Transactions

Overview

- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - ***A**tomicity*: Entire transaction or nothing
  - ***C**onsistency*: Transaction, executed completely, takes database from one consistent state to another
  - ***I**solation*: Concurrent transactions *appear* to run in isolation
  - ***D**urability*: 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..

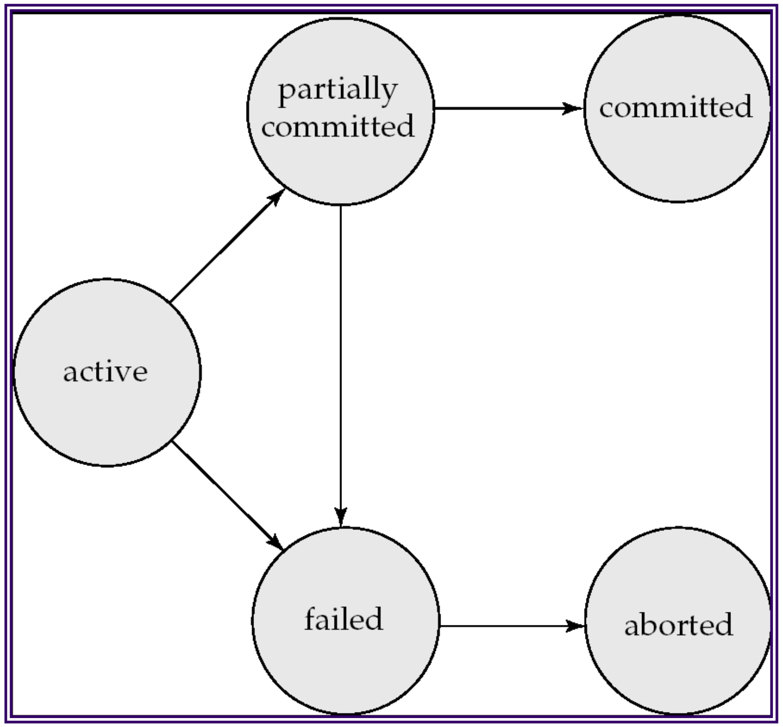
- .. this relate to *queries* that we discussed ?
  - Queries don’t update data, so *durability* and *consistency* not relevant
  - Would want *concurrency*
    - Consider a query computing balance at the end of the day
  - Would want *isolation*
    - What if somebody makes a *transfer* while we are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

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.**
    - Modern apps don't assume this; replicate data
  - 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

- active – initial state, while executing
- partially committed – after final statement
- failed – after discover that can not proceed
- aborted – after rolled back and DB restored
- committed – after successful completion



Next…

- Concurrency control schemes
    - A CC scheme is used to guarantee that concurrency does not lead to problems
    - For simplicity, we will ignore durability for now
        - So no crashes
        - 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.

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
    - If each xact is consistent, then whole app will remain consistent
    - But Slow
        - Processor resources being wasted
        - Disk / memory resources may be wasted
        - Throughput and latency suboptimal

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

Constraint on A+B is satisfied.

Since each Xact maintains constraint, any serial schedule maintains constraint

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

Same as first schedule we saw.
So isolation is maintained.
So this schedule OK in retrospect.

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

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

Constraint on A + B violated

Isolation violated

Not serializable

Serializability

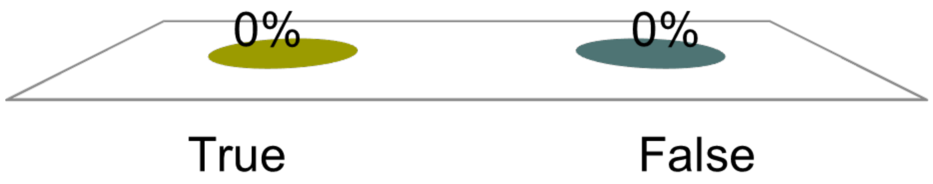
- A schedule is called *serializable* if:
  - *its final effect is the same as that of a serial schedule*
- Serializability + individual xacts maintain constraints→ app maintains constraints
- App programmers very good at reasoning about consistency of individual xacts
  - But very bad at reasoning across xacts (much harder to think about)
  - Database that guarantees serializability much easier for app developer
    - App constraints often get violated without serializability guarantees
    - *But serializability comes with a performance cost*
      - *Commercial DB systems allow relaxed isolation levels*
      - *Must think very hard before using them*

Is this schedule equivalent to serial sechedule?

[✓ A. True]

B. False

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	A = A – 20
	write(A)
read(A)	
A=A+5	
write(A)	
	read(B)
	B = B+ 42
	write(B)



Have picture want to share, but want wife not to see it. Issue 2 xacts:

- Transaction 1: Remove wife’s permissions to see my “Pictures” folder
- Transaction 2: Upload pic to Pictures folder

Serializable database will prevent my wife from seeing pic if issue 2 xacts:

A. True

[✓ B. False]

- Transaction 1: Remove wife's permissions to see my "Pictures" folder
- Transaction 2: Upload pic to "Pictures" folder

Serializable DBs can order transactions however they want!!!!!

If want to maintain order, must place these two requests in same xact, or use **strictly serializable** DB (see slide coming up soon on that subject) and wait until xact 1 commits before issuing xact 2.

Have picture want to share, but want wife not to see it. Issue 2 xacts:

- Transaction 1: Remove wife’s permissions to see my “Pictures” folder
- WAIT FOR CONFIRMATION THAT XACT 1 COMMITTED!!!!!
- Transaction 2: Upload pic to Pictures folder

Serializable database will prevent my wife from seeing pic if issue:

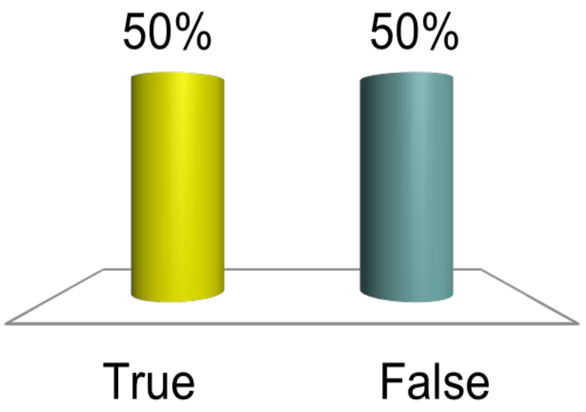
A. True

[✓ B. False]

- Transaction 1: Remove wife’s permissions to see my “Pictures” folder
- WAIT FOR CONFIRMATION THAT XACT 1 COMMITTED!!!!!
- Transaction 2: Upload pic to Pictures folder

Serializable DBs can order transactions however they want!!!!!

See next 2 slides which explain this example.



Explaining the example

- T1: Write(P) (P is permissions)
  commit
- T2: Write(F) (F is the folder where the pic goes)
  commit
- T3: Query run by wife:

```
Read(P)
IF (P == true)
  Read(F)
commit
```

Serializability says we can run equivalent to any serial order. System can choose T2, T3, T1.

Even if T1 committed before T2 started!

(nothing from the definition of serializability prevents this!)

Serializability by itself isn't that useful

- Most users have an expectation of notion of earlier and later for transactions
  - A database that returns the empty set for every query is serializable
    - Because initial database was empty
    - This is not useful
- Strict serializability guarantees that all transactions submitted to DB after committed xact, see writes of that committed xact
  - Not discussed in your textbook, but important to know about

Not on exam!!

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

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)

Effect:	Before	After
A	100	45
B	50	105

==

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

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

Another example

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, but serializable
- Mainly because of the +/- only operations
  - Requires analysis of the actual operations, not just read/write operations

Conflict serializability: why we care

- All conflict serializable schedules are serializable
- Really easy to enforce conflict serializable schedules with locking

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)

Example Schedules (Cont.)

A "bad" schedule

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

Transaction blocks

(can't get lock on A until T1 completes)

Will not be schedules with locks on exam!!

Enforce conflict serializability with locking

- Transactions must *acquire* locks before using data
  - locking usually handled by transaction statements
- Two types:
  - *Shared* (S) locks (*read locks)*
    - Obtained if we want to only read an item
  - *Exclusive* (X) locks (*write locks)*
    - Obtained for updating a data item
- Transactions *release* locks after commit
  - Can sometimes release earlier, but most DBs wait until after commit
  - This is called strict 2PL

Reduced Isolation Levels

- READ UNCOMMITTED
  - OK to read uncommitted data
- READ COMMITTED
  - Only read committed data, but reads might not be repeatable
- REPEATABLE READ
  - Read committed + reading a data item twice results in same value
  - But if iterate through table multiple times, may see new data inserted in later iterations
- SERIALIZABLE

Databases: So much more to learn!!

- How to recover partially completed transactions after crash?
- How to write multi-threaded transaction managers?
- How to physically lay out data on storage?
- How does the DBMS buffer manager work?
- How to guarantee ACID across a distributed cluster of machines?
- NoSQL DBs, and how do they differ from SQL DBs?
- How do Google, Amazon, and other large apps manage their large amount of data?
- Find out by taking CMSC 624 / 828N (two different numbers for same class) next semester!
  - Class size will be 20-30, so if you are sick of large classes, this is another reason to take CMSC 624 / 828N
  - If you get an A in CMSC 424 with me, you will definitely be granted permission to take CMSC 624 / 828N
    - All other requests will be handled on a case-by-case basis.