Crash Recovery

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

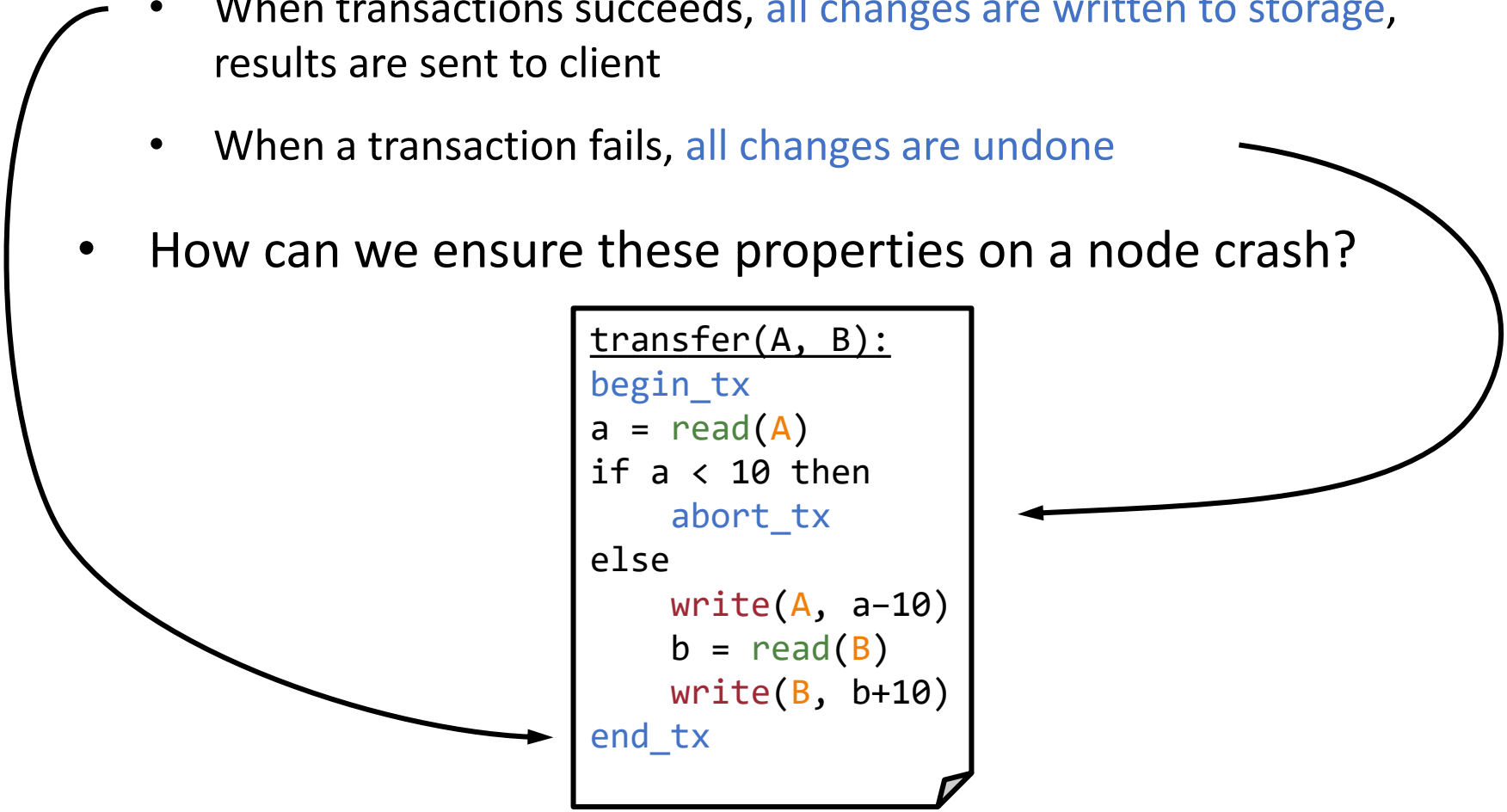
Distributed Systems
ECE419

Overview

- Introduction to crash recovery
- Shadow copy
- Write-ahead logging
- Checkpointing

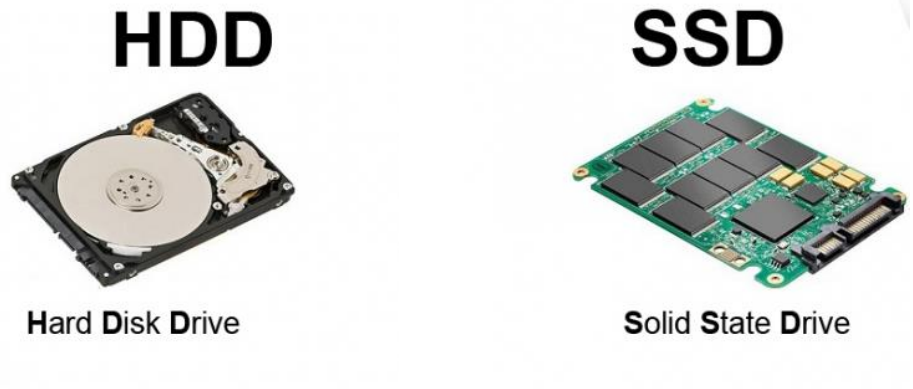
Review of transactions

- We have seen that transactional code may read and write multiple items (e.g., A, B)
  - When transactions succeeds, all changes are written to storage, results are sent to client
  - When a transaction fails, all changes are undone
- How can we ensure these properties on a node crash?



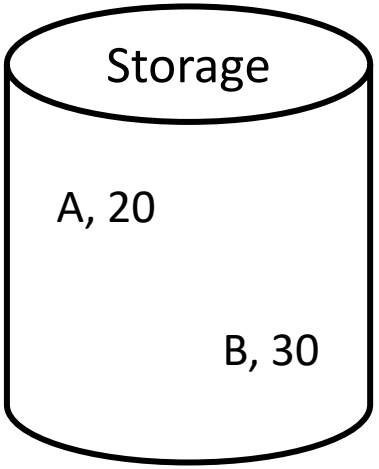
Data storage

- Assume data is cached in memory (DRAM) and stored durably on storage (hard drive, SSD, etc.)
- We will assume crash-recovery failures
  - Crashes are fail-stop
  - Data in memory is lost
  - Data on storage survives crashes, also called stable storage



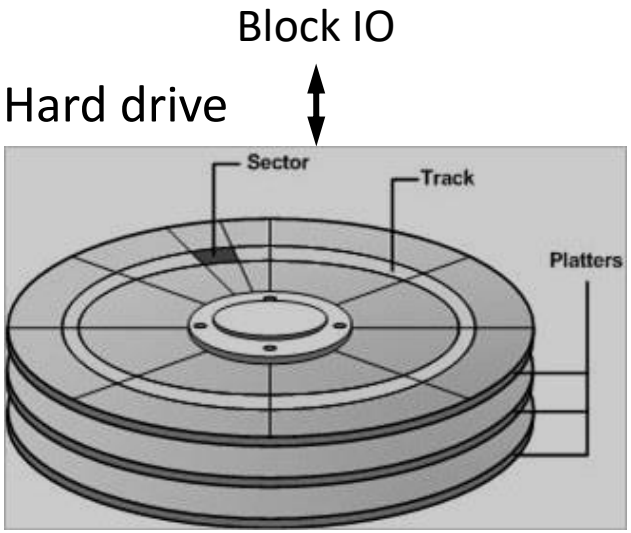
Memory

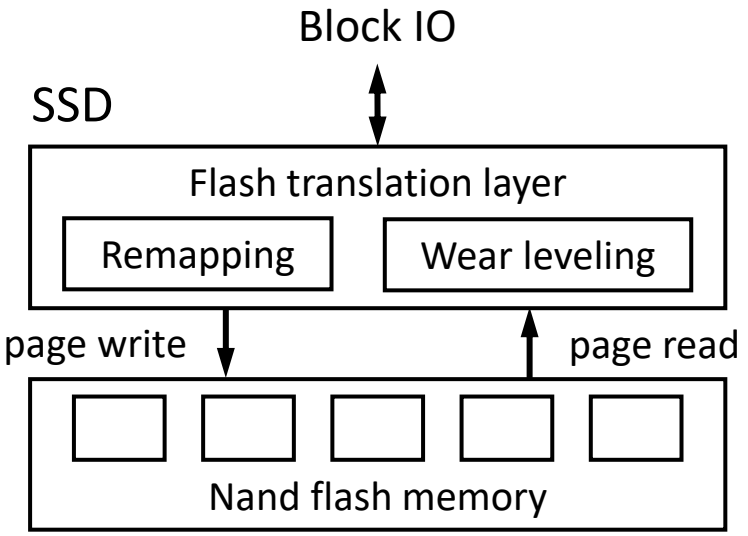
Key	Value
A	20
B	30



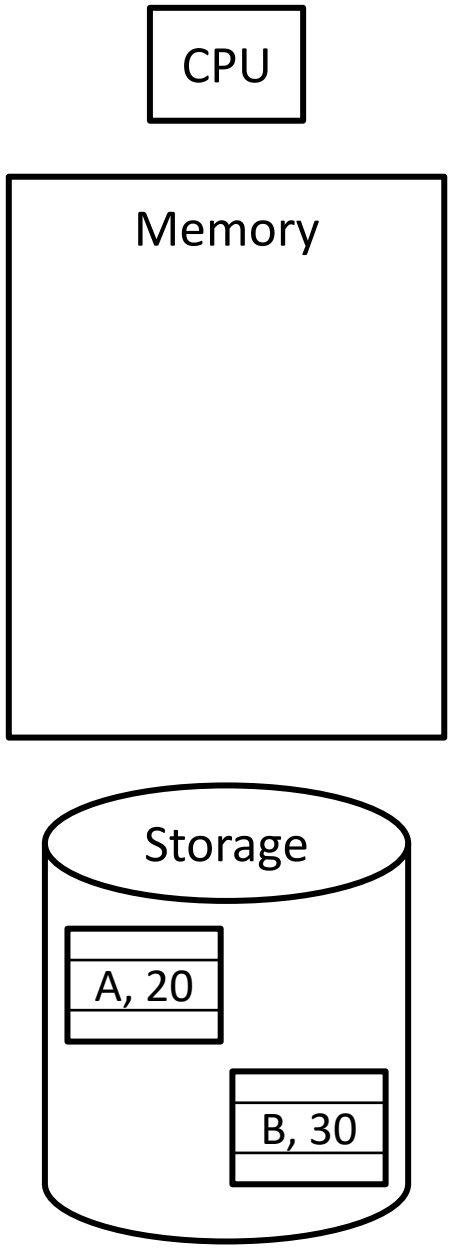
Storage access granularity

- Hard drives and SSDs read and write fixed-size blocks
  - Typically, 512 to 4096 contiguous bytes
  - Blocks are called sectors on hard drives, and pages on SSDs



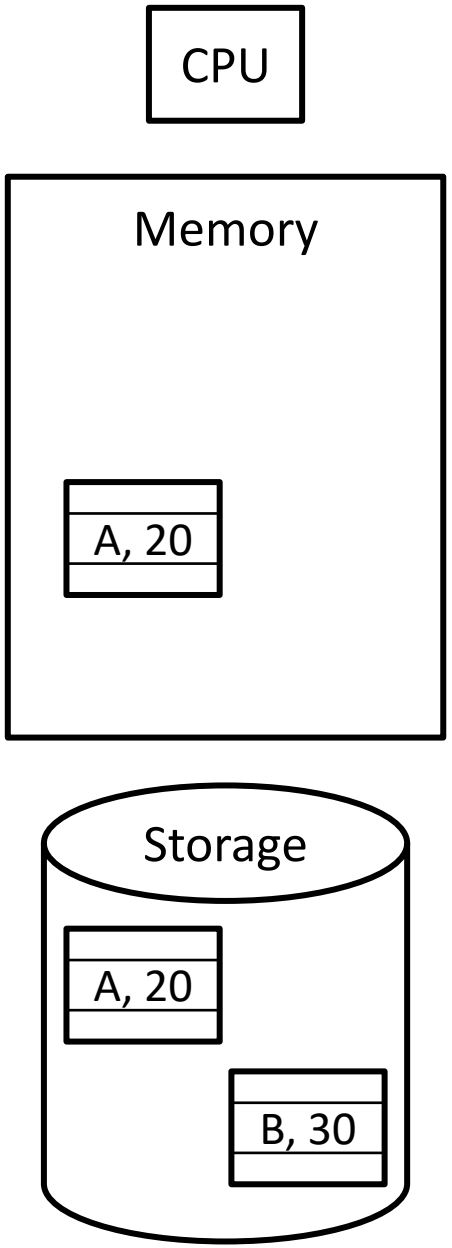


Data access model



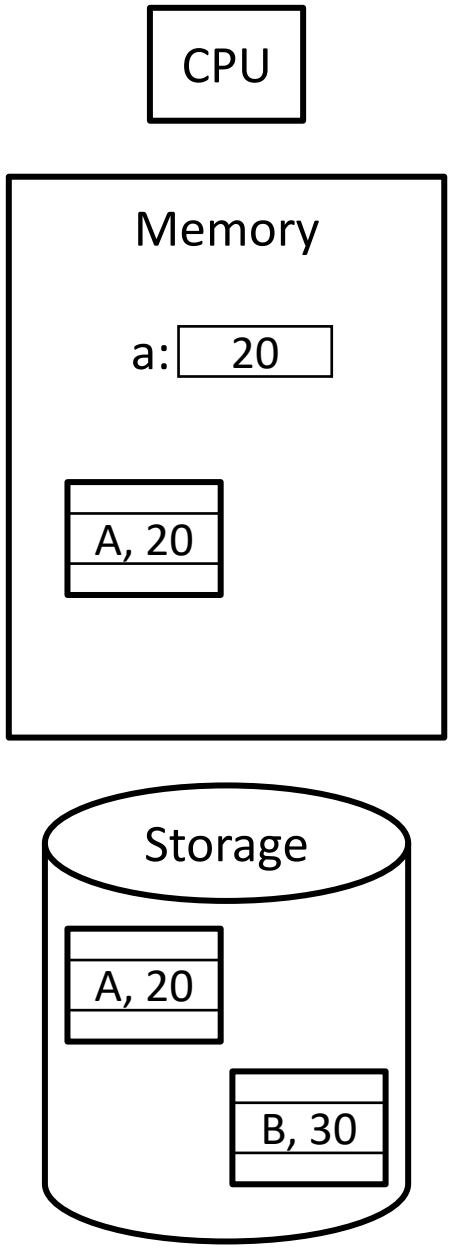
Data access model

- input(A)
  - Read the storage block containing item A from storage into memory (block is cached)



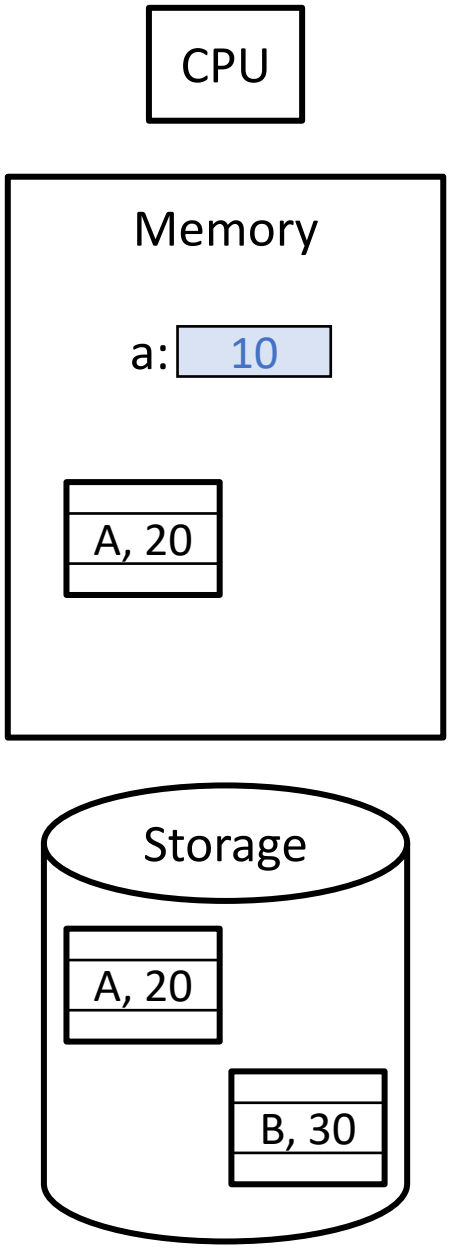
Data access model

- input(A)
  - Read the storage block containing item A from storage into memory (block is cached)
- a = read(A)
  - Read value of item A into a local variable ‘a’, execute input(A) first if necessary



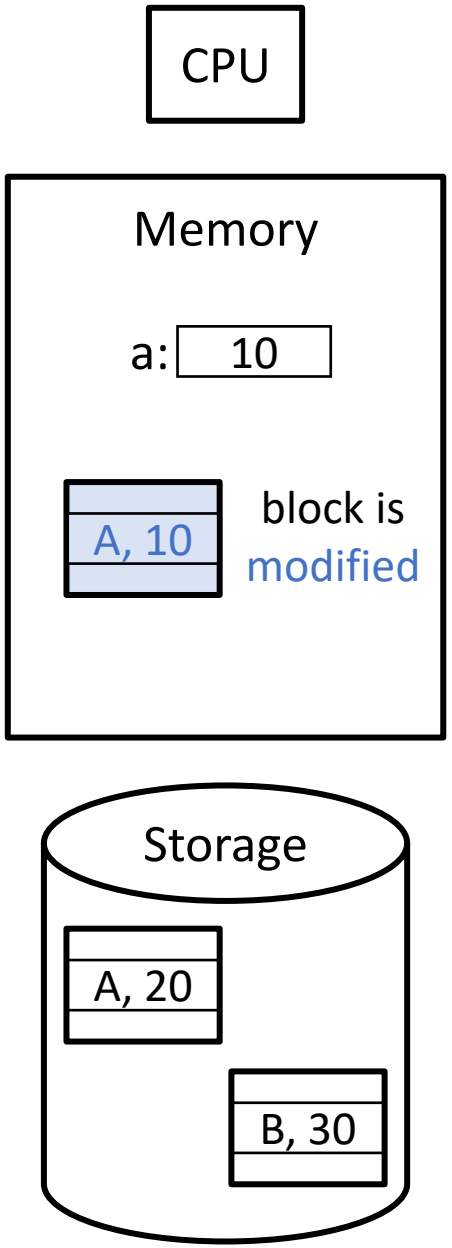
Data access model

- input(A)
  - Read the storage block containing item A from storage into memory (block is cached)
- a = read(A)
  - Read value of item A into a local variable ‘a’, execute input(A) first if necessary
- Programmer modifies variable ‘a’



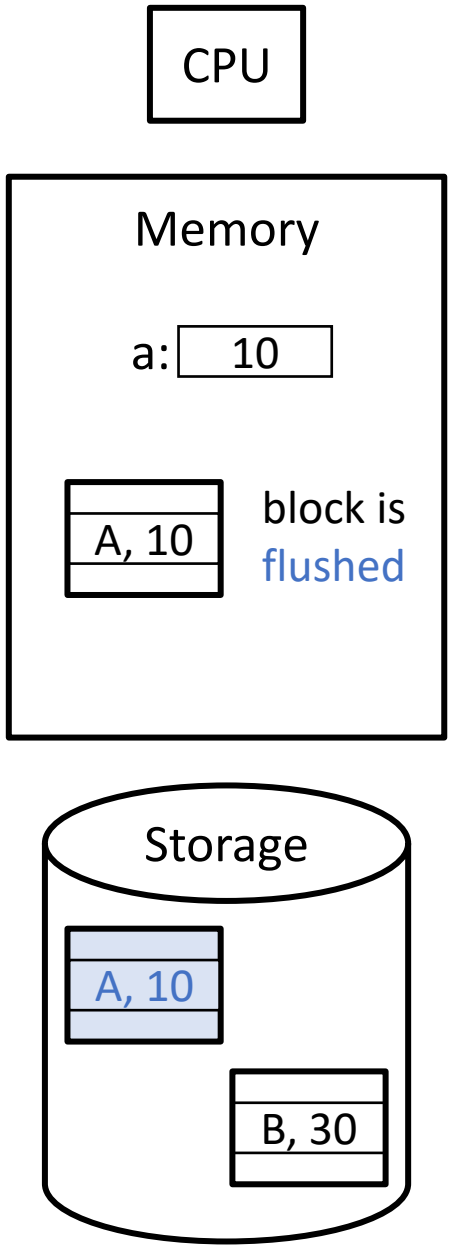
Data access model

- input(A)
  - Read the storage block containing item A from storage into memory (block is cached)
- a = read(A)
  - Read value of item A into a local variable ‘a’, execute input(A) first if necessary
- Programmer modifies variable ‘a’
- write(A, a)
  - Write local variable ‘a’ to item A in memory, execute input(A) if needed



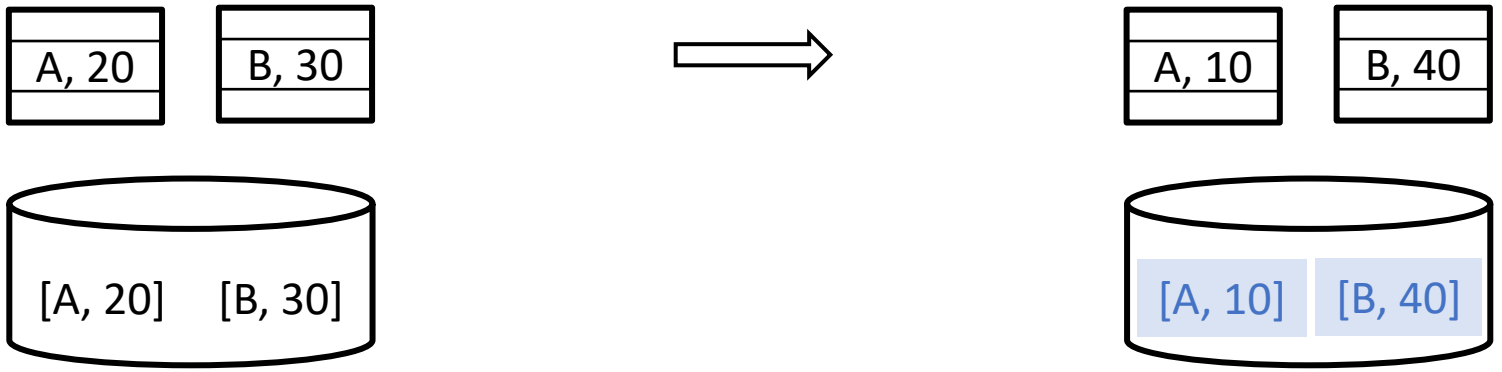
Data access model

- input(A)
  - Read the storage block containing item A from storage into memory (block is cached)
- a = read(A)
  - Read value of item A into a local variable ‘a’, execute input(A) first if necessary
- Programmer modifies variable ‘a’
- write(A, a)
  - Write local variable ‘a’ to item A in memory, execute input(A) if needed
- output(A)
  - Write memory block containing item A to storage durably



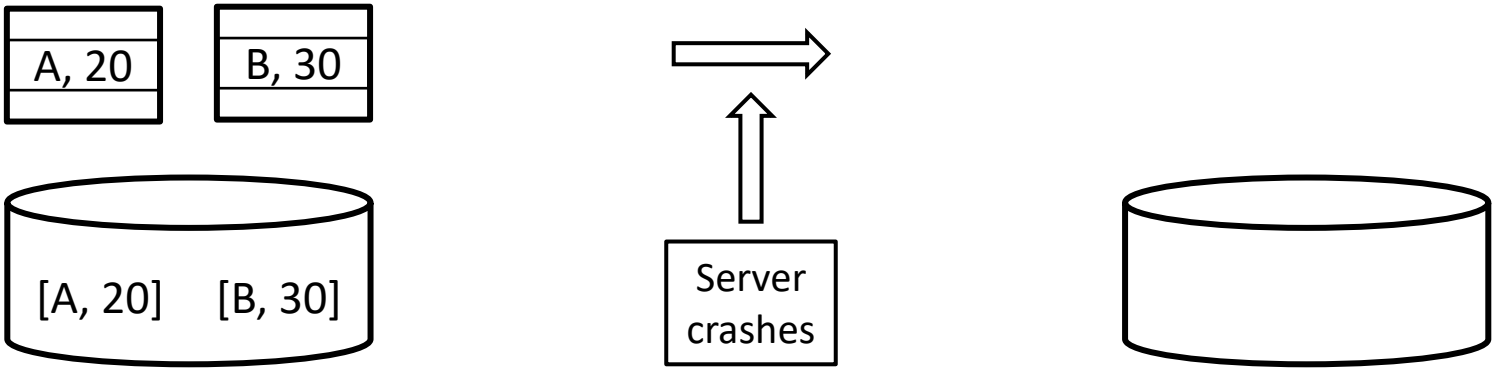
Server operation

- Say client issues transfer(A, B)
- Storage system
  - Updates A and B in memory and storage
  - Responds to client



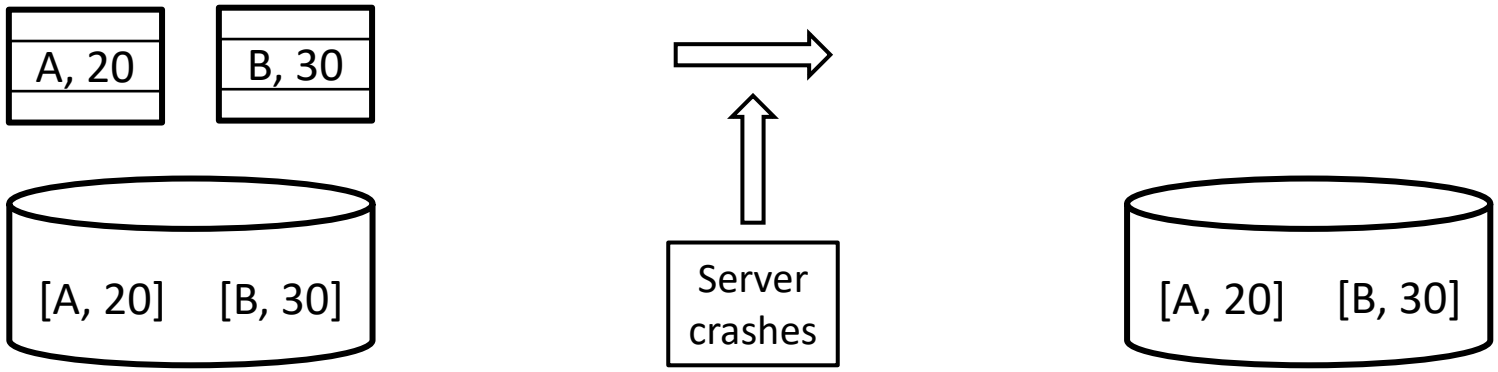
Server crashes

- Suppose server crashes while transfer(A, B) in progress
  - Memory contents are lost on reboot
- What could go wrong?
  1. A and B not updated in storage, client gets response
  2. A and B updated in storage, client doesn't get response
  3. One of A or B updated in storage



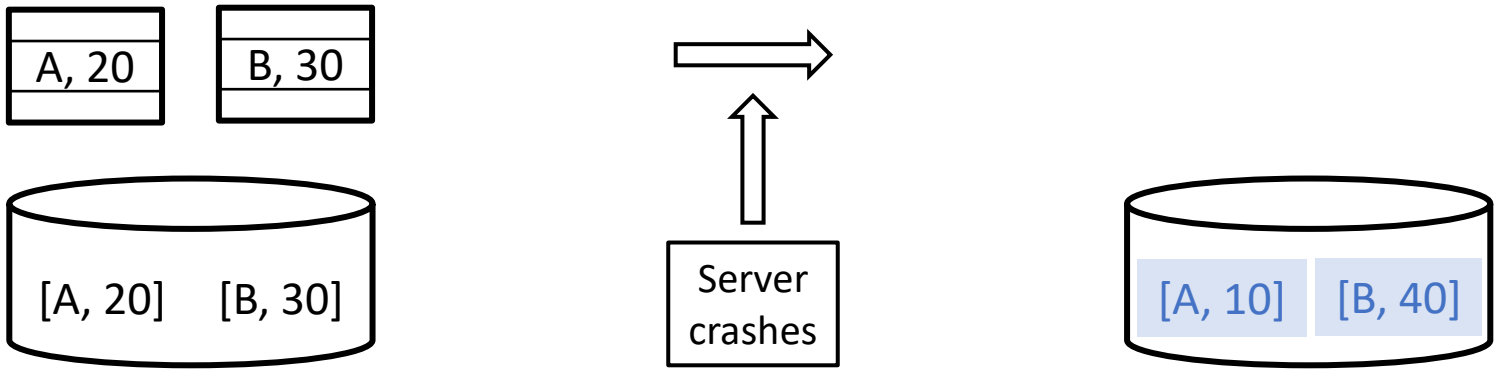
Handling server crashes

1. A and B not updated in storage, client gets response
   - We need to write A and B to storage before sending response, then a completed operation is not lost on failure
   - This property is called durability



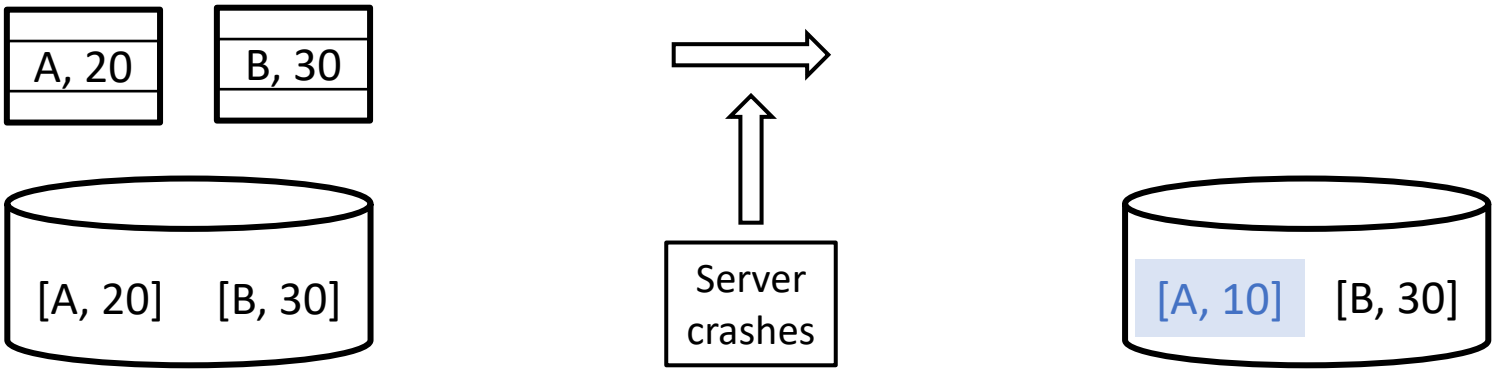
Handling server crashes

2. A and B updated in storage, client doesn't get response
   - Client needs to check whether transaction succeeded, or
   - Client retries request,
     server needs to detect duplicate request, ignore executing it,
     return previous saved result (exactly-once semantics)



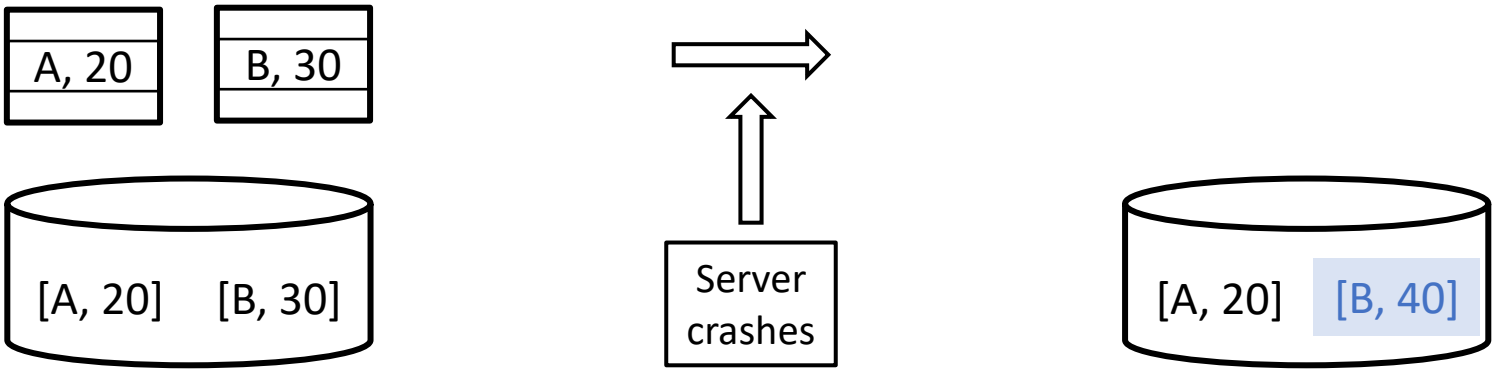
Handling server crashes

3. One of A or B updated in storage
   - If only A updated, client loses



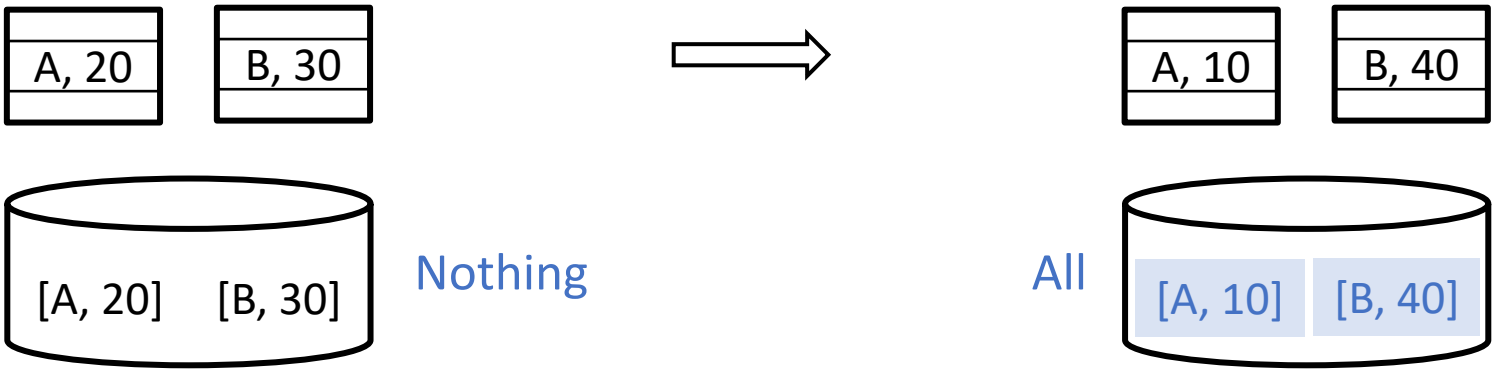
Handling server crashes

3. One of A or B updated in storage
   - If only B updated, bank loses



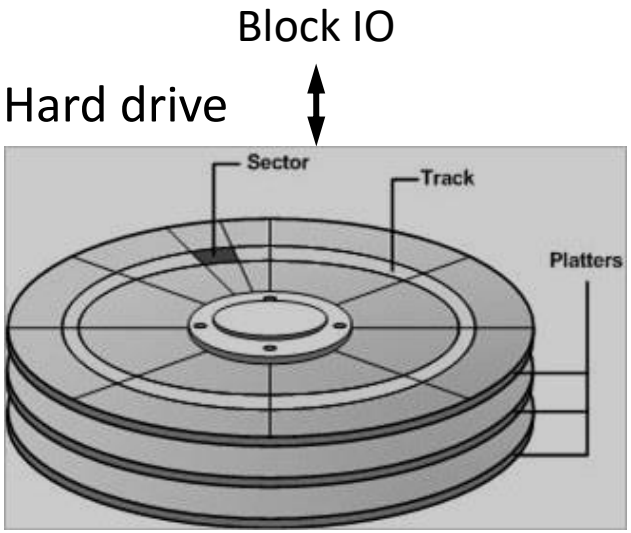
Failure atomicity

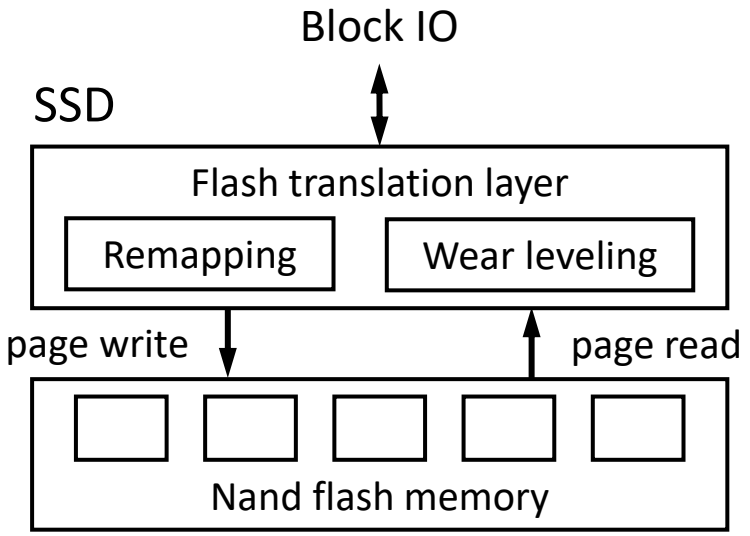
- Say client issues transfer(A, B)
  - For modified A and B, we need to ensure that all updates are on storage, or none of them are on storage
  - This property is called failure (all-or-nothing) atomicity
  - Without it, storage can become inconsistent after crashes
- Ensuring failure atomicity is key challenge with crashes



Storage failure atomicity

- Hard drives and SSDs guarantee that a block is written failure atomically, i.e., block is fully written or not at all, even under system crash or power failure





Why is failure atomicity hard?

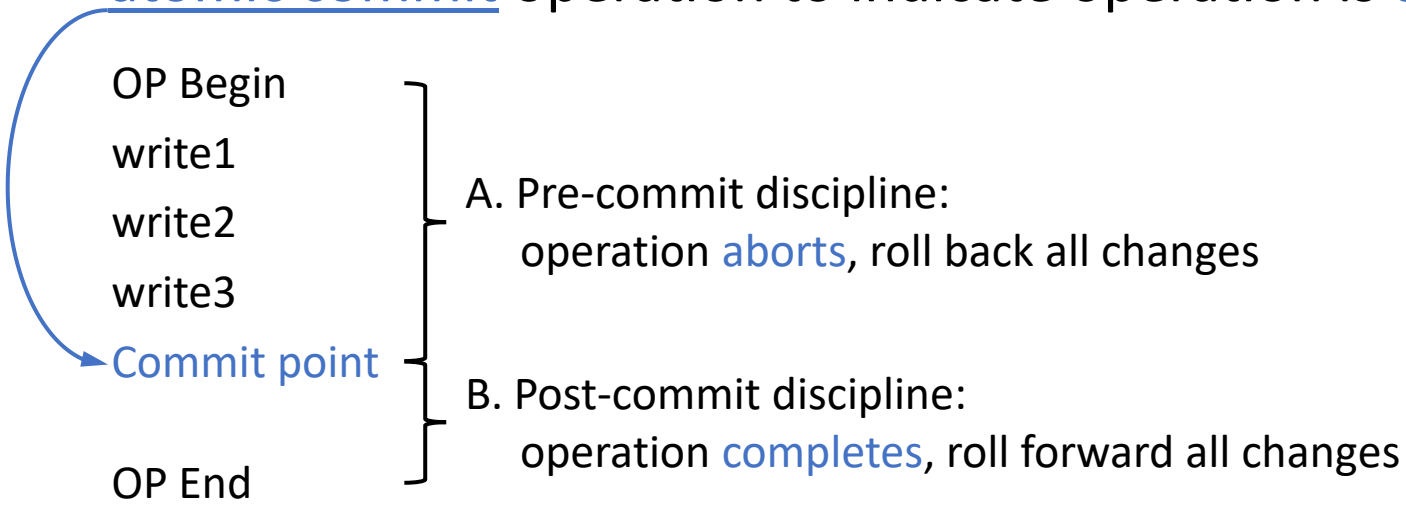
- Say client issues transfer(A, B)
- Problem occurs when A and B lie in different blocks
  - Some (but not all) of these blocks could be written on crash
  - After a crash, it is not possible to revert these writes
- Problem is worse with more complex data structures
  - If items are larger than block size
  - If items are variable sized, reallocated when resizing
  - If we use a separate index (e.g., hash table) to look up keys, then inserts require updating items and index on storage
  - If we use separate checksum blocks to detect storage errors
  - …

Keys ideas for ensuring failure atomicity

- Idea 1: When modifying a block,
  make a copy of a block on storage,
  then we have both old and new block version on storage
  - If operation doesn't complete, use the old version
  - If operation completes, use the new version
  - But how do we know when an operation is complete?

Keys ideas for ensuring failure atomicity

- Idea 2: After operation's writes, perform an atomic commit operation to indicate operation is done



A. If crash occurs before commit point, we abort operation by rolling back all blocks to their old version

B. Otherwise, we commit operation by rolling forward all blocks to their new version

- Doing A or B after a crash ensures failure atomicity and is called crash recovery

Shadow copy

Shadow copy

- Used by editors, compilers, etc., to ensure that files remain intact on crash
  - Pre-commit
    - Create a complete working copy of the file to be modified
    - Make changes to the working copy, ensure it is not visible to others
  - Commit point
    - Atomically exchange working copy with original copy
    - Requires lower-level atomic method, e.g., rename system call
  - Post-commit
    - Release space occupied by original copy
  - Recovery
    - What should be done on abort, commit?
- Shadow copy requires making a full copy, expensive

Write-Ahead Logging

Write-ahead logging (WAL)

- A general technique for providing failure atomicity
  - Key idea: log modified item before overwriting it on storage
- Logging: append a record for each modified item into a log
  - Log contains copy of data item
  - Append ensures no data in log is overwritten
- WAL: flush log record for modified item before flushing item to storage
  - Then copy is written to storage before original is overwritten
  - This ordering ensures roll back or roll forward is possible

Recovery schemes using WAL

- Let's look at two recovery schemes that use WAL
  - Undo logging: performs roll back only
  - Redo logging: performs roll forward only

- Log record format:

Id	Type	Item	Value

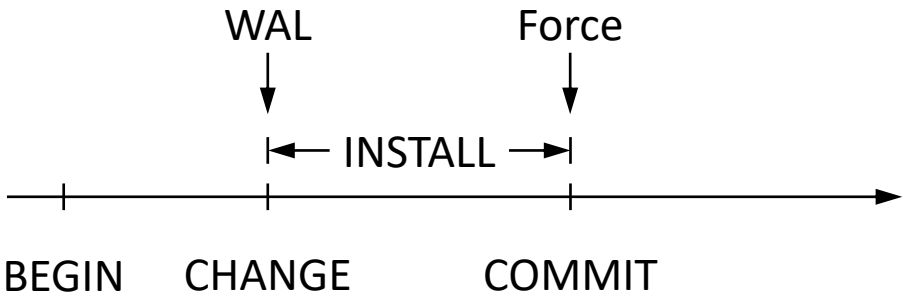
  - Id: operation id
  - Type: BEGIN, CHANGE, COMMIT, END
  - Item: physical location of item on storage (block id, offset)
  - Value: physical value of item (physical logging)

Undo logging

- Log old value of modified item in the log record
- Undo logging discipline
  - WAL: flush log record before flushing modified item
  - Force: flush all modified items before flushing commit record to log
- Recovery
  - Pre-commit crash: roll back updates using old values from log
  - Post-commit crash: all updates of the operation have been applied
- Commonly used in database systems

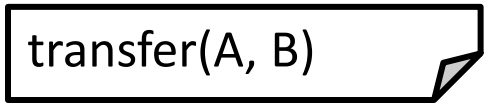
Undo logging operation

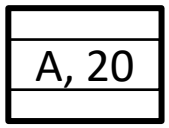
- Logging API
  - Pre-commit
    - id = LOG(BEGIN) // start operation
    - LOG(id, CHANGE, item, old_value)
  - Commit point
    - LOG(id, COMMIT)
  - Post-commit
    - Nothing

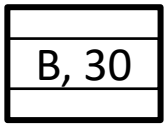


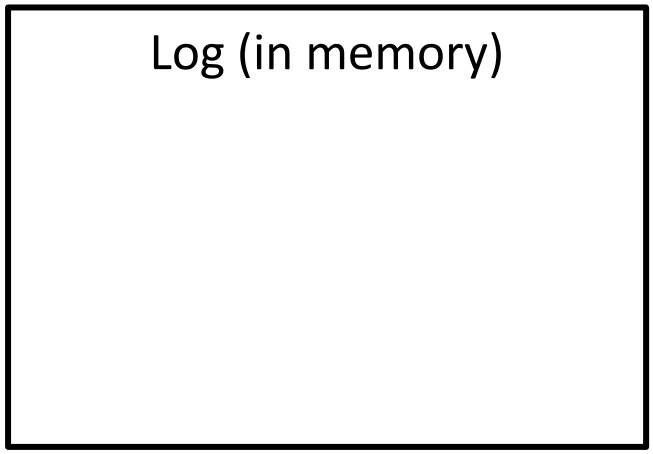
- Install API
  - INSTALL(item, value) // flush item's value
    - new_value during normal operation between CHANGE and COMMIT
    - old_value from log during (pre-commit) crash recovery

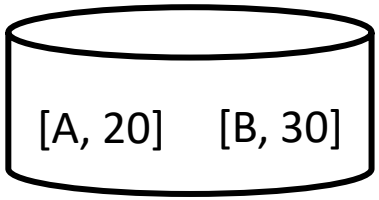
Undo logging example

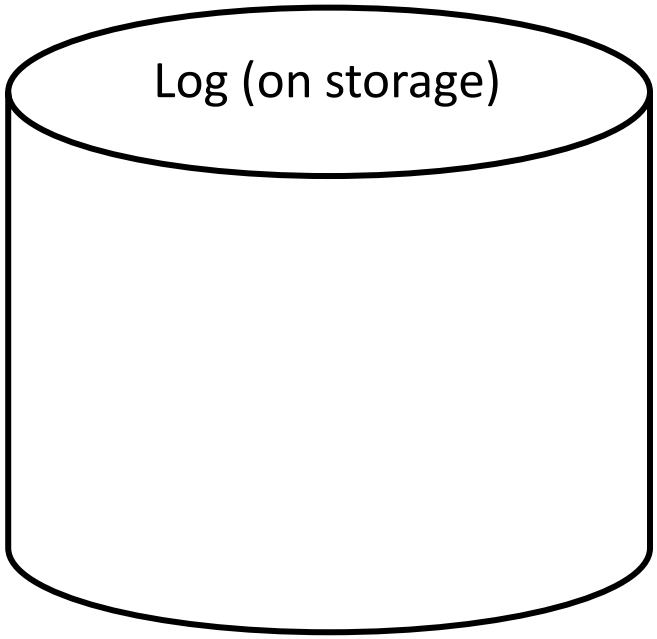




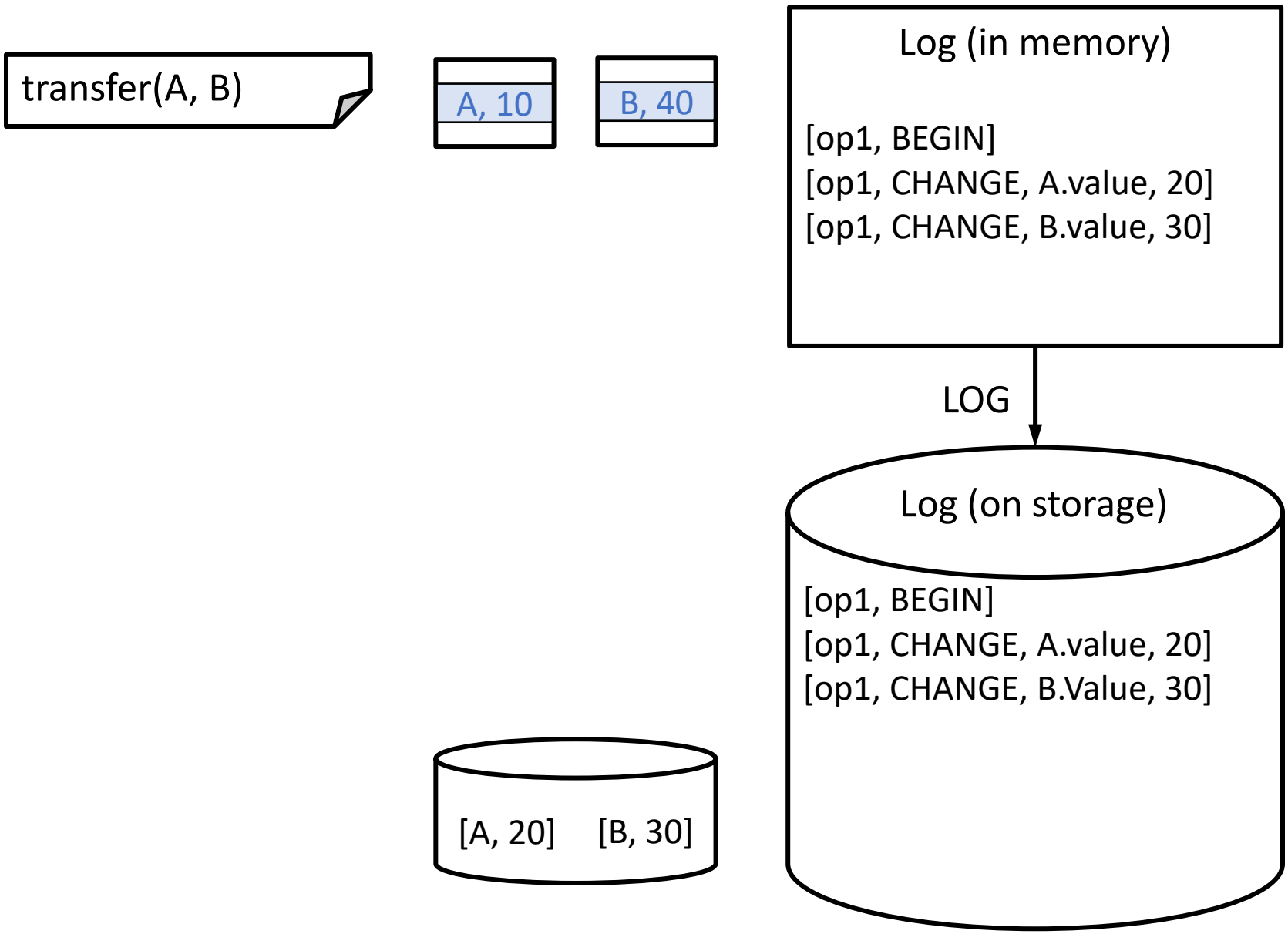




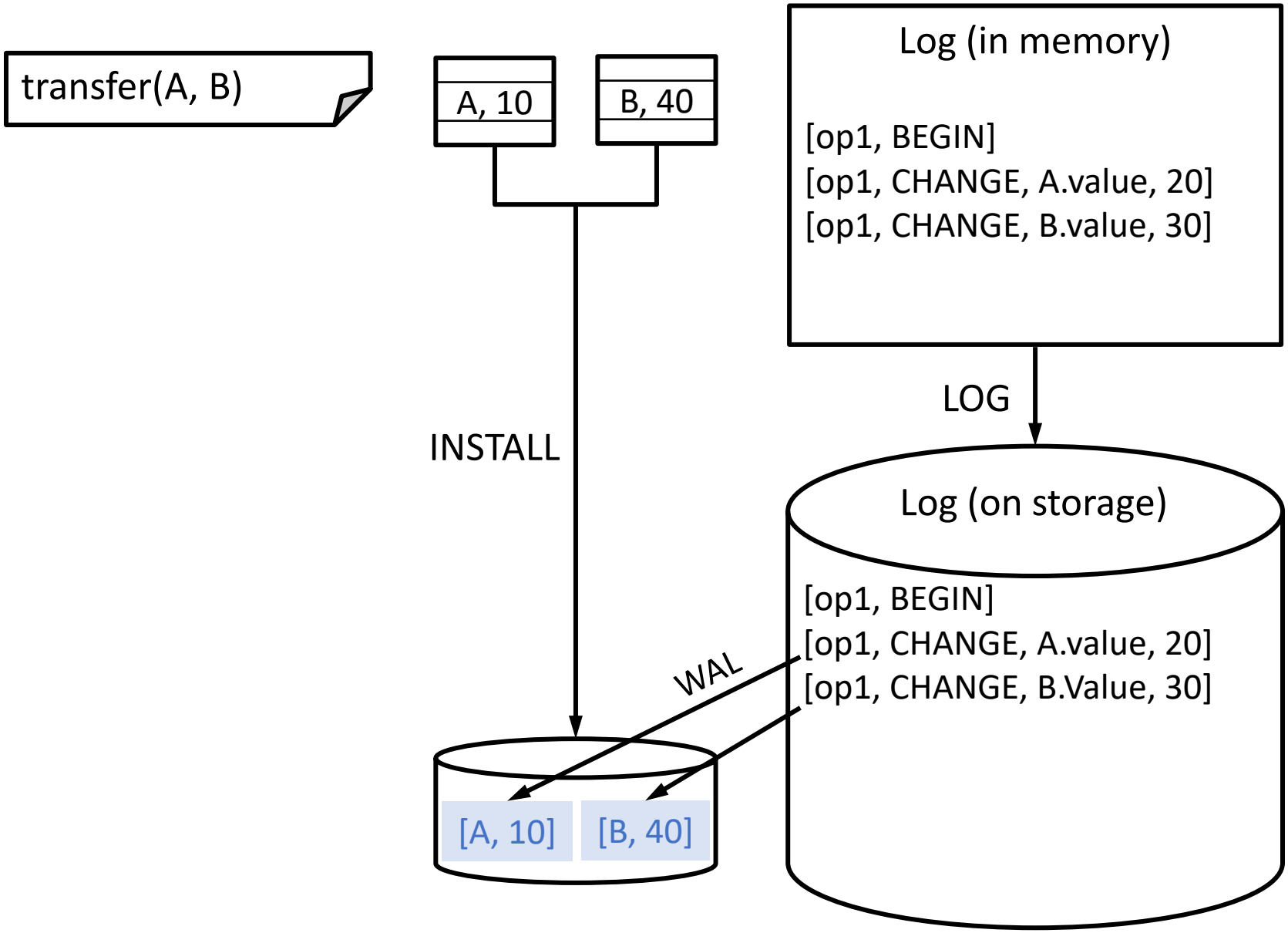




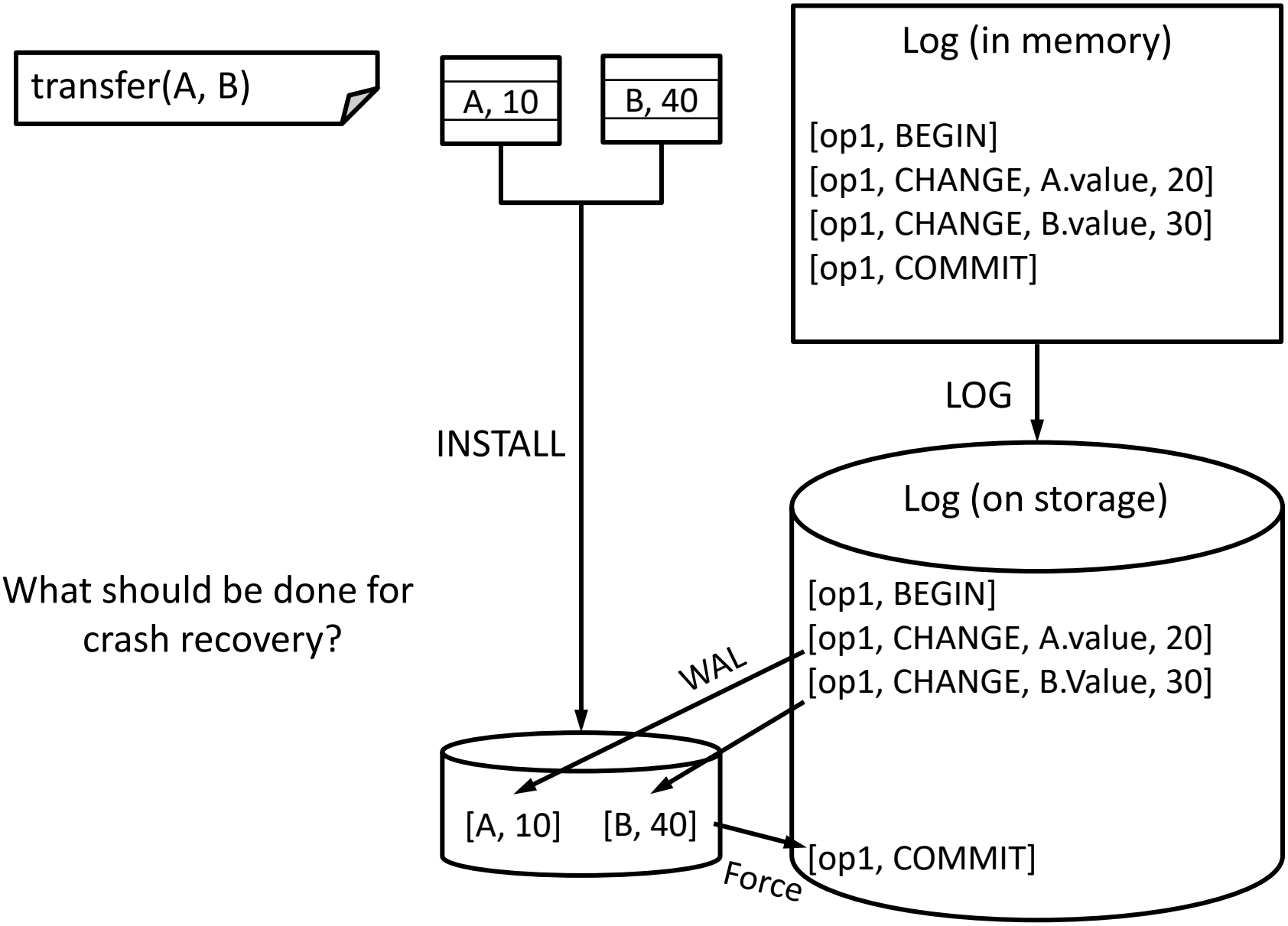
Undo logging example



Undo logging example



Undo logging example

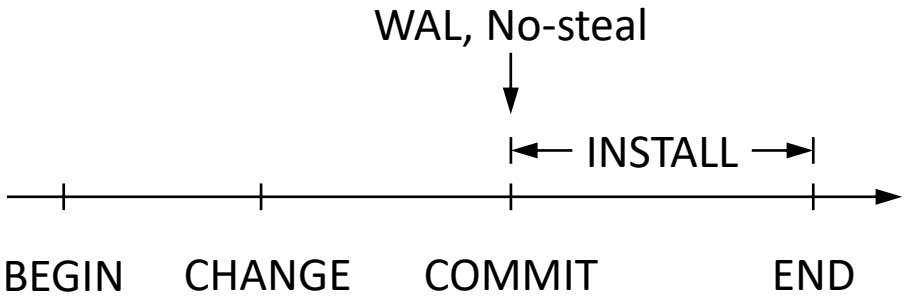


Redo logging

- Log new value of modified item in the log record
- Redo logging discipline
  - WAL: flush log record before flushing modified item
  - No-steal: flush commit record before flushing any modified items
- Recovery
  - Pre-commit crash: no updates of the operation have been applied
  - Post-commit crash: roll forward updates using new values from log
- Commonly used
  - Replicated systems (e.g., Raft, ZooKeeper)
  - File systems (e.g., Ext4, ZFS intent log)
  - Persistent key-value stores (e.g., RocksDB)

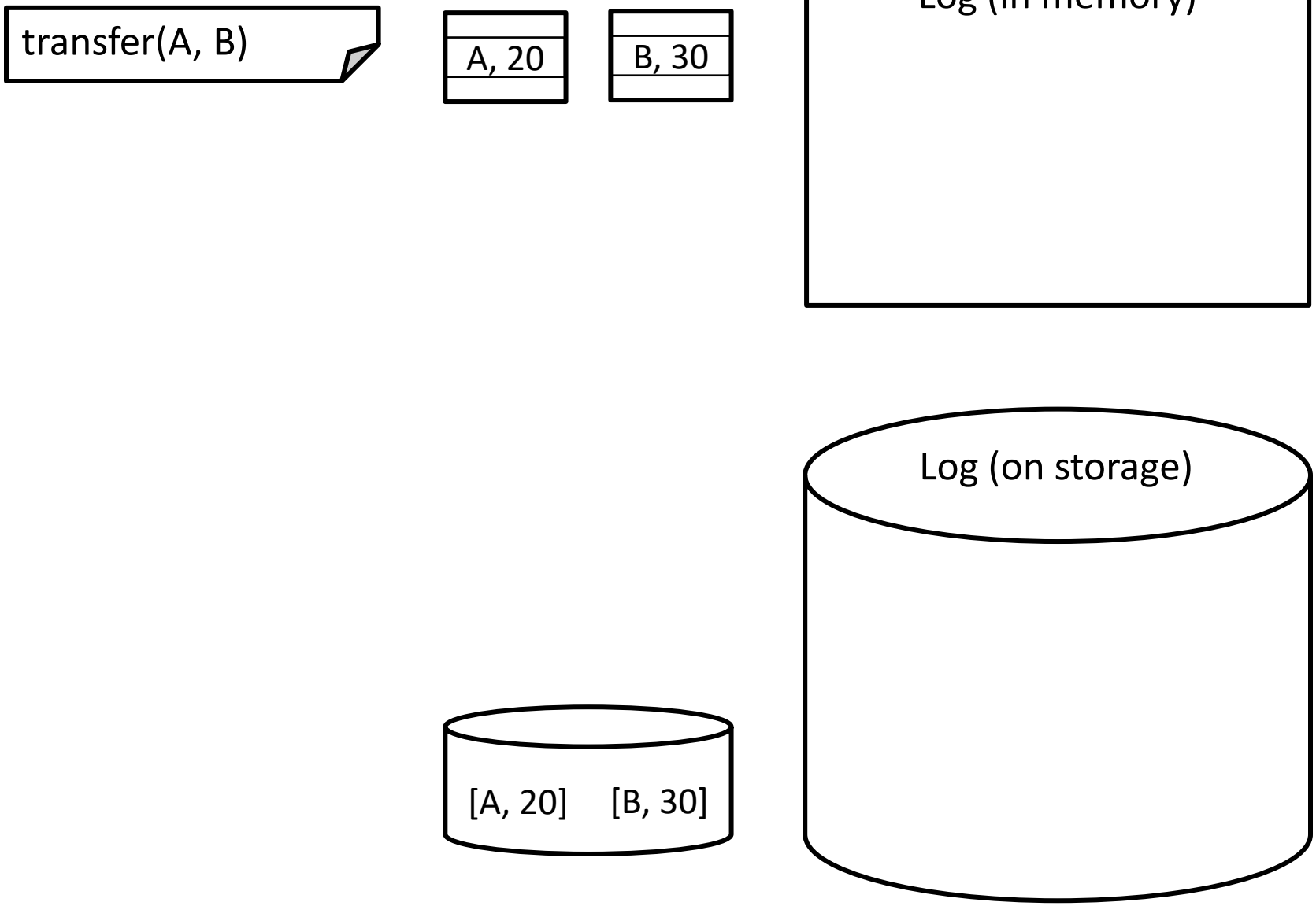
Redo logging operation

- Logging API
  - Pre-commit
    - id = LOG(BEGIN) // start operation
    - LOG(id, CHANGE, item, new_value)
  - Commit point
    - LOG(id, COMMIT)
  - Post-commit
    - LOG(id, END)

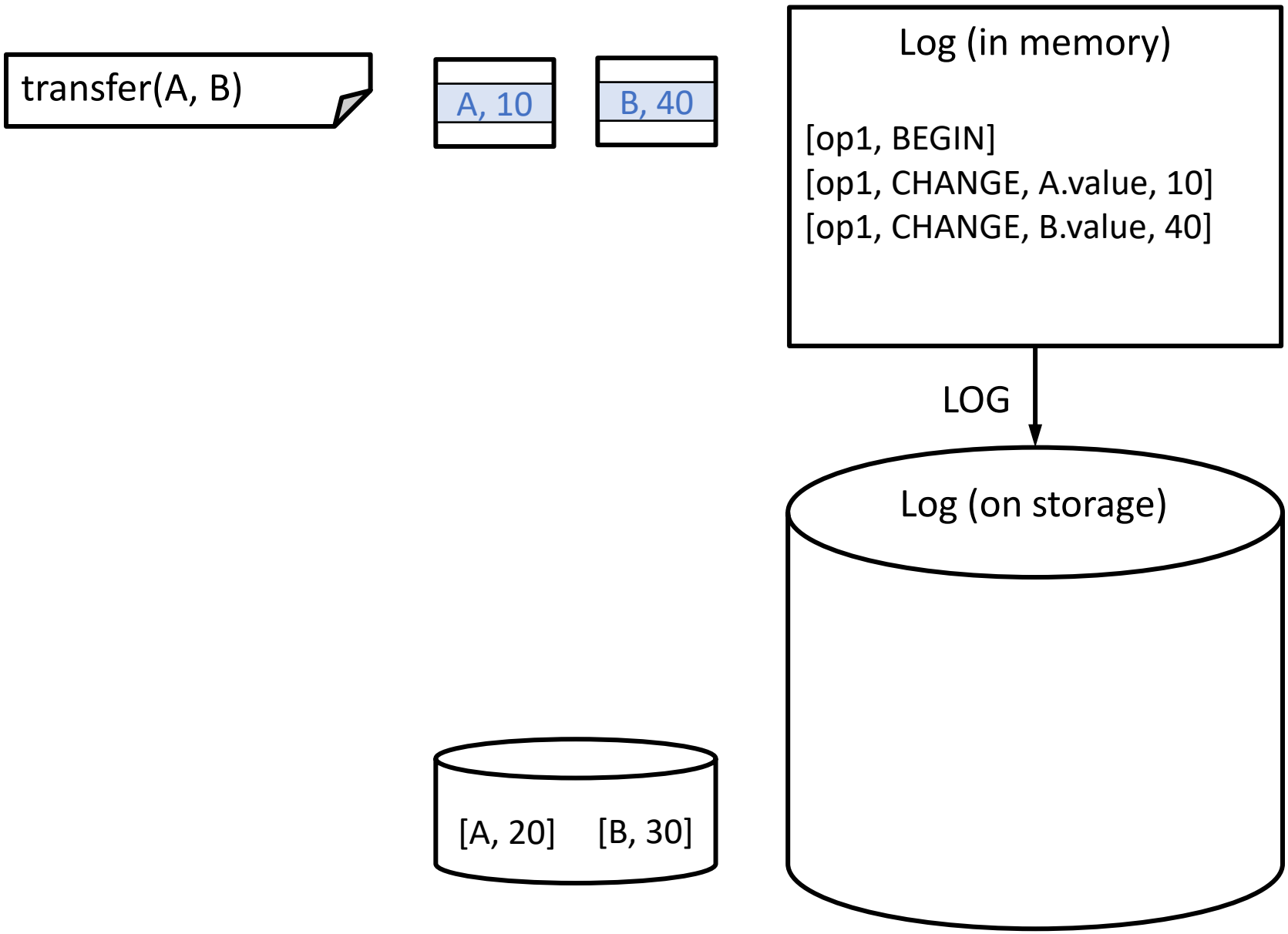


- Install API
  - INSTALL(item, value) // flush item's value
    - new_value during normal operation between COMMIT and END
    - new_value during (post-commit) crash recovery

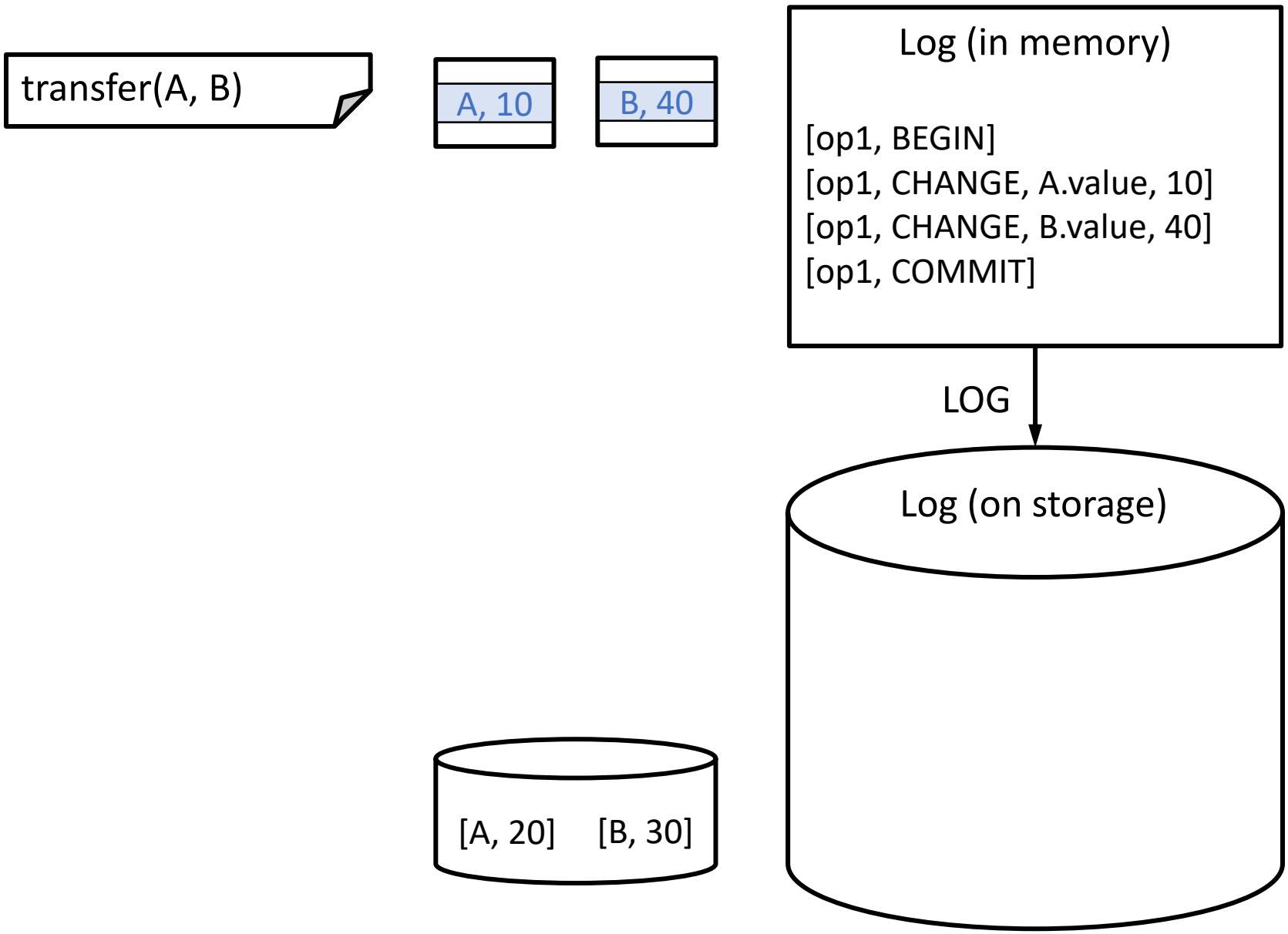
Redo logging example



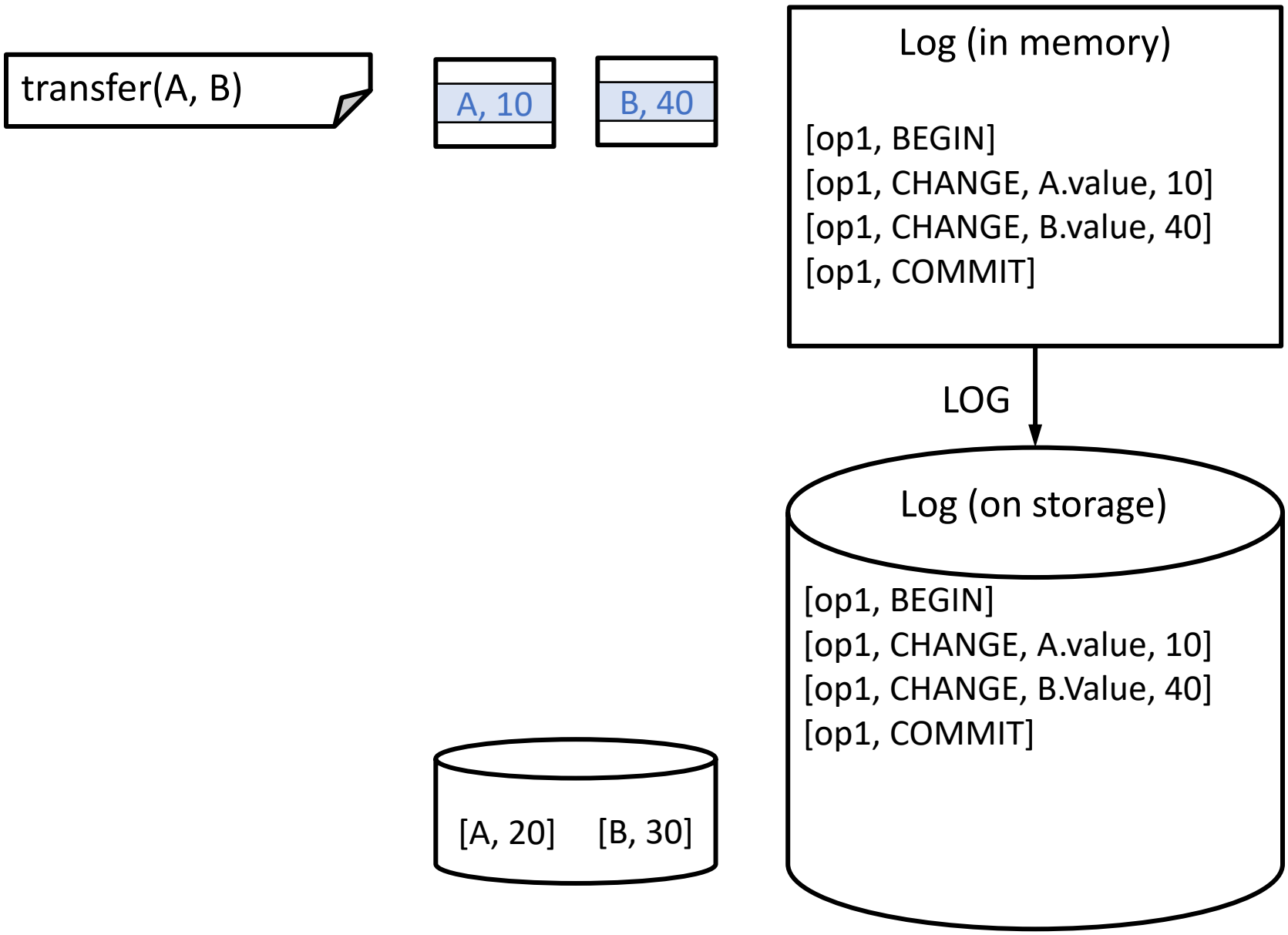
Redo logging example



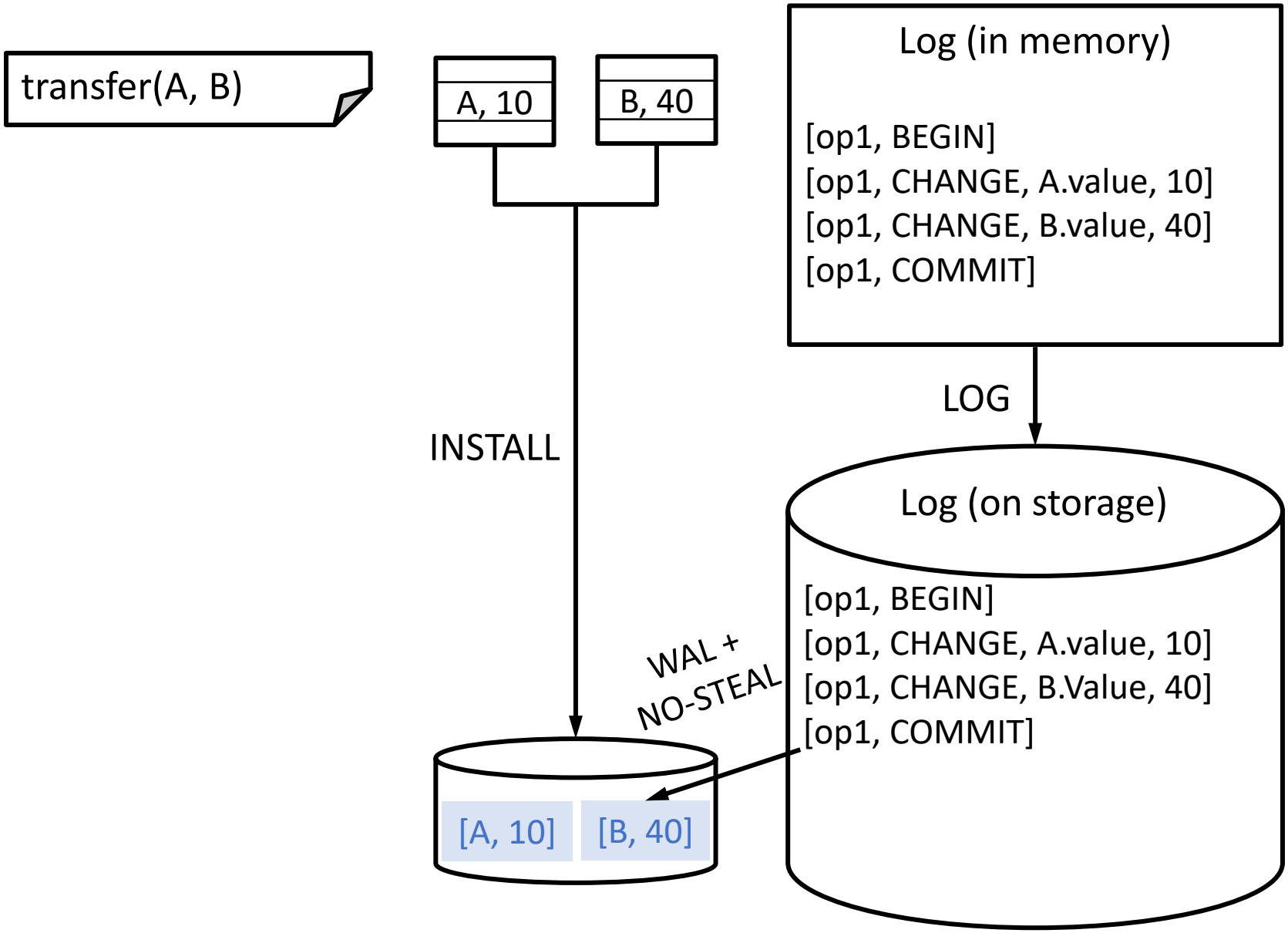
Redo logging example



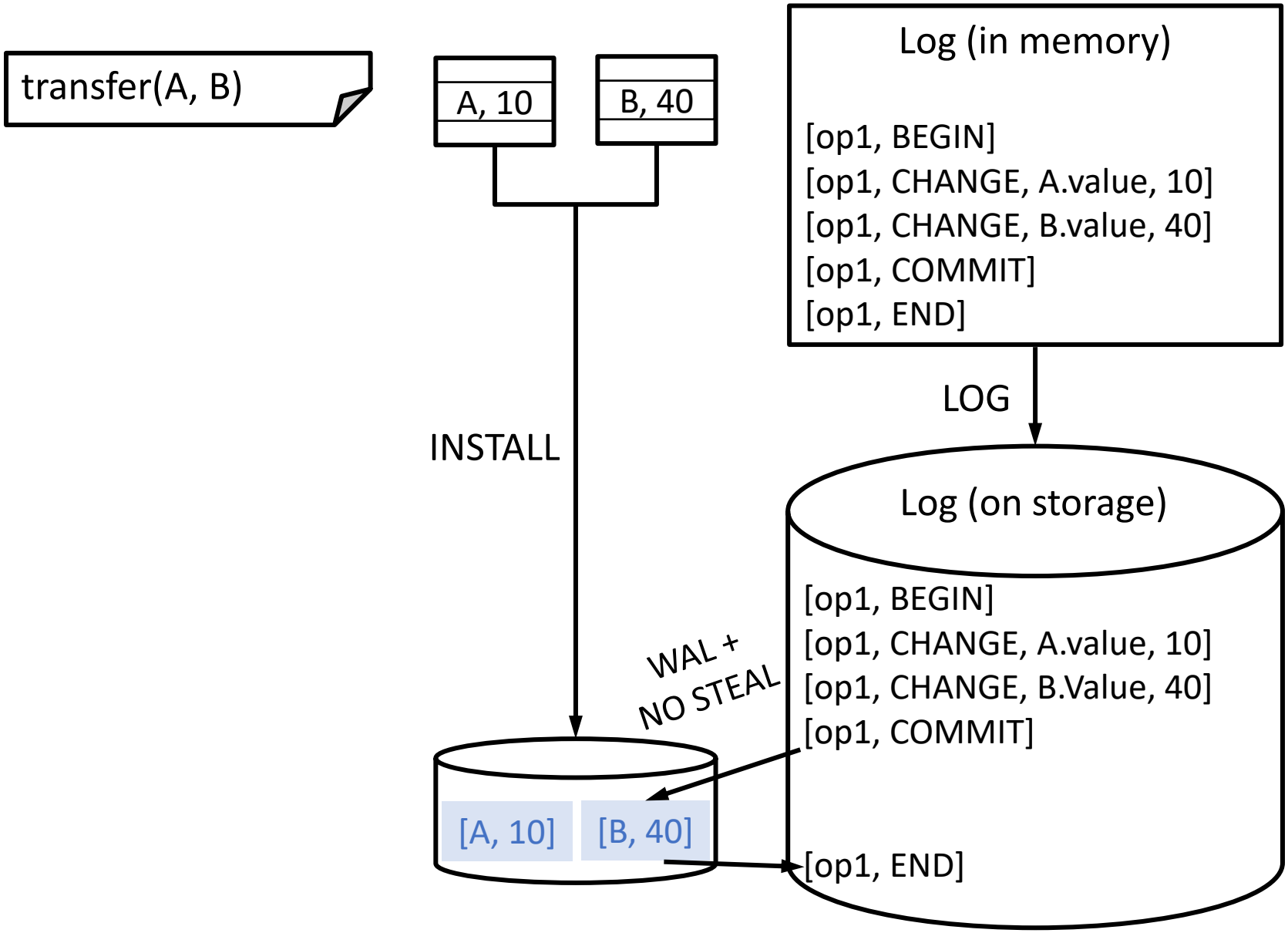
Redo logging example



Redo logging example



Redo logging example



Crash recovery with redo logging

```
Recovery() { // requires one backward + one forward pass over log
  committed = NULL;
  // collect all ops with COMMIT records that don't have END record
  foreach record starting from end of log to beginning {
    if ((record.type == COMMIT) and
        (END record for record.id was not found previously)) {
      committed = committed + record.id;
    }
  }
  // perform INSTALL for committed operations
  foreach record starting from beginning of log to end {
    if ((record.id in committed) and (record.type == CHANGE)) {
      // this change must be idempotent
      INSTALL(record.item, record.new_value);
    }
  }
  // mark committed operations as ended
  foreach id in committed {
    LOG(id, END);
  }
}
```

Undo versus Redo

- Undo
  - Force: requires blocks to be flushed before commit
    - Provides durability, but requires undo for atomicity
    - Delays commit, can cause high operation latency
- Redo
  - No-steal: requires blocks to be pinned in memory until commit
    - Provides atomicity, but requires redo for durability
    - Inefficient memory utilization, can cause low throughput
- In practice, most database systems (e.g., Oracle, MySQL innoDB storage engine) use undo-redo logging, which avoids force and allows steal (no-force, no-steal)
  - However, it requires logging both old and new values

Logging costs

- With logging, every update requires two writes
- However, logging costs are lower than expected because
  - Log writes are performed sequentially to the log
    - Sequential writes are must faster than random writes on hard drives
  - For redo logging, blocks are flushed asynchronously after commit
    - Only sequential log writes occur before commit

Improving logging performance

- Use battery-backed RAM for logging
  - Need to ensure that battery remains in good condition
- Use a separate hard drive or SSD for logging
  - Increases costs
- Buffer log records
  - Synchronously flushing each log record to storage is very expensive
  - We can buffer log records for an operation until commit
    - Need to still ensure WAL, force/no-steal requirements
  - When processing commit, if the last log block isn't full, in some cases, we can delay the flush until the block is full
    - Batches multiple commits (called group commit)
    - Increases operation latency but improves throughput

Checkpointing

Log size and recovery time

- Currently, write-ahead logging has two problems
  - Log grows over time
  - Crash recovery scans entire log, so recovery time grows over time
- How can we reduce log size and speed up crash recovery?
  - We can purge log records for operations that have completed

Checkpointing

- A checkpoint reduces log scan time during recovery by writing information about current system state to storage
  - Checkpoint information varies across systems, may involve
    - Writing a checkpoint record to the log
    - Flushing data blocks, e.g., creating a full snapshot of the in-memory state of a system atomically, to allow restoring system state after crash
- Checkpointing and logging are often used together
  - Periodically create a checkpoint
  - Use the checkpoint to prune log records that are no longer needed
    - E.g., if the checkpoint contains a full snapshot, then all log records before the checkpoint record can be pruned
  - Helps reduce log size and speed up recovery time

Checkpointing with redo logging

- Suppose the redo logging system maintains the current state of each operation: begin, commit, end
- Periodically, append a checkpoint record containing operations that have committed but have not yet ended
- These operations may require redo recovery

Redo recovery using checkpoints

- Say the log contains the following records:

  begin(2) … checkpoint(2, 3) … commit(4) … end(3) … commit(5) … end(5)

- Scan log backwards until checkpoint record
  - Collect set of committed ops that have not ended: [4]
  - Add ops from checkpoint that have not ended to set: [4, 2]
- Continue scanning backwards until the begin record of all operations in the set are found (generally takes short time)
  - All earlier records can be purged, why?
- Start forward pass
  - Only need to redo ops 2, 4 during recovery

Conclusions

- Atomicity and durability hide failures
  - Atomicity: an operation either executes completely, or not at all
  - Durability: an operation that completes is not lost
  - Help system provide same guarantees as fail-free operation, e.g., linearizability, serializability
- Shadow copy and write-ahead logging are two general techniques for ensuring these properties
  - Shadow copy uses a copy-on-write technique to atomically swap old and new data versions
  - Write-ahead logging logs a modified item before overwriting it, allowing partial modifications to be rolled back and completed modifications to be rolled forward
    - Checkpointing helps reduce log size and improve recovery time