CIS 560 – Database System Concepts

Lecture 21

Concurrency Control

October 23, 2013

Credits for slides: Chang, Ullman, Whitehead.

Outline

Last:

- Serial and serializable schedules 18.1
- Conflict serializability 18.2
- Locks 18.3

Today:

- Locks 18.3
- Timestamps 18.8

Next:

- Indexes and B-trees 14.1-14.2

2

Review

- Schedule
- Serial schedule
- Serializable schedule
- Conflict serializable schedule
- Precedence graph
- Locks
- Two phase locking (2PL)

3

What about Aborts?

- 2PL enforces conflict-serializable schedules
- But what if a transaction releases its locks and then aborts?
- Serializable schedule definition only considers transactions that commit
  - Relies on assumptions that aborted transactions can be undone completely

4

Example with Abort

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	

5

Recoverable Schedules

- A schedule is *recoverable* if whenever a transaction T commits, all transactions who have written elements read by T have already committed.

6

Is this schedule recoverable?

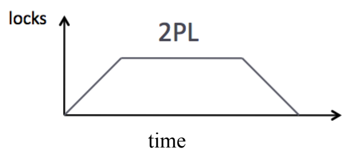
T1	T2
L_1(A); L_1(B); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s);
	L_2(B); **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); U_2(A); U_2(B);
Commit	
	Commit

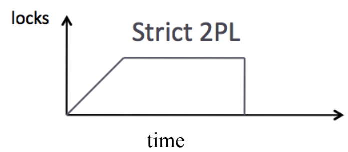
Cascading Aborts

- If a transaction T aborts, then we need to abort any other transaction T' that has read an element written by T.
- A schedule is said to *avoid cascading aborts* if whenever a transaction read an element, the transaction that has last written it has already committed.

Strict 2PL

- Strict 2PL: All locks held by a transaction are released when the transaction is completed.
- Ensures that schedules are recoverable
  - Transactions commit only after all transactions whose changes they read also commit.
- Avoids cascading rollbacks.





9

Deadlock

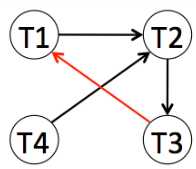
- Transaction T_1 waits for a lock held by T_2;
- But T_2 waits for a lock held by T_3;
- While T_3 waits for . . . .
- . . .
- . . .and T_{73} waits for a lock held by T_1!
- Could be prevented/detected (see textbook 19.2);

10

Deadlock Example

T1	T2	T3	T4
L(A)			
R(A)			
	L(B)		
	W(B)		
L(B)			
		L(C)	
		R(C)	
	L(C)		
			L(B)
		L(A)	

Waits-for graph



Deadlock!

Commercial systems use *wait-for graphs* to detect deadlocks (then abort at least one transaction).

11

Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)
- U = update lock
  - Initially like S
  - Later may be upgraded to X
- I = increment lock (for A := A + something)
  - Increment operations commute

Recommended reading 18.4!

12

Lock Granularity

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

Recommended reading 18.6!

13

The Locking Scheduler

Task 1:
Add lock/unlock requests to transactions

- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- Ensure Strict 2PL!

14

The Locking Scheduler

Task 2:
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

Recommended reading 18.5!

15

Phantom Problem [18.6.3]

- 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

16

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)

Conflict serializable! But not serializable due to phantoms

Dealing with Phantoms

- 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
- Expensive ways of dealing with phantoms:
  - Lock the entire table, or
  - Lock the index entry for 'blue' (if index is available)

Serializable transactions are very expensive

8

Concurrency Control Mechanisms

- Pessimistic:
  - *Intuition: assume that things will go wrong unless transactions are prevented in advance from engaging in nonserializable behavior.*
    - Locks
- Optimistic
  - *Intuition: assume that no unserializable behavior will occur and only fix things up when a violation is apparent!*
    - Timestamp based: basic, multiversion
    - Validation
    - Snapshot isolation: a variant of both

19

Timestamps

- Each transaction receives a unique timestamp TS(T), when the transaction first notifies the scheduler that it is beginning.
- Could be:
  - The system's clock
  - A unique counter, incremented by the scheduler

20

Timestamps

Main invariant:

The timestamp order defines
the serialization order of the transactions.

21

Main Idea

- For any two conflicting actions, ensure that their order is the serialized order:

In each of these cases

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

Possible conflicts

When T requests $r_T(X)$ or $w_T(X)$,
need to check TS(U) <= TS(T)

22

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 1 element = 1 page, then these are associated with each page X in the buffer pool

23

Ensuring Recoverable Schedules

- Recall the definition: if a transaction reads an element, then the transaction that wrote it must have already committed
- Use the commit bit C(X) to keep track if the transaction that last wrote X has committed

24

Simplified Timestamp-based Scheduling

Note: simple version that ignores the commit bit

- Only for transactions that do not abort
- Otherwise, may result in non-recoverable schedule

Transaction wants to read element X
If TS(T) < WT(X) then ROLLBACK
Else READ and update RT(X) to larger of TS(T) or RT(X)

Transaction wants to write element 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)

25

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!

26

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!

27

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)

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

Don’t write X at all !
(Thomas’ rule)

28

T1	T2	T3	A	B	C
200	150	175	RT =0 WT=0	RT =0 WT=0	RT =0 WT=0
$r_1(B)$					
	$r_2(A)$				
		$r_3(C)$			
$w_1(B)$					
$w_1(A)$					
	$w_2(C)$				
		$w_3(A)$			

29

T1	T2	T3	A	B	C
200	150	175	RT =0 WT=0	RT =0 WT=0	RT =0 WT=0
$r_1(B)$				RT=200	
	$r_2(A)$		RT=100		
		$r_3(C)$			RT=175
$w_1(B)$				WT=200	
$w_1(A)$			WT=200		
	$w_2(C)$ **rollback**				
		$w_3(A)$			

30