Chapter 11: Concurrency Control

Zhaonian Zou

Massive Data Computing Research Center
School of Computer Science and Technology
Harbin Institute of Technology, China
Email: znzou@hit.edu.cn

Spring 2019

Outline[1]

[1] Updated on May 8, 2019

11.1 Overview of Concurrency Control

11.2 Schedules

Schedules (调度)

A schedule is a sequence of the important actions taken by one or more transactions

Example (Schedules)

T_1	T_2
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)

Serial Schedules (串行调度)

A schedule is serial (串行的) if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on

Example (Serial Schedules)

T_1	T_2
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)

T_1	T_2
	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)	

Nonserial Schedules (非串行调度)

Schedules other than serial schedules are called nonserial schedules

Example (Nonserial Schedules)

T_1	T_2
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)

T_1	T_2
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)

The Correctness Principle

Every transaction, if executed in isolation, will transform any consistent state of the database to another consistent state

Example (The Correctness Principle)

Assume that the only consistency constraint is that $A = B$

T_1	A	B
	25	25
READ(A, t)		
t := t + 100		
WRITE(A, t)	125	
READ(B, t)		
t := t + 100		
WRITE(B, t)		125

T_2	A	B
	25	25
READ(A, s)		
s := s * 2		
WRITE(A, s)	50	
READ(B, s)		
s = s * 2		
WRITE(B, s)		50

The Correctness Principle (Cont'd)

Every serial schedule will preserve consistency of database state

Example (Correct Serial Schedules)

T_1	T_2	A	B
		25	25
`READ(A, t)`			
`t := t + 100`			
`WRITE(A, t)`		125	
`READ(B, t)`			
`t := t + 100`			
`WRITE(B, t)`			125
	`READ(A, s)`		
	`s := s * 2`		
	`WRITE(A, s)`	250	
	`READ(B, s)`		
	`s := s * 2`		
	`WRITE(B, s)`		250

The Correctness Principle (Cont'd)

In general, the final state of the database is not expected to be independent of the order of transactions

Example (Correct Serial Schedules)

T_1	T_2	A	B
		25	25
	`READ(A, s)`		
	`s := s * 2`		
	`WRITE(A, s)`	50	
	`READ(B, s)`		
	`s := s * 2`		
	`WRITE(B, s)`		50
`READ(A, t)`			
`t := t + 100`			
`WRITE(A, t)`		150	
`READ(B, t)`			
`t := t + 100`			
`WRITE(B, t)`			150

The Correctness Principle (Cont'd)

A nonserial schedule may not preserve consistency and cause anomalies

Example (Incorrect Nonserial Schedules)

T_1	T_2	A	B
		25	25
`READ(A, t)`			
`t := t + 100`			
`WRITE(A, t)`		125	
	`READ(A, s)`		
	`s := s * 2`		
	`WRITE(A, s)`	250	
	`READ(B, s)`		
	`s := s * 2`		
	`WRITE(B, s)`		50
`READ(B, t)`			
`t := t + 100`			
`WRITE(B, t)`			150

Anomalous Behavior 1: Reading Uncommitted Data

The value of A written by T_1 is read by T_2 before T_1 commits

Example (Reading Uncommitted Data)

T_1	T_2	A	B
		25	25
`READ(A, t)`			
`t := t + 100`			
`WRITE(A, t)`		125	
	`READ(A, s)`		
	`s := s * 2`		
	`WRITE(A, s)`	250	
	`READ(B, s)`		
	`s := s * 2`		
	`WRITE(B, s)`		50
`READ(B, t)`			
`t := t + 100`			
`WRITE(B, t)`			150

Anomalous Behavior 2: Overwriting Uncommitted Data

T_1 overwrites the value of A, which has already been modified by T_2, while T_2 is still in progress

Example (Overwriting Uncommitted Data)

T_1	T_2	A	B
		25	25
`READ(A, t)`			
`t := t + 100`			
	`READ(A, s)`		
	`s := s * 2`		
	`WRITE(A, s)`	250	
`WRITE(A, t)`		125	
	`READ(B, s)`		
	`s := s * 2`		
`READ(B, t)`			
`t := t + 100`			
`WRITE(B, t)`			125
	`WRITE(B, s)`		50

Anomalous Behavior 3: Unrepeatable Reads

T_2 changes the value of A that has been read by T_1. If T_1 tries to read the value of A again, it will get a different result, even though T_1 has not modified A in the meantime

Example (Unrepeatable Reads)

Assume that the consistency constraint is $A \geq 0$

T_1	T_2	A
		1
`READ(A, t)`		
	`READ(A, s)`	
	`s := s - 1`	
	`WRITE(A, s)`	0
`READ(A, t)`		
`t := t - 1`		
`WRITE(A, t)`		-1

11.2 Schedules
Serializable Schedules

Serializable Schedules (可串行化调度)

A schedule S is serializable (可串行化) if there is a serial schedule S' such that for all initial database states, the effects of S and S' are the same

Example (Serializable Schedules)

T_1	T_2	A	B	T_1	T_2
		25	25		
`READ(A, t)`				`READ(A, t)`	
`t := t + 100`				`t := t + 100`	
`WRITE(A, t)`		125		`WRITE(A, t)`	
	`READ(A, s)`			`READ(B, t)`	
	`s := s * 2`			`t := t + 100`	
	`WRITE(A, s)`	250		`WRITE(B, t)`	
`READ(B, t)`					`READ(A, s)`
`t := t + 100`					`s := s * 2`
`WRITE(B, t)`			125		`WRITE(A, s)`
	`READ(B, s)`				`READ(B, s)`
	`s := s * 2`				`s := s * 2`
	`WRITE(B, s)`		250		`WRITE(B, s)`

Notation

- $r_T(X)$: transaction T reads database element X
- $w_T(X)$: transaction T writes database element X
- $r_i(X) = r_{T_i}(X)$, $w_i(X) = w_{T_i}(X)$

Example (Simplified Notation)

T_1	T_2
`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)`

- $T_1 : r_1(A); w_1(A); r_1(B); w_1(B)$
- $T_2 : r_2(A); w_2(A); r_2(B); w_2(B)$
- $S : r_1(A); w_1(A); r_2(A); w_2(A);$
  $r_1(B); w_1(B); r_2(B); w_2(B)$

Conflict-Serializability (冲突可串行化)

- Commercial DBMS generally enforce conflict-serializability, a condition that is stronger than serializability
- A conflict-serializable schedule must be serializable
- A serializable schedule may not be conflict-serializable

Conflicts (冲突)

A pair of consecutive actions in a schedule is a conflict if when their order is interchanged, the behavior of at least one of the transactions involved can change. Particularly,

1. Two consecutive actions of the same transaction conflict
2. $w_i(X); w_j(X)$ is a conflict
3. $r_i(X); w_j(X)$ is a conflict
4. $w_i(X); r_j(X)$ is a conflict

Conflict-Equivalence (冲突等价)

Two schedules are conflict-equivalent if we can turn one schedule into the other by a sequence of nonconflicting swaps of adjacent actions

Example (Confilit-Equivalence)

Step	Schedule
1	$r_1(A); w_1(A); r_2(A); \underbrace{w_2(A); r_1(B)}; w_1(B); r_2(B); w_2(B)$
2	$r_1(A); w_1(A); \underbrace{r_2(A); r_1(B)}; w_2(A); w_1(B); r_2(B); w_2(B)$
3	$r_1(A); w_1(A); r_1(B); r_2(A); \underbrace{w_2(A); w_1(B)}; r_2(B); w_2(B)$
4	$r_1(A); w_1(A); r_1(B); \underbrace{r_2(A); w_1(B)}; w_2(A); r_2(B); w_2(B)$
5	$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict-Serializability (冲突可串行化)

- A schedule is conflict-serializable if it is conflict-equivalent to a serial schedule
- A conflict-serializable schedule must be serializable, but the reverse may not be true
- Enforcing or testing conflict-serializability turns out to be much more easier than other types of serializability

Example (Confilit-Serializable Schedules)

Step	Schedule
1	$r_1(A); w_1(A); r_2(A); \underbrace{w_2(A); r_1(B)}; w_1(B); r_2(B); w_2(B)$
2	$r_1(A); w_1(A); \underbrace{r_2(A); r_1(B)}; w_2(A); w_1(B); r_2(B); w_2(B)$
3	$r_1(A); w_1(A); r_1(B); r_2(A); \underbrace{w_2(A); w_1(B)}; r_2(B); w_2(B)$
4	$r_1(A); w_1(A); r_1(B); \underbrace{r_2(A); w_1(B)}; w_2(A); r_2(B); w_2(B)$
5	$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict-Serializability (Cont'd)

Example (Confilit-Serializable Schedules)

T_1	T_2		T_1	T_2
`READ(A, t)`			`READ(A, t)`	
`t := t + 100`			`t := t + 100`	
`WRITE(A, t)`			`WRITE(A, t)`	
	`READ(A, s)`		`READ(B, t)`	
	`s := s * 2`		`t := t + 100`	
	`WRITE(A, s)`	$\Rightarrow$	`WRITE(B, t)`	
`READ(B, t)`				`READ(A, s)`
`t := t + 100`				`s := s * 2`
`WRITE(B, t)`				`WRITE(A, s)`
	`READ(B, s)`			`READ(B, s)`
	`s := s * 2`			`s := s * 2`
	`WRITE(B, s)`			`WRITE(B, s)`

Test for Conflict-Serializability (冲突可串行化测试)

Given a schedule S involving transactions $T_1, \ldots, T_n$, T_i precedes (领先于) T_j, written $T_i <_S T_j$, if there are actions A_i of T_i and A_j of T_j such that

1. A_i is ahead of A_j in S
2. Both A_i and A_j involve the same database element
3. At least one of A_i and A_j is a write action

Example (Precedence of Transactions)

Given a schedule
$S = r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$, we have

- $T_1 <_S T_2$ because

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

- $T_2 <_S T_3$ because

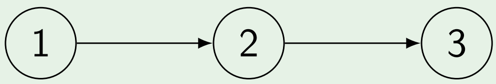
$$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$$

Test for Conflict-Serializability (Cont'd)

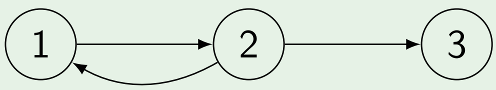
A precedence graph (领先图) for a schedule S is a graph where vertices represent transactions involved in S, and there is an arc from node i to node j if $T_i <_S T_j$

Example (Precedence Graphs)

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



- $S' = r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

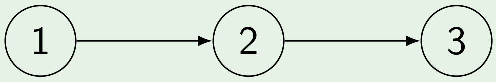


Test for Conflict-Serializability (Cont'd)

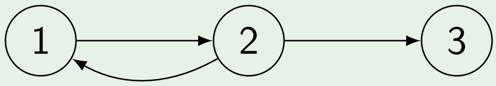
1. If the precedence graph for a schedule S is acyclic (无环), S is conflict-serializable, and any topological order (拓扑排序) of the nodes is a conflict-equivalent serial order
2. If the precedence graph for a schedule S is cyclic (有环), S is not conflict-serializable

Example (Test for Conflict-Serializability)

- $S = r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$ is conflict-serializable, and S is conflict-equivalent to the following serial schedule $r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B); r_3(A); w_3(A)$



- $S' = r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$ is not conflict-serializable



11.3 Lock-based Concurrency Control

11.3 Lock-based Concurrency Control
11.3.1 Locking Protocols

Locking-based Concurrency Control (基于锁的并发控制)

- A locking protocol (锁协议) is a set of rules to be followed by every transaction to ensure that only serializable schedules are allowed
- A lock (锁) is a small bookkeeping object associated with a database element
- Different locking protocols use different types of locks

Consistent Transactions

A transaction is consistent if it obeys the following rules:

1. (读写前须先加锁) The transaction can only read and write an element if it was granted a lock on that element and has not yet released the lock
2. (加锁后还须解锁) If the transaction locks an element, it must later unlock that element

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details

Consistent Transactions (Cont'd)

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

Example (Consistent Transactions)

T_1	A	B
$l_1(A)$		
$r_1(A)$		
`A := A + 100`		
$w_1(A)$		
$l_1(B)$		
$u_1(A)$		
$r_1(B)$		
`B := B + 100`		
$w_1(B)$		
$u_1(B)$		

T_1	A	B
$l_1(A)$		
$r_1(A)$		
`A := A + 100`		
$w_1(A)$		
$u_1(A)$		
$l_1(B)$		
$r_1(B)$		
`B := B + 100`		
$w_1(B)$		
$u_1(B)$		

Legal Schedules (合法调度)

A schedule is legal if no two transactions may have locked the same element without one having first released the lock

Example (Legal Schedules)

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

Illegal Schedules (不合法调度)

A schedule is illegal if two transactions may have locked the same element without one having first released the lock

Example (Illegal Schedules)

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

The Locking Scheduler (锁调度器)

Legal schedules are enforced by the locking scheduler

Scheduling Rules

1. A request is granted if and only if the request will result in a legal schedule
2. If a request is not granted, the requesting transaction is delayed and waits until the scheduler grants its request at a later time

The Locking Scheduler (Cont'd)

Example (Scheduling of Transactions)

T_1	T_2	Actions of the Scheduler
$l_1(A)$; $r_1(A)$		The lock on A is granted to T_1
`A := A + 100`		
$w_1(A)$; $l_1(B)$		The lock on B is granted to T_1
$u_1(A)$		The lock on A is released
	$l_2(A)$; $r_2(A)$	The lock on A is granted to T_2
	`A := A * 2`	
	$w_2(A)$	
	$l_2(B)$	The request is denied, and T_2 waits
$r_1(B)$		
`B := B + 100`		
$w_1(B)$; $u_1(B)$		The lock on B is released
		The lock on B is granted to T_2; T_2 resumes
	$u_2(A)$; $r_2(B)$	The lock on A is released
	`B := B * 2`	
	$w_2(B)$; $u_2(B)$	The lock on B is released

Problem with Legal Schedules

A legal schedule of consistent transactions may not be conflict-serializable

Example (A legal but nonserializable schedule)

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

11.3 Lock-based Concurrency Control
11.3.2 Two-Phase Locking (2PL)

Two-Phase Locking (2PL, 两阶段锁)

A transaction is called a 2PL transaction if it obeys the 2PL locking rule

The 2PL Locking Rule

In every transaction, all lock actions precede all unlock actions

Example (2PL Transactions)

T_1	T_2
$l_1(A)$	$l_2(A)$
$r_1(A)$	$r_2(A)$
`A := A + 100`	`A := A * 2`
$w_1(A)$	$w_2(A)$
$l_1(B)$	$l_2(B)$
$u_1(A)$	$u_2(A)$
$r_1(B)$	$r_2(B)$
`B := B + 100`	`B := B * 2`
$w_1(B)$	$w_2(B)$
$u_1(B)$	$u_2(B)$

Example (Not 2PL Transactions)

T_1	T_2
$l_1(A)$	$l_2(A)$
$r_1(A)$	$r_2(A)$
`A := A + 100`	`A := A * 2`
$w_1(A)$	$w_2(A)$
$u_1(A)$	$u_2(A)$
$l_1(B)$	$l_2(B)$
$r_1(B)$	$r_2(B)$
`B := B + 100`	`B := B * 2`
$w_1(B)$	$w_2(B)$
$u_1(B)$	$u_2(B)$

Two-Phase Locking (Cont'd)

Any legal schedule of consistent and 2PL transactions is conflict-serializable

Example (Conflict-Serializable Schedules Enforced by 2PL)

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

Problems with 2PL

1. A transaction must take a lock on a database element even if it only wants to read it but not write it
2. Several transactions could not read a database element at the same time even if none is allowed to write the element

Shared and Exclusive Locks (共享锁与互斥锁)

- Shared locks (read locks): Transaction T can only read database element X if T was granted a shared lock on X and has not yet released that lock
- Exclusive locks (write locks): Transaction T can only write database element X if T was granted an exclusive lock on X and has not yet released that lock
- Transaction T can also read database element X if T was granted an exclusive lock on X and has not yet released that lock
- For every database element X, there can can be either one exclusive lock on X, or no exclusive lock but any number of shared locks

Notation

- $sl_i(X)$: Transaction T_i requests a shared lock on database element X
- $xl_i(X)$: Transaction T_i requests an exclusive lock on database element X
- $u_i(X)$: Transaction T_i releases its lock(s) on database element X

Locking Rules using Shared and Exclusive Locks

Rules Guaranteeing Consistent Transactions

1. (读前须加共享锁) A read action $r_i(X)$ must be preceded by $sl_i(X)$ or $xl_i(X)$, with no intervening $u_i(X)$
2. (写前须加互斥锁) A write action $w_i(X)$ must be preceded by $xl_i(X)$, with no intervening $u_i(X)$
3. (加锁后还须解锁) All locks must be unlocked

Rules Guaranteeing 2PL Transactions

1. (先全加锁后全解锁) All lock actions precede all unlock actions

Rules Guaranteeing Legal Schedules

1. If $xl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$ or $sl_j(X)$, for $j \neq i$, without an intervening $u_i(X)$
2. If $sl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$, for $j \neq i$, without an intervening $u_i(X)$

Locking Rules using Shared and Exclusive Locks (Cont'd)

Any legal schedule of consistent and 2PL transactions using shared and exclusive locks is conflict-serializable

Example (Shared and Exclusive Locks)

T_1	T_2	A	B
$sl_1(A); r_1(A)$		$\subset \boxed{s}(T_1)$	
	$sl_2(A); r_2(A)$	$\subset \boxed{s}(T_1) \subset \boxed{s}(T_2)$	
	$sl_2(B); r_2(B)$	$\subset \boxed{s}(T_1) \subset \boxed{s}(T_2)$	$\subset \boxed{s}(T_2)$
	$u_2(A); u_2(B)$	$\subset \boxed{s}(T_1)$	
$xl_1(B); r_1(B); w_1(B)$		$\subset \boxed{s}(T_1)$	$\subset \boxed{x}(T_1)$
$u_1(A); u_1(B)$			

Scheduling using Shared and Exclusive Locks

Legal schedules of transactions using shared and exclusive locks are enforced by the scheduler (调度器) or lock manager (锁管理器)

Scheduling Rule (调度规则)

A request for lock on database element X in lock type C can be granted if and only if for every row R in the compatibility matrix, if a lock on X in lock type R has been granted to some other transaction, there is a "Yes" in column C

Compatibility Matrix (相容矩阵)

		Lock Requested	
		Shared lock	Exclusive lock
Lock	Shared lock	Yes	No
Granted	Exclusive lock	No	No

Scheduling using Shared and Exclusive Locks (Cont'd)

Example (Scheduling using Shared and Exclusive Locks)

T_1	T_2	Actions of the Scheduler
$sl_1(A); r_1(A)$		A shared lock on A is granted to T_1
	$sl_2(A); r_2(A)$	A shared lock on A is granted to T_2
	$sl_2(B); r_2(B)$	A shared lock on B is granted to T_2
$xl_1(B)$		This request is denied, and T_1 waits
	$u_2(A)$	The shared lock on A granted to T_2 is released
	$u_2(B)$	The shared lock on B granted to T_2 is released
		The exclusive lock on B is granted to T_1
		T_1 is resumed
$r_1(B); w_1(B)$		
$u_1(A)$		The shared lock on A granted to T_1 is released
$u_1(B)$		The exclusive lock on B granted to T_1 is released

Problems with shared and exclusive locks: $r_1(B)$ is delayed, but it is not necessary to delay $r_1(B)$

11.3 Lock-based Concurrency Control
11.3.3 Lock Conversions

Upgrading Locks (锁升级)

A transaction that wants to read and write a new value of X first takes a shared lock on X, and only later, when T was ready to write the new value, upgrade the lock to exclusive (i.e., request an exclusive lock on X in addition to its already held shared lock on X)

Example (Upgrading Locks)

T_1	T_2	Actions of the Scheduler
$sl_1(A); r_1(A)$		A shared lock on A is granted to T_1
	$sl_2(A); r_2(A)$	A shared lock on A is granted to T_2
	$sl_2(B); r_2(B)$	A shared lock on B is granted to T_2
$sl_1(B); r_1(B)$		A shared lock on B is granted to T_1
$xl_1(B)$		This upgrading request is denied, and T_1 waits
	$u_2(A)$	The shared lock on A granted to T_2 is released
	$u_2(B)$	The shared lock on B granted to T_2 is released
		The exclusive lock on B is granted to T_1
		T_1 is resumed
$w_1(B)$		
$u_1(A)$		The shared lock on A granted to T_1 is released
$u_1(B)$		The exclusive lock on B granted to T_1 is released

Upgrading Locks (Cont'd)

Upgrading locks can cause a deadlock (死锁)

Example (Deadlocks Due to Upgrading Locks)

T_1	T_2	Actions of the Scheduler
$sl_1(A)$		A shared lock on A is granted to T_1
$r_1(A)$		
	$sl_2(A)$	A shared lock on A is granted to T_2
	$r_2(A)$	
$xl_1(A)$		This upgrading request is denied, and T_1 waits
	$xl_2(A)$	This upgrading request is denied, and T_2 waits

Update Locks (更新锁)

- An update lock $ul_i(X)$ gives transaction T_i only the privilege to read database element X, but not to write X
- Only update locks can be upgraded to exclusive locks
- Shared locks cannot be upgraded to exclusive locks

Compatibility Matrix

Granted\Requested	Shared lock	Exclusive lock	Update lock
Shared lock	Yes	No	Yes
Exclusive lock	No	No	No
Update lock	No	No	No

Update Locks (Cont'd)

Example (Update Locks)

T_1	T_2	Actions of the Scheduler
$ul_1(A)$		An update lock on A is granted to T_1
$r_1(A)$		
	$ul_2(A)$	This request is denied, and T_2 waits
$xl_1(A)$		An exclusive lock on A is granted to T_1
$w_1(A)$		
$u_1(A)$		Release the locks on A granted to T_1
		An update lock on A is granted to T_2
		T_2 is resumed
	$r_2(A)$	
	$xl_2(A)$	An exclusive lock on A is granted to T_2
	$w_2(A)$	
	$u_2(A)$	The locks on A granted to T_2 are released

11.3 Lock-based Concurrency Control
11.3.4 Recoverable Schedules

Schedules Involving Aborted Transactions

A transaction T_2 starts after the abort of another transaction T_1

Example (Serial Schedules Involving Aborted Transactions 1)

T_1	T_2	A	B
		25	25
READ(A, t)			
t := t + 100			
WRITE(A, t)		125	
ABORT		25	
	READ(A, s)		
	s := s * 2		
	WRITE(A, s)	50	
	READ(B, s)		
	s := s * 2		
	WRITE(B, s)		50
	COMMIT		

Schedules Involving Aborted Transactions (Cont'd)

An aborted transaction T_1 starts after the commit of another transaction T_2

Example (Serial Schedules Involving Aborted Transactions 2)

T_1	T_2	A	B
		25	25
	READ(A, s)		
	s := s * 2		
	WRITE(A, s)	50	
	READ(B, s)		
	s := s * 2		
	WRITE(B, s)		50
	COMMIT		
READ(A, t)			
t := t + 100			
WRITE(A, t)		150	
ABORT		50	

Schedules Involving Aborted Transactions (Cont'd)

- A transaction T_2 has read a value for database element A written by another transaction T_1 that subsequently aborts
- Aborting T_1 will lead to a cascading abort (级联中止) of T_2

Example (Nonserial Schedules Involving Aborted Transactions)

T_1	T_2	A	B
		25	25
`READ(A, t)`			
`t := t + 100`			
`WRITE(A, t)`		125	
	`READ(A, s)`		
	`s := s * 2`		
	`WRITE(A, s)`	250	
`ABORT`		125	
		25	
	`READ(B, s)`		
	`s := s * 2`		
	`WRITE(B, s)`		
	`COMMIT`		

Schedules Involving Aborted Transactions (Cont'd)

- A transaction T_2 has read a value for database element A written by another transaction T_1 that aborts after the commit of T_2
- T_2 has already committed, so we cannot undo its actions
- When T_1 is aborted, the changes made by T_2 are lost
- Such a schedule is unrecoverable (不可恢复)

Example (Unrecoverable Schedules)

T_1	T_2	A	B
		25	25
`READ(A, t)`			
`t := t + 100`			
`WRITE(A, t)`		125	
	`READ(A, s)`		
	`s := s * 2`		
	`WRITE(A, s)`	250	
	`READ(B, s)`		
	`s := s * 2`		
	`WRITE(B, s)`		50
	`COMMIT`		
`ABORT`		25	

Recoverable Schedules (可恢复调度)

- In a recoverable schedule, transactions commit only after all transactions whose changes they read commit
- A recoverable schedule avoids cascading aborts
- Recoverable schedules do not require cascading aborts, and actions of aborted transactions can be undone by restoring the original values of modified database elements

Strict Schedules (严格调度)

- A schedule is strict if a value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits
- Strict schedules are recoverable

Strict 2PL (严格两阶段锁协议)

- Under Strict 2PL, each 2PL transaction holds its (exclusive) locks until it commits or aborts
- Strict 2PL improves on 2PL by guaranteeing that every allowed schedule is strict in addition to being conflict serializable

Example (Strict 2PL)

T_1	T_2	A	B	Actions of the Scheduler
		25	25	
$xl_1(A); r_1(A)$				The exclusive lock on A is granted to T_1
`A := A + 100`				
$w_1(A)$		125		
	$xl_2(A)$			The request is denied, and T_2 waits
`ABORT`		25		
$u_1(A)$				The exclusive lock on A is released The exclusive lock on A is granted to T_2
	$r_2(A)$			
	`A := A * 2`			
	$w_2(A)$	50		
	$xl_2(B); r_2(B)$			The exclusive lock on B is granted to T_2
	`B := B * 2`			
	$w_2(B)$		50	
	$u_2(A); u_2(B)$			The locks on A and B are released

11.3 Lock-based Concurrency Control
11.3.5 Resolving Deadlocks

Deadlocks (死锁)

- A set of concurrently executing transactions reach a deadlock state if each of the transactions is waiting for a resource held by one of the others, and none can make progress
- Deadlock detection (死锁检测)
  - Deadlock detection by timeout (超时)
  - Deadlock detection by wait-for graphs (等待图)
- Deadlock prevention (死锁预防)
  - Deadlock prevention by ordering elements

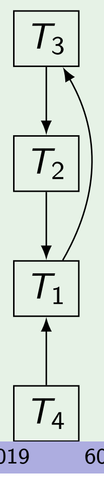
Waits-for Graphs (等待图)

Each node represents a transaction that currently holds a lock or is waiting for one. There is an arc from node T to node U if there is some database element X such that:

1. U holds a lock on X
2. T is waiting for a lock on X
3. T cannot get a lock on X in its desired type unless U first releases its lock on X

Example (Waits-for Graphs)

T_1	T_2	T_3	T_4
$l_1(A); r_1(A)$			
	$l_2(C); r_2(C)$		
		$l_3(B); r_3(B)$	
			$l_4(D); r_4(D)$
	$l_2(A)$ (denied)		
		$l_3(C)$ (denied)	
			$l_4(A)$ (denied)
$l_1(B)$ (denied)			

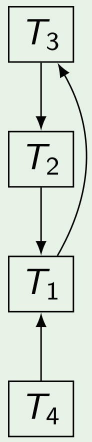


Deadlock Detection (死锁检测) by Waits-for Graphs

- If the waits-for graph is acyclic (无环), then there is no deadlock
- If the waits-for graph is cyclic (有环), then no transaction in the cycle can ever make progress, so there is a deadlock

Example (Waits-for Graphs)

T_1	T_2	T_3	T_4
$l_1(A); r_1(A)$			
	$l_2(C); r_2(C)$		
		$l_3(B); r_3(B)$	
			$l_4(D); r_4(D)$
	$l_2(A)$ (denied)		
		$l_3(C)$ (denied)	
			$l_4(A)$ (denied)
$l_1(B)$ (denied)			

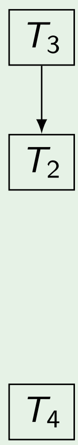


Deadlock Resolution (死锁解除) by Waits-for Graphs

To resolve a deadlock, abort any transaction on the cycle

Example (Waits-for Graphs)

T_1	T_2	T_3	T_4
$l_1(A); r_1(A)$			
	$l_2(C); r_2(C)$		
		$l_3(B); r_3(B)$	
			$l_4(D); r_4(D)$
	$l_2(A)$ (denied)		
		$l_3(C)$ (denied)	
			$l_4(A)$ (denied)
$l_1(B)$ (denied)			
abort	granted	granted	granted



Deadlock Prevention (死锁预防) by Ordering Elements

- Order database elements in some arbitrary but fixed order
- Every transaction is required to request locks on elements in order
- If so, there can be no deadlock

Example (Deadlock Prevention by Ordering Elements)

The locking order is $A \to B \to C \to D$

T_1	T_2	T_3	T_4
$l_1(A); r_1(A)$			
	$l_2(A)$ (denied)		
		$l_3(B); r_3(B)$	
			$l_4(A)$ (denied)
		$l_3(C); w_3(C)$	
		$u_3(B); u_3(C)$	
$l_1(B); w_1(B)$			
$u_1(A); u_1(B)$			
	$l_2(A); l_2(C)$		
	$r_2(C); w_2(A)$		
	$u_2(A); u_2(C)$		
			$l_4(A); r_4(D)$
			$r_4(D); w_4(A)$
			$u_4(D); u_4(D)$

Excercises

1. Design a 2PL lock manager
2. Work out how the schedule in the example of unrecoverable schedules is disallowed by Strict 2PL but not by 2PL

11.4 Transaction Isolation Levels

Transaction Isolation Levels (事务隔离级别)

- Serializability allows programmers to ignore issues related to concurrency when they code transactions
- However, the lockinig protocols required to ensure serializability may allow too little concurrency for certain applications
- In these cases, weaker levels of consistency are used, which places additional burdens on programmers for ensuring database consistency
- The isolation level controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently

Transaction Isolation Levels (Cont'd)

Four isolation levels specified by the SQL standard are as follows

- Read Uncommitted (读未提交): Uncommitted data are allowed to be read
- Read Committed (读提交): Only committed data are allowed to be read
- Repeatable Read (可重复读)[2]: Only committed data are allowed to be read, and between two reads of a database element by a transaction, no other transaction is allowed to update it
- Serializable (可串行化)

[2] The default isolation level of MySQL is Repetable Read

Read Uncommitted (读未提交)

Changes to database elements made by an uncommitted transaction are exposed to other transactions

- Dirty read (脏读): possible
- Unrepeatable read (不可重复读): possible

Example (Read Uncommitted)

T_1	T_2	A	Variables
		A = 1	
READ(A, t)			t = 1
	READ(A, s)		s = 1
	s := s * 2		s = 2
	WRITE(A, s)	A = 2	
READ(A, x)			x = 2
	COMMIT		
READ(A, y)			y = 2
COMMIT			
READ(A, z)			z = 2

Read Committed (读提交)

Only changes to database elements made by an committed transaction are exposed to other transactions

- Dirty read: no
- Unrepeatable read: possible

Example (Read Committed)

T_1	T_2	A	Variables
		A = 1	
READ(A, t)			t = 1
	READ(A, s)		s = 1
	s := s * 2		s = 2
	WRITE(A, s)	A = 2	
READ(A, x)			x = 1
	COMMIT		
READ(A, y)			y = 2
COMMIT			
READ(A, z)			z = 2

Repeatable Read (可重复读)

The value for a database element read by a transaction must always be equal to the value it would read at the start time

- Dirty read: no
- Unrepeatable read: no

Example (Repeatable Read)

T_1	T_2	A	Variables
		A = 1	
READ(A, t)			t = 1
	READ(A, s)		s = 1
	s := s * 2		s = 2
	WRITE(A, s)	A = 2	
READ(A, x)			x = 1
	COMMIT		
READ(A, y)			y = 1
COMMIT			
READ(A, z)			z = 2

Repeatable Read (Cont'd)

- Phantom read (幻读): possible

Example (Repeatable Read)

R

a	b
1	1
2	2

T_1	T_2

`SELECT a FROM R`
`WHERE b = 1 FOR UPDATE;`
Result: 1	

	`UPDATE R SET b = 1 WHERE a = 2;`
`SELECT a FROM R WHERE b = 1;` Result: 1, 2	

Isolation Levels

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read (幻读)
Read Uncommitted	Possible	Possible	Possible
Read Committed	No	Possible	Possible
Repeatable Read	No	No	Possible
Serializable	No	No	No

Summary

Thank You!

For any questions and comments, please contact me at

znzou@hit.edu.cn