

CSE 444: Database Internals

Lectures 14 Transactions: Locking

CSE 444 - Spring 2016

1

Announcements

- Lab 2 is due tonight
- Lab 2 quiz is on Wednesday in class
 - Same format as lab 1 quiz
- Lab 3 is available
 - Fastest way to do lab 3 is to do it very slowly
- Hw5 is due on Friday

CSE 444 - Spring 2016

2

Review of Schedules

Serializability

- Serial
- Serializable
- Conflict serializable
- View serializable

Recoverability

- Recoverable
- Avoids cascading aborts

CSE 444 - Spring 2016

3

Scheduler

- The scheduler:
- Module that schedules the transaction's actions, ensuring serializability
- Two main approaches
- **Pessimistic**: locks
- **Optimistic**: timestamps, multi-version, validation

CSE 444 - Spring 2016

4

Pessimistic 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)

CSE 444 - Spring 2016

5

Notation

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

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

CSE 444 - Spring 2016

6

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)	

CSE 444 - Spring 2016

7

Example

T1	T2
L ₁ (A); READ(A, t)	
t := t+100	
WRITE(A, t); U ₁ (A); L ₁ (B)	
	L ₂ (A); READ(A, s)
	s := s*2
	WRITE(A, s); U ₂ (A);
	L ₂ (B); DENIED...
READ(B, t)	
t := t+100	
WRITE(B, t); U ₁ (B);	
	...GRANTED; READ(B, s)
	s := s*2
	WRITE(B, s); U ₂ (B);

Scheduler has ensured a conflict-serializable schedule

8

But...

T1	T2
L ₁ (A); READ(A, t)	
t := t+100	
WRITE(A, t); U ₁ (A);	
	L ₂ (A); READ(A, s)
	s := s*2
	WRITE(A, s); U ₂ (A);
	L ₂ (B); READ(B, s)
	s := s*2
	WRITE(B, s); U ₂ (B);
L ₁ (B); READ(B, t)	
t := t+100	
WRITE(B, t); 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
- This ensures conflict serializability! (will prove this shortly)

CSE 444 - Spring 2016

10

Example: 2PL transactions

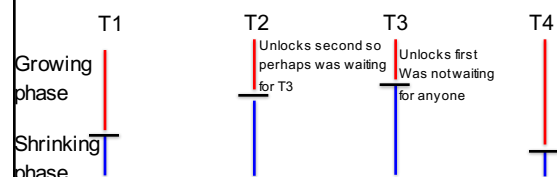
T1	T2
L ₁ (A); L ₁ (B); READ(A, t)	
t := t+100	
WRITE(A, t); U ₁ (A)	
	L ₂ (A); READ(A, s)
	s := s*2
	WRITE(A, s);
	L ₂ (B); DENIED...
READ(B, t)	
t := t+100	
WRITE(B, t); U ₁ (B);	
	...GRANTED; READ(B, s)
	s := s*2
	WRITE(B, s); U ₂ (A); U ₂ (B);

Now it is conflict-serializable

CSE 444 - Spring 2016

11

Example with Multiple Transactions



Equivalent to each transaction executing entirely the moment it enters shrinking phase

CSE 444 - Spring 2016

12

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

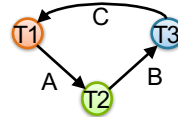
CSE 444 - Spring 2016

13

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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



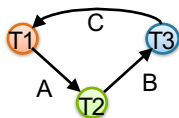
CSE 444 - Spring 2016

14

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:

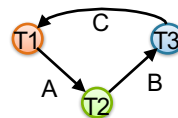
CSE 444 - Spring 2016

15

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?

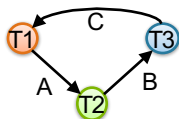
CSE 444 - Spring 2016

16

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?

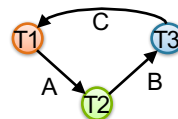
CSE 444 - Spring 2016

17

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

CSE 444 - Spring 2016

A New Problem: Non-recoverable Schedule

T1	T2
$L_1(A)$; $L_1(B)$; READ(A, t)	
t := t+100	
WRITE(A, t); $U_1(A)$	
	$L_2(A)$; READ(A,s)
	s := s*2
	WRITE(A,s);
	$L_2(B)$; DENIED...
READ(B, t)	
t := t+100	
WRITE(B,t); $U_1(B)$;	
	...GRANTED; READ(B,s)
	s := s*2
	WRITE(B,s); $U_2(A)$; $U_2(B)$;
	Commit

Abort

CSE 444 - Spring 2016

19

Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed; release happens at the time of COMMIT or ROLLBACK
- Schedule is recoverable
- Schedule avoids cascading aborts
- Schedule is strict: read book

CSE 444 - Spring 2016

20

Strict 2PL

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A);	
	$L_2(A)$; DENIED...
$L_1(B)$; 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

21

Summary of Strict 2PL

- Ensures serializability, recoverability, and avoids cascading aborts
- Issues: implementation, lock modes, granularity, deadlocks, performance

CSE 444 - Spring 2016

22

The Locking Scheduler

Task 1: -- act on behalf of the transaction

- Add lock/unlock requests to transactions
- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- On COMMIT/ROLLBACK release all locks
- Ensures Strict 2PL !

CSE 444 - Spring 2016

23

The Locking Scheduler

Task 2: -- act on behalf of the system

Execute the locks accordingly

- Lock table: a big, critical data structure in a DBMS !
- When a lock is requested, check the lock table
 - Grant, or add the transaction to the element's wait list
- When a lock is released, re-activate a transaction from its wait list
- When a transaction aborts, release all its locks
- Check for deadlocks occasionally

CSE 444 - Spring 2016

24

Lock Modes

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

Lock compatibility matrix:

	None	S	X
None	OK	OK	OK
S	OK	OK	Conflict
X	OK	Conflict	Conflict

CSE 444 - Spring 2016

25

Lock Granularity

- **Fine granularity locking** (e.g., tuples)
 - High concurrency
 - High overhead in managing locks
- **Coarse grain locking** (e.g., tables, predicate locks)
 - Many false conflicts
 - Less overhead in managing locks
- **Alternative techniques**
 - Hierarchical locking (and intentional locks) [commercial DBMSs]
 - Lock escalation

CSE 444 - Spring 2016

26

Hierarchical Locking

- To enable both coarse- and fine-grained locking
- Consider database as a hierarchy
 - Relations are largest lockable elements
 - Relations consist of blocks
 - Blocks contain tuples
- To place a lock on an element, start at the top
 - If at element to lock, get an S or X lock on it
 - If want to lock an element deeper in the hierarchy
 - Leave an **intentional** lock: IS or IX

CSE 444 - Spring 2016

27

Hierarchical Locking

	IS	IX	S	SIX	X
IS	y	y	y	y	n
IX	y	y	n	n	n
S	y	n	y	n	n
SIX	y	n	n	n	n
X	n	n	n	n	n

Table 2: Compatibility Matrix for Regular and Intention Locks

To Get	Must Have on all Ancestors
IS or S	IS or IX
IX, SIX, or X	IX or SIX

Table 3: Hierarchical Locking Rules

From Franklin97. See readings posted on course website.

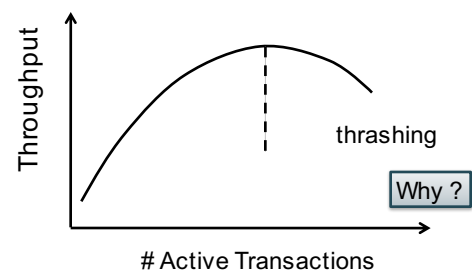
Deadlocks

- **Cycle in the wait-for graph:**
 - T1 waits for T2
 - T2 waits for T3
 - T3 waits for T1
- **Deadlock detection**
 - Timeouts
 - Wait-for graph
- **Deadlock avoidance**
 - Acquire locks in pre-defined order
 - Acquire all locks at once before starting

CSE 444 - Spring 2016

29

Lock Performance



CSE 444 - Spring 2016

30

The Tree Protocol

- An alternative to 2PL, for tree structures
- E.g. B-trees (the indexes of choice in databases)
- Because
 - Indexes are hot spots!
 - 2PL would lead to great lock contention

CSE 444 - Spring 2016

31

The Tree Protocol

Rules:

- The first lock may be any node of the tree
- Subsequently, a lock on a node A may only be acquired if the transaction holds a lock on its parent B
- Nodes can be unlocked in any order (no 2PL necessary)
- "Crabbing"
 - First lock parent then lock child
 - Keep parent locked only if may need to update it
 - Release lock on parent if child is not full
- The tree protocol is NOT 2PL, yet ensures conflict-serializability!

CSE 444 - Spring 2016

32

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

CSE 444 - Spring 2016

33

Phantom Problem

T1	T2
SELECT *	
FROM Product	
WHERE color='blue'	
	INSERT INTO Product(name, color)
	VALUES ('gizmo','blue')
SELECT *	
FROM Product	
WHERE color='blue'	

Is this schedule serializable?

CSE 444 - Spring 2016

34

Phantom Problem

T1	T2
SELECT *	
FROM Product	
WHERE color='blue'	
	INSERT INTO Product(name, color)
	VALUES ('gizmo','blue')
SELECT *	
FROM Product	
WHERE color='blue'	

Suppose there are two blue products, X1, X2:

R1(X1), R1(X2), W2(X3), R1(X1), R1(X2), R1(X3)

CSE 444 - Spring 2016

35

Phantom Problem

T1	T2
SELECT *	
FROM Product	
WHERE color='blue'	
	INSERT INTO Product(name, color)
	VALUES ('gizmo','blue')
SELECT *	
FROM Product	
WHERE color='blue'	

Suppose there are two blue products, X1, X2:

R1(X1), R1(X2), W2(X3), R1(X1), R1(X2), R1(X3)

This is conflict serializable! What's wrong??

36

Phantom Problem

T1

T2

```
SELECT *
FROM Product
WHERE color='blue'
```

```
INSERT INTO Product(name, color)
VALUES ('gizmo', 'blue')
```

```
SELECT *
FROM Product
WHERE color='blue'
```

Suppose there are two blue products, X1, X2:

```
R1(X1), R1(X2), W2(X3), R1(X1), R1(X2), R1(X3)
```

Not serializable due to **phantoms**

37

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 !

CSE 444 - Spring 2016

38

Phantom Problem

- In a **static** database:
 - Conflict serializability implies serializability
- In a **dynamic** database, this may fail due to phantoms
- Strict 2PL guarantees conflict serializability, but not serializability

CSE 444 - Spring 2016

39

Dealing With Phantoms

- Lock the entire table, or
- 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 !

CSE 444 - Spring 2016

40

Isolation Levels in SQL

- “Dirty reads”
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
- “Committed reads”
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
- “Repeatable reads”
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
- Serializable transactions
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

ACID

CSE 444 - Spring 2016

41

1. Isolation Level: Dirty Reads

- “Long duration” **WRITE** locks
 - Strict 2PL
- No **READ** locks
 - Read-only transactions are never delayed

Possible pbs: dirty and inconsistent reads

CSE 444 - Spring 2016

42

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

CSE 444 - Spring 2016

43

3. Isolation Level: Repeatable Read

- “Long duration” WRITE locks
 - Strict 2PL
- “Long duration” READ locks
 - Strict 2PL

This is not serializable yet !!!

Why ?

CSE 444 - Spring 2016

44

4. Isolation Level Serializable

- “Long duration” WRITE locks
 - Strict 2PL
- “Long duration” READ locks
 - Strict 2PL
- Deals with phantoms too

CSE 444 - Spring 2016

45

READ-ONLY Transactions

```
Client 1: START TRANSACTION
INSERT INTO SmallProduct(name, price)
VALUES ('P1', 0.99)
SELECT pname, price
FROM Product
WHERE price <= 0.99

DELETE FROM Product
WHERE price <= 0.99
COMMIT

Client 2: SET TRANSACTION READ ONLY
START TRANSACTION
SELECT count(*)
FROM Product

SELECT count(*)
FROM SmallProduct
COMMIT
```

May improve
performance

46