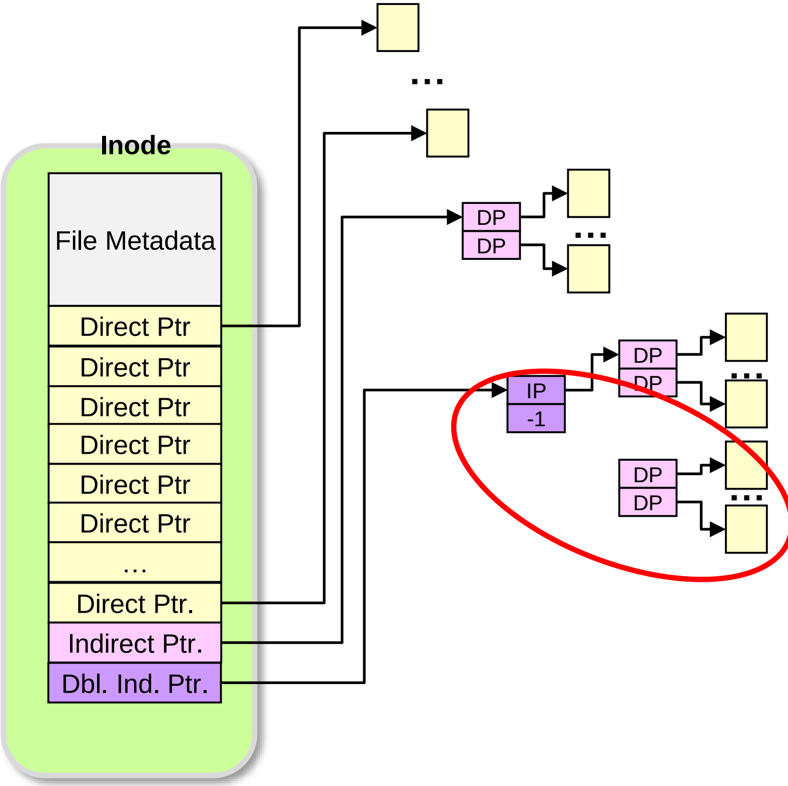
CSCI 350
Ch. 14 – Reliable Storage & Transactions

Mark Redekopp
Michael Shindler & Ramesh Govindan

Introduction

- Seeking reliability and consistency of file system
  - Consistency: If adding multiple blocks and we need to update the indirect pointers, a poorly timed crash could leave the file in an inconsistent state
  - Reliability: Data can get corrupted or lost due to mechanical/electrical issues
- Solutions
  - Transactions (we will focus on these)
  - Redundancy / Error-correction
    - RAID, ECC/Parity codes, checksums, etc.
    - See earlier units



Transactions

- A transaction is a set of updates to the state of one or more objects
- Terminology
  - Committed: If a transaction commits (succeeds) then the new state of the objects will be seen going forward [i.e. all updates occur]
  - Rollback: If a transaction rolls back (fails) then the object will remain in its original state (as if no updates to any part of the state were made) [i.e. no updates occur]

```
void threadTask(void* arg)
{
  /* Do local computation */

  /* checkpoints/saves state */
  begin_transaction(val1,val2) {

  /* Do some computation/updates */
  val1 -= amount;
  val2 += amount;
  } // end_transaction
  abort {
    // restore/re-read val1, val2
    // restart
  }

}
```

We have seen this before briefly in the context of multi-object synchronization. Now we'll focus on its application to file systems.

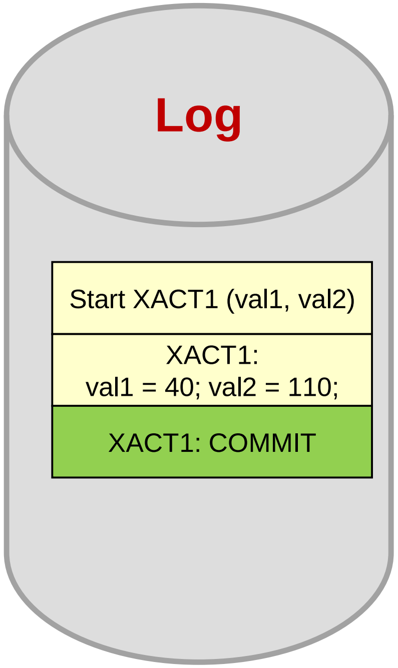
ACID Properties

- Transactions help achieve the ACID properties
  - Atomicity: Update appears as indivisible (all or nothing); no partial updates are visible
  - Consistency: Old state and new, updated state meet certain necessary invariants
    - E.g. No orphaned blocks, etc.
  - Isolation: Idea of serializability (transactions T appears to execute entirely before T' or vice versa)
  - Durability: Committed transactions are persistent

Logging

- Logging is a common way to achieve transactions
  - Maintains a log of "records" in persistent storage
- Steps:
  - Write intent (i.e. updates) to log
  - Write 'commit' to log (if no errors)
    - No going back now
  - Perform update
    - Actually carry out the updates described in the intent
  - Garbage collect (log entries, etc.)
    - Once the intentions are carried out successfully, we can now delete the log entry and any other temporary data

Original
val1 = 50; val2 = 100;
amount=10;



Start XACT1 (val1, val2)

XACT1:
val1 = 40; val2 = 110;

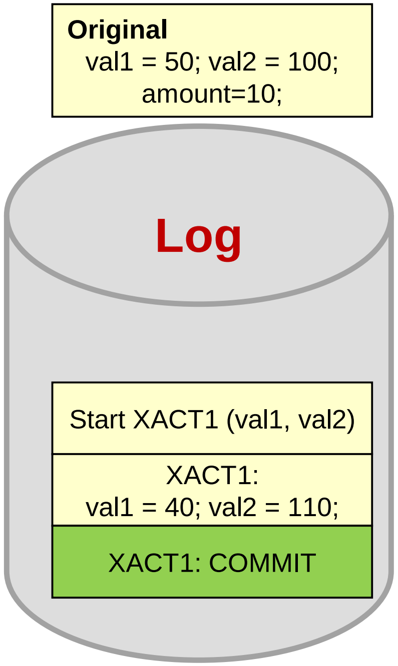
XACT1: COMMIT

Updated
val1 = 40; val2 = 110;
amount=10;

Recovery

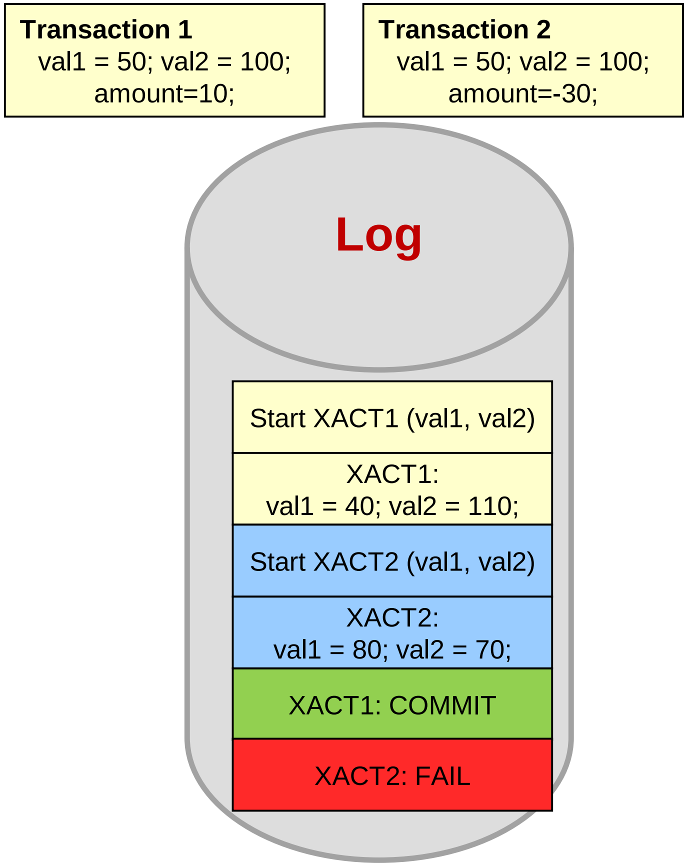
- If crash occurs before COMMIT is written, the transaction effectively is rolled back (original state is still present) and the log entry will be reclaimed on restart
- If crash occurs after step 2 completes, then the intentions/commit in the log will be replayed upon restart until all the intentions are carried out

1. Write intent (i.e. updates) to log
2. Write 'commit' to log
3. Perform update
4. Garbage collect (log entries, etc.)



Handling Concurrency

- Suppose two transactions attempt to execute concurrently
- Only 1 can successfully commit
- The other will need to roll back



Transaction 1
val1 = 50; val2 = 100;
amount=10;

Transaction 2
val1 = 50; val2 = 100;
amount=-30;

Log

Start XACT1 (val1, val2)

XACT1:
val1 = 40; val2 = 110;

Start XACT2 (val1, val2)

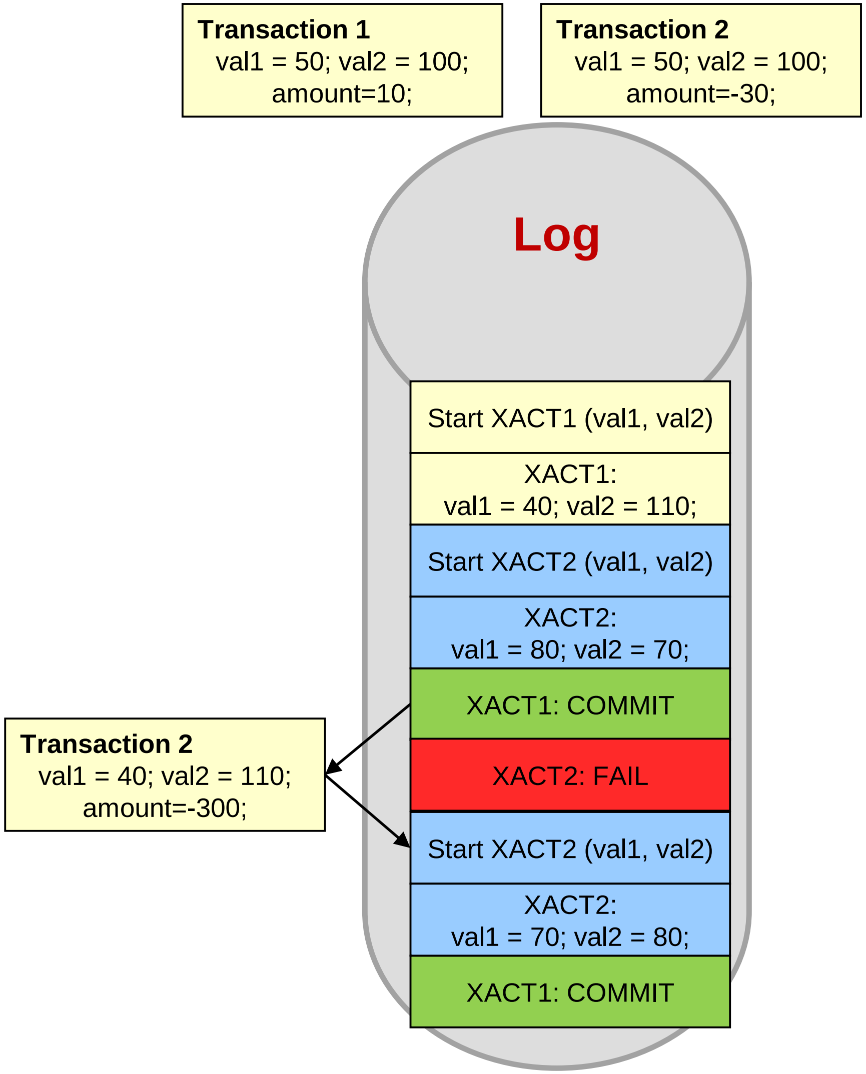
XACT2:
val1 = 80; val2 = 70;

XACT1: COMMIT

XACT2: FAIL

Handling Concurrency

- After rollback the second transaction will need to restart and thus use the update values
- It could potentially fail again based on some new transaction that commits before it, in which case it would replay again
  - Some priority can be used to help "older" transactions commit before "newer" ones



Transaction 1
val1 = 50; val2 = 100;
amount=10;

Transaction 2
val1 = 50; val2 = 100;
amount=-30;

Log

Start XACT1 (val1, val2)

XACT1:
val1 = 40; val2 = 110;

Start XACT2 (val1, val2)

XACT2:
val1 = 80; val2 = 70;

XACT1: COMMIT

XACT2: FAIL

Start XACT2 (val1, val2)

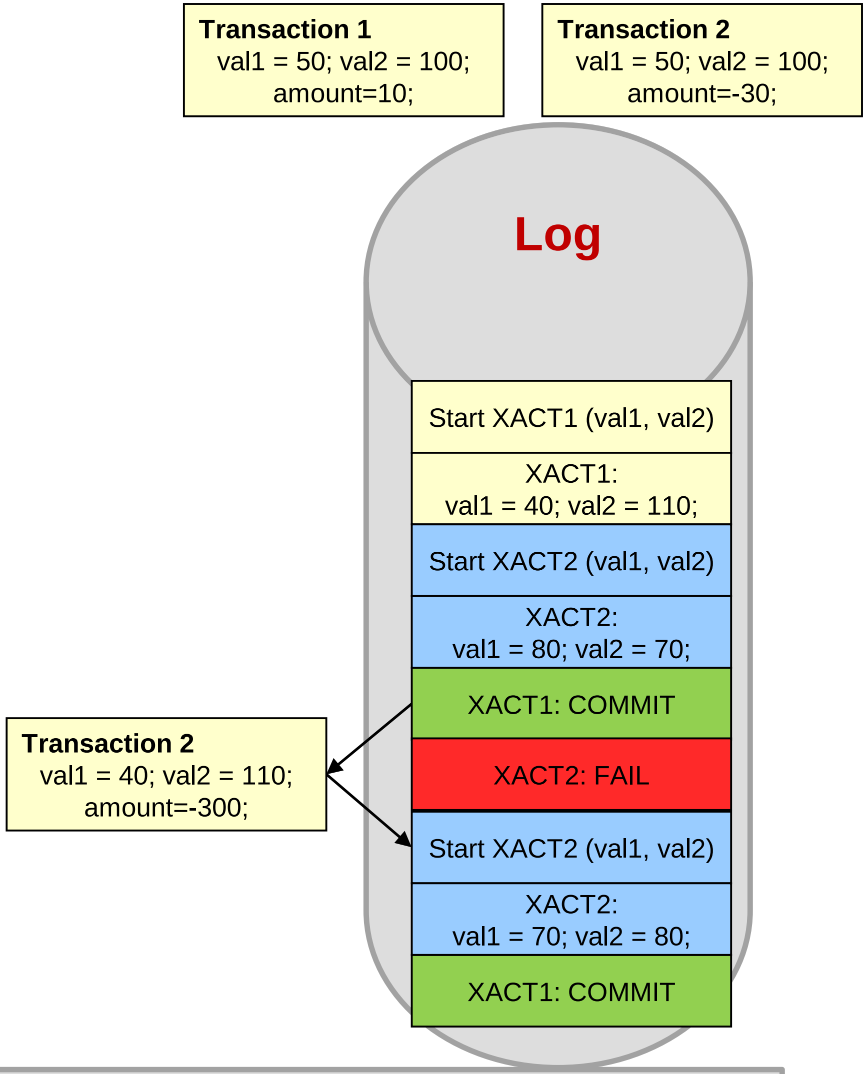
XACT2:
val1 = 70; val2 = 80;

XACT1: COMMIT

Transaction 2
val1 = 40; val2 = 110;
amount=-300;

Redo Logging

- The process outlined in the past several slides are known as "redo logging"
  - On a crash, the committed transactions will be "redone"
  - If another crash before the transaction can be "redone" it will simply try again on the next restart and continue retrying until successful
- Alternative: "Undo Logging"
  - Make updates in place but write old values to the log
  - On rollback, replace the new values with the old ones in the log



Transaction 1
val1 = 50; val2 = 100;
amount=10;

Transaction 2
val1 = 50; val2 = 100;
amount=-30;

Log

Start XACT1 (val1, val2)

XACT1:
val1 = 40; val2 = 110;

Start XACT2 (val1, val2)

XACT2:
val1 = 80; val2 = 70;

XACT1: COMMIT

XACT2: FAIL

Start XACT2 (val1, val2)

XACT2:
val1 = 70; val2 = 80;

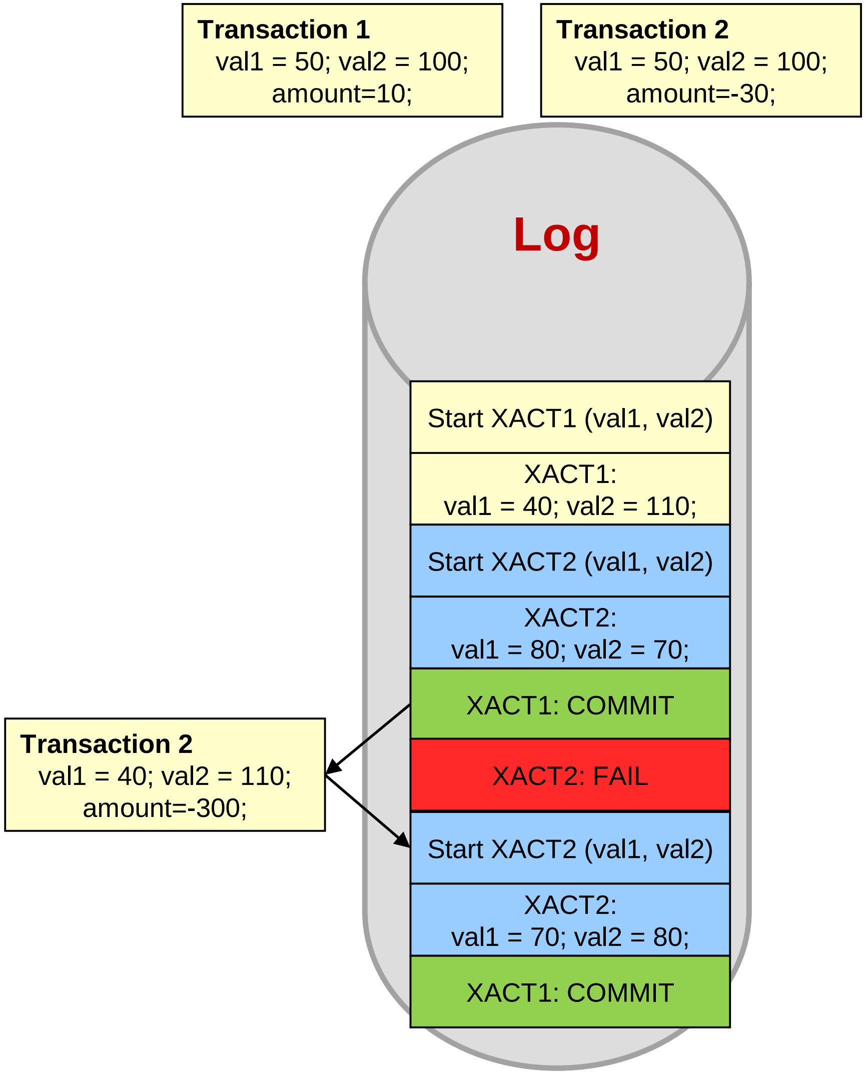
XACT1: COMMIT

Transaction 2
val1 = 40; val2 = 110;
amount=-300;

Which to use? Each has their advantages. What do we expect more of: successful or failed transactions?

Idempotent Operations

- Updates must be **idempotent** (i.e. redoing it once compared to many times leaves the same result)
- Notice the log store the values we wanted to write to the variables
  - Writes are idempotent (e.g. writing 40 to val1 once and then repeating it will still leave val1 with 40)
- If our log store val1 -= 10 then each replay would deduct another 10 from val1



Transaction 1
val1 = 50; val2 = 100;
amount=10;

Transaction 2
val1 = 50; val2 = 100;
amount=-30;

Log

Start XACT1 (val1, val2)

XACT1:
val1 = 40; val2 = 110;

Start XACT2 (val1, val2)

XACT2:
val1 = 80; val2 = 70;

XACT1: COMMIT

XACT2: FAIL

Start XACT2 (val1, val2)

XACT2:
val1 = 70; val2 = 80;

XACT1: COMMIT

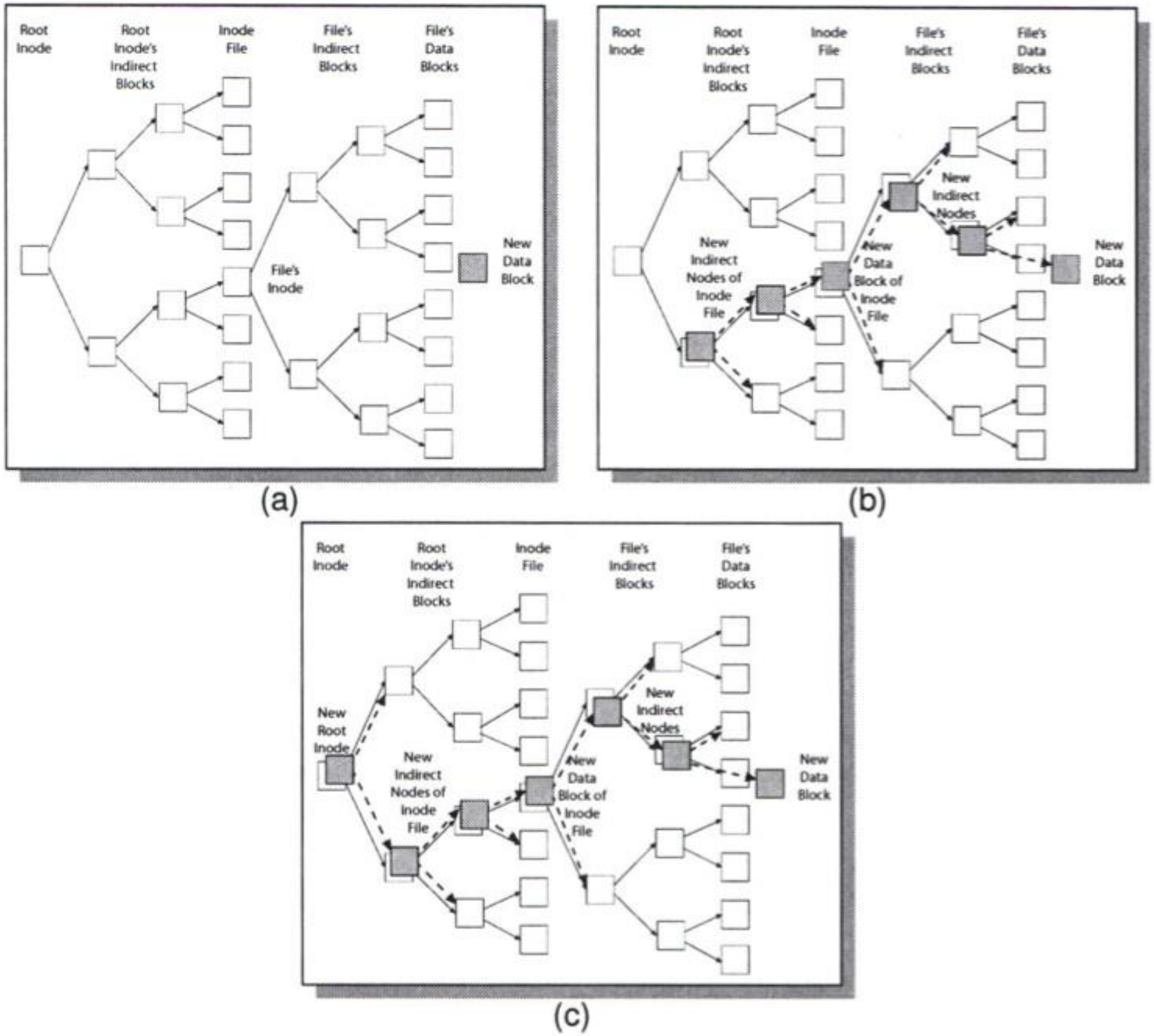
Transaction 2
val1 = 40; val2 = 110;
amount=-300;

Performance of Redo Logging

- Transactions may seem like a lot of overhead but…
  - Writes to the log are sequential
    - We've learned how sequential writes are faster than random writes
  - Actual updates (step 3) can be asynchronous
    - Updates can be batched together and performed at an "opportune" time
    - Caller can return and proceed as soon as commit is written
    - Don't wait too long though as then recovery time is slower due to "replay" of many updates and log itself takes more space since a transaction in the log can't be reclaimed until it is completed
    - Writes can be scheduled as a batch (rather than FIFO)

Logging and File Systems

- Need to ensure all metadata is updated according to ACID principles



Use of Logging In File Systems

- Two variants
  - Journaling:
    - Use of a logging for updates to metadata (i.e. inodes, free-space map, etc.)
    - But actual data is updated in place (so file data itself can be inconsistent)
    - Used by NTFS, Apple's HFS+, and Linux's XFS
      - Linux's ext3 and ext4 FS can be configured for journaling
  - Logging
    - Use of a log for both metadata and file data
      - Linux's ext3 and ext4 can also be configured to do logging
- COW file systems are inherently transactional
  - Only when the root node (uberblock) is update does new data become visible (i.e. transaction commits)