SFU

Database Systems I

CMPT 354 Summer 2024

Zhengjie Miao

July 26 2024

Announcements (Fri. July 26)

- Last exam on Friday, Aug 2
  - Releasing sample questions tonight
  - Covering Lec9-Lec18, emphasizing Lec14-Lec18
    - Details coming 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

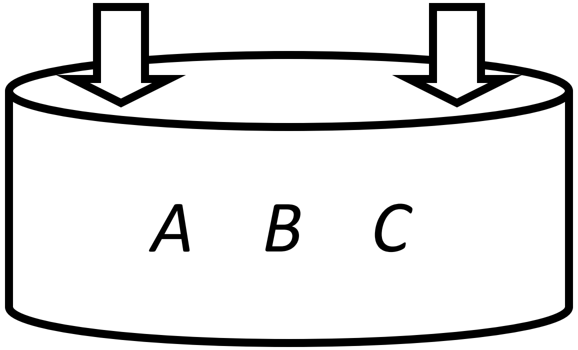
```
-- Begins implicitly
SELECT …;
UPDATE …;
ROLLBACK | COMMIT;
```

Concurrency control

- Goal: ensure the “I” (isolation) in ACID

Each transaction is
a sequence of reads, writes,

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)

Read of A in second transaction depends on the write on the first transaction

Bad!

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

Read 400

400 is stale here

Read 400

Write 400 – 100

Write 400 – 50

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
- Challenge: how to reorder operations to allow more concurrency

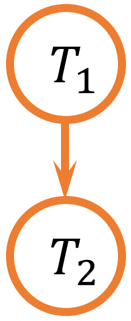
Conflicting operations

- 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

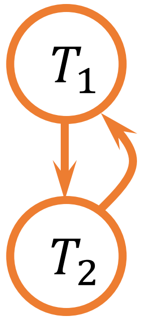
- A node for each transaction
- A directed edge from T_i to T_j if an operation of T_i precedes and conflicts with an operation of T_j in the schedule

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



Good: no cycle

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



Bad: cycle

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

- A mechanism to ensure serializability
- 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

Why one or the other? Why not have both exclusive and shared locks?

Mode of the lock requested

Mode of lock(s) currently held by other transactions

	S	X
S	Yes	No
X	No	No

Grant the lock?

Compatibility matrix

Basic locking is not enough

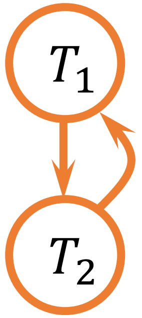
Add 1 to both *A* and *B* (preserve *A*=*B*)

Multiply both *A* and *B* by 2 (preserves *A*=*B*)

	T_1	T_2	
	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
1 lock-X(A)	
r(A)	
w(A)	
2 lock-X(B)	
3 unlock(A)	
	lock-X(A) 4
	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!	

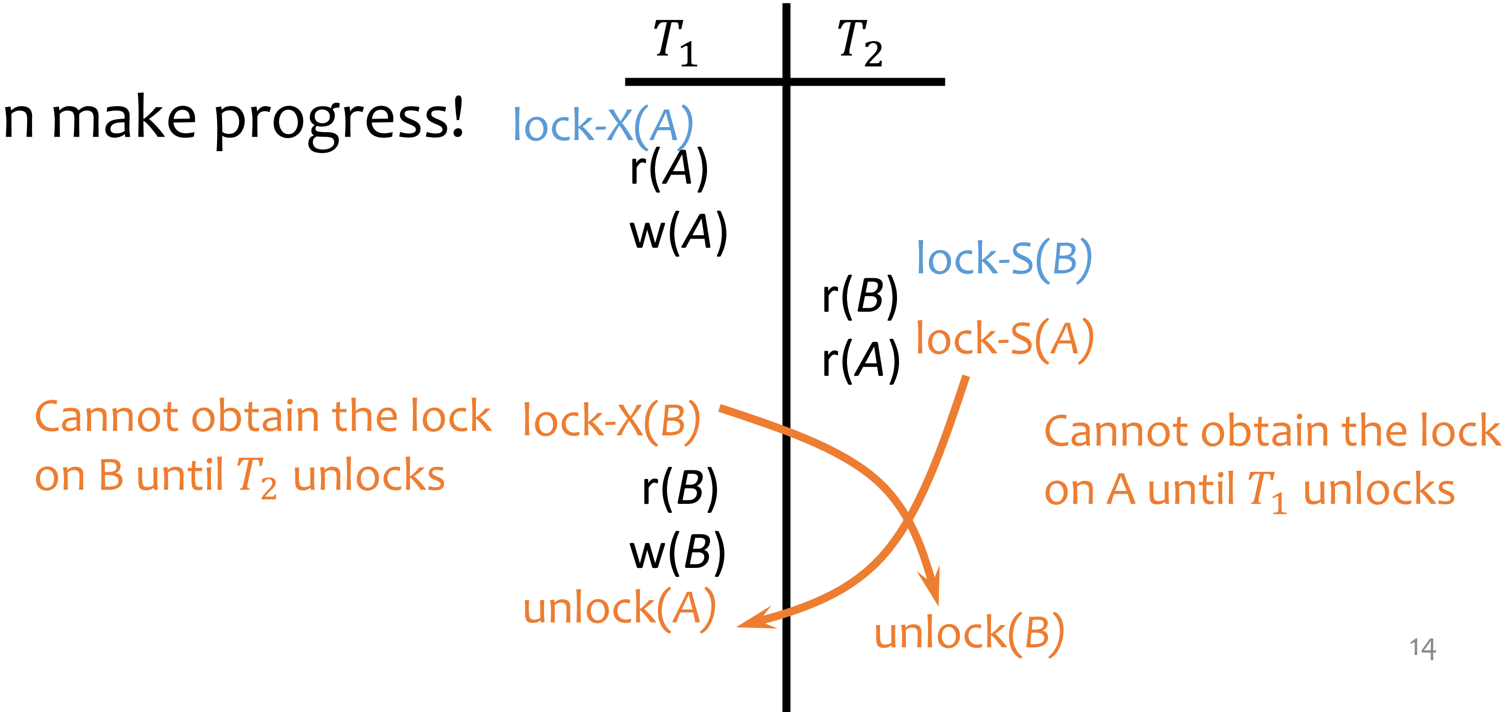
- T_2 has read uncommitted data written by T_1
- If T_1 aborts, then T_2 must abort as well
- Cascading aborts possible if other transactions have read data written by T_2

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

Deadlocks

- A transaction is deadlocked if it is blocked and will remain blocked until there is an intervention.
- Locking-based concurrency control algorithms may cause deadlocks requiring abort of one of the transactions

Neither T_1 nor T_2 can make progress!



Strict 2PL

- Only release X-locks at commit/abort time
  - A writer will block all other readers until the writer commits or aborts
- All DBMS using locking use strict 2PL
  - Avoids cascading aborts
  - All recoverable
  - But deadlocks are still possible
- Conservative 2PL: acquire all locks at the beginning of a txn
  - Avoids deadlocks but often not practical

☞*But recall locking is not not the only mechanism that can enforce serializability*
  - *E.g., PostgreSQL uses multi-version concurrency control*

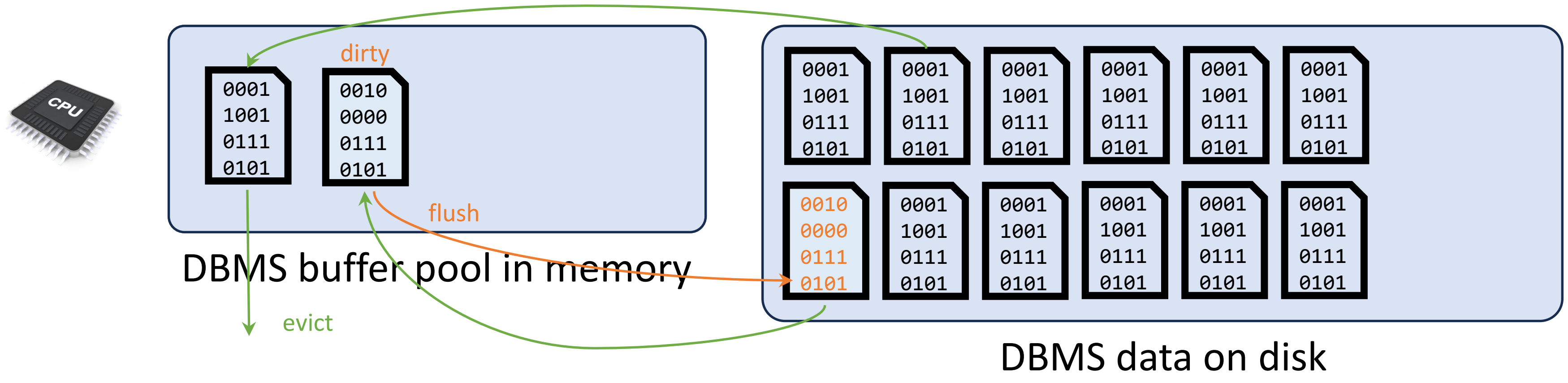
Recovery

- Goal: ensure “A” (atomicity) and “D” (durability)
  - Naïve approaches
  - Logging for undo and redo

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 -- durability

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

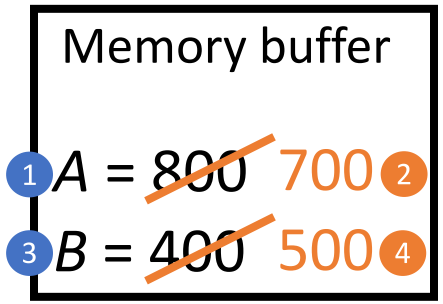
1 read(A, a); $a = a - 100$;

2 write(A, a);

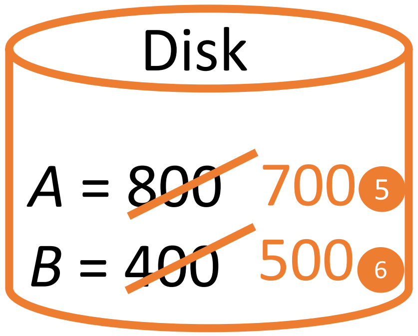
3 read(B, b); $b = b + 100$;

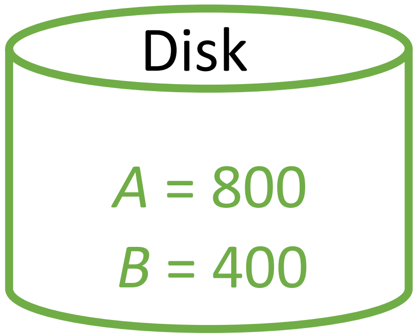
4 write(B, b);

7 commit;



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

Naïve approach: No steal -- atomicity

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

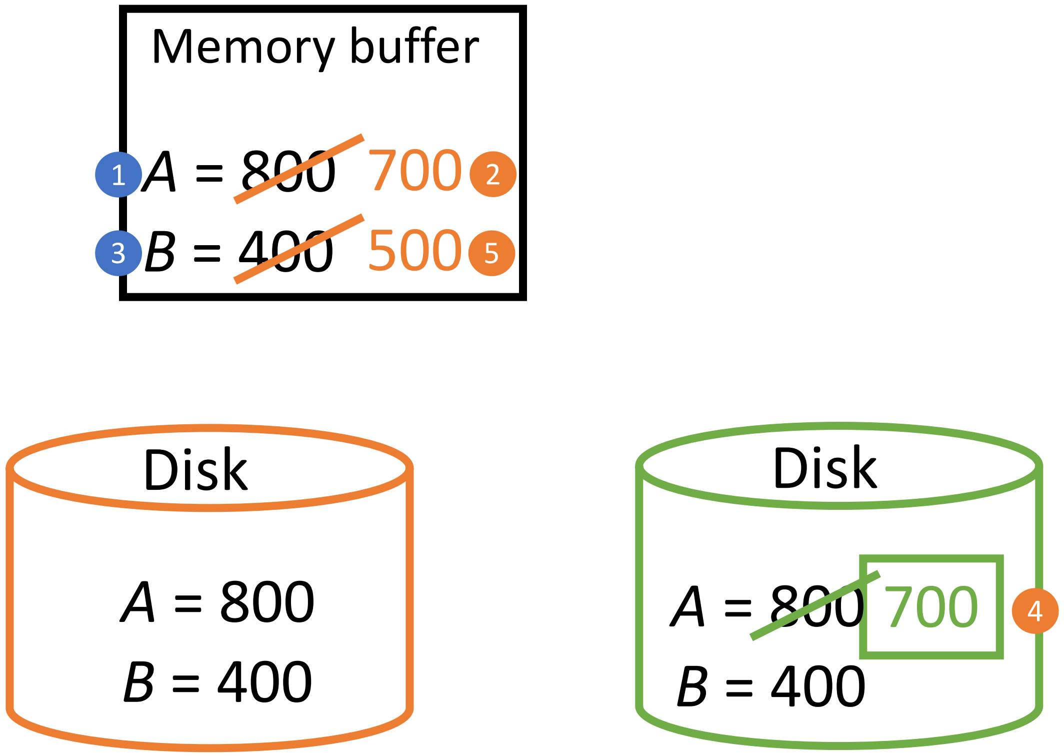
1 read(A, a); $a = a - 100$;

2 write(A, a);

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

5 write(B, b);

6 commit;



No steal: Writes of a transaction can only be flushed to disk at commit time

With steal: some writes are on disk before T commits,

☞Problem of Force: Lots of random writes hurt performance

Naïve approach

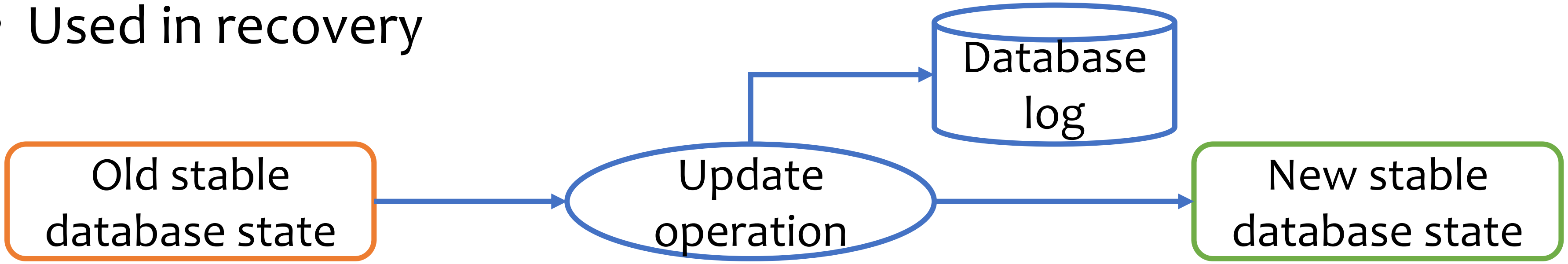
- Force: When a transaction commits, all writes of this transaction must be reflected on disk
  - Without force, if system crashes right after *T* commits, effects of *T* will be lost

  ☞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

The writes to the log are not random since t

Undo/redo logging rules

- When a transaction T_i starts, log ⟨ T_i, start ⟩
  - T_i is transaction id
- Record values before and after each modification:
  ⟨ T_i, X, *old_value_of_X*, *new_value_of_X* ⟩
  - X identifies the data item
- A transaction T_i is committed when its commit log record ⟨ T_i, commit ⟩ is written to disk

Undo/redo logging rules

- 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 been 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)

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

3 write(A, a);

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

6 write(B, b);

9 commit;

Memory buffer

2 $A = \cancel{800}$ 700 3

5 $B = \cancel{400}$ 500 6

Disk

$A = \cancel{800}$ 700 8

$B = \cancel{400}$ 500 10

Log

1 $\langle T_1$, start $\rangle$

4 $\langle T_1, A$, 800, 700 $\rangle$

7 $\langle T_1, B$, 400, 500 $\rangle$

9 $\langle T_1$, commit $\rangle$

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
  - Finish all active transactions
  - Take a database dump
- To recover:
  - Start from last checkpoint

http://www.saintlouischeckpoints.com/wp-content/uploads/2013/08/dui20checkpoint200220172011.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 phase 1: analysis & redo

Goal: *repeats history from last completed checkpoint and 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 phase 2: undo

Goal: undo the actions of incomplete transactions (U)

- 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)