Transactions; Concurrency; Recovery

Amol Deshpande
CMSC424

Spring 2020 – Online Instruction Plan

- Week 1: File Organization and Indexes
- Week 2: Query Processing
- Week 3: Query Optimization; Parallel Databases 1
- Week 4: Parallel Databases; Mapreduce; Transactions 1
  - Map-reduce and Apache Spark
  - Parallel Databases 2: Execution and Other Issues
  - Transactions 1: ACID, SQL Transactions
  - Homework Due April 24
- Week 5: Transactions 2 (Homework Due May 1)
- Week 6: Miscellaneous Topics (Reading Homework Due May 8)

Transactions: Overview

- Book Chapters
  - 14.1, 14.2, 14.3, 14.4, 14.5
- Key topics:
  - Transactions and ACID Properties
  - Different states a transaction goes through
  - Notion of a ”Schedule”
  - Introduction to Serializability

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:
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)
- 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: Transaction programmer needs to guarantee that
  - 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 total 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

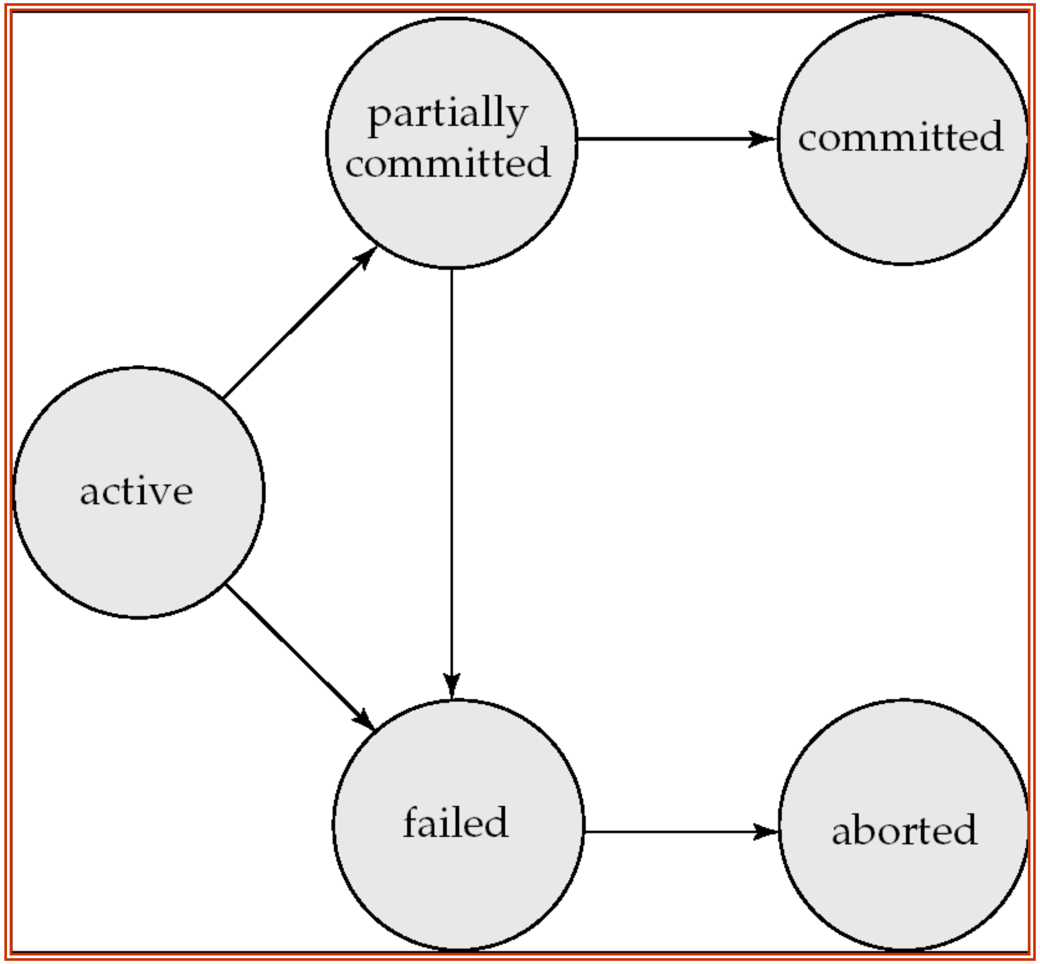
Assumptions and Goals

- 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

Transaction states

Initial State – stays in this during execution



Successful Completion

Any changes have been rolled back

Next…

- Concurrency: Why?
  - Increased processor and disk utilization
  - Reduced average response times
- Concurrency control schemes
  - A CC scheme is used to guarantee that concurrency does not lead to problems
  - For now, we will assume durability is not a problem
    - 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)

Effect:	Before	After
A	100	45
B	50	105

Each transaction obeys the constraint.

This schedule does too.

Schedules

- A *schedule* is simply a (possibly interleaved) execution sequence of transaction instructions

- *Serial Schedule:* A schedule in which transaction appear one after the other
  - ie., No interleaving

- Serial schedules satisfy isolation and consistency
  - Since each transaction by itself does not introduce inconsistency

Example Schedule

- 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 must be 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

- A schedule is called *serializable* if its final effect is the same as that of a *serial schedule*

- Serializability → schedule is fine and doesn't cause inconsistencies
  - Since serial schedules are fine

- Non-serializable schedules unlikely to result in consistent databases

- We will ensure serializability
  - Typically relaxed in real high-throughput environments

- Not possible to look at all *n!* serial schedules to check if the effect is the same
  - Instead we ensure serializability by allowing or not allowing certain schedules

Example Schedule with More Transactions

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)				

Summary

- Transactions is how we update data in databases

- ACID properties: foundations on which high-performance transaction processing systems are built
  - From the beginning, consistency has been a key requirement
  - Although "relaxed" consistency is acceptable in many cases (originally laid out in 1975)

- NoSQL systems originally eschewed ACID properties
  - MongoDB was famously bad at guaranteeing any of the properties
  - Lot of focus on what's called "eventual consistency"

- Recognition today that strict ACID is more important than that
  - Hard to build any business logic if you have no idea if your transactions are consistent