Concurrency control & recovery system

Transaction Processing (optional)

Introduction to Database Management

CS348 Fall 2022

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;
```

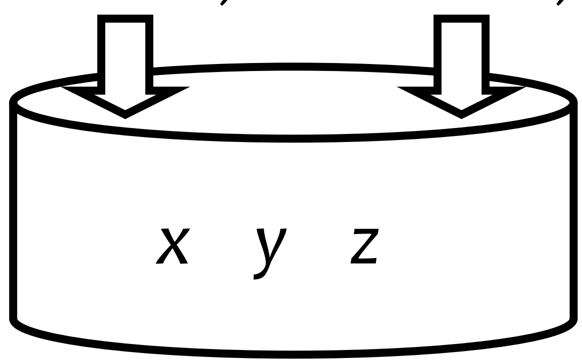
Outline

- Concurrency control -- isolation
  - Review serializable execution histories
  - Locking-based concurrency control
- Recovery – atomicity and durability
  - Naïve approaches
  - 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 versus bad execution histories

Serial

Good!

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

Bad!

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

Good! Why?

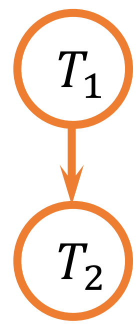
T_1	T_2
r1(x)	
w1(x)	
	r2(x)
	w2(x)
r1(y)	
	r2(C)
w1(y)	
H_c	w2(C)

Good versus bad execution histories

Serializable

Good!

Serialization graph
(Lecture 17)



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

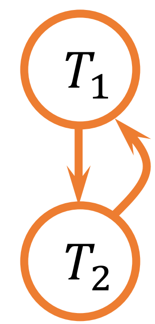
H_c

Good versus bad execution histories

Not serializable

Bad!

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



How to avoid this?

Concurrency control

Possible classification

- Pessimistic – assume that conflicts will happen and take preventive action
  - Two-phase locking (2PL)
  - Timestamp ordering
- Optimistic – assume that conflicts are rare and run transactions and fix if there is a problem
- We will only review 2PL

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

Mode of lock(s) currently held by other transactions

	S	X
S	Yes	No
X	No	No

Grant the lock?

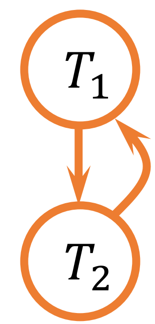
Compatibility matrix

Basic locking is not enough

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

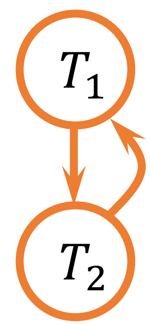
Add 1 to both x and y (preserve x=y)

Multiply both x and y by 2 (preserves x=y)

	T_1	T_2	
	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

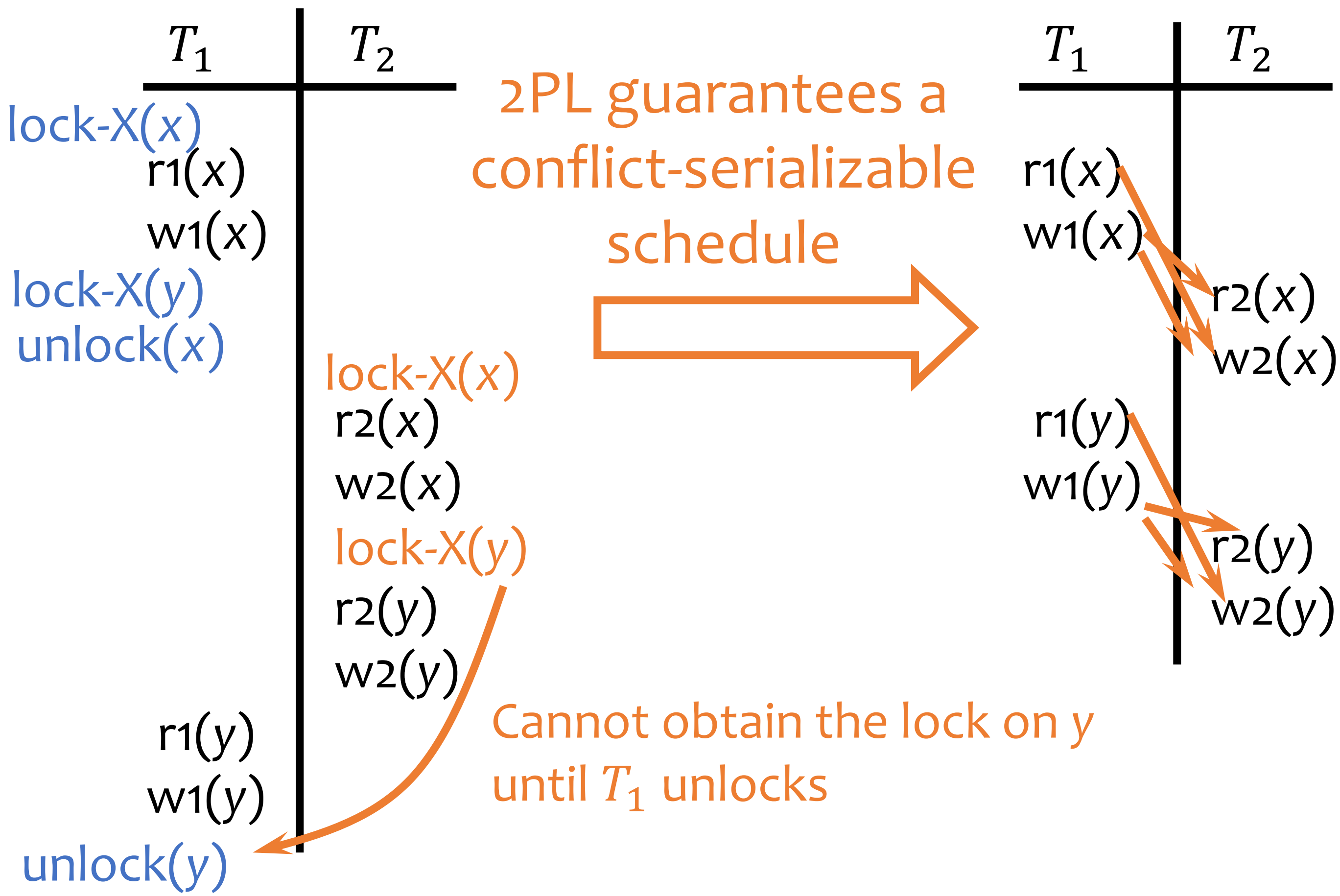
But still not conflict-serializable!



$x \neq y$!

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
r1(x)	
w1(x)	
	r2(x)
	w2(x)
r1(y)	
w1(y)	
	r2(y)
	w2(y)
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
- Consider the partial history
    - Neither T_1 nor T_2 can make progress

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

Cannot obtain the lock on y until T_2 unlocks

Cannot obtain the lock on y until T_1 unlocks

Strict 2PL

- Only release X-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

- Why do we use strict 2PL? (assignment question)

Outline

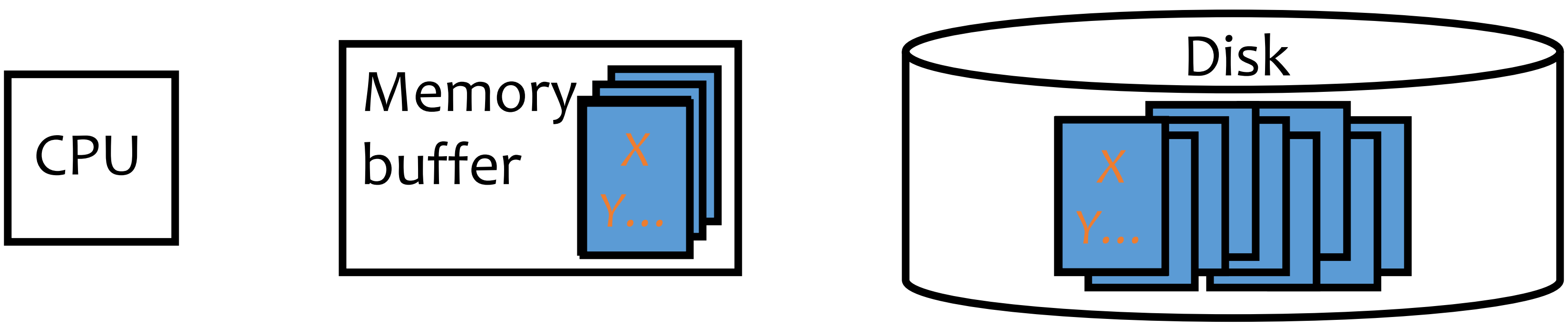
- Concurrency control -- isolation
  - Review serializable execution histories
  - Locking-based concurrency control

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

Execution model

To read/write *X*

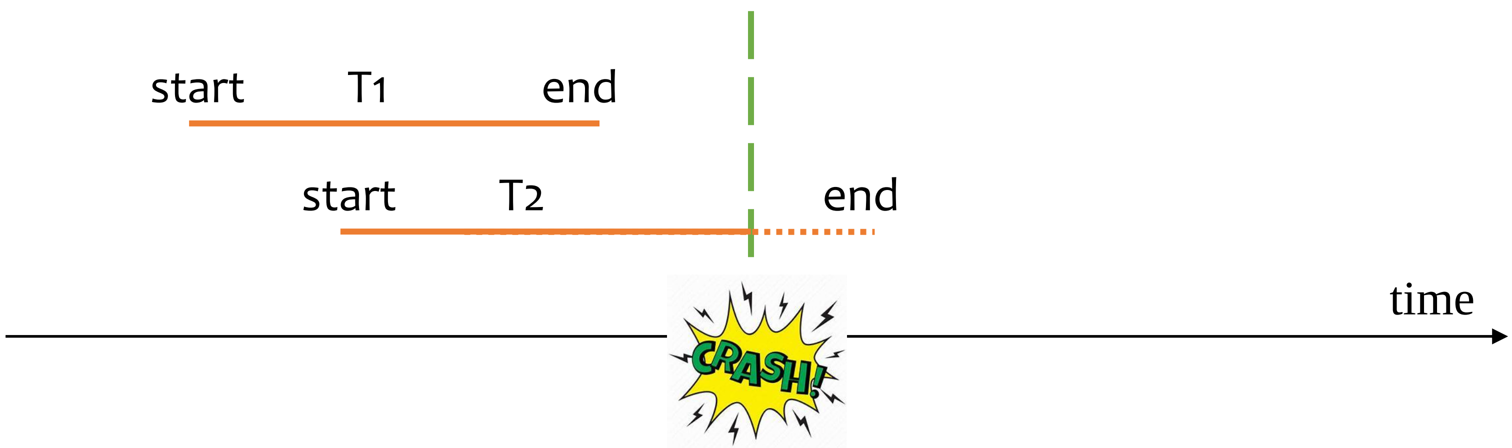
- The disk block containing *X* must be first brought into memory
- *X* is read/written in memory
- The memory block containing *X*, if modified, must be written back (flushed) to disk eventually



Failures

- System crashes 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*)

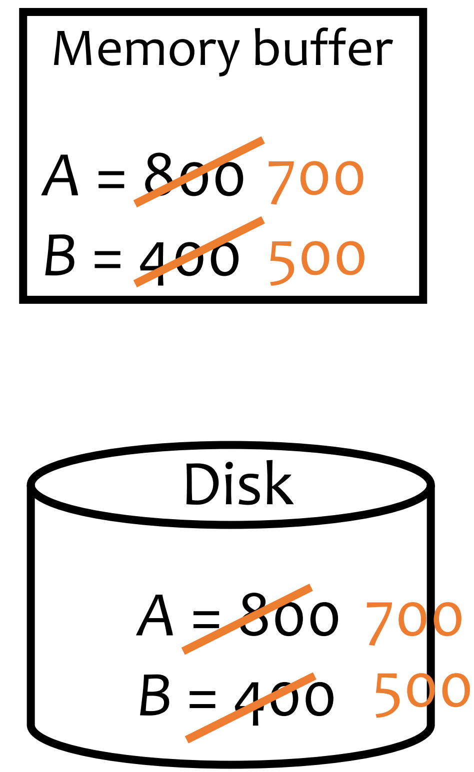
read(A, a); $a = a - 100$;

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

Disk

A = 800

B = 400

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

Bad!

Without force: not all writes are on disk when T1 commits

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;

Memory buffer

A = ~~800~~ 700

B = ~~400~~ 500

Disk

A = ~~800~~ 700

B = 400

Bad!

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.

With steal: some writes are on disk before T commits

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

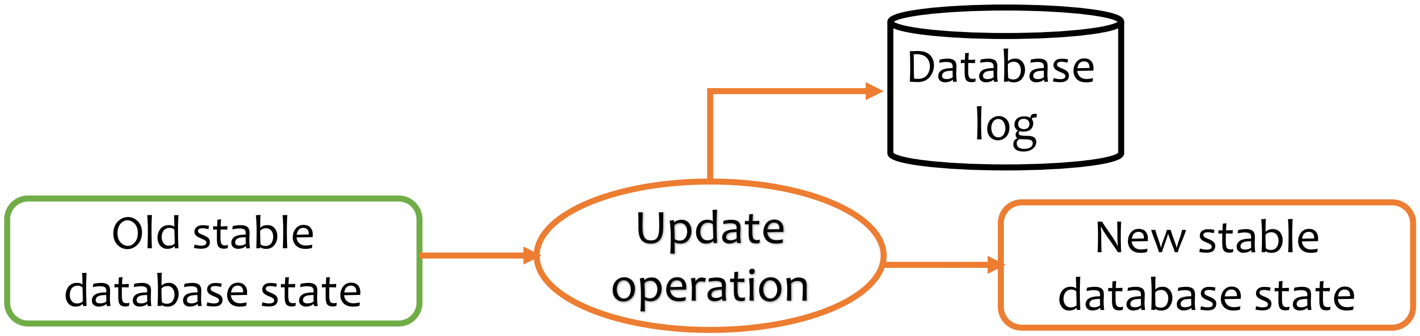
Naïve approach

- 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

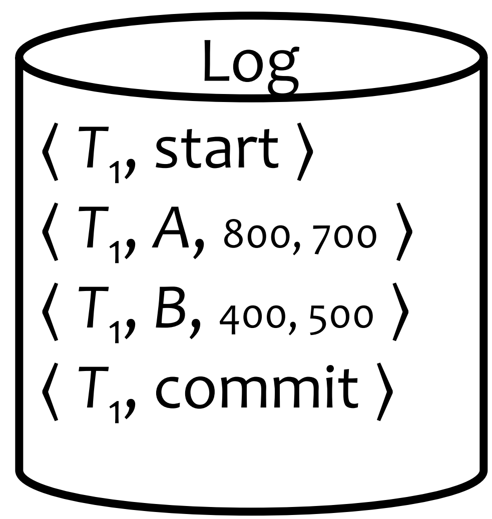
- Database log: sequence of log records, recording all changes made to the database, written to stable storage (e.g., disk) during normal operation



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

Log format

- When a transaction T_i starts
  - $\langle\, T_i, \text{start} \,\rangle$
- Record values before and after each modification:
  - $\langle\, T_i, X, \mathit{old\_value\_of\_X}, \mathit{new\_value\_of\_X} \,\rangle$
  - T_i is transaction id
  - X identifies the data item
- A transaction T_i is committed when its commit log record is written to disk
  - $\langle\, T_i, \text{commit} \,\rangle$



When to write log records into stable store?

- 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

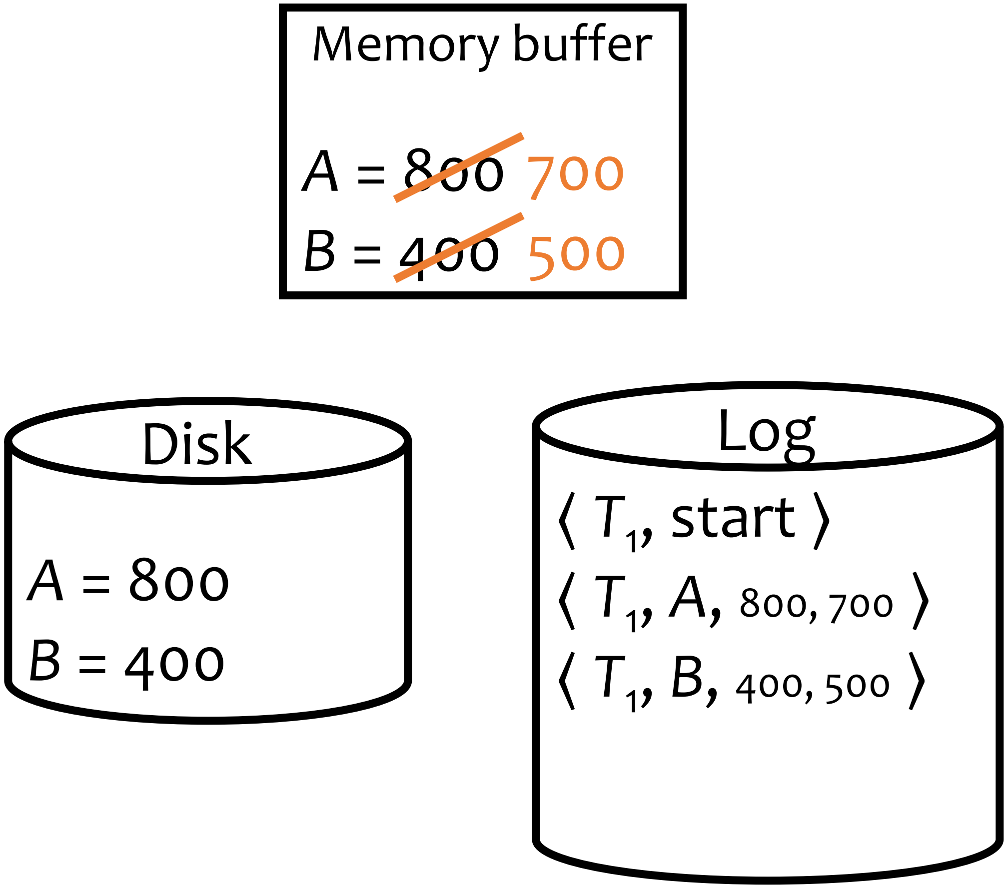
T1 (balance transfer of \$100 from *A* to *B*)

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

Steal: can flush before commit

Disk

A = ~~800~~ 700

B = 400

Log

⟨ T_1, start ⟩

$\boxed{\langle\ T_1,\ A,\ 800,\ 700\ \rangle}$

⟨ 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;

Memory buffer

$A = \cancel{800}$ 700

$B = \cancel{400}$ 500

Disk

$A = 800$

$B = 400$

Log

$\langle T_1, \text{start} \rangle$

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

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

$\langle T_1, \text{commit} \rangle$

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 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

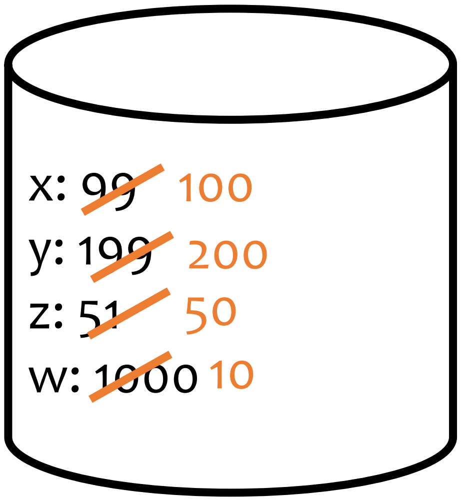
End of log

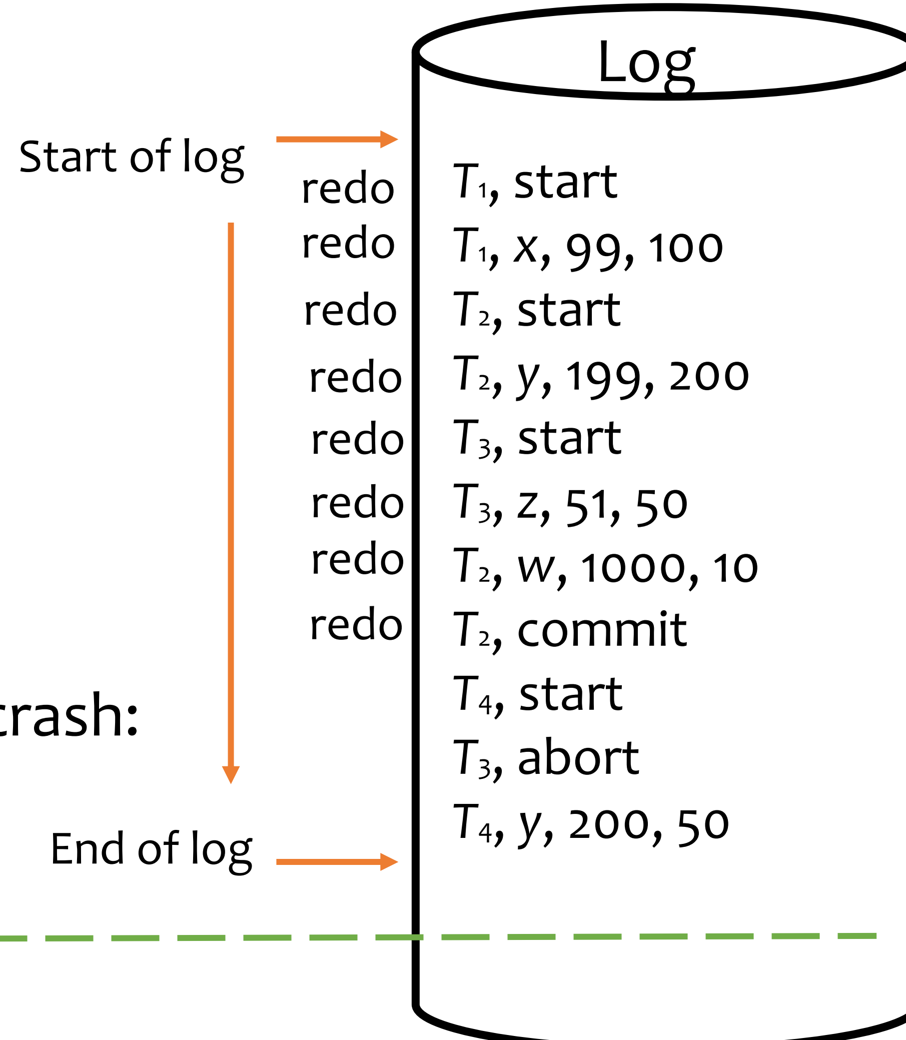
Log

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:





List of active transactions at crash:

T1 ~~T2~~ T3

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

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
redo	T_4, start
	T_3, abort
	T_4, y, 200, 50

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

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
redo	T_4, start
redo	T_3, abort
	T_4, y, 200, 50

Log example

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ ~~200~~ 50
z: ~~51~~ ~~50~~ 51
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
redo T_3, abort
redo T_4, y, 200, 50

End of log

Log example

- 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

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

End of log

T_4, abort
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*, commit | abort ⟩, remove *T* from *U*
  - For a log record ⟨ *T*, *X*, *old*, *new* ⟩, issue write(*X*, *new*)

  ☞ Basically repeats history!

- 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

Checkpointing

- Shortens the amount of log that need to be undone or redone when a failure occurs

- A checkpoint record contains a list of active transactions

- Steps:
    1. Write a begin_checkpoint record into the log
    2. Collect the checkpoint data into the stable storage
    3. Write an end_checkpoint record into the log

Summary

- Concurrency control
  - 2PL: guarantees a conflict-serializable schedule
  - Deadlock problem

- Recovery: undo/redo logging
  - Normal operation: write-ahead logging, no force, steal
  - Recovery: first redo (forward), and then undo (backward)