Transactions

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer $50 from account A to account B:

```
begin
    read(A)
    A := A – 50
    write(A)
    read(B)
    B := B + 50
    write(B)
end
```

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

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

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

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* *<u>sector</u>* *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

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

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)

time

Effect:		Before	After
	A	100	45
	B	50	105

Each transaction obeys the constraint.

The schedule does too.

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

Another serial schedule

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

Effect:	Before	After
A	100	40
B	50	110

Consistent ?

Constraint is satisfied.

Since each Xion is consistent, any serial schedule is also consistent

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:	Before	After
A	100	45
B	50	105

Consistent.

So this schedule is okay too.

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:	Before	After
A	100	45
B	50	105

Further, the effect same as the serial schedule 1.

Called *serializable*

Example Schedules (Cont.)

A "bad" schedule

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

Effect:	Before	After
A	100	50
B	50	60

Not consistent

Serializability (chapters 17+18)

- 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 typically relaxed in real high-throughput environments…*

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

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

Equivalence by Swapping

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)

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

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

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)

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	
	read(B)
write(B)	
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

! ==

Effect:	Before	After
A	100	45
B	50	55

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

Equivalence by Swapping

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)

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

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

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)

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:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Example Schedules (Cont.)

A “bad” schedule

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

Can’t move Y below X
read(B) and write(B) conflict

Other options don’t work either

Not Conflict Serializable

View-Serializability

- Similarly, following not conflict-serializable

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

BUT, it is serializable

- Intuitively, this is because 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’

Other notions of serializability

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

- 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
- *Conflict-Free Replicated Data Types* (CRDTs)

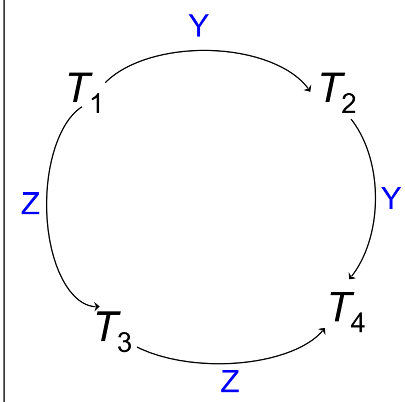
Testing for conflict-serializability

1. Draw a *precedence-graph* over the transactions:
   - A directed edge from T1 to T2, iff:
     - they have conflicting instructions, and
     - T1's conflicting instruction executed first
2. 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
3. If there is none, conflict-serializable

- Testing for view-serializability is NP-hard.

Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



No cycle, so
conflict-serializable