CS411
Database Systems

14: Concurrency Control

Kazuhiro Minami

Approaches for Concurrency Control

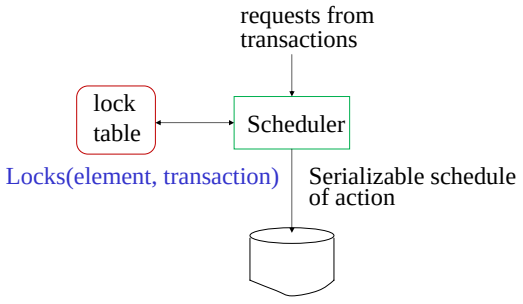
- Locking
  - Maintain a lock on each database element
- Timestamping
  - Assign a "timestamp" to each transaction and database element
- Validation
  - Maintain a record of what active transactions are doing

Locks are the basis of most protocols to guarantee serializability.

- Prevent orders of actions that lead to an unserializable schedule using locks
- Maintain a lock on each database element
- Transactions must obtain a lock on a database element if they want to perform any operation on that element

Locks

- A scheduler uses a lock table to guide decisions



Requirements for the use of locks

- Consistency of transactions
  - A transaction can only read or write an element if it previously requested a lock on that element and hasn't yet released the lock
  - If a transaction locks an element, it must later unlock that element
- Legality of schedulers
  - No two transactions may have locked the same element without one having first released the lock

Notation for locks

- $l_i(X)$: Transaction T_i requests a lock on database element X
- $u_i(X)$: Transaction T_i releases its lock on database element X

Example 1

- A legal, but not serializable schedule

T_1	T_2	A	B
$l_1(A)$; $r_1(A)$;		25	25
A := A+100;			
$w_1(A)$;$u_1(A)$;		125	
	$l_2(A)$;$r_2(A)$;		
	A := A*2;		
	$w_2(A)$; $u_2(A)$;	250	
	$l_2(B)$; $r_2(B)$;		
	B := B*2;		50
	$w_2(B)$; $u_2(B)$;		
$l_1(B)$; $r_1(B)$;			150
B := B+100;			
$w_1(B)$;$u_1(B)$;			

Example 2

- T_1 and T_2 lock B before releasing the lock on A

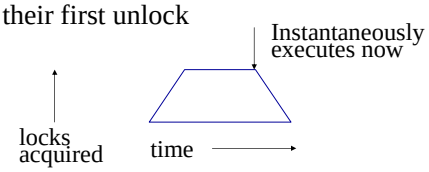
T_1	T_2	A	B
$l_1(A)$; $r_1(A)$;		25	25
A := A+100;			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$;$r_2(A)$;		
	A := A*2;		
	$w_2(A)$;	250	
	$l_2(B)$; Denied		
$r_1(B)$; B := B+100;			125
$w_1(B)$; $u_1(B)$;			
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		250
	B := B*2;		
	$w_2(B)$; $u_2(B)$;		

2-Phase Locking (2PL): no new locks once you've given one up

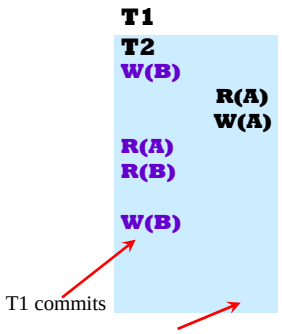
- In every transaction, all lock requests precede all unlock requests
- Guaranteed that a legal schedule of consistent transactions is conflict-serializable

Why Two-Phase Locking Works

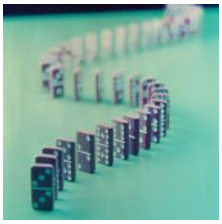
- Each two-phase-locked transaction may be thought to execute in its entirety at the instant it issues its first unlock request
- The conflict-equivalent serial schedule for a schedule S of 2PL transactions is the one in which the transactions are ordered in the same order as their first unlock



2PL doesn't solve every potential problem.



We should never have let T1 commit.



Cascading rollback

How do we deal with this?

Commit trans T only after all transactions that wrote data that T read have committed

Or only let a transaction read an item after the transaction that last wrote this item has committed

Strict 2PL: 2PL + a transaction releases its locks only after it has committed.

How does Strict 2PL prevent cascading rollback?

Concurrency Control by Timestamps

Timestamping for Concurrency Control

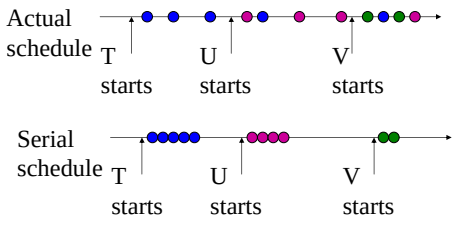
- Assign a "timestamp" to each transaction
- Record the timestamps of transactions that last read and write each database element

Timestamps

- Scheduler assigns each transaction T a timestamp of its starting time TS(T)
- Each database element X is associated with
  - RT(X): read time, the highest timestamp of a transaction that has read X
  - WT(X): write time, the highest timestamp of a transaction that has written X
  - c(X): the commit bit of X, which is true iff the most recent transaction to write X has already committed

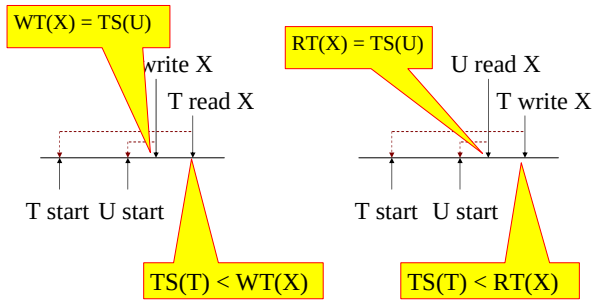
Assumed Serial Schedule

- Conflict serializable schedule that is equivalent to a serial schedule in which the timestamp order of transactions is the order to execute them



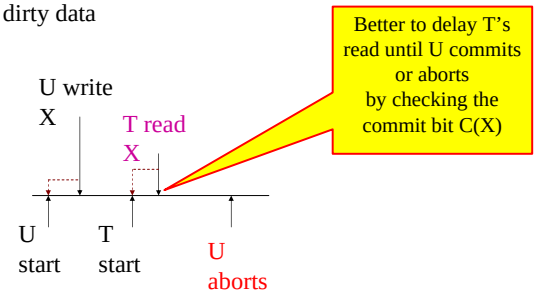
Detecting Physically Unrealizable Behaviors Using Timestamps

1. Read too late
2. Write too late



Problem with Dirty Data

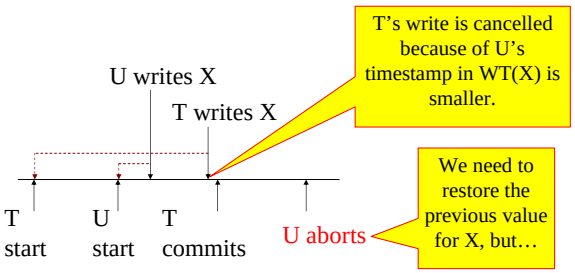
The commit bit is designed to help deal with the problem of dirty data



Another Problem with Dirty Data

- Thomas write rule: T's write can be skipped if

TS(T) < WT(X)



Scheduler's Response to a T's request for Read(X)/Write(X)

1. Grant the request
2. Abort and restart (roll back) T with a new timestamp
3. Delay T and later decide whether to abort T or to grant the request

Rules for Timestamp-Based Scheduling

Request $r_T(X)$:

1. If TS(T) >= WT(X), the read is physically realizable
   - I. If C(X) is true, grant the request. If TS(T) > RT(X), set RT(X) := TS(T); otherwise do not change RT(X)
   - II. If C(X) is false, delay T until C(X) becomes true or the transaction that wrote X aborts
2. If TS(T) < WT(X), the read is physically unrealizable. Rollback T; abort T and restart it with a new, larger timestamp

Rules for Timestamp-Based Scheduling

Request $w_T(X)$:

1. If TS(T) >= RT(X) and TS(T) >= WT(X), the write is physically realizable and must be performed
   1. Write the new value for X
   2. Set WT(X) := TS(T), and
   3. Set C(X) := false
2. If TS(T) >= RT(X), but TS(T) < WT(X), then the write is physically realizable, but there is already a later value in X. If C(X) is true, then ignore the write by T. If C(X) is false, delay T
3. If TS(T) < RT(X), then the write is physically unrealizable

Example

Transactions			Database elements		
T_1	T_2	T_3	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=150		
	Writing too late!				RT=175
$w_1(B)$				WT=200	
$w_1(A)$			WT=200		
	$w_2(C)$				
	Abort;				
		$w_3(A)$			WT=175

Multiversion Timestamps

- Maintain old versions of database elements
- Allow read $r_T(X)$ that would cause T to abort to proceed by reading the version of X

T_1	T_2	T_3	T_4	A
150	200	175	225	RT=0 WT=0
$r_1(A)$; $w_1(A)$;				RT=150 WT=150
	$r_2(A)$; $w_2(A)$;			RT=200 WT=200
		$r_3(A)$; Abort;		
			$r_4(A)$;	RT=225

Multiversion Timestamping Scheduler

- When $w_T(X)$ occurs, if it's legal, a new version of X, X_t where $t = TS(T)$, is created.
- When $r_T(X)$ occurs, find the version X_t of X s.t. $t <= TS(T)$, but no Xt' with $t < t' <= TS(T)$
- Write times are associated with versions of an element, and they never change
- Read times are also associated with versions
- When X_t has a write time t s.t. no active transaction has a timestamp less than t, we can delete any version of X previous to X_t

Example

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225			
$r_1(A)$; $w_1(A)$;				RT=150	WT=150	
	$r_2(A)$; $w_2(A)$;				RT=200	WT=200
		$r_3(A)$;			RT=175	
			$r_4(A)$;			RT=225

Timestamps vs. Locks

Time stamps

- Superior if
  - most transactions are read-only
  - rare that concurrent transactions will read or write the same element
- In high-conflict situations, rollback will be frequent, introducing more delays than a locking system

Locks

- Superior in high-conflict situations
- Frequently delay transactions as they wait for locks

Concurrency Control by Validation

Concurrency Control by Validation

- Another type of optimistic concurrency control
- Maintains a record of what active transactions are doing
- Just before a transaction starts to write, it goes through a "validation phase"
- If a there is a risk of physically unrealizable behavior, the transaction is rolled back

Validation-based Scheduler

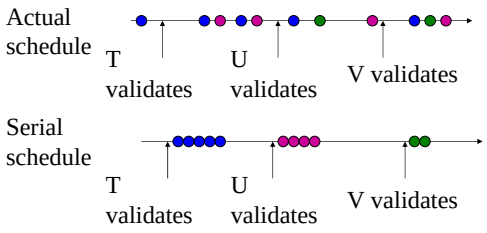
- Keep track of each transaction T's
  - Read set RS(T): the set of elements T read
  - Write set WS(T): the set of elements T write
- Execute transactions in three phases:
  1. *Read*. T reads all the elements in RS(T)
  2. *Validate*. Validate T by comparing its RS(T) an WS(T) with those in other transactions. If the validation fails, T is rolled back
  3. *Write*. T writes its values for the elements in WS(T)

Scheduler Maintains Information Sets

- START: the set of transactions that have started, but not yet completed validation. For each T, maintain (T, START(T))
- VAL: the set of transactions that have been validated, but not yet finished. For each T, maintain (T, START(T), VAL(T))
- FIN: the set of transaction that have completed. For each T, maintain (T, START(T), VAL(T), FIN(T))

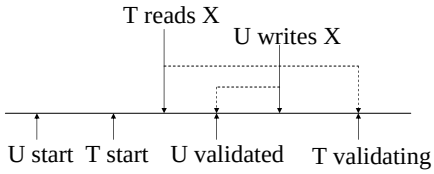
Assumed Serial Schedule for Validation

- We may think of each transaction that successfully validates as executing at the moment that it validates



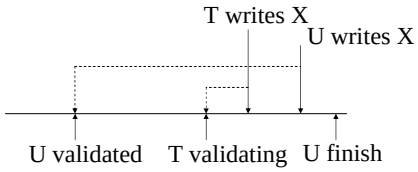
Potential Violation of the Serial Order

- Transactions T and U such that
  - U has validated
  - START(T) < FIN(U)
  - RS(T) ∩ WS(U) is not empty



Another Potential Violation of the Serial Order

- Two transactions T and U such that
  - U is in VAL
  - VAL(T) < FIN(U)
  - WS(T) ∩ WS(U) is not empty

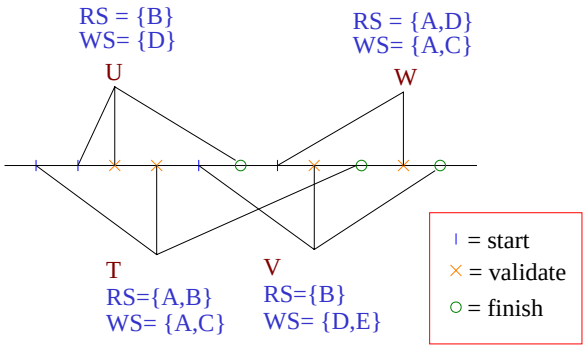


Validation Rules

To validate a transaction T,

1. Check that RS(T) ∩ WS(U) is an empty set for any *validated* U and START(T) < FIN(U)
2. Check that WS(T) ∩ WS(U) is an empty set for any *validated* U that did not finish before T validated, i.e., if VAL(T) < FIN(U)

Example



Comparison of Three Mechanisms

- Storage utilization
  - Locks: space in the lock table is proportional to the number of database elements locked
  - Timestamps: Read and write times for recently accessed database elements
  - Validation: timestamps and read/write sets for each active transaction, plus a few more transactions that finished after some currently active transaction began

Comparison of Three Mechanisms

- Delay
  - Locking delays transactions but avoids rollbacks, even when interaction is high
  - If interference is low, neither timestamps nor validation will cause many transactions
  - When a rollback is necessary, timestamps catch some problems earlier than validation

Summary