Crash Recovery

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

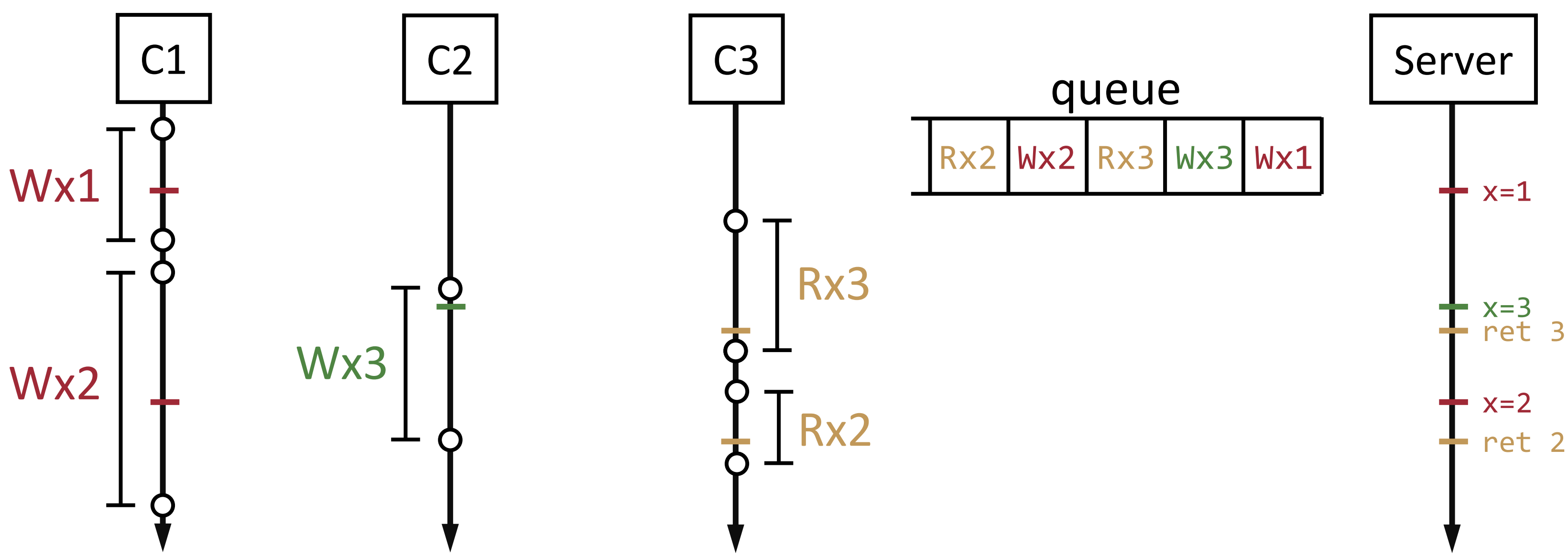
Distributed Systems
ECE419

Overview

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

Until now

- We needed exactly-once semantics for linearizability
- Server crashes complicate exactly-once implementation
- Now, we will look at how to handle server crashes



Storage API

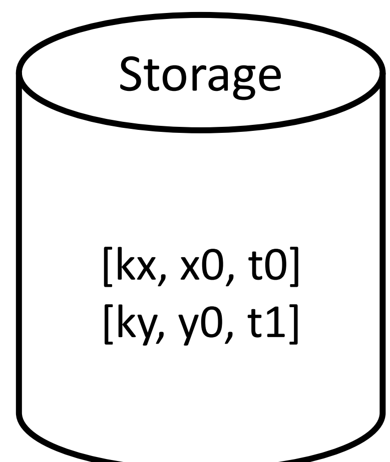
- Assume storage system provides following operations:
  - put(key, value, T) // create or update key with value
    - Client generates unique timestamp T for put() request
  - value = get(key) // return the value of key

Data storage

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

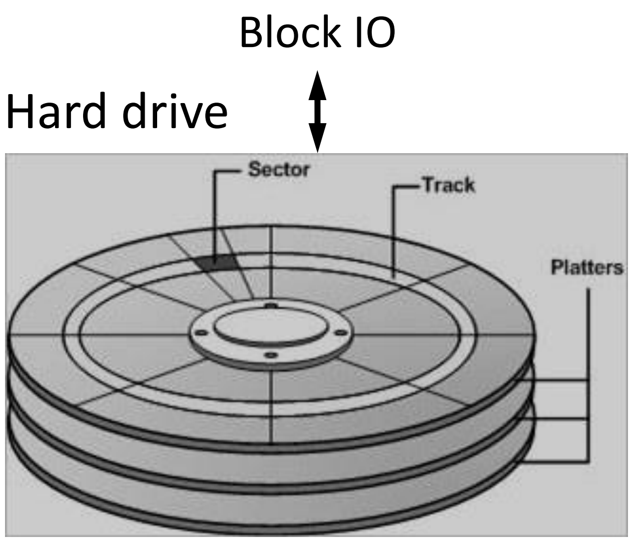
Memory

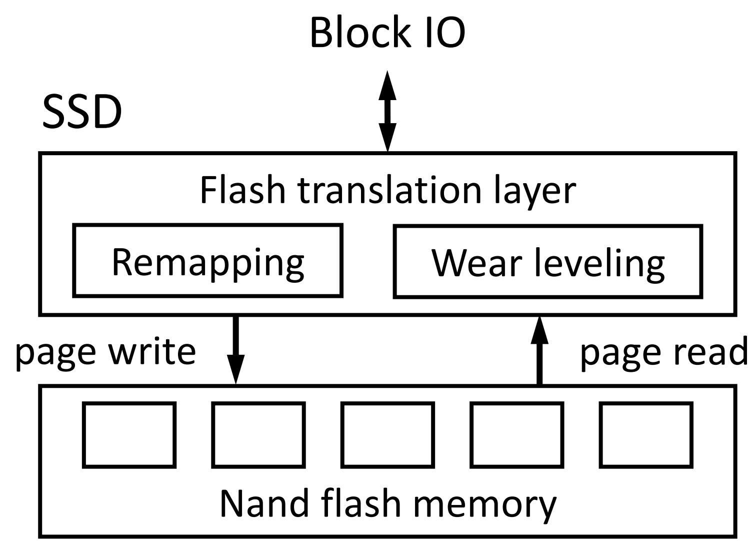
	key	value	time
x:	kx	x0	t0
y:	ky	y0	t1



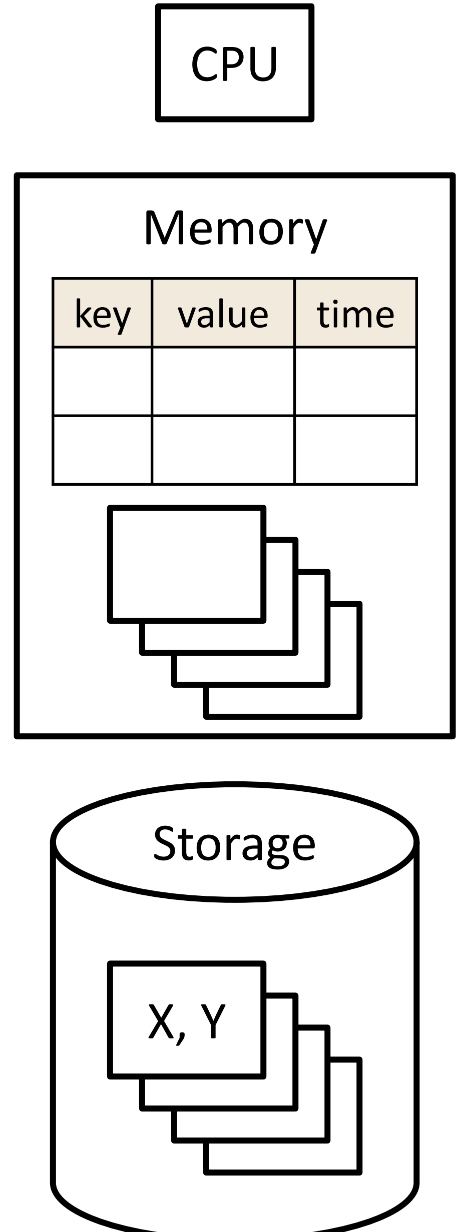
Storage access granularity

- 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



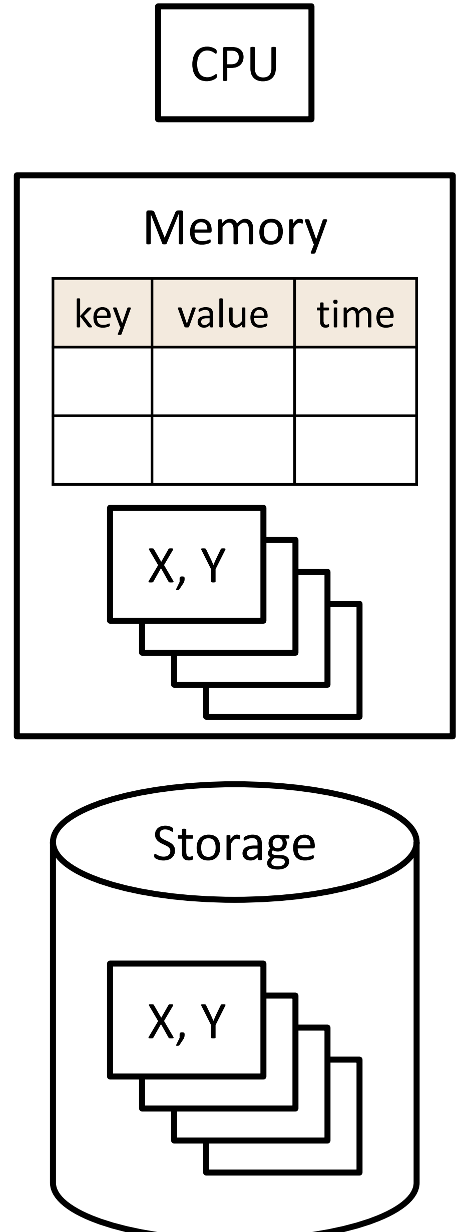


Data access model



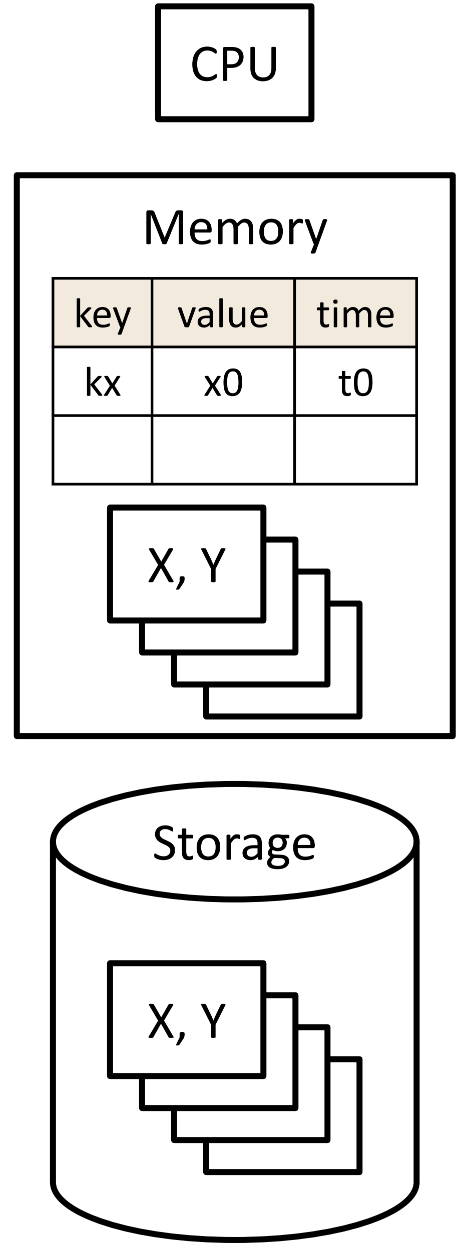
Data access model

- input(X)
  - Read the storage block containing record X from storage into memory (block is cached)



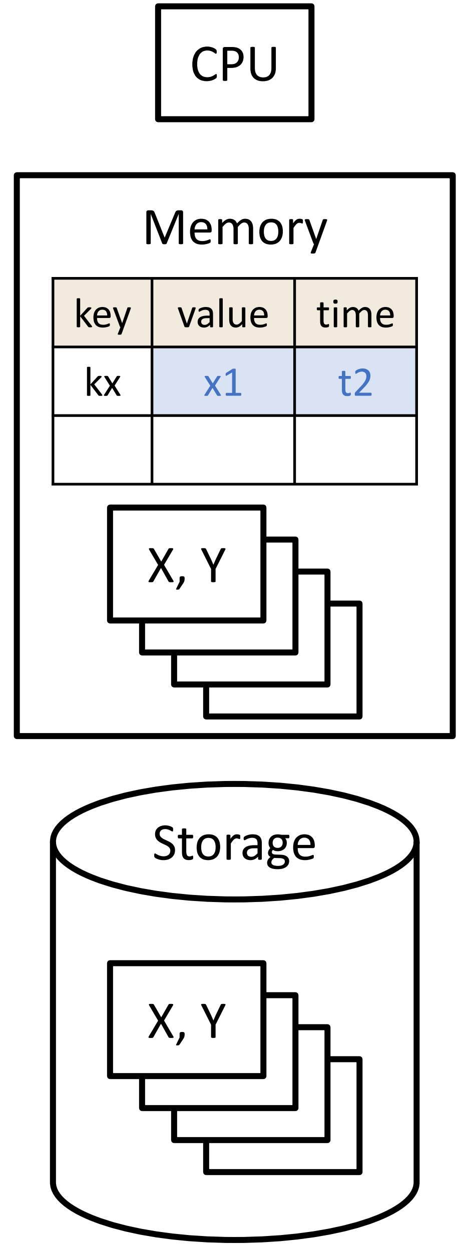
Data access model

- input(X)
  - Read the storage block containing record X from storage into memory (block is cached)
- x = read(X)
  - Read value of record X into a local variable x, execute input(X) first if necessary



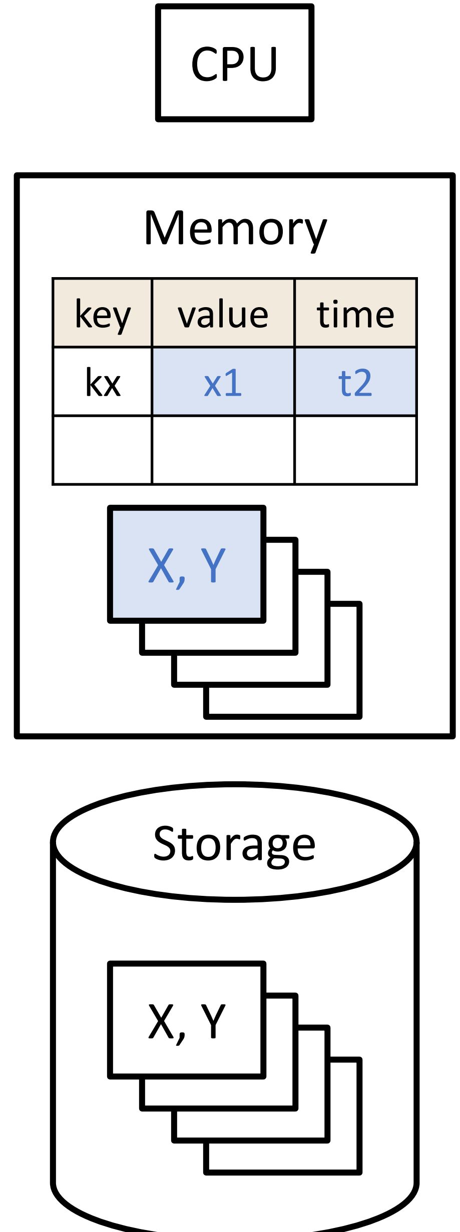
Data access model

- input(X)
  - Read the storage block containing record X from storage into memory (block is cached)
- x = read(X)
  - Read value of record X into a local variable x, execute input(X) first if necessary
- Say client issues put(kx, x1, t2)



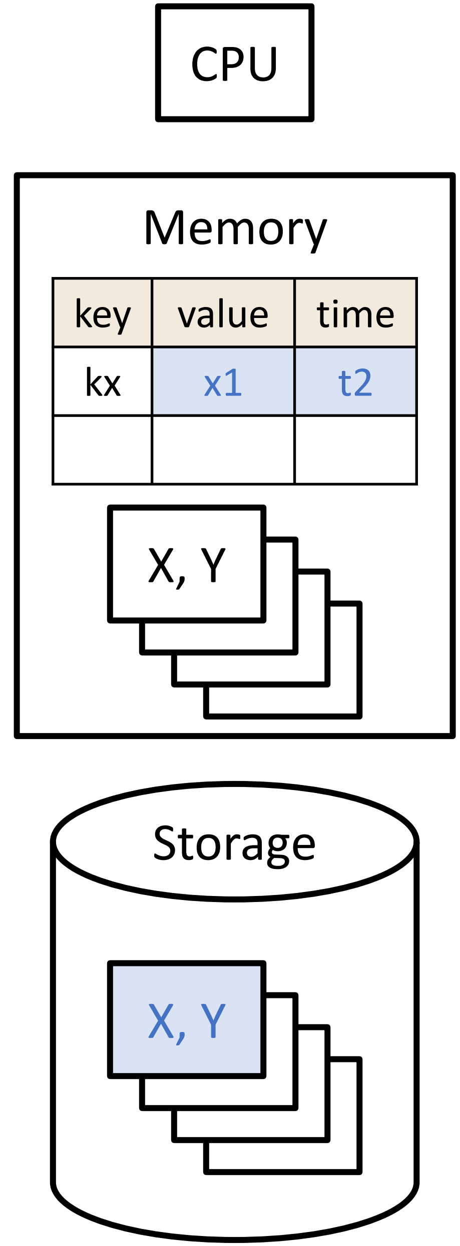
Data access model

- input(X)
  - Read the storage block containing record X from storage into memory (block is cached)
- x = read(X)
  - Read value of record X into a local variable x, execute input(X) first if necessary
- Say client issues put(kx, x1, t2)
- write(X, x)
  - Write x to record X in memory, execute input(X) if needed (block is modified)



Data access model

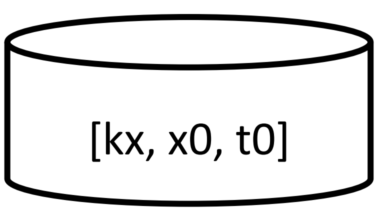
- input(X)
  - Read the storage block containing record X from storage into memory (block is cached)
- x = read(X)
  - Read value of record X into a local variable x, execute input(X) first if necessary
- Say client issues put(kx, x1, t2)
- write(X, x)
  - Write x to record X in memory, execute input(X) if needed (block is modified)
- output(X)
  - Write memory block containing record X to storage durably (modified block is flushed)



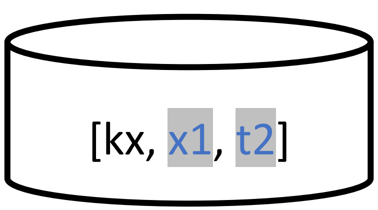
Server operation

- Say client issues put(kx, x1, t2)
- Storage system
  - Performs duplicate detection using Timestamps t0 and t2
  - Updates value and timestamp of Key kx in memory and storage
  - Responds to client

key	value	time
kx	x0	t0
ky	y0	t1



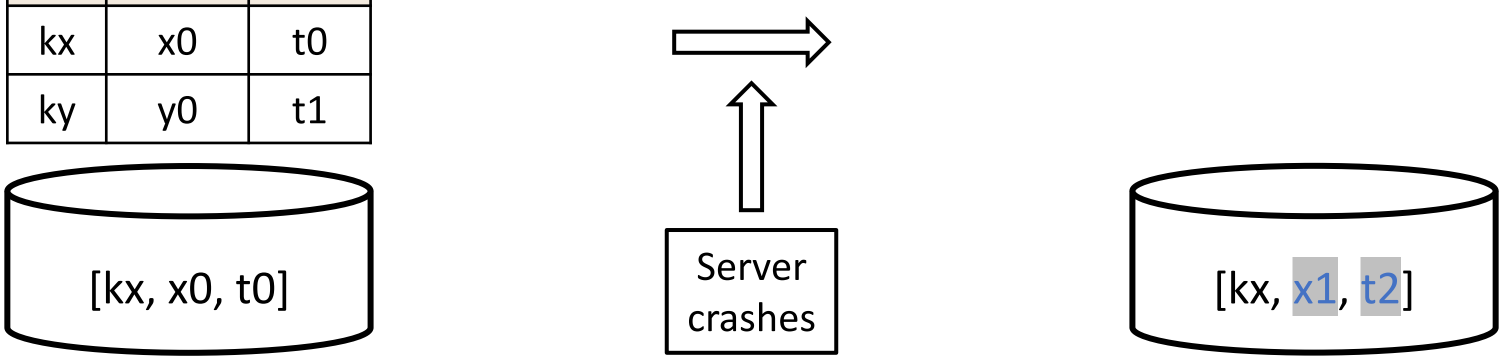
key	value	time
kx	x1	t2
ky	y0	t1



Server crashes

- Suppose server crashes sometime in between
  - Memory contents are lost on reboot
- What could go wrong?
  1. Both x1 and t2 not on storage, client receives response
  2. Both x1 and t2 on storage, client doesn't receive response
  3. One of x1 or t2 on storage

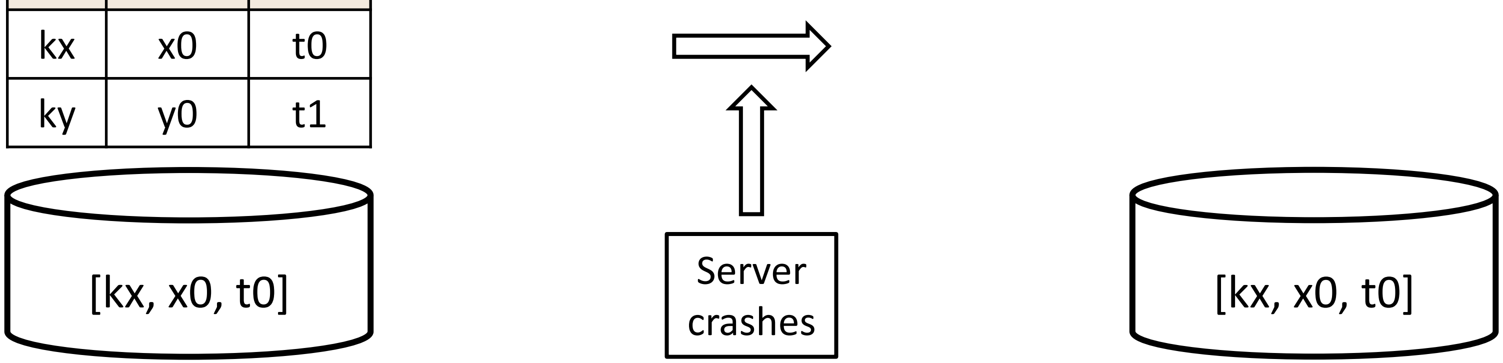
key	value	time
kx	x0	t0
ky	y0	t1



Handling server crashes

1. Both x1 and t2 not on storage, client receives response
   - We need to write x1 and t2 to storage before sending response, then a completed operation is not lost on failure
   - This property is called durability

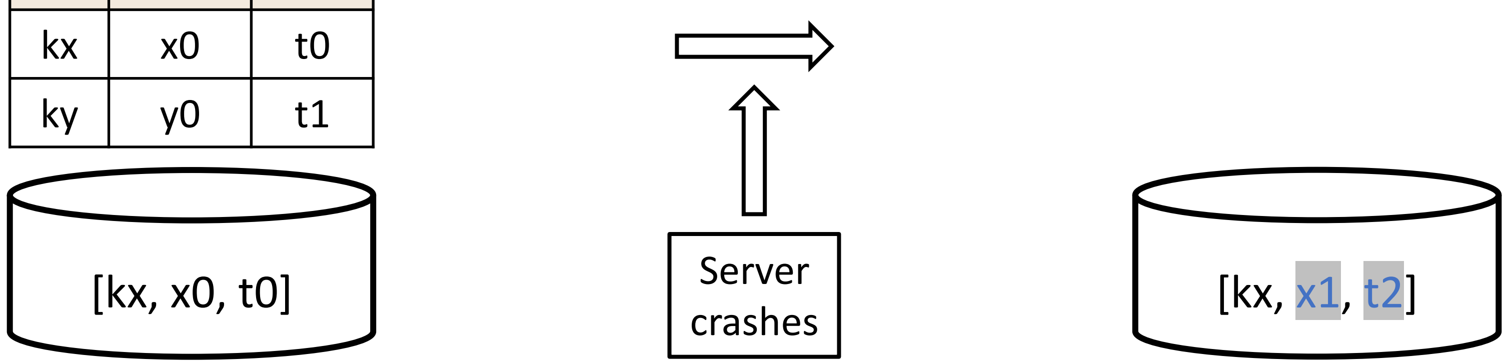
key	value	time
kx	x0	t0
ky	y0	t1



Handling server crashes

2. Both x1 and t2 on storage, client doesn't receive response
    - Client retries with timestamp t2
    - Can detect duplicate request, ignore executing it, return previous saved result

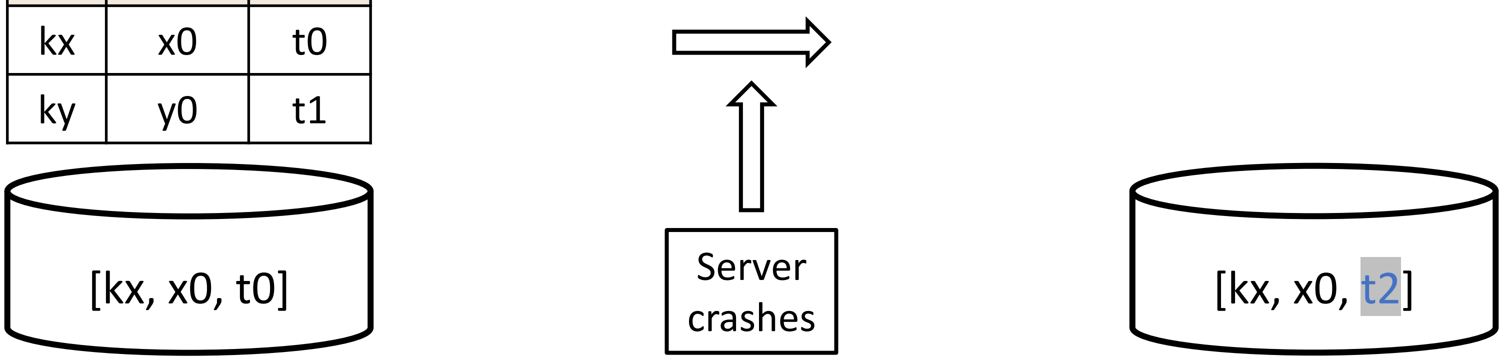
key	value	time
kx	x0	t0
ky	y0	t1



Handling server crashes

3. One of x1 or t2 on storage
   - Only t2 on storage: client's retry will be ignored, x1 will be lost

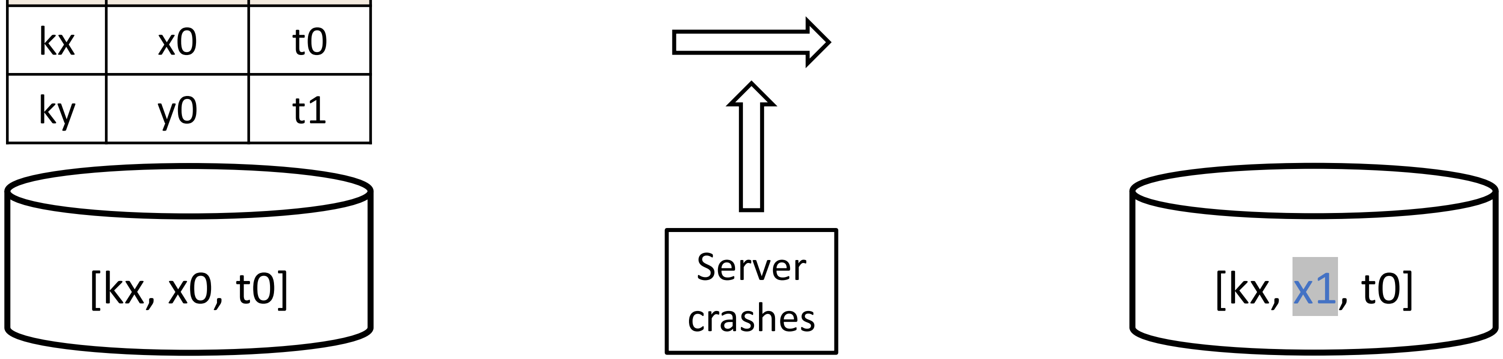
key	value	time
kx	x0	t0
ky	y0	t1



Handling server crashes

3. One of x1 or t2 on storage
   - Only t2 on storage: client's retry will be ignored, x1 will be lost
   - Only x1 on storage: duplicate request will be accepted, e.g., what if update operation was incrementing x0

key	value	time
kx	x0	t0
ky	y0	t1

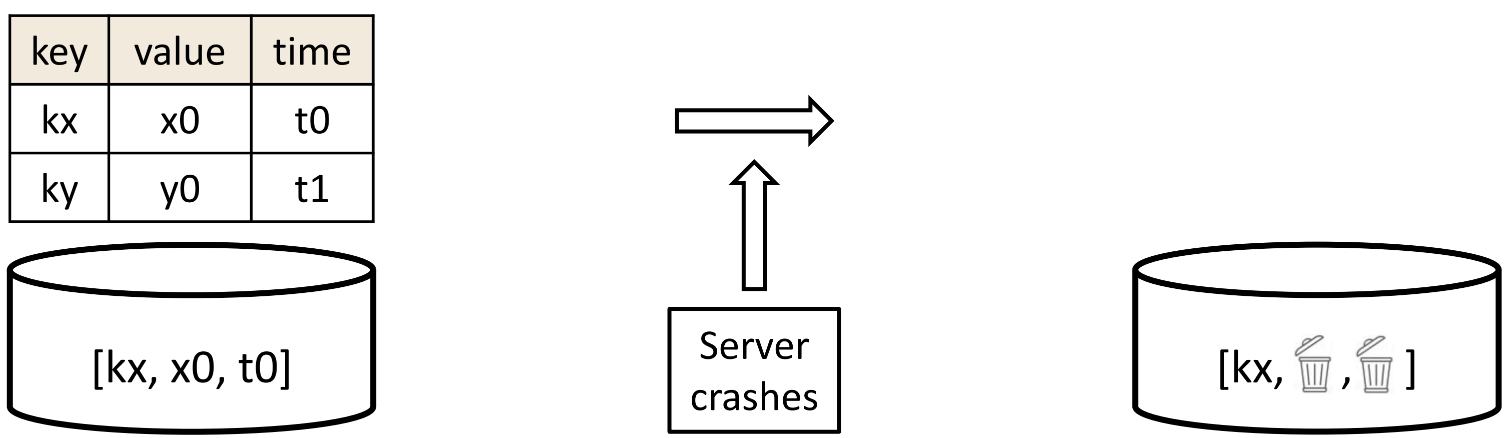


Handling server crashes

3. One of x1 or t2 on storage

- Only t2 on storage: client's retry will be ignored, x1 will be lost
- Only x1 on storage: duplicate request will be accepted,
  e.g., what if update operation was incrementing x0
- Worse, if x1 or t2 are written partially,
  storage is inconsistent after reboot

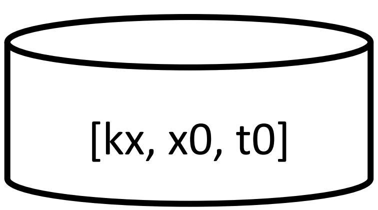
key	value	time
kx	x0	t0
ky	y0	t1



Failure atomicity

- Say client issues put(kx, x1, t2)
  - For modified x1 and t2, 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 reboot
- Ensuring failure atomicity is key challenge with crashes

key	value	time
kx	x0	t0
ky	y0	t1

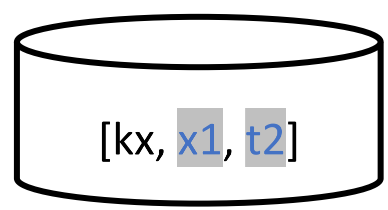


Nothing

⟹

key	value	time
kx	x1	t2
ky	y0	t1

All

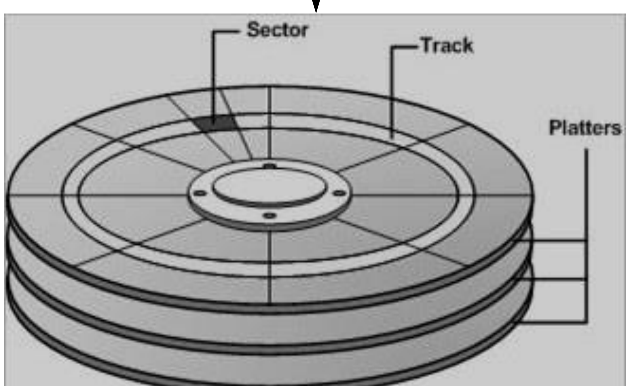


Storage failure atomicity

- 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

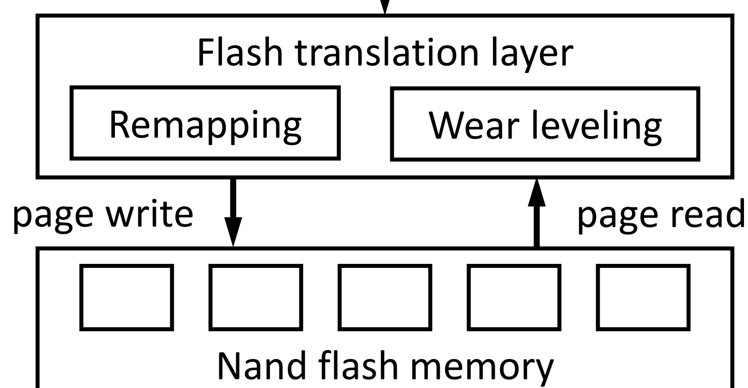
Block IO

Hard drive



Block IO

SSD



Why is failure atomicity hard?

- Say client issues put(kx, x1, t2)
- Problem occurs when x1 and t2 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 values are variable sized and stored in a separate heap area
  - If values are larger than block size
  - If we use a separate hash table index to look up keys,
    and key-value pairs are reallocated when resizing
  - If we add checksums 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

OP Begin
write1
write2
write3
Commit point

OP End

A. Pre-commit discipline:
operation aborts, roll back all changes

B. Post-commit discipline:
operation completes, roll forward all changes

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 item is flushed
  - Then copy is written to storage before original is overwritten
  - This ordering ensures roll back or 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

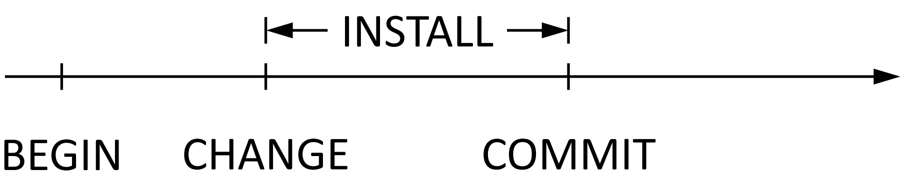
  - Id: operation id
  - Type: BEGIN, CHANGE, COMMIT, END
  - Item: physical location of item on storage (block id, offset)
  - Value: physical value of item (physical logging)

Undo logging

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

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 during pre-commit crash recovery

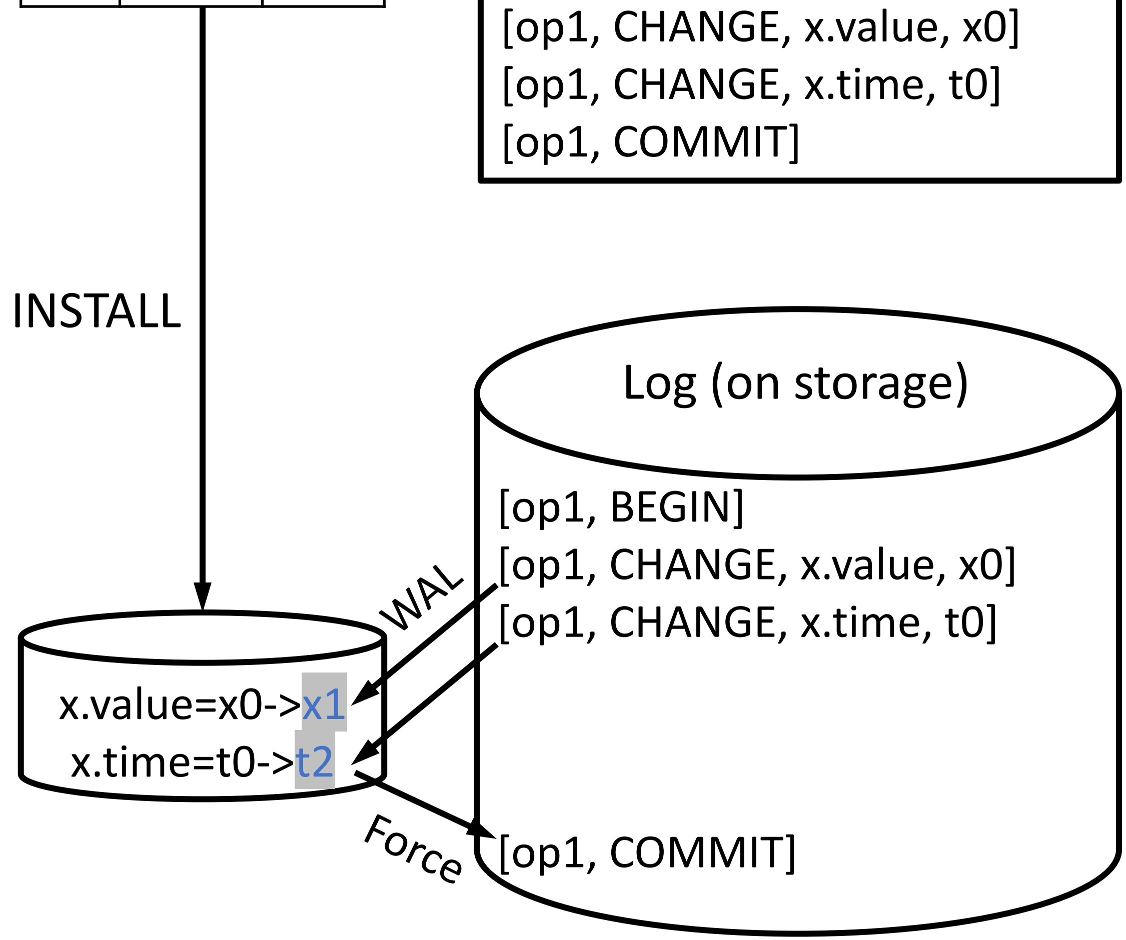
Undo logging example

put(kx, x1, t2)

key	value	time
kx	x1	t2
ky	y0	t1

Log (in memory)

[op1, BEGIN]
[op1, CHANGE, x.value, x0]
[op1, CHANGE, x.time, t0]
[op1, COMMIT]

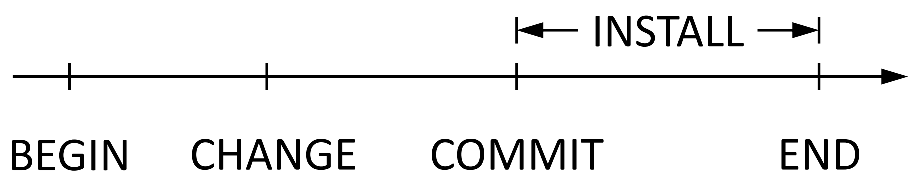


Redo logging

- Log new value of modified item in the log record
- Redo logging discipline
  - WAL: flush log record before modified item
  - No steal: flush all log records before any modified items are flushed
- Recovery
  - Pre-commit crash: no updates of the operation have been applied
  - Post-commit crash: roll forward updates using new values from log

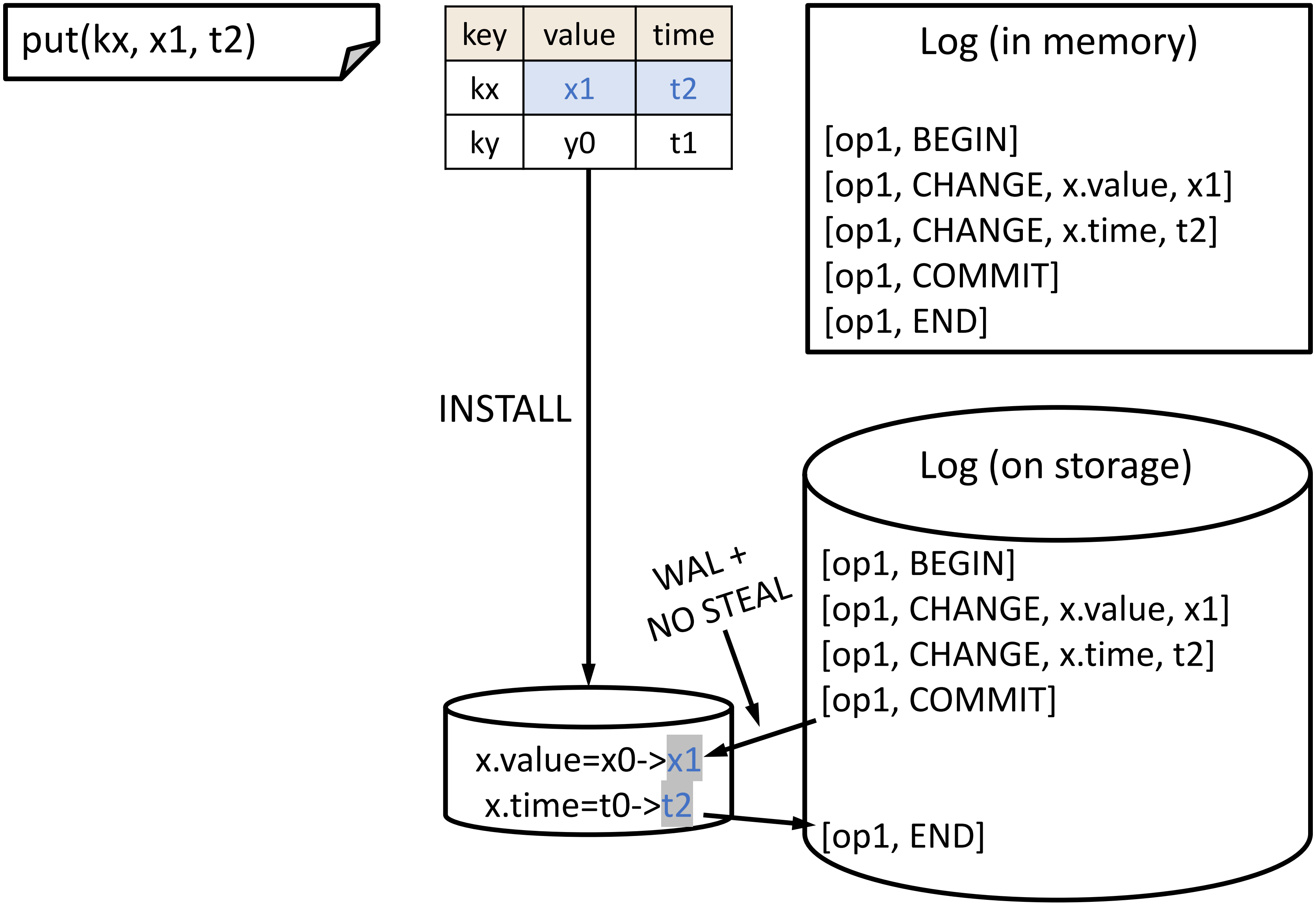
Redo logging operation

- 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



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, leads to 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, leads to low throughput
- In practice, systems use undo-redo logging, which avoids force and allows 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
  - When should an in-memory log record by flushed to storage?
  - Synchronously flushing each log record to storage is very expensive
  - We can buffer log records for an operation until commit
  - 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 complete snapshot of the in-memory state of a system, 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 complete 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

- For linearizability, a storage system must provide atomicity and durability under crash failures
  - Atomicity: an operation either executes completely, or not at all
  - Durability: an operation that completes is not lost
- Shadow copy and write-ahead logging are two general techniques for ensuring these properties
  - Shadow copy uses a copy-on-write technique to atomically switch between the old and the 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