CMPT 321
FALL 2017

Concurrent DB operations

Lecture 05.04

By Marina Barsky

Why concurrent execution

- It is possible for multiple queries to be submitted at approximately the same time
- Many queries are both complex and time consuming: finishing these queries would make other queries wait a long time for a chance to execute
- Disk usage can be optimized for several queries running in parallel (recall – elevator algorithm)

So, in practice, the DBMS may be running **many different queries at about the same time (concurrently)**

Interleaving

- DBMS has to *interleave* the actions of several transactions
- **Interleaving** of transactions may **lead to anomalies** even if each individual transaction preserves all the database constraints

Recording transactions

- To reason about the order of interleaving transactions, we can abstract each transaction into a sequence of reads and writes of disk data
- For example, withdrawing of money from the account can be written as:

  $\mathbf{r_1(A); w_1(A)}$

That means that transaction T_1 reads database element A, does something with it in main memory and writes it back to the database

Recording sequence of commands

- Then we can record the sequence of commands from 2 transactions received by DBMS as:

$r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$

Transactions and Schedules: notation

- To ensure that interleaving does not lead to anomalies, DBMS *schedules* the execution of each action in a certain way
- A ***schedule*** is a list of actions for a set of interleaved transactions

Possible schedule:

T1	T2
r(A)	
	r(A)
	w(A)
	commit
w(A)	
commit	

Anomalies of interleaving: case 1

- Consider two transactions T1 and T2, each of which, when running alone preserves database consistency:
  - T1 transfers $100 from A to B (e.g. from checking to saving account)
  - T2 increments both A and B by 1% (e.g. daily interest)

- The list of actions received by DBMS:

  r1(A); w1(A);r1(B);w1(B);r2(A);w2(A);r2(B);w2(B)

Anomalies of interleaving transactions: possible schedule

DBMS decides on the following schedule:

T1	T2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
	commit
r(B)	
w(B)	
commit	

What is the problem?

Anomalies of interleaving transactions: case 1

T1	T2	
r(A)		T1 deducted $100 from A
w(A)		
	r(A)	T2 incremented both A and B by 1%
	w(A)	
	r(B)	
	w(B)	
	commit	
r(B)		T1 added $100 to B
w(B)		
commit		

Anomalies of interleaving: reading uncommitted data

T1	T2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
	commit
r(B)	
w(B)	
commit	

T1 deducted $100 from A

T2 incremented both A and B by 1%

T1 added $100 to B

The problem is that the bank didn't pay interest on the $100 that was being transferred. This happened because T2 was **reading uncommitted** values.

Anomalies of interleaving transactions: case 2

- Suppose that A is the number of copies available for a book.
- Transactions T1 and T2 both place an order for this book. First they check the availability of the book.
- Consider the following scenario:
  1. T1 checks whether A is greater than 1.
     Suppose T1 sees (reads) value 1.
  2. T2 also reads A and sees 1.
  3. T2 decrements A to 0.
  4. T2 commits.
  5. T1 tries to decrement A, which is now 0, and gets an error because some integrity check doesn't allow it.

Anomalies of interleaving: unrepeatable reads

1. T1 checks whether A is greater than 1.
   Suppose T1 sees (reads) value 1.
2. T2 also reads A and sees 1.
3. T2 decrements A to 0.
4. T2 commits.
5. T1 tries to decrement A, which is now 0, and gets an error because some integrity check doesn't allow it.

The problem is that because value of A has been changed by T1, when T2 reads A for the second time, before updating it, **the value is different from that when T2 started.**

Anomalies of interleaving transactions: case 3

- Suppose that Larry and Harry are two employees, and their salaries **must be kept equal**. T1 sets their salaries to $1000 and T2 sets their salaries to $2000.
- Now consider the following schedule:

T1	T2
r(Larry)	
w(Larry)	
	r(Harry)
	w(Harry)
r(Harry)	
w(Harry)	
	r(Larry)
	w(Larry)
	commit
commit	

What is the problem?

Anomalies of interleaving: overriding uncommitted data

- Suppose that Larry and Harry are two employees, and their salaries **must be kept equal**. T1 sets their salaries to $1000 and T2 sets their salaries to $2000.
- Now consider the following schedule:

T1	T2	
r(Larry)		
w(Larry)		
	r(Harry)	
	w(Harry)	
r(Harry)		
w(Harry)		← $1000 to Harry
	r(Larry)	
	w(Larry)	← $2000 to Larry
	commit	
commit		

Anomalies of interleaving

- Reading uncommitted data
- Unrepeatable reads
- Overriding uncommitted data

None of these would happen if we were executing transactions one after another: **serial schedules**

Notations

- A **transaction** (model) is a *sequence* of *r* and *w* requests on database elements

- A **schedule** is a *sequence* of reads/writes actions performed by a DBMS: to achieve interleaving and at the same time preserve consistency

- **Serial Schedule** = All actions for each transaction are consecutive.

  r1(A); w1(A); r1(B); w1(B); r2(A); w2(A); r2(B); w2(B); …

- **Serializable Schedule**: A schedule whose "**effect**" is equivalent to that of some serial schedule.

Serializable schedules

Sufficient condition for serializability

Equivalent schedules and conflicts

- Two transactions ***conflict*** if they access the same data element and *at least one of the actions is a write*.
- $r_i(X); r_j(Y) \equiv r_j(Y); r_i(X)$ (even when X=Y) — No conflict
- We can flip $r_i(X); w_j(Y)$ as long as $X \neq Y$ — No conflict
- However, $r_i(X); w_j(X) \neq w_j(X); r_i(X)$ — Conflict!
- We can flip $w_i(X); w_j(Y);$ provided $X \neq Y$ — No conflict
- However, $w_i(X); w_j(X) \neq w_j(X); w_i(X);$ — Conflict!

  The final value of X may be different depending on which write occurs last.

Conflicts: summary

There is a conflict if one of these two conditions hold:

1. A read and a write of the same X, or
2. Two writes of the same X

- Such actions conflict in general and *may **not** be swapped in order*.
- All other events (reads/writes) of 2 different transactions may be swapped without changing the **effect** of the schedule.

Sufficient condition for serializable schedule

A schedule is ***conflict-serializable*** if it can be converted into a **serial** schedule by a series of **non-conflicting swaps** of adjacent elements

Example:

What can we say about the original schedule?

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

Non-conflicting swaps:

$r_1(A); w_1(A); r_2(A); \underline{r_1(B);}\ \underline{w_2(A);}\ w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); \underline{r_1(B);}\ \underline{r_2(A);}\ w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); \underline{w_1(B);}\ \underline{w_2(A);}\ r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); \underline{w_1(B);}\ \underline{r_2(A);}\ w_2(A); r_2(B); w_2(B)$

Result: serial schedule

Conflict-serializability

Sufficient condition for serializability but **not necessary**.

Example

S1: $w_1(Y)$; $w_1(X)$; $w_2(Y)$; $w_2(X)$; $w_3(X)$; ← This is serial

S2: $w_1(Y)$; $w_2(Y)$; **$w_2(X)$; $w_1(X)$; $w_3(X)$;** ←

This is called **view-serializable**, and requires from scheduler to understand what each action is doing, not just its type

S2 isn't conflict-serializable, but it is serializable. It has the same effect as S1.

Intuitively, the values of X written by T1 and T2 have no effect, since T3 overwrites them.

Serializability/precedence Graphs

- Non-swappable pairs of actions represent potential conflicts between transactions.
- The existence of non-swappable actions enforces an **ordering** on the transactions that include these actions.

We can represent this order by a **graph**

- **Nodes**: transactions $\{T_1,\ldots,T_k\}$
- **Arcs**: There is a directed edge from T_i to T_j if they have conflicting access to the same database element X and T_i is first:

  written $T_i <_S T_j$.

Precedence graphs: example 1

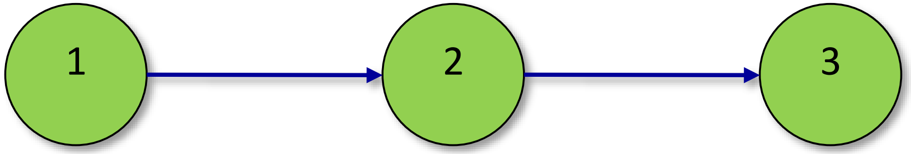
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Note the following:

- $w_1(B) <_S r_2(B)$
- $r_2(A) <_S w_3(A)$

➢ These are conflicts since they contain a read/write on the same element

➢ They cannot be swapped. Therefore $T_1 < T_2 < T_3$



Conflict-serializable

Precedence graphs: example 2

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

Note the following:

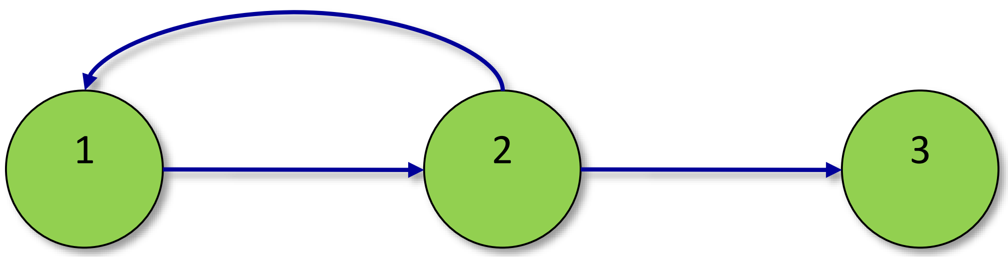
- $r_1(B) <_S w_2(B)$
- $w_2(A) <_S w_3(A)$
- $r_2(B) <_S w_1(B)$

➢ Here, we have

$T_1 < T_2 < T_3$,

but we also have

$T_2 < T_1$



Not conflict-serializable

Precedence graphs: test for conflict-serializability

- **If there is a cycle in the graph,** then there is **no** serial schedule which is conflict-equivalent to S.
  - Each arc represents a requirement on the order of transactions in a conflict-equivalent *serial schedule*.
  - A cycle puts too many requirements on any *linear order* of transactions.

- **If there is no cycle in the graph,** then *any* **topological order*** of the graph suggests a conflict-equivalent schedule.

---

*A topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges

Enforcing serializability by locks

- If scheduler allows multiple transactions access the same element, this may result in non-serializable schedule
- To prevent this, before reading or writing an element X, a transaction T_i requests a lock on X from the scheduler.
- The scheduler can either grant the lock to T_i or make T_i wait for the lock.
- If granted, T_i should eventually unlock (release) the lock on X.

- Notations:

  $L_i(X)$ = "transaction T_i requests a lock on X"

  $u_i(X)$ (or **$uL_i(X)$**)= "T_i unlocks/releases the lock on X"

Legal schedule with locks

Schedule with locks - constraints:

Consistency of Transactions:

- Read or write X only when hold a lock on X.

  $r_i(X)$ or $w_i(X)$ must be preceded by some $L_i(X)$ with no intervening $u_i(X)$.

- If T_i locks X, T_i must eventually unlock X.

  Every $L_i(X)$ must be followed by $u_i(X)$.

Legality of Schedules:

- Two transactions may not have locked the same element X without one having first released the lock.

  A schedule with $L_i(X)$ cannot have another $L_j(X)$ until $u_i(X)$ appears *in between.*

T_1	T_2	A	B
		25	25
$\mathbf{L_1(A)}$; $r_1(A)$			
A = A + 100			
$w_1(A)$;$\mathbf{u_1(A)}$		125	
	$\mathbf{L_2(A)}$;$r_2(A)$		
	A = A * 2		
	$w_2(A)$;$\mathbf{u_2}(A)$	250	
	$\mathbf{L_2(B)}$;$r_2(B)$		
	B = B * 2		
	$w_2(B)$;$\mathbf{u_2(B)}$		50
$\mathbf{L_1(B)}$;$r_1(B)$			
B = B + 100			
$w_1(B)$;$\mathbf{u_1(B)}$			150

T1 unlocks A so T2 is free to lock it

Legal schedule doesn't mean serializable!

- T1 adds 100 to both A and B
- T2 doubles both A and B
- Expected result: A=B, and should be 250 for both by the end

Two-Phase Locking

There is a simple condition, which guarantees conflict-serializability:

In every transaction, all lock requests (phase 1) precede all unlock requests (phase 2).

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	
	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)		
	B = B * 2		
	w_2(B);**u_2(B)**		250

Simple locks are too restrictive

- While simple locks + 2PL guarantee conflict-serializability, **they do not allow two readers of DB element X at the same time.**

- **But having multiple readers is not a problem for conflict-serializability (since read actions commute)!**

Shared/Exclusive Locks

Solution: Two types of locks:

I. **Shared lock $sL_i(X)$** allows T_i to read, but not write X.
   It prevents other transactions from writing X but not from reading X.

II. **Exclusive lock $xL_i(X)$** allows T_i to read and/or write X.
   No other transaction may read or write X.

Shared/Exclusive Locks: changes

Consistency of transactions:

- A read $\mathbf{r_i(X)}$ must be preceded by $\mathbf{sL_i(X)}$ or $\mathbf{xL_i(X)}$, with no intervening $\mathbf{u_i(X)}$.
- A write $\mathbf{w_i(X)}$ must be preceded by $\mathbf{xL_i(X)}$, with no intervening $\mathbf{u_i(X)}$.

Legal schedules:

- No two exclusive locks on the same element.

  If $\mathbf{xL_i(X)}$ appears in a schedule, then there cannot be a $\mathbf{xL_j(X)}$ until after a $\mathbf{u_i(X)}$ appears.
- No shared locks on exclusively locked element.

  If $\mathbf{xL_i(X)}$ appears, there can be no $\mathbf{sL_j(X)}$ until after $\mathbf{u_i(X)}$.
- No writing in shared lock mode

  If $\mathbf{sL_i(X)}$ appears, there can be no $\mathbf{w_j(X)}$ until after $\mathbf{u_i(X)}$.

2PL condition:

- No transaction may have a $\mathbf{sL(X)}$ or $\mathbf{xL(X)}$ after a $\mathbf{u(Y)}$.

Scheduler rules for shared/exclusive locks

- When there is more than one kind of lock, the scheduler needs a rule that says **"if there is already a lock of type A on DB element X, can I grant a lock of type B on X?"**

- The compatibility matrix answers the question.

Compatibility Matrix for Shared/Exclusive Locks

	S	X
S	yes	no
X	no	no

Scheduling with locks: example

r1(A); r2(B); r3(C); r1(B); r2(C); r3(D); w1(A); w2(B); w3(C);

T1	T2	T3
xl(A); r1(A)		
	xl(B); r2(B)	
		xl(C); r3(C)
sl(B) **denied**		
	sl(C) **denied**	
		sl(D); r3(D); ul(D)
w1(A);		
	w2(B);	
		w3(C); ul(C)
	sl(C); r2(C); ul(B); ul(C)	
sl(B); r1(B); ul(A); ul(B)		

Upgrading Locks

- Instead of taking an exclusive lock immediately, a transaction can take a *shared* lock on X, read X, and then upgrade the lock to *exclusive* so that it can write X.

T1	T2
sl1(A); r1(A);	
	sl2(A); r2(A);
	sl2(B); r2(B);
sl1(B); r1(B);	
xl1(B) Denied	
	ul2(A); ul2(B);
xl1(B); w1(B);	
ul1(A); ul1(B);	

Upgrading Locks allows more concurrent operations:

Had T1 asked for an exclusive lock on B before reading B, the request would have been denied, because T2 already has a shared lock on B.

Scheduling with upgrade locks: example

r1(A); r2(B); r3(C); r1(B); r2(C); r3(D); w1(A); w2(B); w3(C);

T1	T2	T3
sl(A); r1(A);		
	sl(B); r2(B);	
		sl(C); r3(C);
sl(B); r1(B);		
	sl(C); r2(C);	
		sl(D); r3(D);
xl(A); w1(A); ul(A); ul(B);		
	xl(B); w2(B); ul(B); ul(C);	
		xl(C); w3(C); ul(C); ul(D);

Compared to slide 38: no waiting

Possibility of Deadlocks

Example:T1 and T2 each reads X and later writes X.

T1	T2
sL1(X)	
	sL2(X)
xL1(X) denied	
	xL2(X) denied

Problem: when we allow upgrades, it is easy to get into a deadlock situation.

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Possible solution: Update Locks

Update lock $udL_i(X)$

- Only an **update lock** (**not shared lock**) can be upgraded to **exclusive lock** (if there are no shared locks anymore).
- A transaction that will read and later on write some element A, **asks initially for an update** lock on A, and then asks for an exclusive lock on A. Such transaction doesn't ask for a shared lock on A.

Legal schedules

- Read action permitted when there is either a shared or update lock.
- An update lock can be granted while there is a shared lock, but the scheduler will not grant a shared lock when there is an update lock.

2PL condition

- No transaction may have an **sl(X), udl(X)** or **xl(X)** after a **u(Y)**.

Update Locks: scheduler rules

Compatibility Matrix for Shared/Exclusive/Update Locks

	S	X	U
S	yes	no	yes
X	no	no	no
U	no	no	no

Schedule with update locks: example

T1	T2	T3
sL(A); r(A)		
	udL(A); r(A)	
		sL(A) **Denied**
	xL(A) **Denied**	
u(A)		
	xL(A); w(A)	
	u(A)	
		sL(A); r(A)
		u(A)

(No) Deadlock Example

T_1 and T_2 each read X and later write X.

T1	T2
sL1(X);	
	sL2(X);
xL1(X); denied	
	xL2(X); denied

Deadlock when using sL and xL locks only.

T1	T2
udl1(X); r(X);	
	udL2(X); denied
xL1(X); w(X); u(X);	
	udl2(X); r2(X); xl2(X); w2(X); u2(X)

Fine when using update locks.

Scheduling with 3 types of locks: example

r1(A); r2(B); r3(C); r1(B); r2(C); r3(D); w1(A); w2(B); w3(C);

T1	T2	T3
uL(A); r1(A);		
	uL(B); r2(B);	
		uL(C); r3(C);
sL(B); **denied**		
	sL(C); **denied**	
		sL(D); r3(D);
xl(A); w1(A);		
	xL(B); w2(B);	
		xL(C); w3(C);
		uL(D); uL(C);
	sL(C); r2(C);	
	uL(B); uL(C);	
sL(B); r1(B);		
uL(A); uL(B);		

Benefits of Update Locks

sl – shared lock
udl – update lock
xl – exclusive lock
u - unlock

T1	T2	T3	T4	T5	T6	T7	T8	T9
sl(A);r(A)								
	sl(A);r(A)							
		sl(A);r(A)						
			sl(A);r(A)					
				udl(A);r(A)				
					sl(A);denied			
						sl(A);denied		
							sl(A);denied	
								sl(A);denied
u(A)								
	u(A)							
		u(A)						
			u(A)					
				xl(A);w(A)				
				u(A)				
					s(A);r(A)			
						s(A);r(A)		
							s(A);r(A)	
								s(A);r(A)