Concurrency

Database Systems Lecture 16

Natasha Alechina

In This Lecture

- Concurrency control
- Serialisability
  - Schedules of transactions
  - Serial & serialisable schedules
- Locks
- 2 Phase Locking protocol (2PL)
- For more information
  - Connolly and Begg chapter 20
  - Ullman and Widom chapter 8.6

Need for concurrency control

- Previous lecture: transactions running concurrently may interfere with each other, causing various problems (lost updates etc.)
- Concurrency control: the process of managing simultaneous operations on the database without having them interfere with each other.

Lost Update

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

This update
Is lost

Only this update
succeeds

Uncommitted Update
("dirty read")

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	X = X + 5
	Write(X)
ROLLBACK	
	COMMIT

This reads the value of X which it should not have seen

Inconsistent analysis

T1	T2
Read(X)	
X = X - 5	
Write(X)	
	Read(X)
	Read(Y)
	Sum = X+Y
Read(Y)	
Y = Y + 5	
Write(Y)	

Summing up data while it is being updated

Schedules

- A *schedule* is a sequence of the operations by a set of concurrent transactions that preserves the order of operations in each of the individual transactions
- A *serial* schedule is a schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions (each transaction commits before the next one is allowed to begin)

Serial schedules

- Serial schedules are guaranteed to avoid interference and keep the database consistent
- However databases need concurrent access which means interleaving operations from different transactions

Serialisability

- Two schedules are *equivalent* if they always have the same effect.
- A schedule is *serialisable* if it is equivalent to some serial schedule.
- For example:
  - if two transactions only read some data items, then the order is which they do it is not important
  - If T1 reads and updates X and T2 reads and updates a different data item Y, then again they can be scheduled in any order.

Serial and Serialisable

Interleaved Schedule

T1 Read(X)
T2 Read(X)
T2 Read(Y)
T1 Read(Z)
T1 Read(Y)
T2 Read(Z)

This schedule is serialisable:

Serial Schedule

T2 Read(X)
T2 Read(Y)
T2 Read(Z)

T1 Read(X)
T1 Read(Z)
T1 Read(Y)

Conflict Serialisable Schedule

Interleaved Schedule	Serial Schedule
T1 Read(X)	T1 Read(X)
T1 Write(X)	T1 Write(X)
T2 Read(X)	T1 Read(Y)
T2 Write(X)	T1 Write(Y)
T1 Read(Y)	
T1 Write(Y)	T2 Read(X)
T2 Read(Y)	T2 Write(X)
T2 Write(Y)	T2 Read(Y)
	T2 Write(Y)

This schedule is serialisable, even though T1 and T2 read and write the same resources X and Y: they have a conflict

Conflict Serialisability

- Two transactions have a conflict:
  - NO If they refer to different resources
  - NO If they are reads
  - YES If at least one is a write and they use the same resource

- A schedule is *conflict serialisable* if transactions in the schedule have a conflict but the schedule is still serialisable

Conflict Serialisability

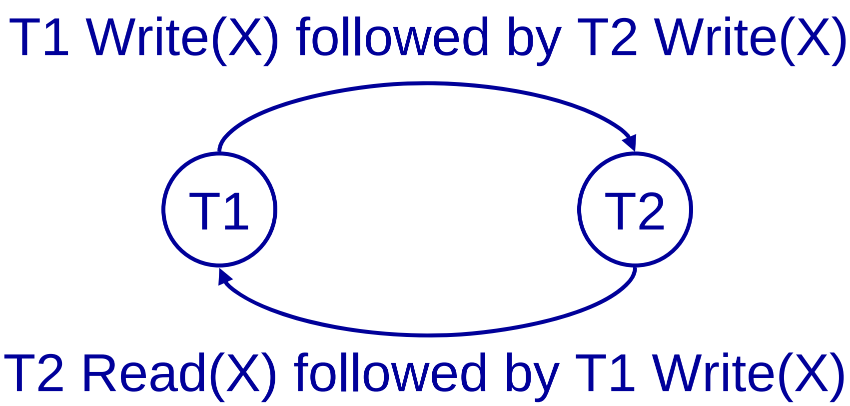
- Conflict serialisable schedules are the main focus of concurrency control
- They allow for interleaving and at the same time they are guaranteed to behave as a serial schedule
- Important questions: how to determine whether a schedule is conflict serialisable
- How to construct conflict serialisable schedules

Precedence Graphs

- To determine if a schedule is conflict serialisable we use a precedence graph
  - Transactions are vertices of the graph
  - There is an edge from T1 to T2 if T1 must happen before T2 in any equivalent serial schedule
- Edge T1 $\rightarrow$ T2 if in the schedule we have:
  - T1 Read(R) followed by T2 Write(R) for the same resource R
  - T1 Write(R) followed by T2 Read(R)
  - T1 Write(R) followed by T2 Write(R)
- The schedule is serialisable if there are no cycles

Precedence Graph Example

- The lost update schedule has the precedence graph:

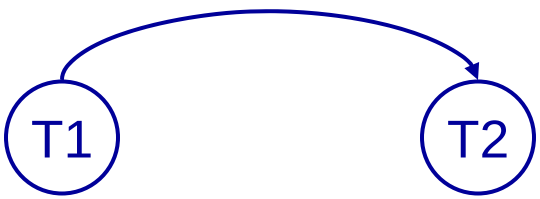


T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

Precedence Graph Example

- No cycles: conflict serialisable schedule

T1 reads X before T2 writes X and
T1 writes X before T2 reads X and
T1 writes X before T2 writes X



T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(X)

Locking

- Locking is a procedure used to control concurrent access to data (to ensure serialisability of concurrent transactions)
- In order to use a 'resource' (table, row, etc) a transaction must first acquire a *lock* on that resource
- This may deny access to other transactions to prevent incorrect results

Two types of locks

- Two types of lock
  - Shared lock (S-lock or read-lock)
  - Exclusive lock (X-lock or write-lock)
- Read lock allows several transactions simultaneously to read a resource (but no transactions can change it at the same time)
- Write lock allows one transaction exclusive access to write to a resource. No other transaction can read this resource at the same time.
- The lock manager in the DBMS assigns locks and records them in the data dictionary

Locking

- Before reading from a resource a transaction must acquire a read-lock
- Before writing to a resource a transaction must acquire a write-lock
- Locks are released on commit/rollback
- A transaction may not acquire a lock on any resource that is write-locked by another transaction
- A transaction may not acquire a write-lock on a resource that is locked by another transaction
- If the requested lock is not available, transaction waits

Two-Phase Locking

- A transaction follows the *two-phase locking protocol* (2PL) if all locking operations precede the first unlock operation in the transaction

- Two phases
  - Growing phase where locks are acquired on resources
  - Shrinking phase where locks are released

Example

- T1 follows 2PL protocol
  - All of its locks are acquired before it releases any of them
- T2 does not
  - It releases its lock on X and then goes on to later acquire a lock on Y

T1	T2
read-lock(X)	read-lock(X)
Read(X)	Read(X)
write-lock(Y)	unlock(X)
unlock(X)	write-lock(Y)
Read(Y)	Read(Y)
Y = Y + X	Y = Y + X
Write(Y)	Write(Y)
unlock(Y)	unlock(Y)

Serialisability Theorem

Any schedule of two-phased transactions is conflict serialisable

Lost Update can't happen with 2PL

	T1	T2	
read-lock(X)	**Read(X)**		
	X = X - 5		
		Read(X)	read-lock(X)
		X = X + 5	
cannot acquire write-lock(X): T2 has read-lock(X)	**Write(X)**		
		Write(X)	cannot acquire write-lock(X): T1 has read-lock(X)
	COMMIT		
		COMMIT	

Uncommitted Update cannot happen with 2PL

	T1	T2	
read-lock(X)	**Read(X)**		
	X = X - 5		
write-lock(X)	**Write(X)**		Waits till T1 releases write-lock(X)
		Read(X)	
		X = X + 5	
		Write(X)	
Locks released	**ROLLBACK**		
		COMMIT	

Inconsistent analysis cannot happen with 2PL

	T1	T2	
read-lock(X)	**Read(X)**		
	X = X - 5		
write-lock(X)	**Write(X)**		
		Read(X)	Waits till T1 releases write-locks on X and Y
		Read(Y)	
		Sum = X+Y	
read-lock(Y)	**Read(Y)**		
	Y = Y + 5		
write-lock(Y)	**Write(Y)**		

Next Lecture

- Deadlocks
  - Deadlock detection
  - Deadlock prevention
- Timestamping
- For more information
  - Connolly and Begg chapter 20
  - Ullman and Widom chapter 8.6