Lecture 20: Transaction

CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**
Sections: 001, 002, 003

Announcements

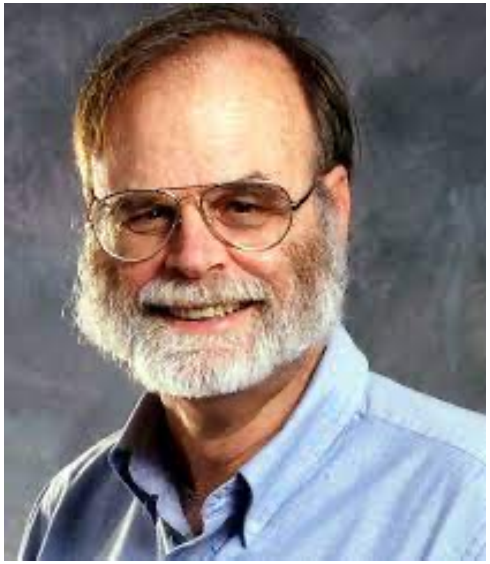
- Assignment 3
  - Check Piazza for online office hours

- Demo of Group Project next week
  - Schedule a demo time with TA before 11:59 PM July 17
  - In-person or online live demo with TA

- Milestone 3 of Group project
  - Due on 11:59 PM July 29

(Recap) Transactions

- ACID properties of transactions (TXs)
  - Atomicity: TXs are either completely done or not done at all
  - Consistency: TXs should leave the database in a consistent state
  - Isolation: TXs must behave as if they execute in isolation (serializable)
  - Durability: Effects of committed TXs are resilient against failures



Jim Gray, Turing Award 1998, who coined this term (as well as data cube and many other things)

Outline for today

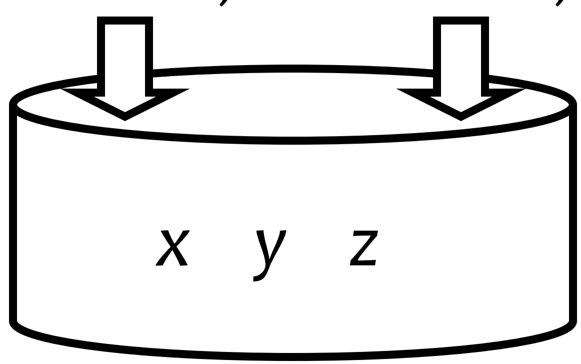
- Concurrency control -- isolation
  - Locking-based control

- Recovery – atomicity and durability
  - Logging for undo and redo

Concurrency control

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

T_1:	T_2:
r1(x);	r2(x);
w1(x);	w2(x);
r1(y);	r2(z);
w1(y);	w2(z);
commit;	commit;



Good v.s. bad execution histories

Good!

T_1	T_2
r1(x)	
w1(x)	
r1(y)	
w1(y)	
	r2(x)
	w2(x)
	r2(z)
	w2(z)

H_a

Bad!

T_1	T_2
r1(x)	
Read 400	r2(x)
Write 400 – 100 w1(x)	Read 400
	w2(x) Write 400 – 50
r1(y)	
	r2(z)
w1(y)	
	w2(z)

H_b

Good! Why?

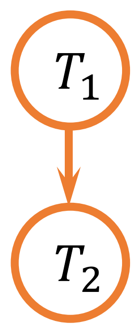
T_1	T_2
r1(x)	
w1(x)	
	r2(x)
	w2(x)
r1(y)	
	r2(C)
w1(y)	
	w2(C)

H_c

Good v.s. bad execution histories

Good!

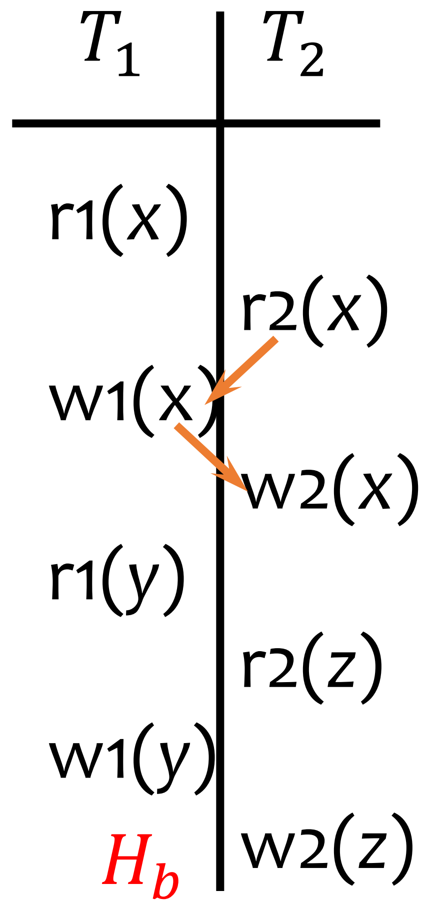
T_1	T_2
r1(x)	
w1(x)	
	r2(x)
	w2(x)
r1(y)	
	r2(C)
w1(y)	
H_c	w2(C)

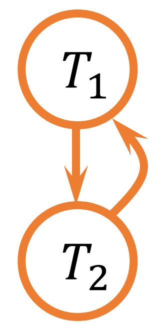


Serializable
(Lecture 19)

Good v.s. bad execution histories

Bad!





Not serializable (Lecture 19)

Locking

(Pessimistic) Assume that conflicts will happen and take preventive action

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

Mode of the lock requested

Mode of lock currently held by other transactions

	lockS	lockX
lockS	Yes	No
lockX	No	No

Grant the lock?

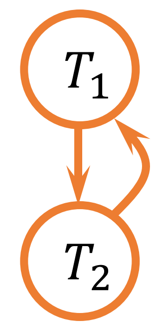
Compatibility matrix

Basic locking is not enough

T_1	T_2
lock-X(x)	
r1(x)	
w1(x)	
unlock(x)	
	lock-X(x)
	r2(x)
	w2(x)
	unlock(x)
	lock-X(y)
	r2(y)
	w2(y)
	unlock(y)
lock-X(y)	
r1(y)	
w1(y)	
unlock(y)	

Possible schedule under locking

But still not conflict-serializable!

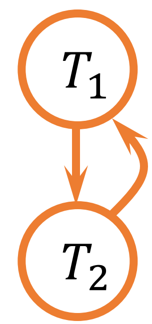


Basic locking is not enough

Add 1 to both *x* and *y* (preserve *x*=*y*)	T_1	T_2	Multiply both *x* and *y* by 2 (preserves *x*=*y*)
	lock-X(x)		
Read 100	r1(x)		
Write 100+1	w1(x)		
	unlock(x)		
		lock-X(x)	
		r2(x)	Read 101
		w2(x)	Write 101*2
		unlock(x)	
		lock-X(y)	
		r2(y)	Read 100
		w2(y)	Write 100*2
		unlock(y)	
	lock-X(y)		
Read 200	r1(y)		
Write 200+1	w1(y)		
	unlock(y)		

Possible schedule under locking

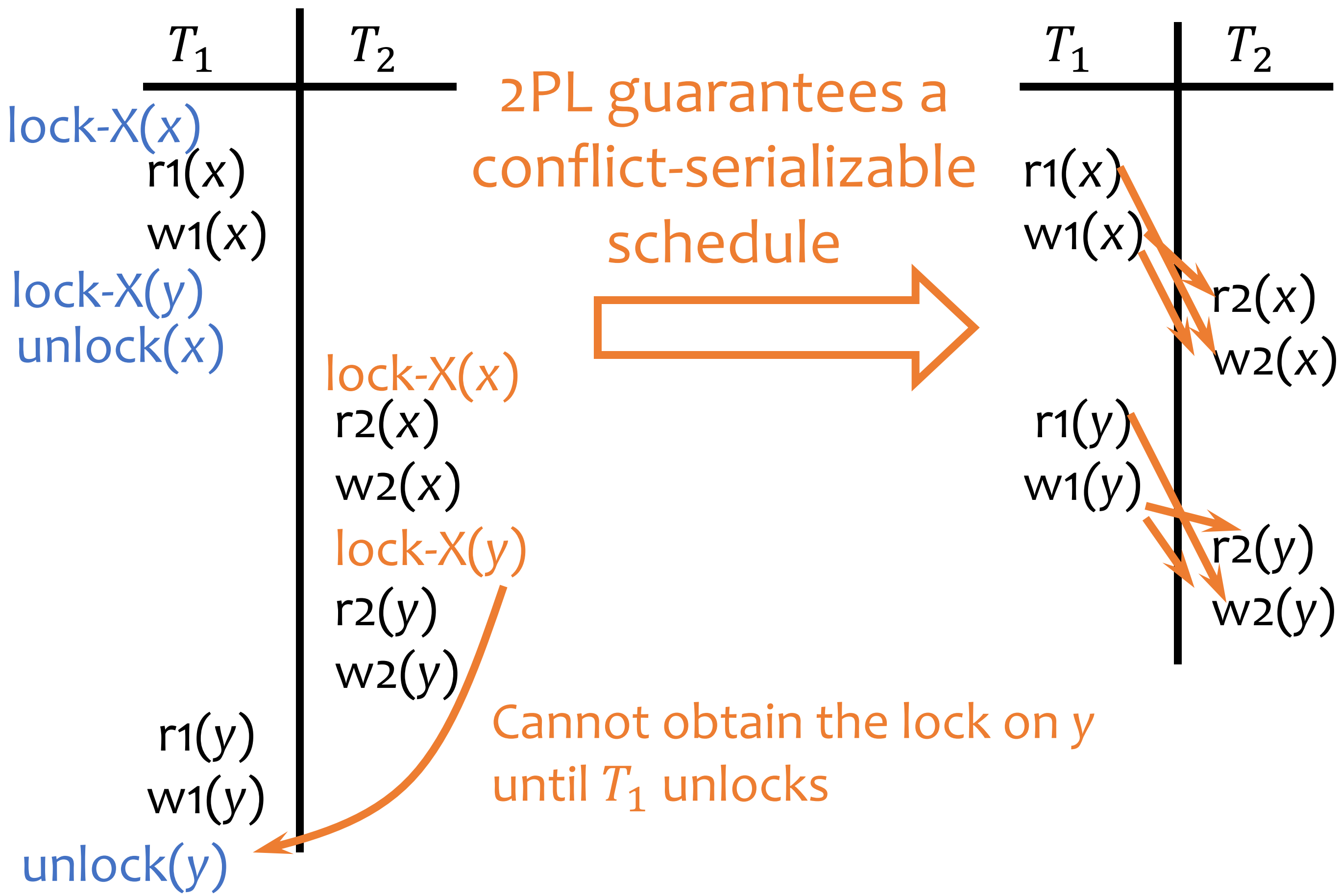
But still not conflict-serializable!



$x \neq y$!

Two-phase locking (2PL)

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



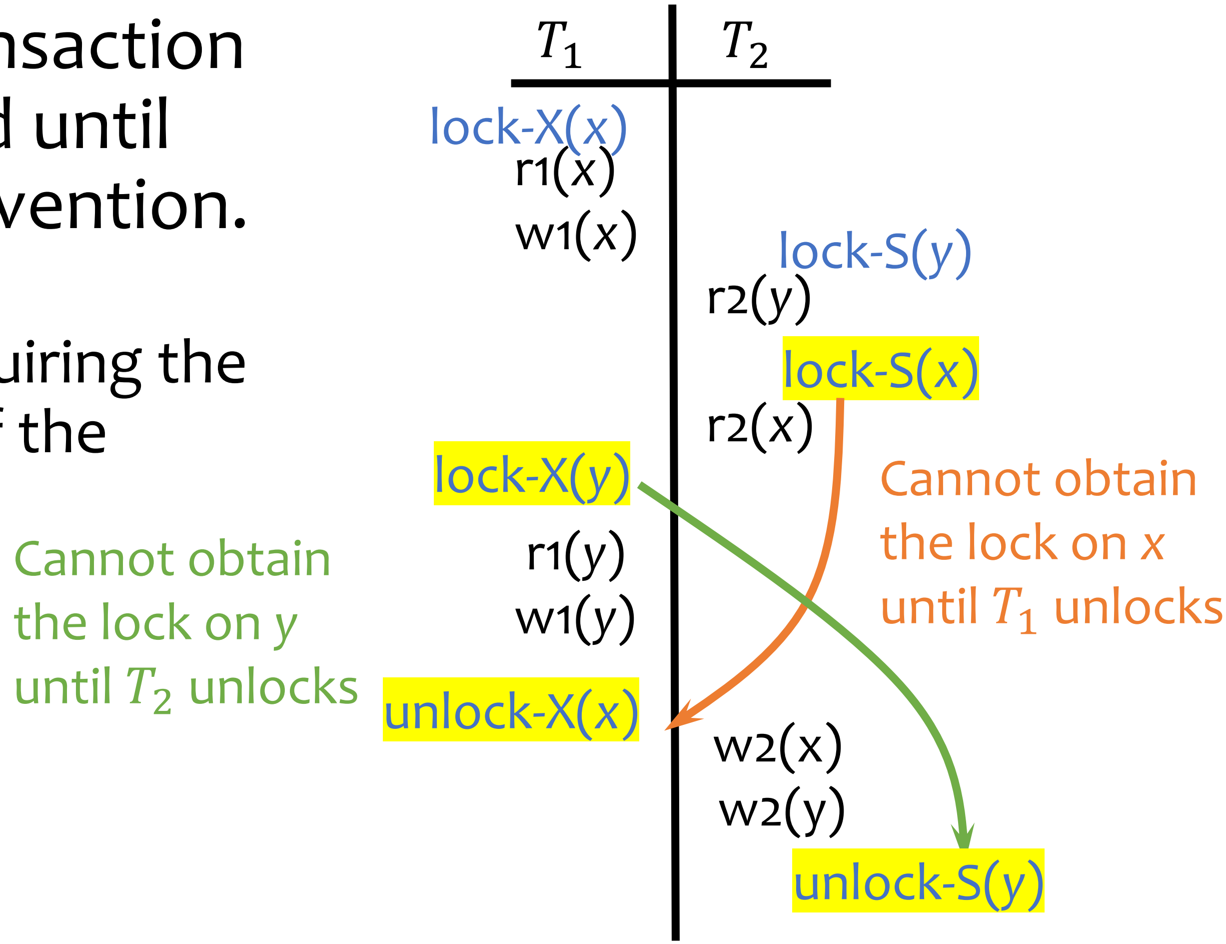
Remaining problems of 2PL

T_1	T_2
lock-X(x)	
r1(x)	
w1(x)	
lock-X(y)	
unlock(x)	
	lock-X(x)
	r2(x)
	w2(x)
r1(y)	
w1(y)	
unlock(y)	
	lock-X(y)
	r2(y)
	w2(y)
	unlock(x)
	unlock(y)
	Commit!
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

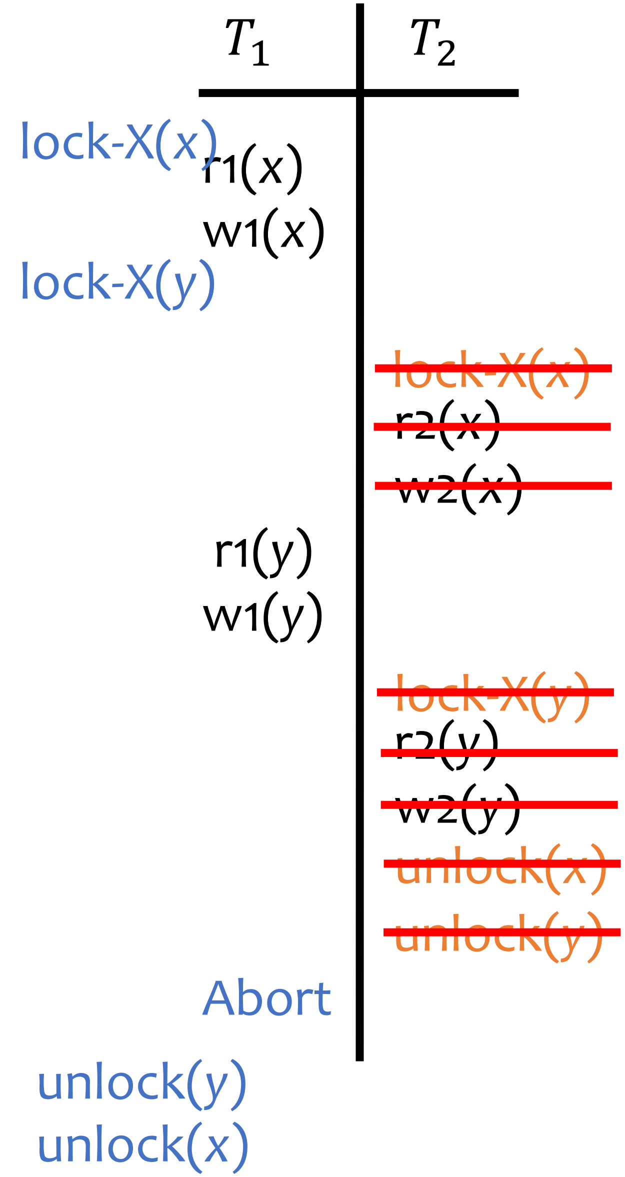
Remaining problems of 2PL

- Deadlock: A transaction remains blocked until there is an intervention.
  - 2PL may cause deadlocks, requiring the abort of one of the transactions



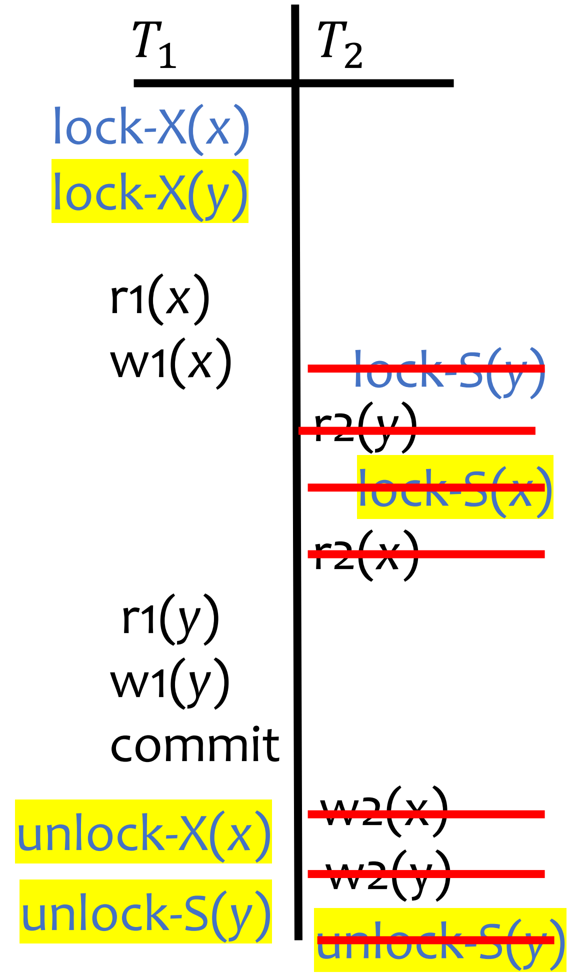
Strict 2PL

- Only release X-locks when commit/abort
    - A write will block all other reads until the write commits or aborts
- Used in many practical DBMSs
    - No cascading rollbacks
    - But can still lead to deadlocks! (see previous slide)
- Also, less concurrency than 2PL



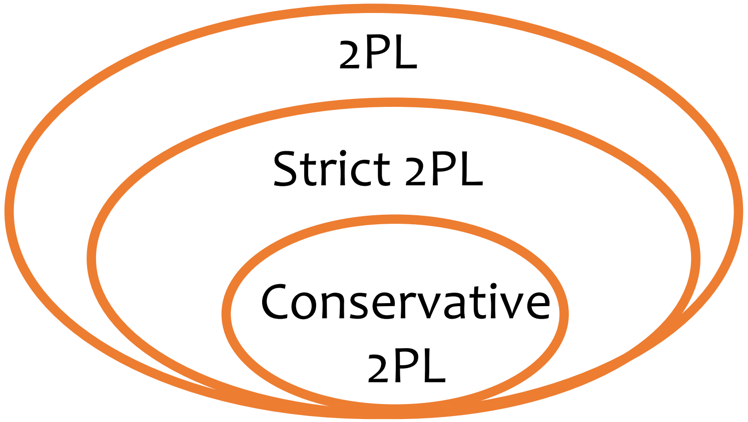
Conservative 2PL

- Only acquire at the beginning of the transaction and release X-locks when commit/abort
- Not practical due to the very limited concurrency
  - No cascading rollbacks
  - No deadlocks



Outline for today

- Concurrency control -- isolation
  - Concurrency: conservative 2PL < strict 2PL < 2PL
  - Serializability: all
  - No cascading aborts: conservative 2PL, strict 2PL
  - No deadlocks: conservative 2PL

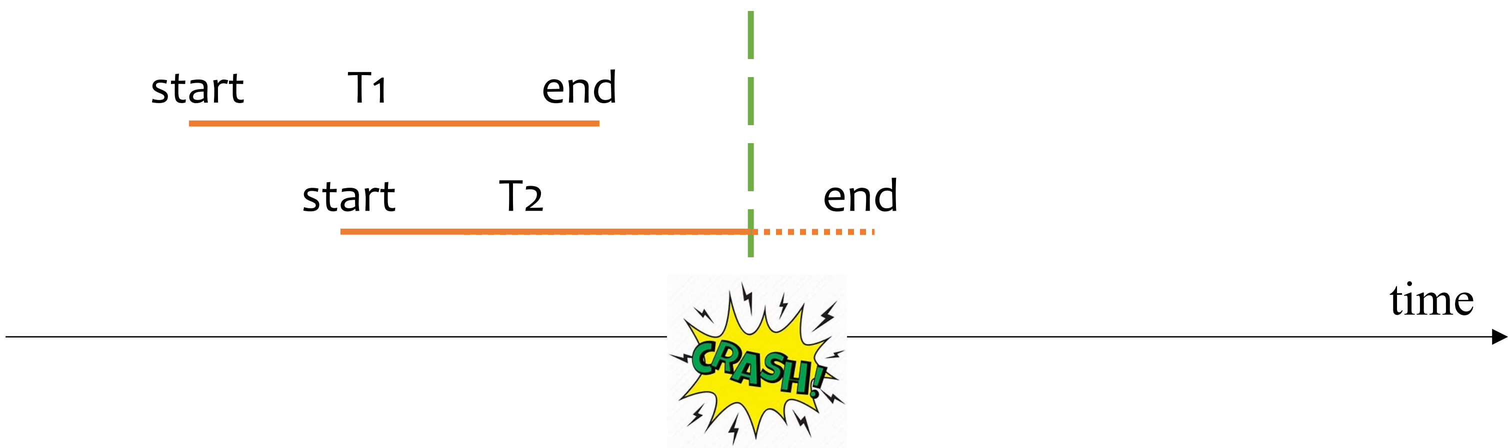


- Recovery – atomicity and durability
  - Logging for undo and redo

Failures

- System crashes right after a transaction *T1* commits; but not all effects of *T1* were written to disk
  - How do we complete/redo *T1* (durability)?

- System crashes in the middle of a transaction *T2*; partial effects of *T2* were written to disk
  - How do we undo *T2* (atomicity)?



Naïve approach: Force -- durability

T1 (balance transfer of \$100 from *A* to *B*)

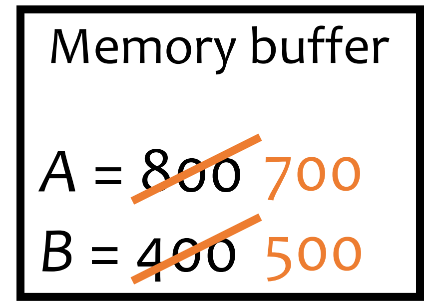
read(*A*, *a*); $a = a - 100$;

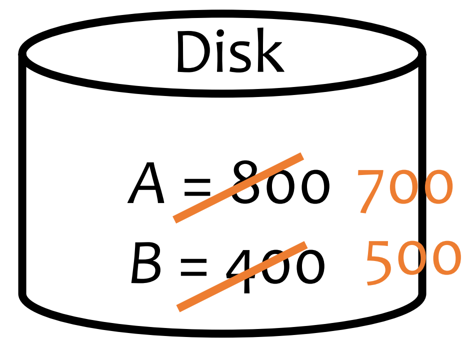
write(*A*, *a*);

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

write(*B*, *b*);

commit;





Force: all writes must be reflected on disk when a transaction commits

Naïve approach: Force -- durability

T1 (balance transfer of $100 from *A* to *B*)

read(*A*, *a*); *a* = *a* – 100;

write(*A*, *a*);

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

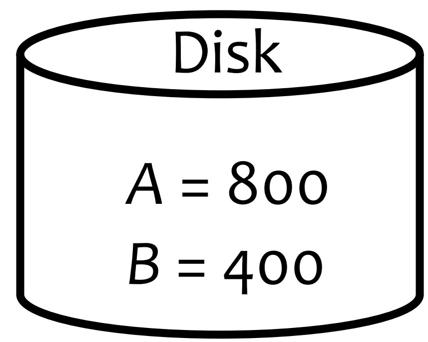
write(*B*, *b*);

commit;

Memory buffer

A = ~~800~~ 700

B = ~~400~~ 500



Force: all writes must be reflected on disk when a transaction commits

Without **Force:** not all writes are on disk when T1 commits

Bad!

If system crashes right after *T1* commits, effects of *T1* will be lost

Naïve approach: No steal -- atomicity

T1 (balance transfer of $100 from *A* to *B*)

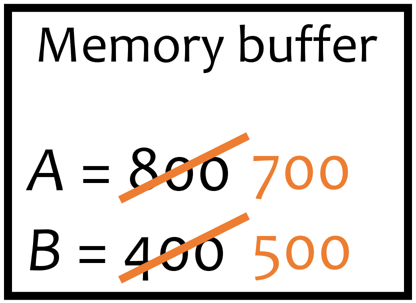
read(*A*, *a*); *a* = *a* – 100;

write(*A*, *a*);

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

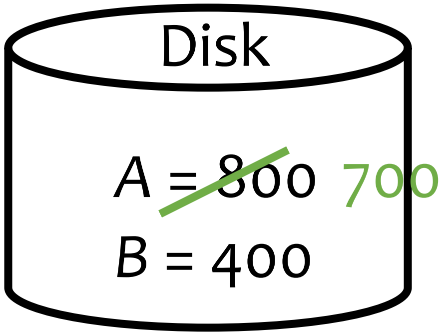
write(*B*, *b*);

commit;



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

- e.g. A=700 cannot be flushed to disk before commit.



Bad!

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

If system crashes before *T1* commits, there is no way to undo the changes

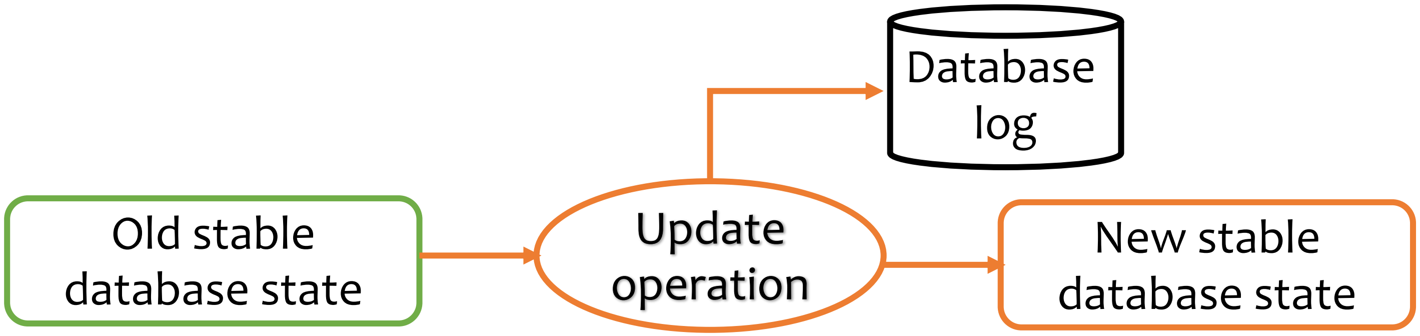
Naïve approach

- Force: When a transaction commits, all writes of this transaction must be reflected on disk
  - Ensures durability
  - ☞Problem of force: Lots of random writes hurt performance

- No steal: Writes of a transaction can only be flushed to disk at commit time
  - Ensures atomicity
  - ☞Problem of no steal: Holding on to all dirty blocks requires lots of memory

Logging

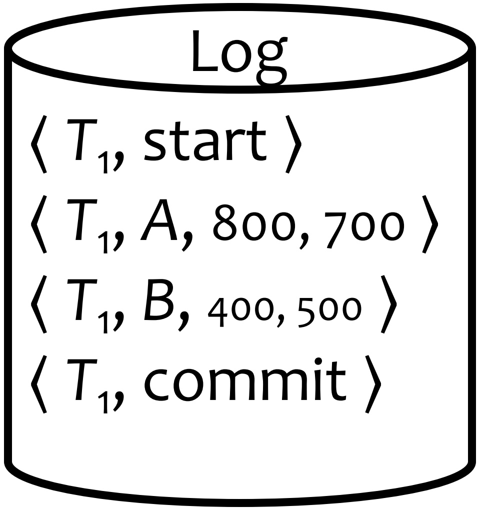
- Database log: sequence of log records, recording all changes made to the database, written to stable storage (e.g., disk) during normal operation



- One change turns into two -- bad for performance?
  - But writes to log are sequential (append to the end of log)

Log

- When a transaction T starts: $\langle T$, start$\rangle$
- Record values before and after each modification of data item X: $\langle T, X, \mathit{old\_value\_of\_X}, \mathit{new\_value\_of\_X} \rangle$
- When a transaction T_i is committed: $\langle T$, commit$\rangle$



When to write log records?

- Before X is modified or after?
- 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

Undo/redo logging example

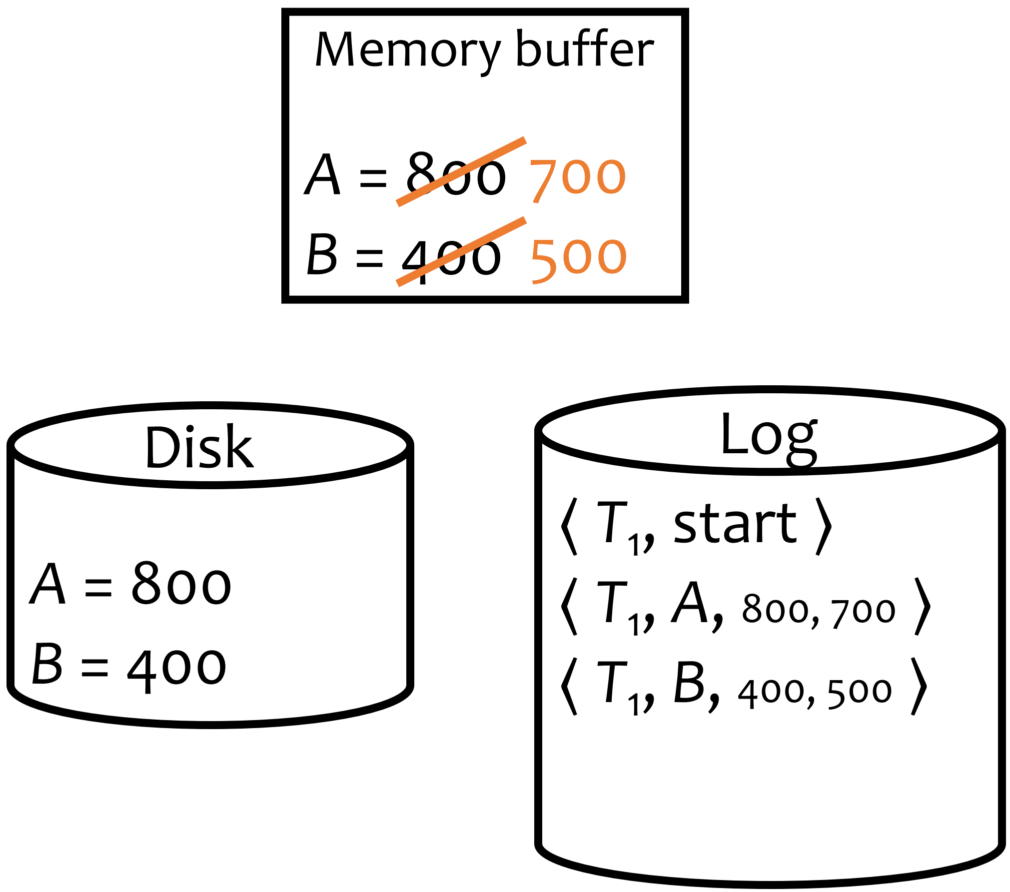
T1 (balance transfer of $100 from *A* to *B*)

read(*A*, *a*); *a* = *a* – 100;

write(*A*, *a*);

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

write(*B*, *b*);



WAL: Before A,B are modified on disk, their log info must be flushed

Undo/redo logging example cont.

T1 (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*);

Memory buffer

A = ~~800~~ 700

B = ~~400~~ 500

Disk

A = ~~800~~ 700

B = 400

Steal: can flush before commit

Log

⟨ T_1, start ⟩

⟨ T_1, A, 800, 700 ⟩

⟨ T_1, B, 400, 500 ⟩

If system crashes before *T1* commits, we have the old value of A stored on the log to **undo** T1

Undo/redo logging example cont.

T1 (balance transfer of $100 from *A* to *B*)

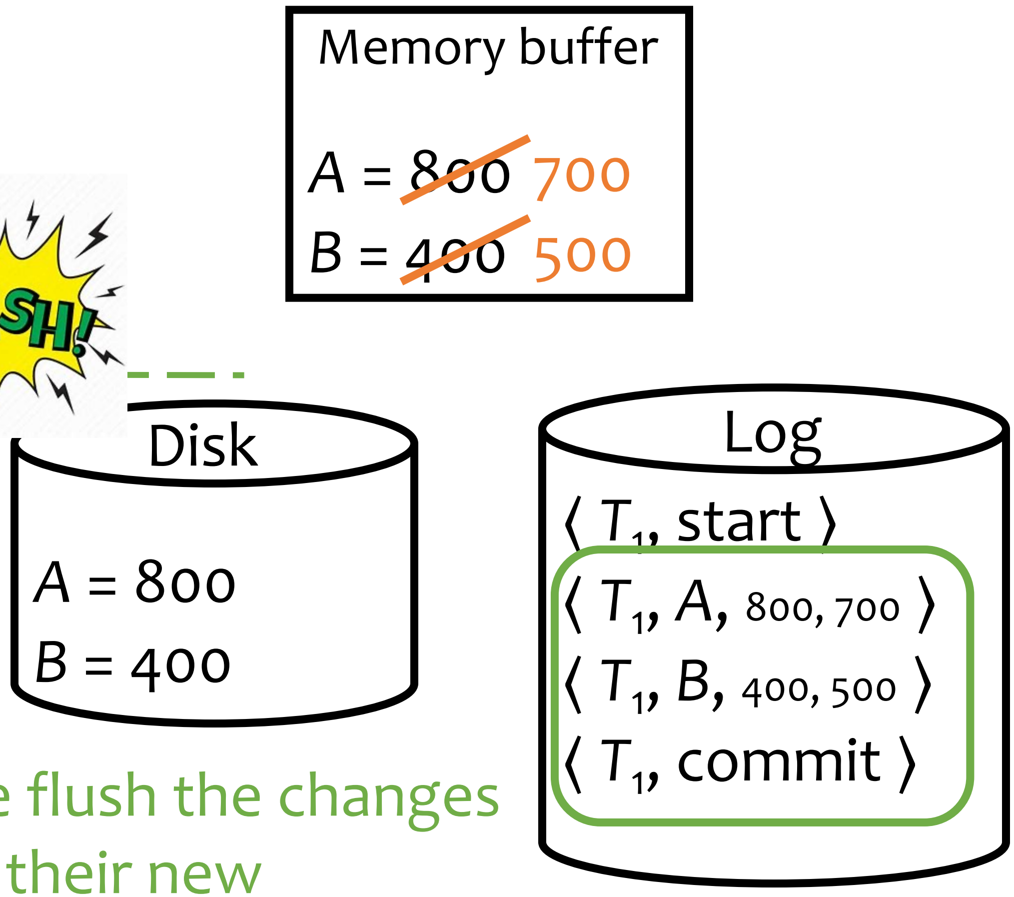
read(*A*, *a*); $a = a - 100$;

write(*A*, *a*);

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

write(*B*, *b*);

commit;



No force: can flush after commit

If system crashes before we flush the changes of A, B to the disk, we have their new committed values on the log to **redo** T1

Log example - redo

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ 50
w: ~~1000~~ 10

List of active transactions at crash:
T1 T2 T3

Start of log →

End of log →

Log

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
	T_2, commit
	T_4, start
	T_3, abort
	T_4, y, 200, 50

Log example

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ 50
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ T3

Start of log →

End of log →

Log

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
	T_4, start
	T_3, abort
	T_4, y, 200, 50

Log example

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ 50
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ T3 T4

Start of log →

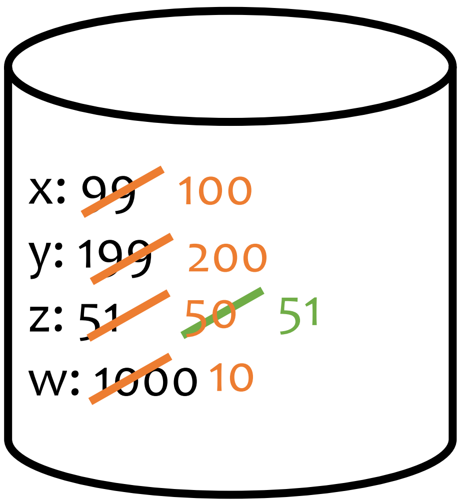
Log

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
redo	T_4, start
	T_3, abort
	T_4, y, 200, 50

End of log →

Log example

- Redo phase:



List of active transactions at crash:

T1 ~~T2~~ T3 T4

Start of log → Log

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
redo	T_4, start
redo	T_3, abort
	T_4, y, 200, 50

End of log →

Log example

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ ~~50~~ 51
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ ~~T3~~ T4

Start of log → (redo, top to bottom)

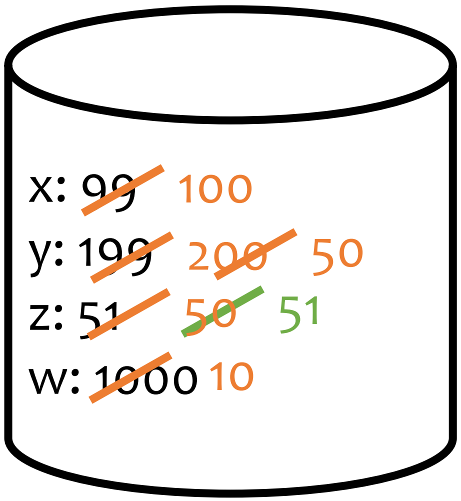
Log

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
redo	T_4, start
redo	T_3, abort
redo	T_4, y, 200, 50

End of log →

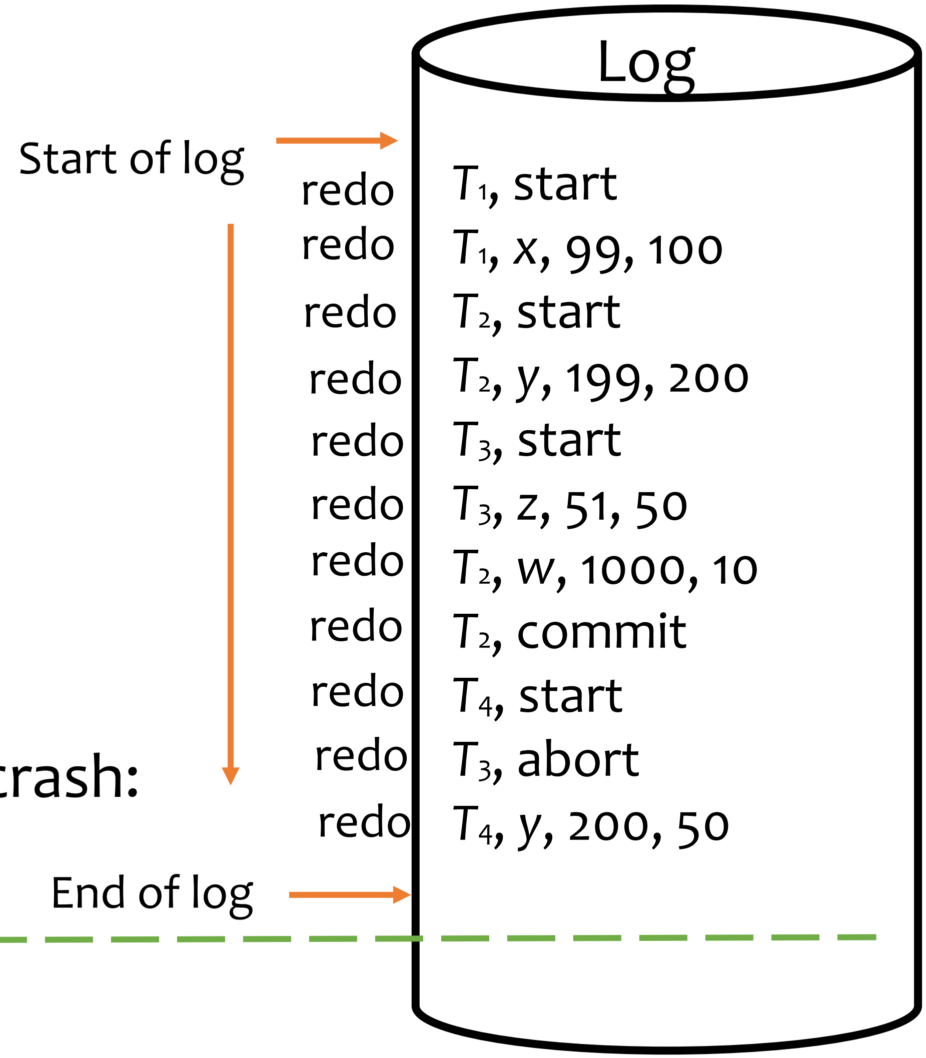
Log example

- Redo phase:



List of active transactions at crash:

T1 ~~T2~~ ~~T3~~ T4



Log example - Undo

- Undo phase: T1, T4

x: ~~99~~ ~~100~~ 99
y: ~~199~~ ~~200~~ ~~50~~ 200
z: ~~51~~ ~~50~~ 51
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ ~~T3~~ T4

Start of log

End of log

Log

*	T_1, start
undo	T_1, x, 99, 100
	T_2, start
	T_2, y, 199, 200
	T_3, start
	T_3, z, 51, 50
	T_2, w, 1000, 10
	T_2, commit
*	T_4, start
	T_3, abort
undo	T_4, y, 200, 50

T_4, y, 200
T_4, abort
T_1, x, 99
T_1, abort

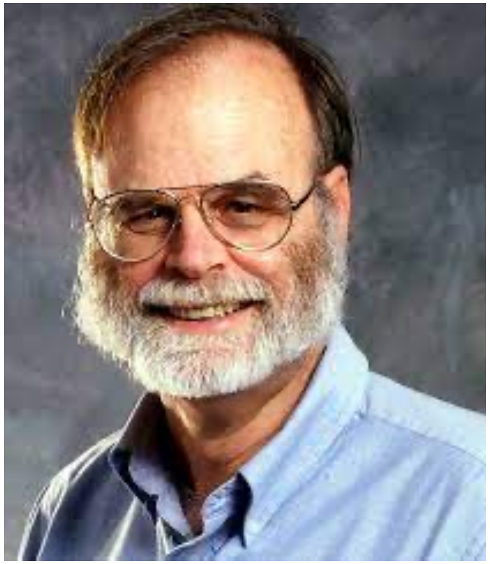
Undo/redo logging

- U: used to track the set of active transactions at crash
- Redo phase: scan forward to end of the log
  - For a log record ⟨ *T*, start ⟩, add *T* to *U*
  - For a log record ⟨ *T*, *X*, *old*, *new* ⟩, issue write(*X*, *new*)
  - For a log record ⟨ *T*, commit | abort ⟩, remove *T* from *U*
    - *If abort, undo changes of T* i.e., add ⟨ *T*, *X*, *old* ⟩ *before logging abort*

  ☞ Basically repeats history!
- Undo phase: scan log backward
  - Undo the effects of transactions in *U*
  - For each log record ⟨ *T*, *X*, *old*, *new* ⟩ where *T* is in *U*, issue write(*X*, *old*), and log this operation too, i.e., add ⟨ *T*, *X*, *old* ⟩
  - Log ⟨ *T*, abort ⟩ when all effects of *T* have been undone

Summary of Transactions

- ACID properties of transactions (TXs)
  - Atomicity: TXs are either completely done or not done at all (logging)
  - Consistency: TXs should leave the database in a consistent state
  - Isolation: TXs must behave as if they execute in isolation (serializable; concurrency control)
  - Durability: Effects of committed TXs are resilient against failures (logging)



Jim Gray, Turing Award 1998, who coined this term (as well as data cube and many other things)

What’s next?

- No lectures next week
- Final review on July 29!