

Transactions

1

Overview

- Transaction: A sequence of database actions enclosed within special tags
- Properties:
 - Atomicity: Entire transaction or nothing
 - Consistency: Transaction, executed completely, takes database from one consistent state to another
 - Isolation: Concurrent transactions appear to run in isolation
 - Durability: Effects of committed transactions are not lost
- Consistency: Programmer needs to guarantee this
 - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees

2

How does..

- .. this relate to *queries* that we discussed ?
 - Queries don't update data, so durability and consistency not relevant
 - Would want concurrency
 - Consider a query computing balance at the end of the day
 - Would want isolation
 - What if somebody makes a *transfer* while we are computing the balance
 - Typically not guaranteed for such long-running queries
- TPC-C vs TPC-H
 - data entry vs decision support

3

Assumptions and Goals

- Assumptions:
 - The system can crash at any time
 - Similarly, the power can go out at any point
 - Contents of the main memory won't survive a crash, or power outage
 - BUT... **disks are durable. They might stop, but data is not lost.**
 - For now.
 - Disks only guarantee *atomic sector writes*, nothing more
 - Transactions are by themselves consistent
- Goals:
 - Guaranteed durability, atomicity
 - As much concurrency as possible, while not compromising isolation and/or consistency
 - Two transactions updating the same account balance... NO
 - Two transactions updating different account balances... YES

4

Next...

- **Concurrency control schemes**
 - A CC scheme is used to guarantee that concurrency does not lead to problems
 - For simplicity, we will ignore durability during this section
 - So no crashes
 - Though transactions may still abort
- **Schedules**
- **When is concurrency okay ?**
 - Serial schedules
 - Serializability

5

A Schedule

Transactions:

T1: transfers \$50 from A to B

T2: transfers 10% of A to B

Database constraint: $A + B$ is constant (*checking+saving accts*)

T1	T2
read(A)	
$A = A - 50$	
write(A)	
read(B)	
$B = B + 50$	
write(B)	
	read(A)
	$tmp = A * 0.1$
	$A = A - tmp$
	write(A)
	read(B)
	$B = B + tmp$
	write(B)

Effect:

	<u>Before</u>	<u>After</u>
A	100	45
B	50	105

Each transaction obeys the constraint.

The schedule does too.

6

Schedules

- A *schedule* is simply a (possibly interleaved) execution sequence of transaction instructions
- *Serial Schedule*: A schedule in which transactions appear one after the other
 - i.e., No interleaving
- Serial schedules satisfy isolation and consistency
 - Since each transaction by itself does not introduce inconsistency

7

Another serial schedule

T1	T2	Effect:		
	read(A)		<u>Before</u>	<u>After</u>
	tmp = A*0.1	A	100	40
	A = A - tmp	B	50	110
	write(A)			
	read(B)			
	B = B + tmp			
	write(B)			
read(A)				
A = A - 50				
write(A)				
read(B)				
B = B + 50				
write(B)				

Consistent ?
Constraint is satisfied.

Since each Xion is consistent, any
serial schedule must be consistent

8

Another schedule

T1	T2
read(A) A = A - 50 write(A)	
	read(A) tmp = A * 0.1 A = A - tmp write(A)
read(B) B = B + 50 write(B)	
	read(B) B = B + tmp write(B)

Is this schedule okay ?

Lets look at the final effect...

Effect:	<u>Before</u>	<u>After</u>
A	100	45
B	50	105

Consistent.

So this schedule is okay too.

9

Another schedule

T1	T2
read(A) A = A - 50 write(A)	
	read(A) tmp = A * 0.1 A = A - tmp write(A)
read(B) B = B + 50 write(B)	
	read(B) B = B + tmp write(B)

Is this schedule okay ?

Lets look at the final effect...

Effect:	<u>Before</u>	<u>After</u>
A	100	45
B	50	105

Further, the effect same as the serial schedule 1.

Called serializable

10

Example Schedules (Cont.)

A “bad” schedule

T1	T2			
read(A) A = A - 50		Effect:	<u>Before</u>	<u>After</u>
	read(A)	A	100	50
	tmp = A * 0.1	B	50	60
	A = A - tmp			
	write(A)			
	read(B)			
		<u>Not consistent</u>		
write(A) read(B) B = B + 50 write(B)				
	B = B + tmp write(B)			

11

Serializability

- A schedule is called *serializable* if:
 - *its final effect is the same as that of a serial schedule*
- Serializability → database remains consistent
 - Since serial schedules are fine
- Non-serializable schedules are unlikely to result in consistent databases
- We will ensure serializability
 - *Though often relaxed in real high-throughput environments...*

12

Serializability

- Not possible to look at all $n!$ serial schedules to check if the effect is the same
 - Instead ensure serializability by disallowing certain schedules
- Conflict serializability
- View serializability
 - allows more schedules

13

Conflict Serializability

- Two read/write instructions “conflict” if
 - They are by different transactions
 - They operate on the same data item
 - At least one is a “write” instruction
- Why do we care ?
 - If two read/write instructions don’t conflict, they can be “swapped” without any change in the final effect
 - If they conflict they CAN’T be swapped

14

Equivalence by Swapping

T1	T2	T1	T2
read(A) A = A - 50 write(A)	read(A) tmp = A * 0.1 A = A - tmp write(A)	read(A) A = A - 50 write(A)	read(A) tmp = A * 0.1 A = A - tmp
read(B) B = B + 50 write(B)	read(B) B = B + tmp write(B)	read(B) B = B + 50 write(B)	write(A) read(B) B = B + tmp write(B)
Effect: A <u>Before</u> <u>After</u> 100 45 B 50 105		Effect: A <u>Before</u> <u>After</u> 100 45 B 50 105	
==			

15

Equivalence by Swapping

T1	T2	T1	T2
read(A) A = A - 50 write(A)	read(A) tmp = A * 0.1 A = A - tmp write(A)	read(A) A = A - 50 write(A)	read(A) tmp = A * 0.1 A = A - tmp write(A)
read(B) B = B + 50 write(B)	read(B) B = B + tmp write(B)	read(B) B = B + 50	read(B) B = B + tmp write(B)
Effect: A <u>Before</u> <u>After</u> 100 45 B 50 105		Effect: A <u>Before</u> <u>After</u> 100 45 B 50 55	
! ==			

16

Conflict Serializability

- **Conflict-equivalent schedules:**
 - If S can be transformed into S' through a series of swaps, S and S' are called *conflict-equivalent*
 - *conflict-equivalence guarantees same final effect on database*
- A schedule S is *conflict-serializable* if it is conflict-equivalent to a serial schedule

17

Equivalence by Swapping

T1	T2	T1	T2
read(A) A = A - 50 write(A)		read(A) A = A - 50 write(A)	
	read(A) tmp = A * 0.1 A = A - tmp write(A)		read(A) tmp = A * 0.1 A = A - tmp
read(B) B = B + 50 write(B)		read(B) B = B + 50 write(B)	write(A)
	read(B) B = B + tmp write(B)		read(B) B = B + tmp write(B)
Effect: <u>Before</u> <u>After</u>		Effect: <u>Before</u> <u>After</u>	
A 100 45		A 100 45	
B 50 105	==	B 50 105	

18

Equivalence by Swapping

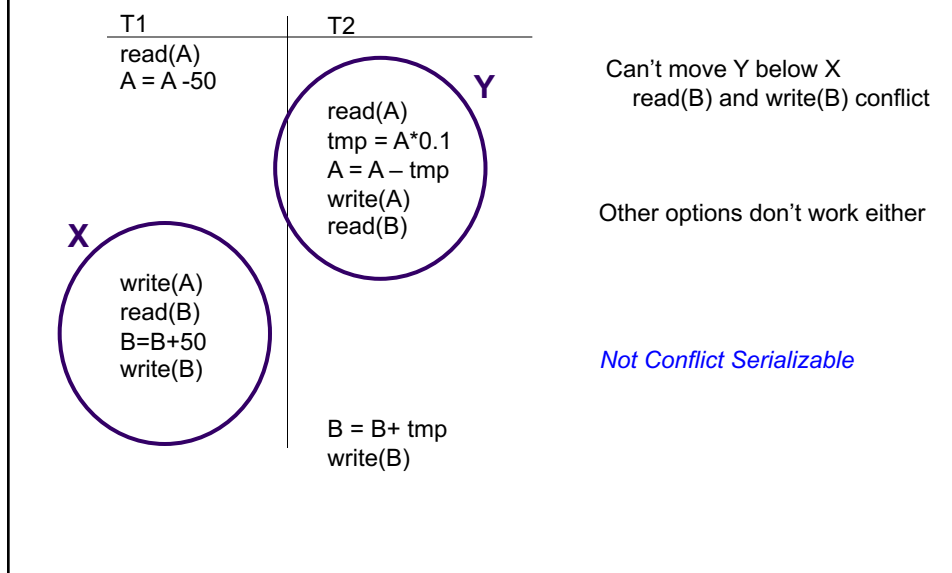
T1	T2	T1	T2
read(A) A = A - 50 write(A)		read(A) A = A - 50 write(A)	
	read(A) tmp = A*0.1 A = A - tmp write(A)	read(B) B=B+50 write(B)	
read(B) B=B+50 write(B)			read(A) tmp = A*0.1 A = A - tmp write(A)
	read(B) B = B+ tmp write(B)		read(B) B = B+ tmp write(B)
Effect:	<u>Before</u> <u>After</u>	Effect:	<u>Before</u> <u>After</u>
A	100 45	A	100 45
B	50 105	B	50 105

==

19

Example Schedules (Cont.)

A "bad" schedule



20

View-Serializability

- Following not conflict-serializable

T_3	T_4	T_6
read(Q)	write(Q)	
write(Q)		
		write(Q)

BUT, it is serializable

- The *conflicting write instructions don't matter!* (in absence of reads)
 - The final write is the only one that matters
- View-serializability, for S' and S , and each datum Q :
 - if T_i reads initial value of Q in S , must also in S'
 - if T_i reads value written from T_j in S , must also in S'
 - if T_i performs final write to Q in S , must also in S'

21

Other notions of serializability

T_1	T_5
read(A)	read(B) $B := B - 10$ write(B)
$A := A - 50$	
write(A)	
read(B)	read(A) $A := A + 10$ write(A)
$B := B + 50$	
write(B)	

- Not conflict-serializable or view-serializable, but serializable
- Mainly because of the +/- only operations
 - Requires analysis of the actual operations, not just read/write operations
- Most high-performance transaction systems will allow these

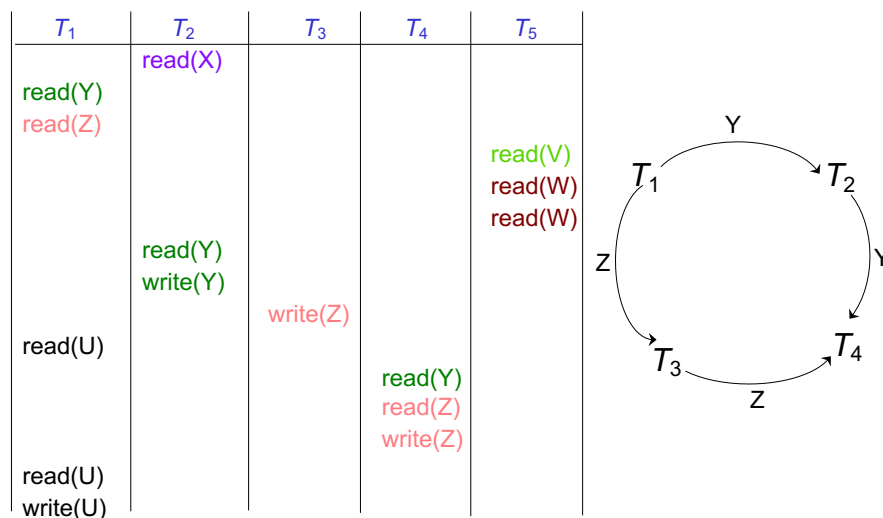
22

Testing for conflict-serializability

- Given a schedule, determine if it is conflict-serializable
- Draw a *precedence-graph* over the transactions
 - A directed edge from T_1 to T_2 , if
 - they have conflicting instructions, and
 - T_1 's conflicting instruction comes first
- If there is a cycle in the graph, not conflict-serializable
 - Can be checked in at most $O(n+e)$ time, where n is the number of vertices, and e is the number of edges
- If there is none, conflict-serializable
- Whereas: testing for view-serializability is NP-hard.

23

Example Schedule (Schedule A) + Precedence Graph



24

Recap so far...

- We discussed:
 - Serial schedules, serializability
 - Conflict-serializability, view-serializability
 - How to check for conflict-serializability
- We haven't discussed:
 - How to guarantee serializability ?
 - Allowing transactions to run, and then aborting them if the schedules aren't serializable can be expensive
 - We can instead use schemes to guarantee that the schedule will be conflict-serializable
 - Also, recoverability ?

25

Recoverability

- Serializability is good for consistency
- What if transactions fail ?
 - T2 has already committed
 - A user might have been notified
 - Now T1 abort creates a problem
 - T2 has seen its effect, so just aborting T1 is not enough. T2 must be aborted as well (and possibly restarted)
 - But T2 is *committed*

T1	T2
read(A) A = A - 50 write(A)	read(A) tmp = A * 0.1 A = A - tmp write(A) COMMIT
read(B) B = B + 50 write(B) ABORT	

26

Recoverability

- **Recoverable** schedule: If T1 has read something T2 has written, T2 must commit before T1
 - Otherwise, if T1 commits, and T2 aborts, we have a problem
- **Cascading rollbacks**: If T10 aborts, T11 must abort, and hence T12 must abort and so on.

T_{10}	T_{11}	T_{12}
read(A) read(B) write(A) abort	 read(A) write(A)	 read(A)

27

Recoverability

- **Dirty read**: Reading a value written by a transaction that hasn't committed yet
- **Cascadeless** schedules:
 - A transaction only reads *committed* values.
 - So if T1 has written A, but not committed it, T2 can't read it.
 - No dirty reads
- **Cascadeless** → No cascading rollbacks
 - That's good
 - We will try to guarantee that as well

28

Recap so far...

- We discussed:
 - Serial schedules, serializability
 - Conflict-serializability, view-serializability
 - How to check for conflict-serializability
 - Recoverability, cascade-less schedules
- We haven't discussed:
 - How to guarantee serializability ?
 - Allowing transactions to run, and then aborting them if the schedules aren't serializable can be expensive
 - We can instead use schemes to guarantee that the schedule will be conflict-serializable
 - Hint: *locks*

29

Concurrency Control

30

Approach, Assumptions etc..

- Approach
 - Guarantee conflict-serializability by limiting concurrency
 - Lock-based
- Assumptions:
 - Still ignoring durability
 - So no crashes
 - Though transactions may still abort
- Goal:
 - Serializability
 - Minimize the bad effect of aborts (cascade-less schedules only)

31

Lock-based Protocols

- Transactions must *acquire* locks before using data
- Two types:
 - *Shared (S) locks* (also called *read locks*)
 - Obtained if we want to only read an item
 - *Exclusive (X) locks* (also called *write locks*)
 - Obtained for updating a data item

32

Lock instructions

- New instructions

- lock-S: shared lock request
- lock-X: exclusive lock request
- unlock: release previously held lock

Example schedule:

T1	T2
read(B)	read(A)
$B \leftarrow B - 50$	read(B)
write(B)	display(A+B)
read(A)	
$A \leftarrow A + 50$	
write(A)	

33

Lock instructions

- New instructions

- lock-S: shared lock request
- lock-X: exclusive lock request
- unlock: release previously held lock

Example schedule:

T1	T2
lock-X(B)	lock-S(A)
read(B)	read(A)
$B \leftarrow B - 50$	unlock(A)
write(B)	lock-S(B)
unlock(B)	read(B)
lock-X(A)	unlock(B)
read(A)	display(A+B)
$A \leftarrow A + 50$	
write(A)	
unlock(A)	

34

Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
 - It decides whether to *grant* a lock request
- Assume T2 holds lock, T1 asks for a lock on same:

<u>Held lock</u>	<u>Lock wanted</u>	<u>Allow?</u>
Shared	Shared	YES
Shared	Exclusive	NO
Exclusive	-	NO

- If *compatible*, grant the lock, otherwise T1 waits in a *queue*.

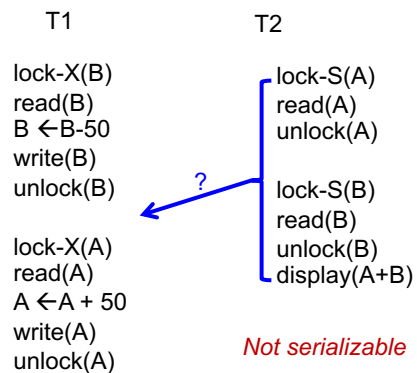
35

Lock instructions

- New instructions
 - **lock-S**: shared lock request
 - **lock-X**: exclusive lock request
 - **unlock**: release previously held lock

Not enough to take minimum locks when you need to read/write something!

Example schedule:



36

2-Phase Locking Protocol (2PL)

- Phase 1: Growing phase
 - Transaction may obtain locks
 - But may not release them
- Phase 2: Shrinking phase
 - Only release locks
- 2PL guarantees *conflict-serializability*
 - *lock-point*: the time at which a transaction acquired last lock
 - if $\text{lock-point}(T1) < \text{lock-point}(T2)$, there can't be an edge from T2 to T1 in the *precedence graph*

T1
 lock-X(B)
 read(B)
 $B \leftarrow B - 50$
 write(B)
 lock-X(A)
 unlock(B)
 read(A)
 $A \leftarrow A + 50$
 write(A)
 unlock(A)

37

2 Phase Locking

- Example: T1 in 2PL

Growing phase

Shrinking phase

T1
lock-X(B)
read(B)
$B \leftarrow B - 50$
write(B)
lock-X(A)
read(A)
$A \leftarrow A - 50$
write(A)
unlock(B)
unlock(A)

38

2 Phase Locking

- Guarantees *conflict-serializability*,
 - but *not cascade-less recoverability*

T1	T2	T3
lock-X(A), lock-S(B) read(A) read(B) write(A) unlock(A), unlock(B)	lock-X(A) read(A) write(A) unlock(A) commit	lock-S(A) read(A) commit
<xction fails>		

39

2 Phase Locking

- Guarantees *conflict-serializability*,
 - but *not recoverability*
 - and *cascades can still happen*
- Guaranteeing just recoverability:
 - If T2 performs a dirty read from T1, T2 can't commit unless T1 either commits or aborts
 - If T1 commits, T2 can proceed with committing
 - If T1 aborts, T2 must abort
- So cascades still happen

40

Strict 2PL

- Release *exclusive* locks only at the very end, together with commit or abort

	T1	T2	T3
Strict 2PL will not allow that	lock-X(A), lock-S(B) read(A) read(B) write(A) unlock(A), unlock(B)	lock-X(A) read(A) write(A) unlock(A) Commit	lock-S(A) read(A) Commit
	<xction fails>		

41

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort

T1	T2	T3
lock-X(A), lock-S(B) read(A) read(B) write(A) unlock(A), unlock(B) commit	lock-X(A) read(A) write(A) unlock(A) commit	lock-S(A) read(A) commit

Works. *Guarantees cascade-less and recoverable schedules.*

42

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort
 - Read locks are ignored
- **Rigorous 2PL:** Release both *exclusive and read* locks only at the very end
 - Makes serializability order === the commit order
 - More intuitive behavior for the users
 - No difference for the system

43

Strict 2PL

- **Lock conversion:**
 - Transaction might not be sure what it needs a write lock on
 - Start with a S lock
 - *Upgrade* to an X lock later if needed
 - Doesn't change any of the other properties of the protocol

44

Implementation of Locking

- A separate process, or a separate module
- Uses a *lock table* to keep track of currently assigned locks and the requests for locks
 - Read up in the book

45

Recap so far...

- Concurrency Control Scheme
 - A way to guarantee serializability, recoverability etc
- Lock-based protocols
 - Use *locks* to prevent multiple transactions accessing the same data items
- 2 Phase Locking
 - Locks acquired during *growing phase*, released during *shrinking phase*
- Strict 2PL, Rigorous 2PL

46