Transaction Processing

Introduction to Databases

CompSci 316 Fall 2016

DUKE
COMPUTER SCIENCE

Announcements (Thu., Dec. 1)

- Homework #4 due next Tuesday
  - Except the last Gradiance problem (due Thursday)
- Project demos—please sign up via Google Doc!
  - Early in-class demos on 12/8
- Final exam Thur. Dec. 15 7-10pm
  - **Different room: LSRC B101**
  - Open-book, open-notes
  - Comprehensive, but with strong emphasis on the second half of the course
  - Sample final to be posted soon

Announcements (Tue., Dec. 6)

- Homework #4 due today
  - Except the last Gradiance problem (due Thursday)
- Project demos to start this Friday
  - Final schedule to be emailed soon
  - Nobody signed up for early in-class demo!
- Final exam Thur. Dec. 15 7-10pm
  - **Different room: LSRC B101**
  - Open-book, open-notes
  - Comprehensive, but with strong emphasis on the second half of the course
  - Sample final posted on Sakai (solution to be posed soon)

Review

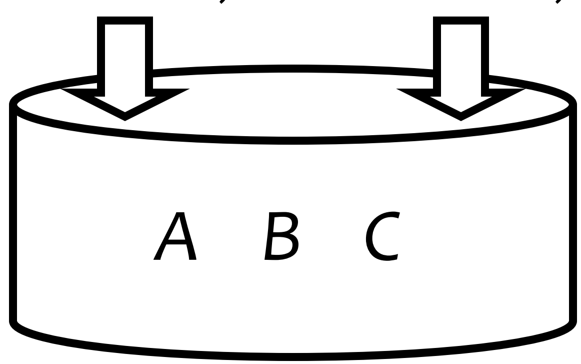
- ACID
  - Atomicity: TX's are either completely done or not done at all
  - Consistency: TX's should leave the database in a consistent state
  - Isolation: TX's must behave as if they are executed in isolation
  - Durability: Effects of committed TX's are resilient against failures
- SQL transactions

```
-- Begins implicitly
SELECT …;
UPDATE …;
ROLLBACK | COMMIT;
```

Concurrency control

- Goal: ensure the "I" (isolation) in ACID

T_1:	T_2:
read(A);	read(A);
write(A);	write(A);
read(B);	read(C);
write(B);	write(C);
commit;	commit;



Good versus bad schedules

Good!

T_1	T_2
r(A)	
w(A)	
r(B)	
w(B)	
	r(A)
	w(A)
	r(C)
	w(C)

Bad!

	T_1	T_2	
Read 400	r(A)		
		r(A)	Read 400
Write 400 – 100	w(A)		
		w(A)	Write 400 – 50
	r(B)		
		r(C)	
	w(B)		
		w(C)	

Good! (But why?)

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
	r(C)
w(B)	
	w(C)

Serial schedule

- Execute transactions in order, with no interleaving of operations
  - T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B), T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C)
  - T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B)

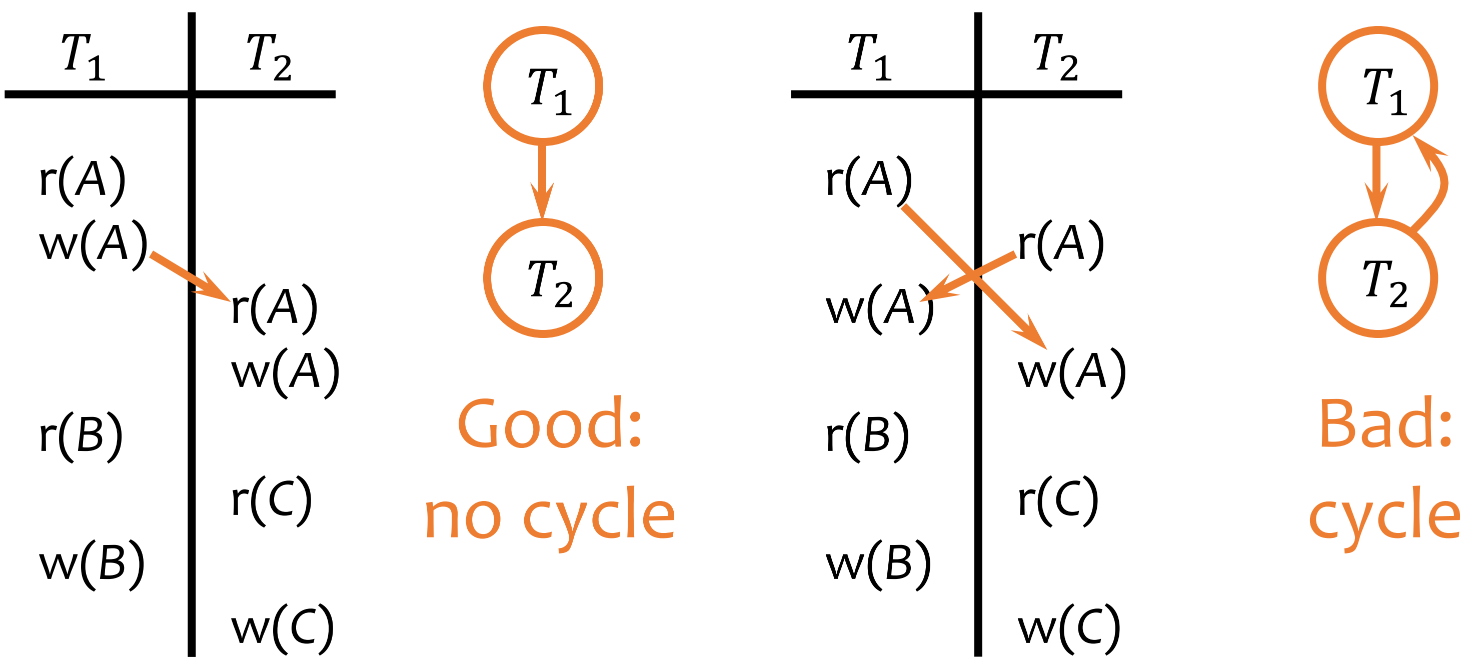
  ☞ Isolation achieved by definition!
- Problem: no concurrency at all
- Question: how to reorder operations to allow more concurrency

Conflicting operations

- Two operations on the same data item conflict if at least one of the operations is a write
  - r(X) and w(X) conflict
  - w(X) and r(X) conflict
  - w(X) and w(X) conflict
  - r(X) and r(X) do not conflict
  - r/w(X) and r/w(Y) do not conflict
- Order of conflicting operations matters
  - E.g., if T_1.r(A) precedes T_2.w(A), then conceptually, T_1 should precede T_2

Precedence graph

- A node for each transaction
- A directed edge from T_i to T_j if an operation of T_i precedes and conflicts with an operation of T_j in the schedule



Conflict-serializable schedule

- A schedule is conflict-serializable iff its precedence graph has no cycles
- A conflict-serializable schedule is equivalent to some serial schedule (and therefore is “good”)
  - In that serial schedule, transactions are executed in the topological order of the precedence graph
  - You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions

Locking

- Rules
  - If a transaction wants to read an object, it must first request a shared lock (S mode) on that object
  - If a transaction wants to modify an object, it must first request an exclusive lock (X mode) on that object
  - Allow one exclusive lock, or multiple shared locks

Mode of the lock requested (columns); *Mode of lock(s) currently held by other transactions* (rows); *Grant the lock?*

	S	X
S	Yes	No
X	No	No

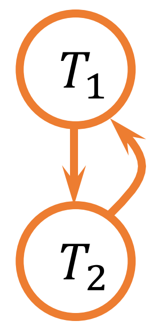
Compatibility matrix

Basic locking is not enough

	T_1	T_2	
Add 1 to both A and B (preserve A=B)			Multiply both A and B by 2 (preserves A=B)
	lock-X(A)		
Read 100	r(A)		
Write 100+1	w(A)		
	unlock(A)		
		lock-X(A)	
		r(A)	Read 101
		w(A)	Write 101*2
		unlock(A)	
		lock-X(B)	
		r(B)	Read 100
		w(B)	Write 100*2
		unlock(B)	
	lock-X(B)		
Read 200	r(B)		
Write 200+1	w(B)		
	unlock(B)		

Possible schedule under locking

But still not conflict-serializable!



$A \neq B$!

Two-phase locking (2PL)

- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks

T_1	T_2
lock-X(A)	
r(A)	
w(A)	
lock-X(B)	
unlock(A)	
	lock-X(A)
	r(A)
	w(A)
	lock-X(B)
	r(B)
	w(B)
r(B)	
w(B)	
unlock(B)	

Cannot obtain the lock on B until T_1 unlocks

2PL guarantees a conflict-serializable schedule

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(B)
	w(B)

Remaining problems of 2PL

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(B)
	w(B)
Abort!	

- T_2 has read uncommitted data written by T_1
- If T_1 aborts, then T_2 must abort as well
- Cascading aborts possible if other transactions have read data written by T_2

- Even worse, what if T_2 commits before T_1?
  - Schedule is not recoverable if the system crashes right after T_2 commits

Strict 2PL

- Only release locks at commit/abort time
  - A writer will block all other readers until the writer commits or aborts
- Used in many commercial DBMS
  - Oracle is a notable exception

Recovery

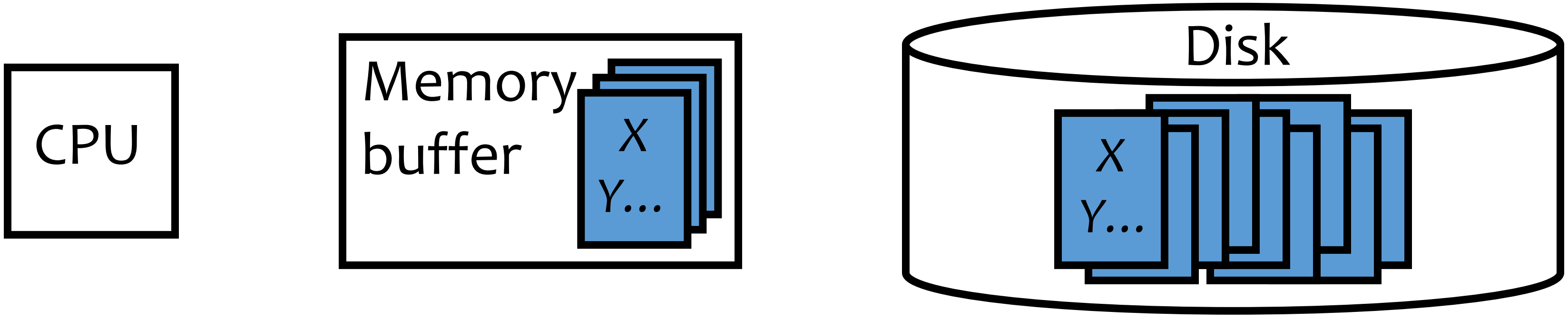
- Goal: ensure "A" (atomicity) and "D" (durability)

http://mnaxe.com/wp-content/uploads/2014/06/Notebook-Tablet-and-Laptop-Data-Recovery.jpg

Execution model

To read/write *X*

- The disk block containing *X* must be first brought into memory
- *X* is read/written in memory
- The memory block containing *X*, if modified, must be written back (flushed) to disk eventually



Failures

- System crashes in the middle of a transaction *T*; partial effects of *T* were written to disk
  - How do we undo *T* (atomicity)?
- System crashes right after a transaction *T* commits; not all effects of *T* were written to disk
  - How do we complete *T* (durability)?

Naïve approach

- Force: When a transaction commits, all writes of this transaction must be reflected on disk
  - Without force, if system crashes right after *T* commits, effects of *T* will be lost
  - ☞Problem: Lots of random writes hurt performance
- No steal: Writes of a transaction can only be flushed to disk at commit time
  - With steal, if system crashes before *T* commits but after some writes of *T* have been flushed to disk, there is no way to undo these writes
  - ☞Problem: Holding on to all dirty blocks requires lots of memory

Logging

- Log
  - Sequence of log records, recording all changes made to the database
  - Written to stable storage (e.g., disk) during normal operation
  - Used in recovery
- Hey, one change turns into two—bad for performance?
  - But writes are sequential (append to the end of log)
  - Can use dedicated disk(s) to improve performance

Undo/redo logging rules

- When a transaction T_i starts, log ⟨ T_i, start ⟩
- Record values before and after each modification: ⟨ T_i, X, *old_value_of_X*, *new_value_of_X* ⟩
  - T_i is transaction id and X identifies the data item
- A transaction T_i is committed when its commit log record ⟨ T_i, commit ⟩ is written to disk
- Write-ahead logging (WAL): Before X is modified on disk, the log record pertaining to X must be flushed
  - Without WAL, system might crash after X is modified on disk but before its log record is written to disk—no way to undo
- No force: A transaction can commit even if its modified memory blocks have not be written to disk (since redo information is logged)
- Steal: Modified memory blocks can be flushed to disk anytime (since undo information is logged)

Undo/redo logging example

T_1 (balance transfer of \$100 from A to B)

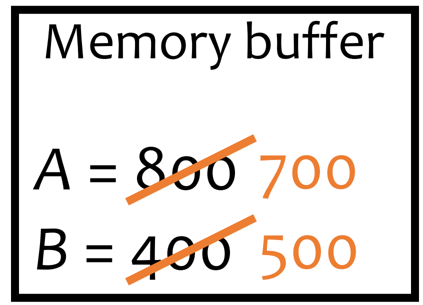
read(A, a); $a = a - 100$;

write(A, a);

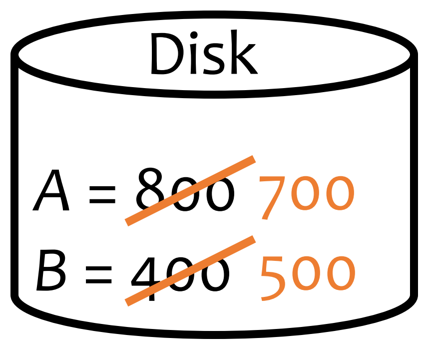
read(B, b); $b = b + 100$;

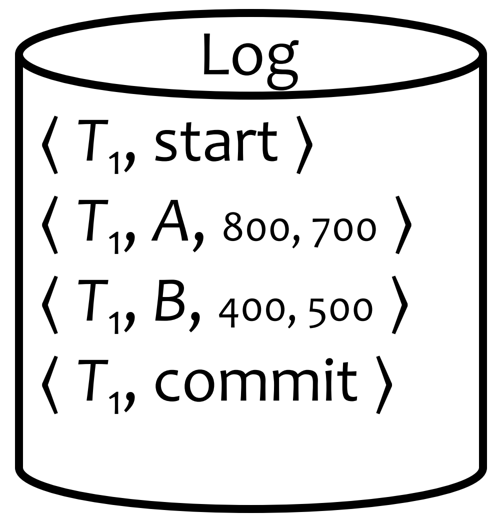
write(B, b);

commit;



Steal: can flush before commit





No force: can flush after commit

No restriction (except WAL) on when memory blocks can/should be flushed

Checkpointing

- Where does recovery start?

Naïve approach:

- To checkpoint:
  - Stop accepting new transactions (lame!)
  - Finish all active transactions
  - Take a database dump
- To recover:
  - Start from last checkpoint

http://www.saintlouischeckpoints.com/wp-content/uploads/2013/08/dui20checkpoint20022017201l.jpg

Fuzzy checkpointing

- Determine S, the set of (ids of) currently active transactions, and log ⟨ begin-checkpoint S ⟩
- Flush all blocks (dirty at the time of the checkpoint) at your leisure
- Log ⟨ end-checkpoint *begin-checkpoint_location* ⟩
- Between begin and end, continue processing old and new transactions

Recovery: analysis and redo phase

- Need to determine *U*, the set of active transactions at time of crash
- Scan log backward to find the last end-checkpoint record and follow the pointer to find the corresponding ⟨ start-checkpoint S ⟩
- Initially, let *U* be *S*
- Scan forward from that start-checkpoint to end of the log
  - For a log record ⟨ *T*, start ⟩, add *T* to *U*
  - For a log record ⟨ *T*, commit | abort ⟩, remove *T* from *U*
  - For a log record ⟨ *T*, *X*, *old*, *new* ⟩, issue write(*X*, *new*)

  ☞ Basically repeats history!

Recovery: undo phase

- Scan log backward
  - Undo the effects of transactions in *U*
  - That is, for each log record ⟨ *T*, *X*, *old*, *new* ⟩ where *T* is in *U*, issue write(*X*, *old*), and log this operation too (part of the "repeating-history" paradigm)
  - Log ⟨ *T*, abort ⟩ when all effects of *T* have been undone

☞ An optimization
  - Each log record stores a pointer to the previous log record for the same transaction; follow the pointer chain during undo

Summary

- Concurrency control
  - Serial schedule: no interleaving
  - Conflict-serializable schedule: no cycles in the precedence graph; equivalent to a serial schedule
  - 2PL: guarantees a conflict-serializable schedule
  - Strict 2PL: also guarantees recoverability
- Recovery: undo/redo logging with fuzzy checkpointing
  - Normal operation: write-ahead logging, no force, steal
  - Recovery: first redo (forward), and then undo (backward)