot at time TS(T) of the database
- When T commits, updated pages are written to disk
- Write/write conflicts resolved by “first committer wins” rule
  - Loser gets aborted
- Read/write conflicts are ignored

Snapshot Isolation (Details)

- Multiversion concurrency control:
  - Versions of X: X_{t1}, X_{t2}, X_{t3}, . . .
- When T reads X, return $X_{TS(T)}$.
- When T writes X: if other transaction updated X, abort
  - Not faithful to "first committer" rule, because the other transaction U might have committed after T. But once we abort T, U becomes the first committer ☺

What Works and What Not

- No dirty reads (Why ?)
- No inconsistent reads (Why ?)
  – A: Each transaction reads a consistent snapshot
- No lost updates (“first committer wins”)
- Moreover: no reads are ever delayed
- However: read-write conflicts not caught ! “Write skew”

Write Skew

Invariant: $X + Y \geq 0$

```
T1:
  READ(X);
  if X >= 50
      then Y = -50; WRITE(Y)
  COMMIT
```

```
T2:
  READ(Y);
  if Y >= 50
      then X = -50; WRITE(X)
  COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

Starting with X=50,Y=50, we end with X=-50, Y=-50.
Non-serializable !!!

Discussions

- Snapshot isolation (SI) is like repeatable reads but also avoids some (not all) phantoms

- If DBMS runs SI and the app needs serializable:
  - use dummy writes for all reads to create write-write conflicts… but that is confusing for developers

- Extension of SI to make it serializable is implemented in postgres

Final Thoughts on Transactions

- Benchmarks: TPC/C; typical throughput: x100’s TXN/second
- New trend: multicores
  - Current technology can scale to x10’s of cores, but not beyond!
  - Major bottleneck: latches that serialize the cores
- New trend: distributed TXN
  - NoSQL: give up serialization
  - Serializable: very difficult e.g.Spanner w/ Paxos

Final/Final Thoughts

- Final is canceled! We will reweight
- Please finish homework 5
- Please submit final project report