Database recovery

CS348 Spring 2023

Instructor: Sujaya Maiyya

Sections: **002 & 004 only**

Announcements

- Assignment 3 due July 20th

- Final demo for projects:
  - Option 1: Online live demo with the TA
  - Option 2: Send a recording to the TA

- Send your choice to your TA by **July 24th**
  - Lose 2 points otherwise

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;
```

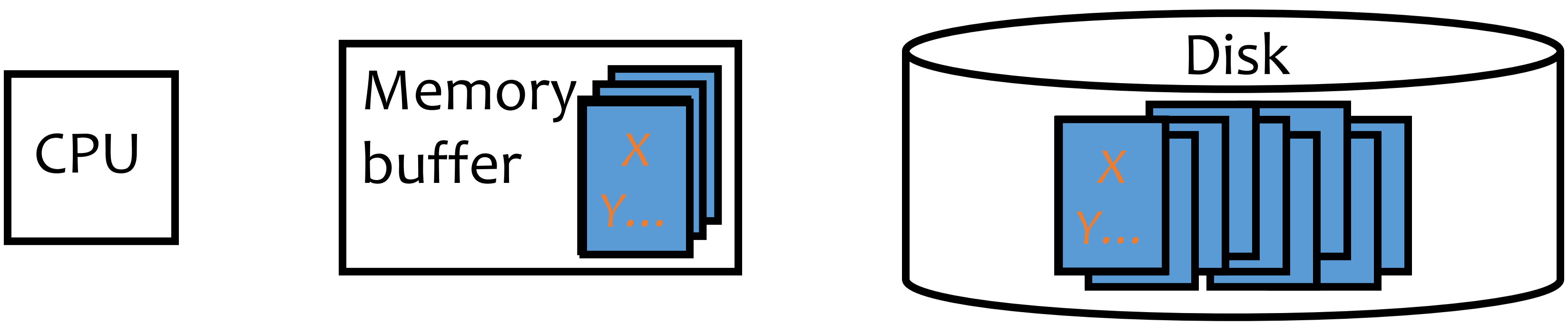
Outline

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

Execution model

To read/write *X*

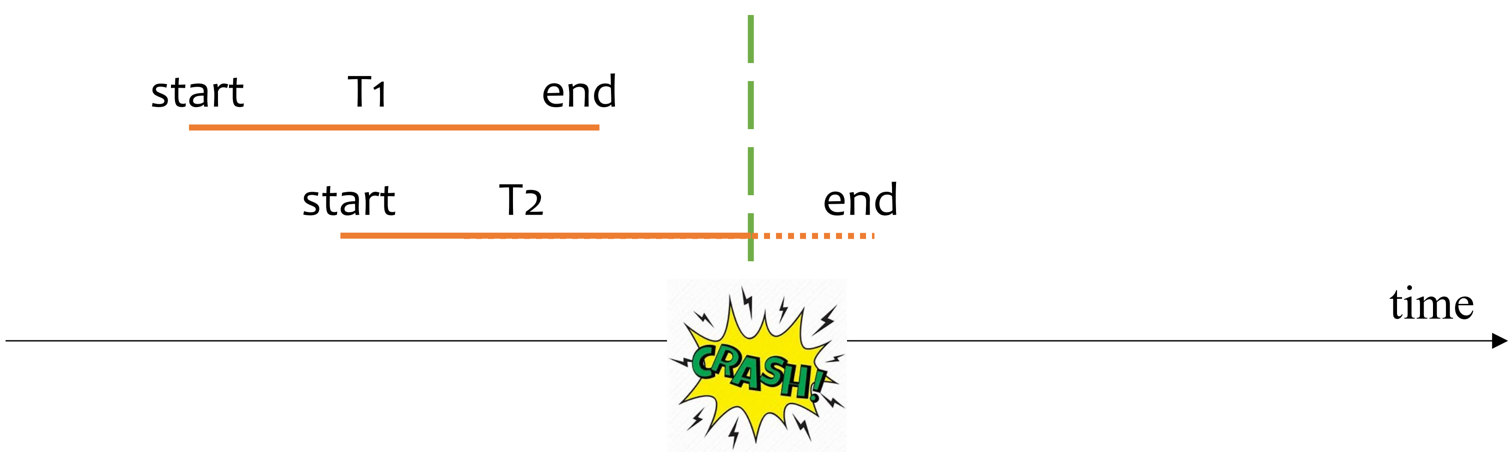
- The disk block containing *X* must be first brought into memory
- *X* is read/written in memory
- The memory block containing *X*, if modified, must be written back (flushed) to disk eventually



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*)

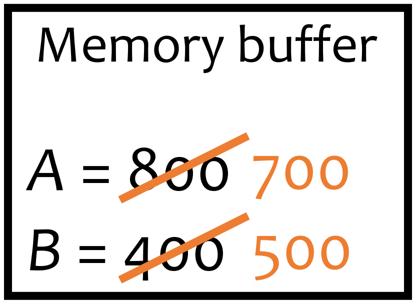
read(*A*, *a*); *a* = *a* – 100;

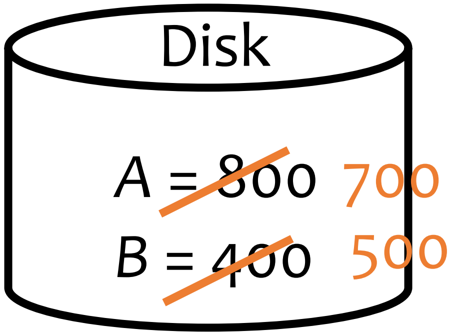
write(*A*, *a*);

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

Bad!

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

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*)

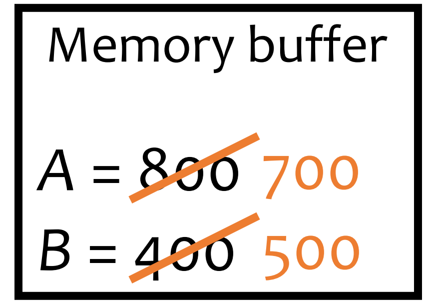
read(*A*, *a*); *a* = *a* – 100;

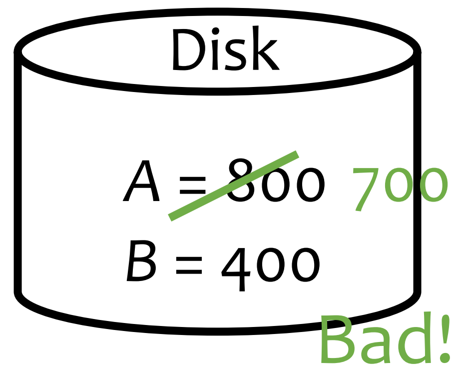
write(*A*, *a*);

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

write(*B*, *b*);

commit;





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

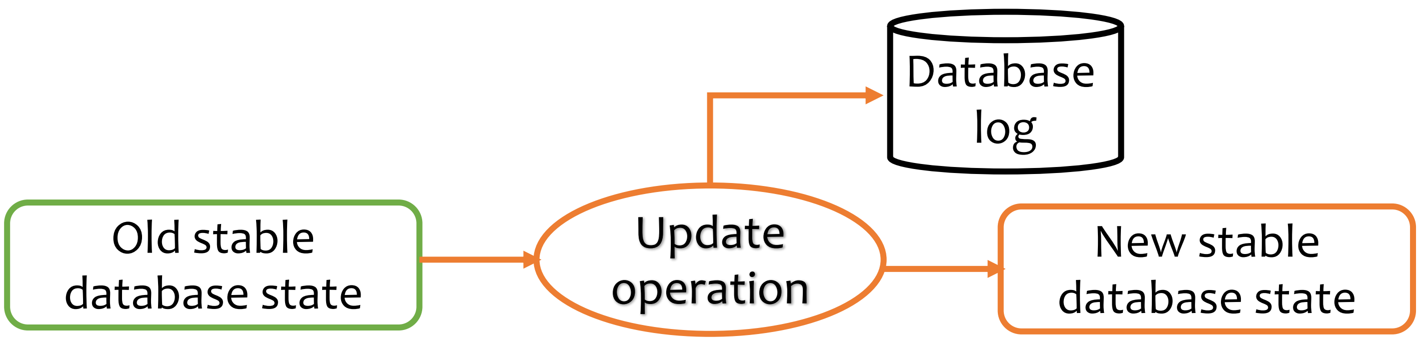
Naïve approach

- 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

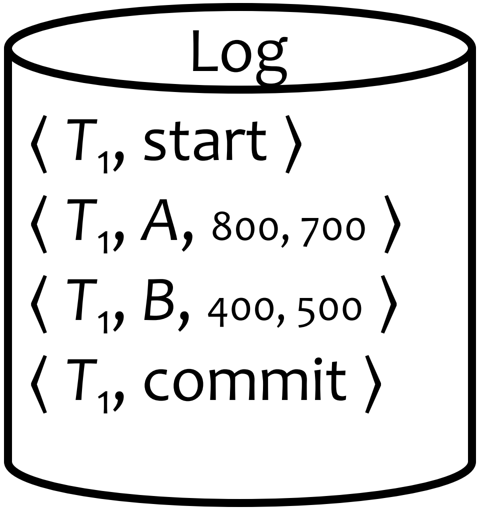
- Database log: sequence of log records, recording all changes made to the database, written to stable storage (e.g., disk) during normal operation



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

Log format

- When a transaction T_i starts
  - ⟨ 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
  - X identifies the data item
- A transaction T_i is committed when its commit log record is written to disk
  - ⟨ T_i, commit ⟩



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

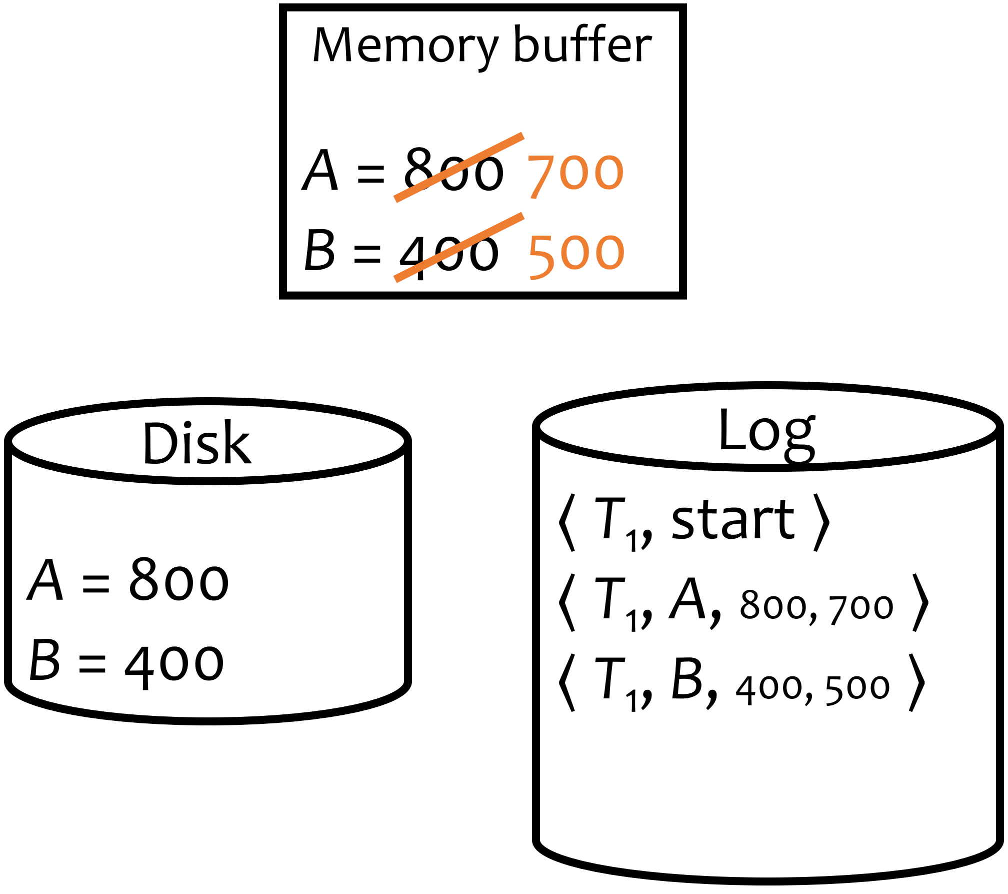
T1 (balance transfer of $100 from *A* to *B*)

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 ⟩

⟨ 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;

Memory buffer

A = ~~800~~ 700

B = ~~400~~ 500

No force: can flush after commit

Disk

A = 800

B = 400

Log

⟨ T_1, start ⟩

⟨ T_1, *A*, 800, 700 ⟩

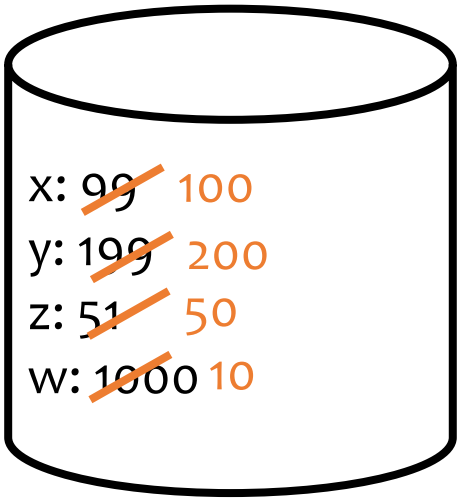
⟨ T_1, *B*, 400, 500 ⟩

⟨ T_1, 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:



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, z, 51
	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

Log

Start of 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, z, 51
	T_3, abort
	T_4, y, 200, 50

End of log →

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

Log

Start of 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, z, 51
	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

When txn manager receives abort, it logs reverse operations before abort

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

Log

Start of 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, z, 51
redo	T_3, abort
	T_4, y, 200, 50

End of log →

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

Log

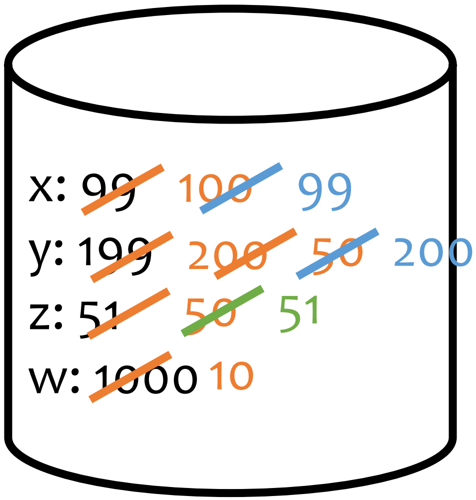
Start of log →

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, z, 51
redo	T_3, abort
redo	T_4, y, 200, 50

End of log

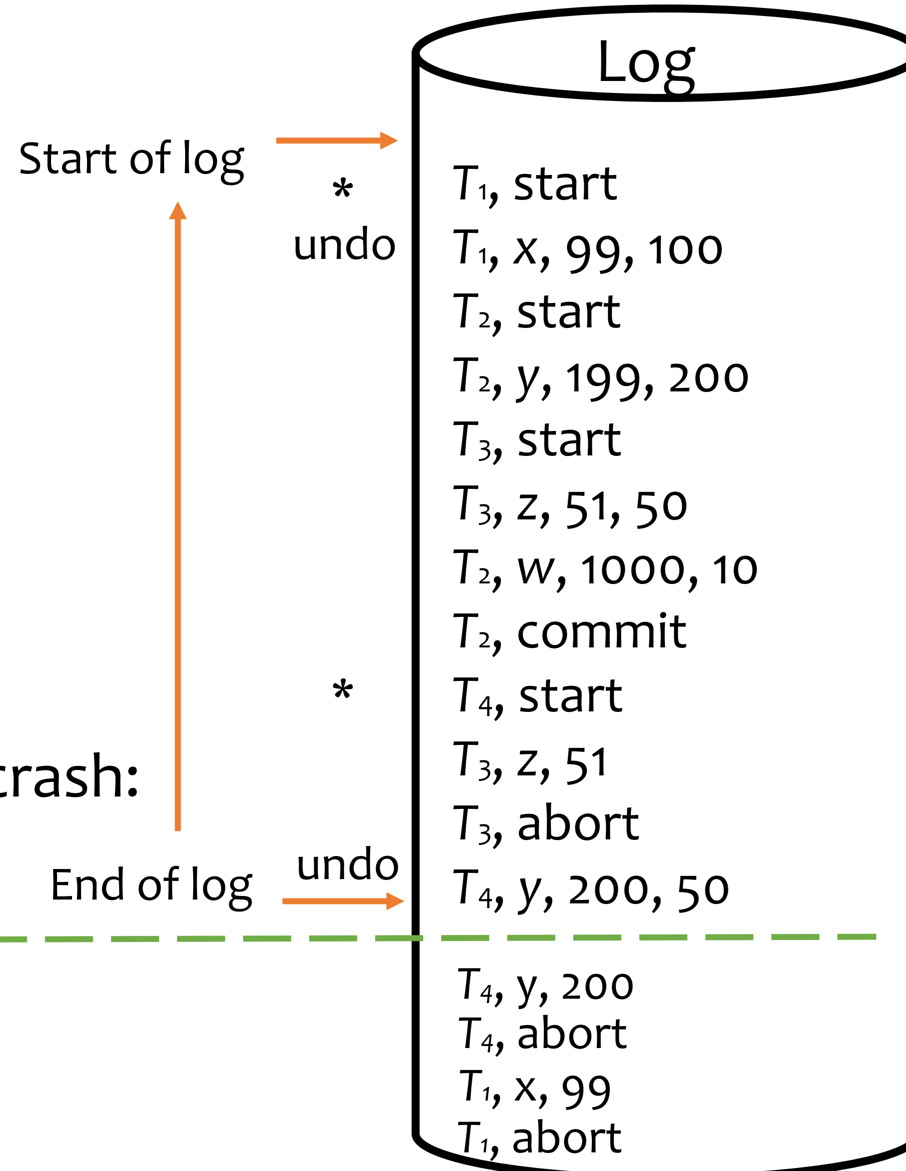
Log example - Undo

- Undo phase: T1, T4



List of active transactions at crash:

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



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*
  - That is, 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

Checkpointing

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

- Assumption: Txns cannot perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress
- Steps:
  - Output to the disk all modified buffer blocks
  - Add to log: <checkpoint *L*>, where L is a list of txns active at the time of the checkpoint
- After a system crash has occurred, the system examines the log to find the last <checkpoint *L*> record
  - The redo operations will start from the checkpoint record
  - The undo operations will start from the end of the log until the list of active transactions is empty

Summary

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

- Next lecture:
  - Other forms of durability: data replication
  - Atomicity when data is stored on different machines
  - Data privacy