Atomicity vs. durability

- How to ensure both upon failure?
  - A single transaction might update multiple disk sectors
  - Upon reboot, any subset of pending sectors might have been updated

Atomicity vs. durability

- Durability without atomicity is easy:
  - Update database files before acknowledging commit
- Atomicity without durability is also easy:
  - Upon reboot, erase the database!

Goal

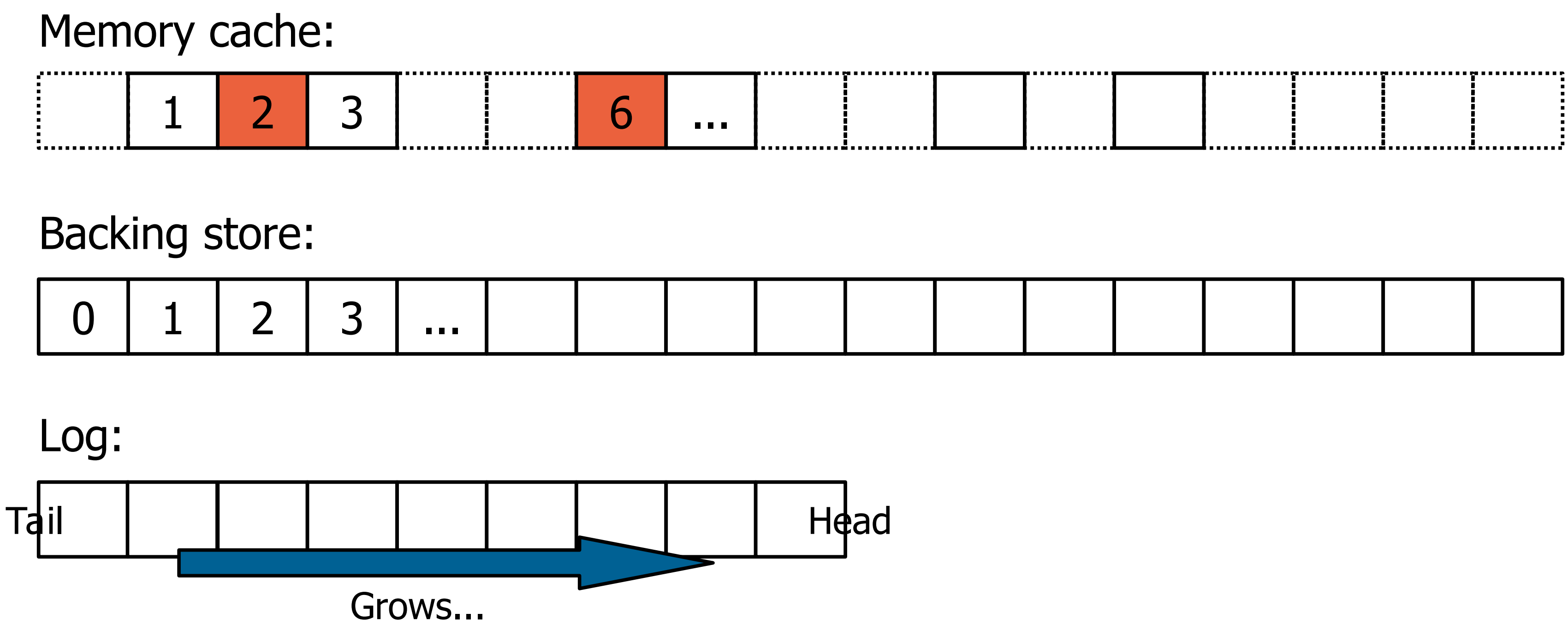
- A sequence of atomic operations is not an atomic operation
- There is a general purpose technique for obtaining an atomic sequence of operations
- Requires that:
  - Individual operations are atomic
  - It is known when individual operations are finished

Redo log

- First method: Re-Do log
- Based on “re-doing missing writes”

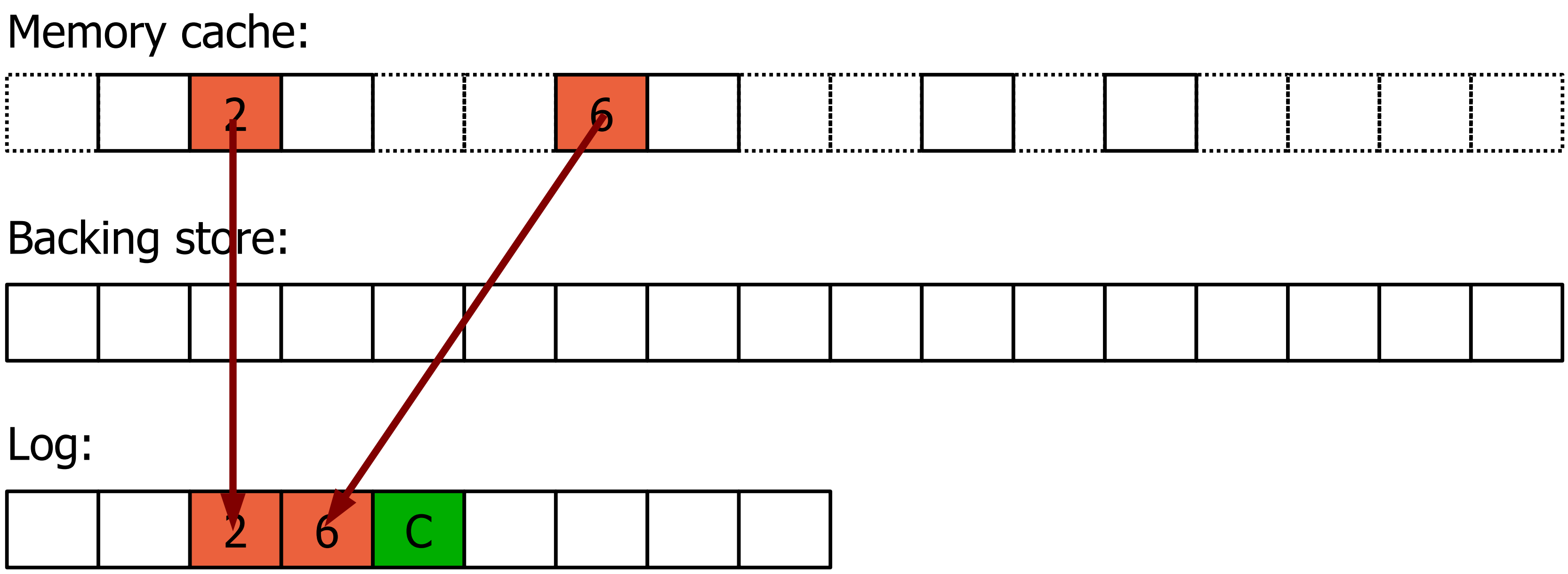
Redo log

- The application changes the memory cache:



Redo log

- Dirty blocks are copied to the log followed by a commit marker:

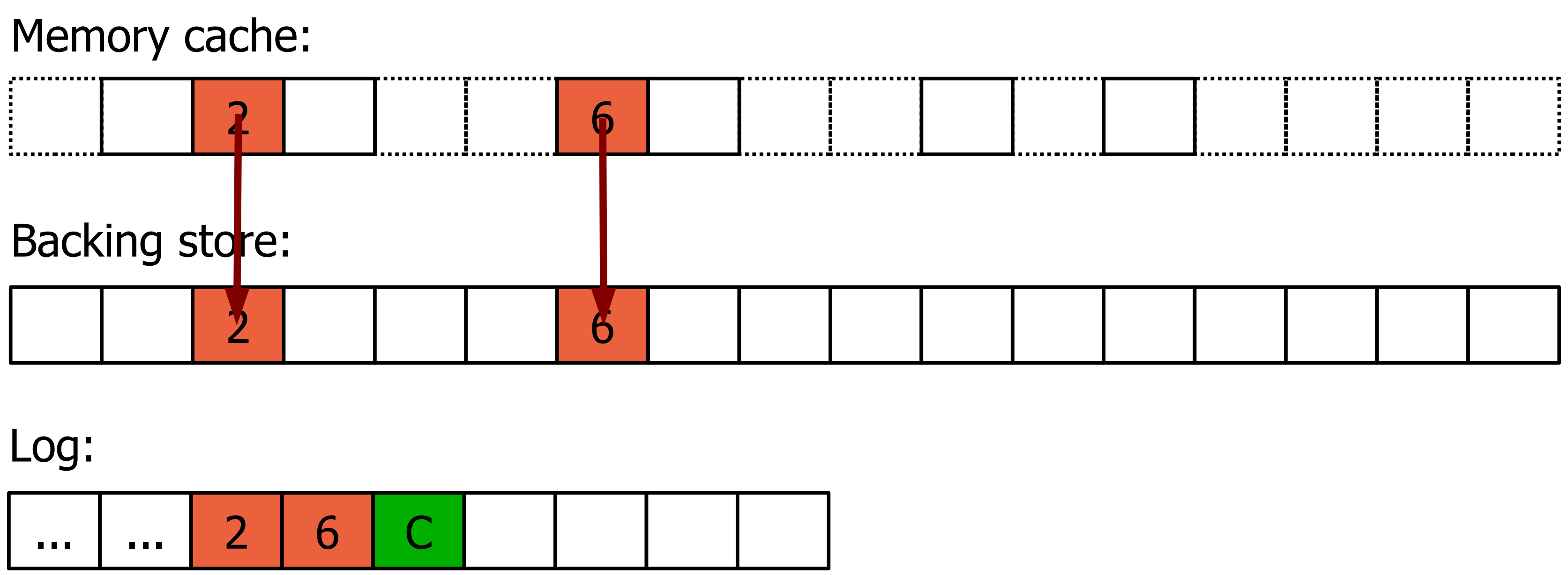


Log format

- Physical: The block itself
  - Clearly idempotent
  - Inefficient
- Logical: The command that changes the block
  - Harder to make it idempotent (INSERT?)
  - Space efficient
- “Physiological”:
  - Either one of the above, depending on the operation, the context, ...

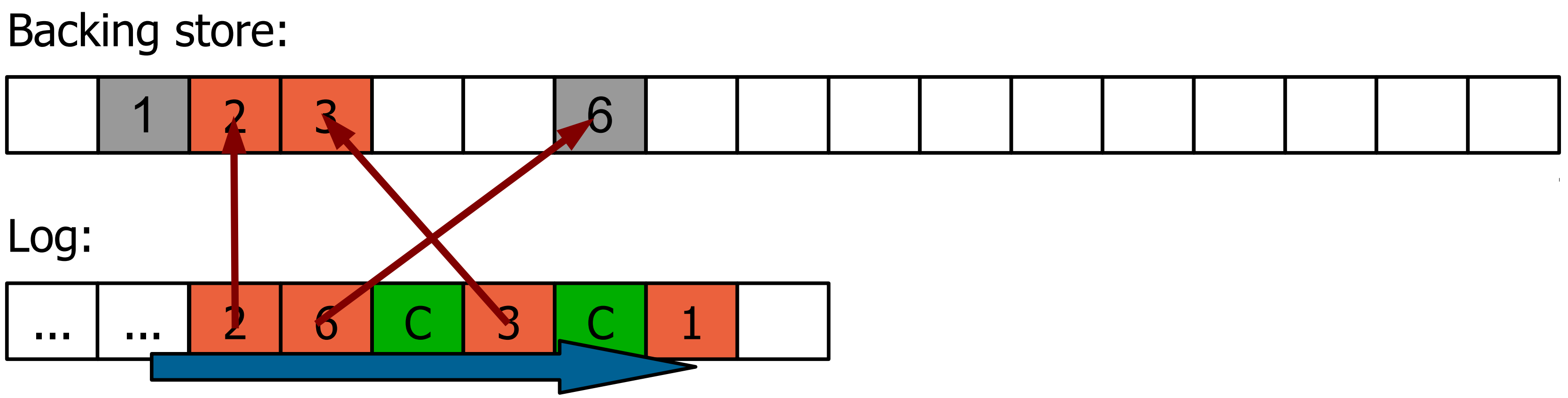
Redo log

- Eventually, dirty blocks are copied to the backing store:



Redo log

- Recovery after restart:
  - Copy blocks of transactions with commit markers from the log to storage
  - Forward (tail to head)



Redo log truncation

- When can a log prefix be removed?
- Naive approach: Write all changes of committed transactions to disk
- How to keep track of which transactions modified each block?
- Cannot write a memory block to disk if it has been modified by a running transaction:
  - No-steal
- What if a block has been modified by two transactions: one finished, one running?

Redo log summary

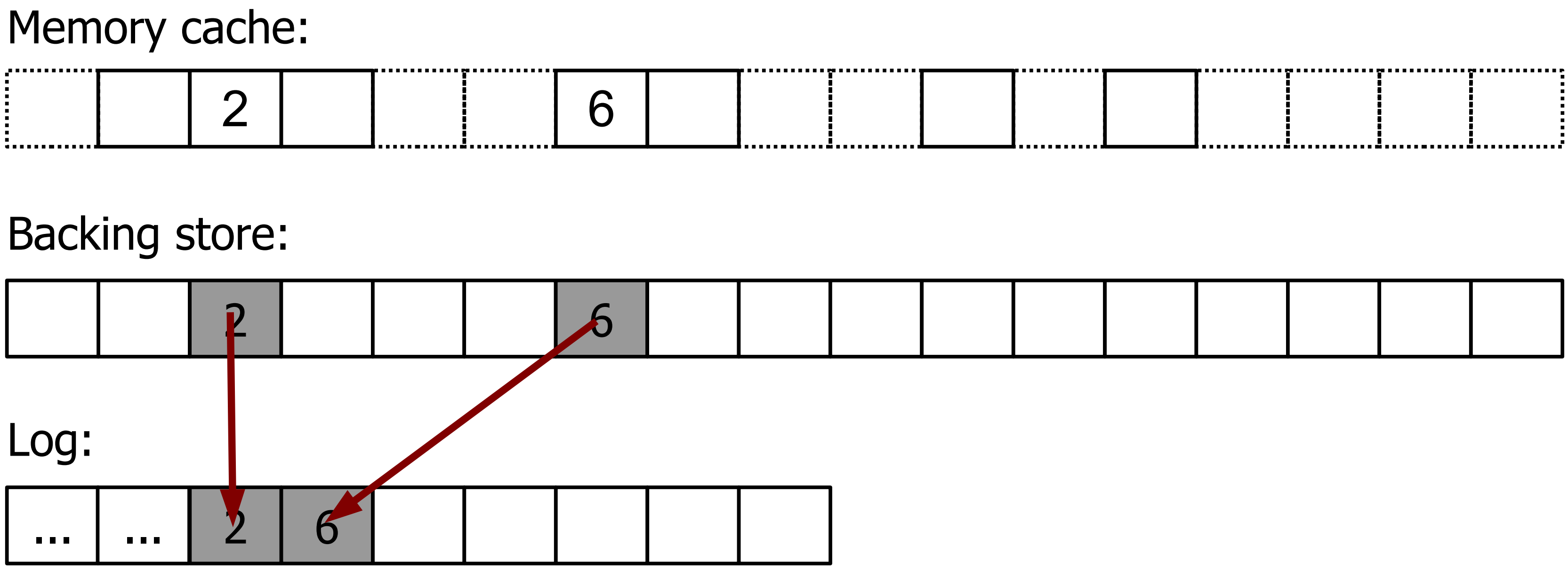
- The good:
  - The transaction can be committed without modifying the backing store (<u>no force</u>)
- The bad:
  - Modifications can only be written to backing store after the transaction begins committing (<u>no steal</u>)
    - Checkpoints are hard
    - Assumes that memory is large enough to hold all modifications

Undo log

- Second method: Un-Do log
- Based on “un-doing unwanted writes”

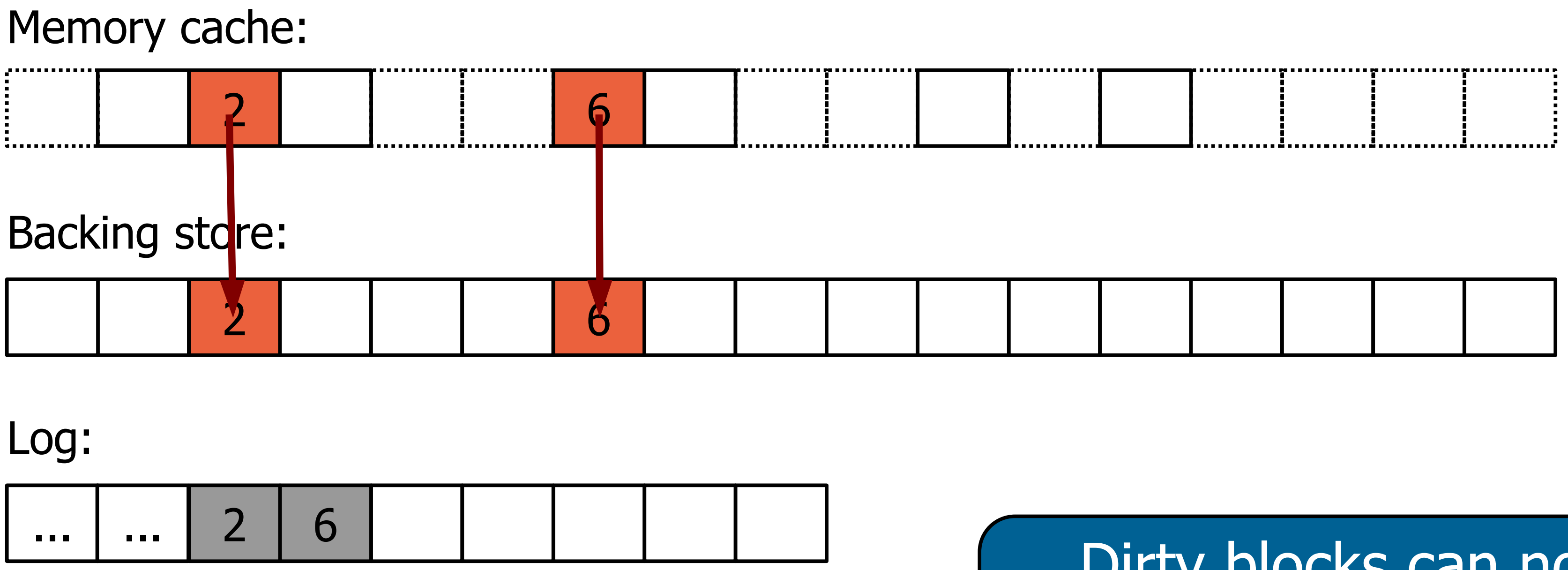
Undo log

- Original values of dirty blocks are copied to the log:



Undo log

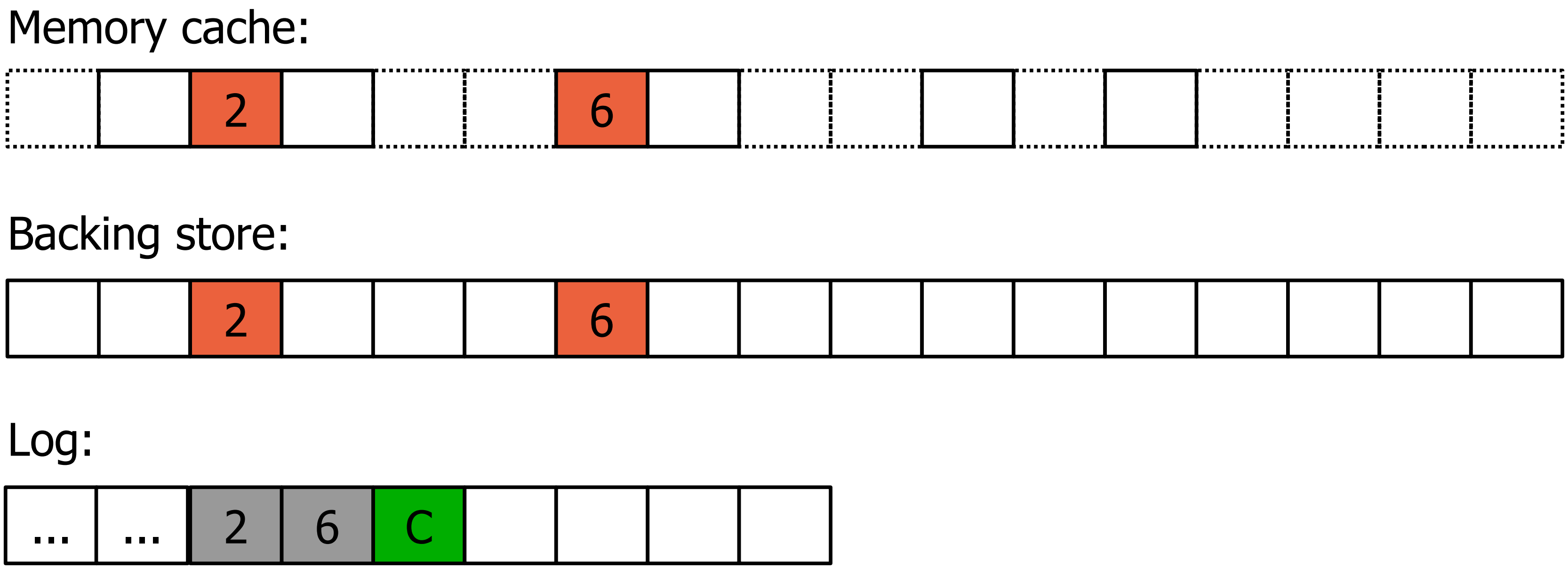
- Modified blocks are copied to the backing store:



Dirty blocks can now be evicted from cache!

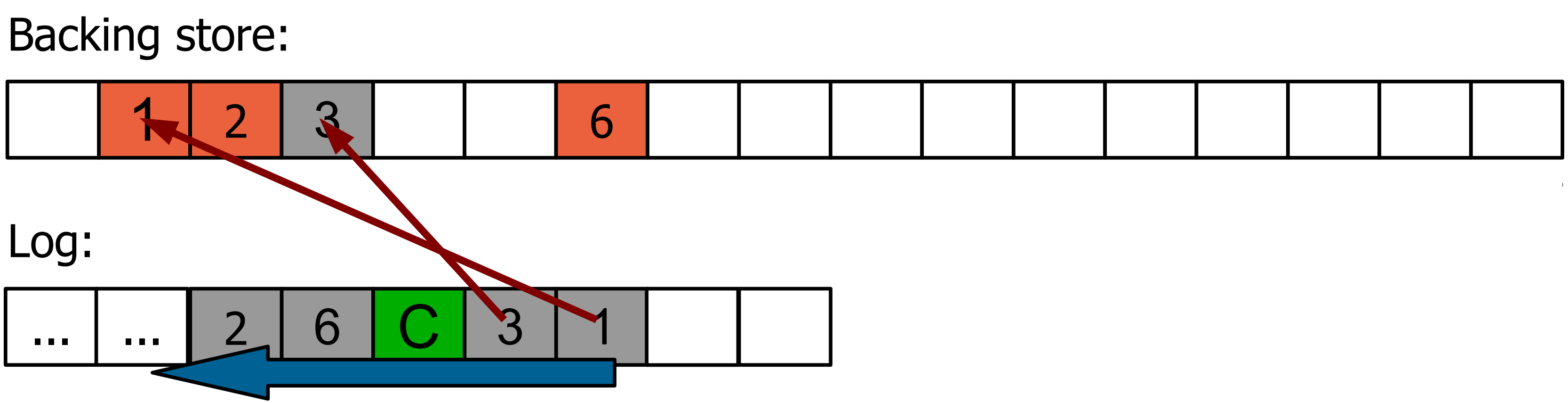
Undo log

- A commit marker is inserted in the log:



Undo log

- Recovery after restart:
  - Copy back entries of transactions without commit markers
  - Backward (head to tail)

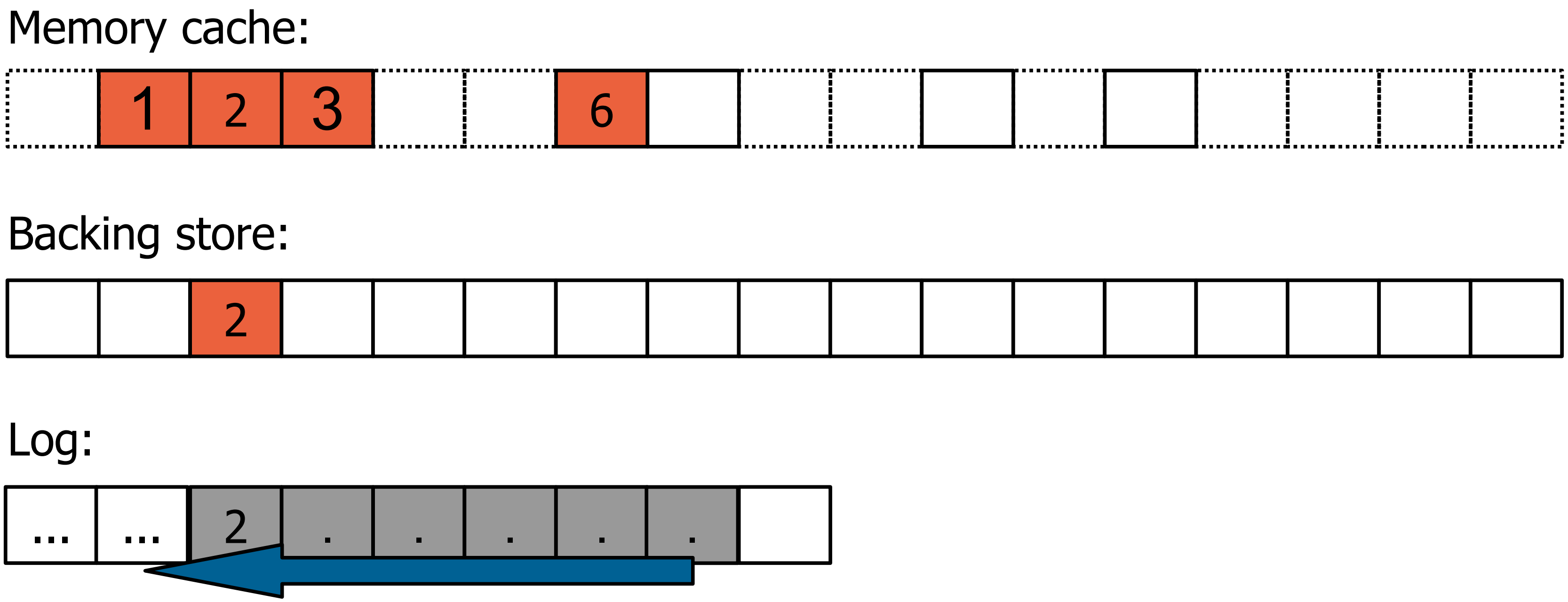


Undo log truncation

- When can a log prefix be removed?
- Can always write all memory to disk, even while there are running transactions
  - Steal

Undo log truncation

- The log must still be kept due to a possible very old running transaction



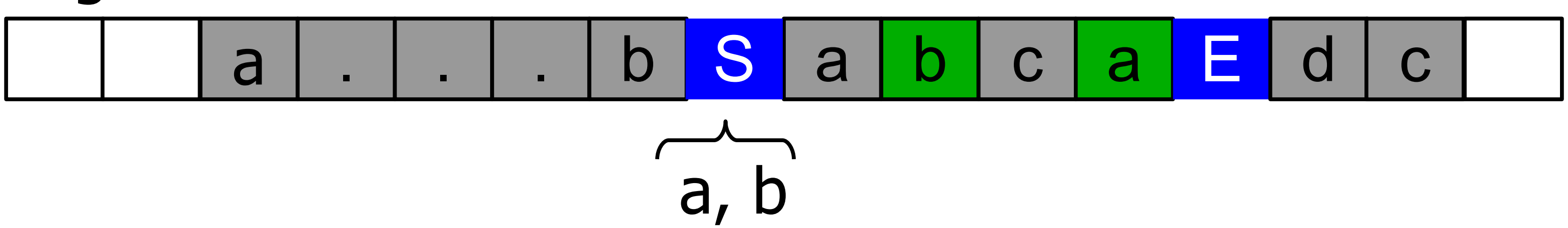
Undo log truncation

- Waiting for all transactions to finish means:
  - Disk is now consistent
  - Log is no longer required
- Can we account for running transactions and avoid stopping the system?

Undo log truncation

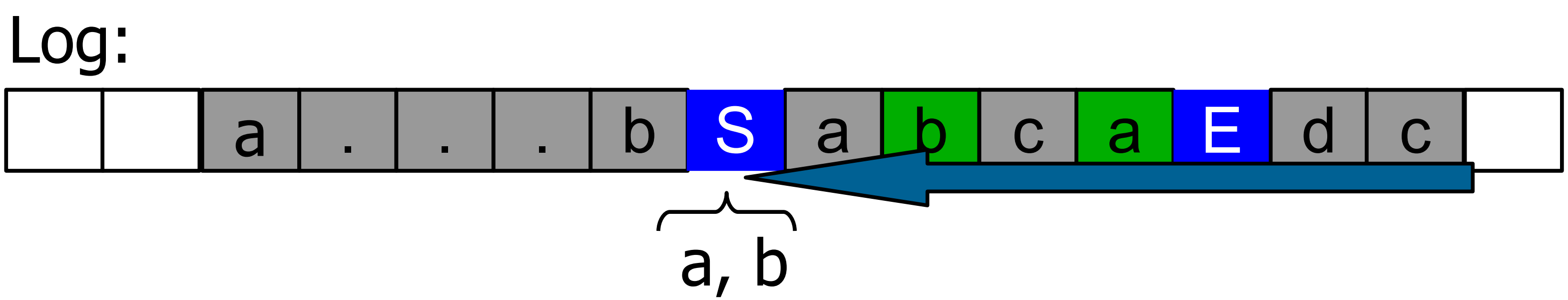
- Non-quiescent (a.k.a. fuzzy) checkpoint:
  - Write to log “start checkpoint” marker with a list of running transactions <T1, ..., Tn>
  - Wait until <T1, ..., Tn> have finished
    - Implicitly writes all modified memory blocks
    - Other transactions may start in between
  - Write “end checkpoint” marker

Log:



Undo log truncation

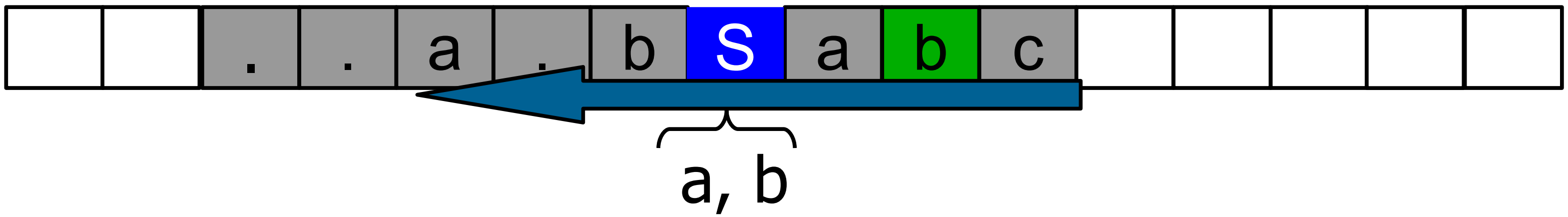
- Recovery to a complete checkpoint:
  - Search log until "start checkpoint"
  - No unfinished transactions exist before that



Undo log truncation

- Recovery to an incomplete checkpoint:
  - Search log until “start checkpoint”
  - Get list of possibly unfinished transactions
  - Continue searching the log until all found (likely to be close…)

Log:



Undo log

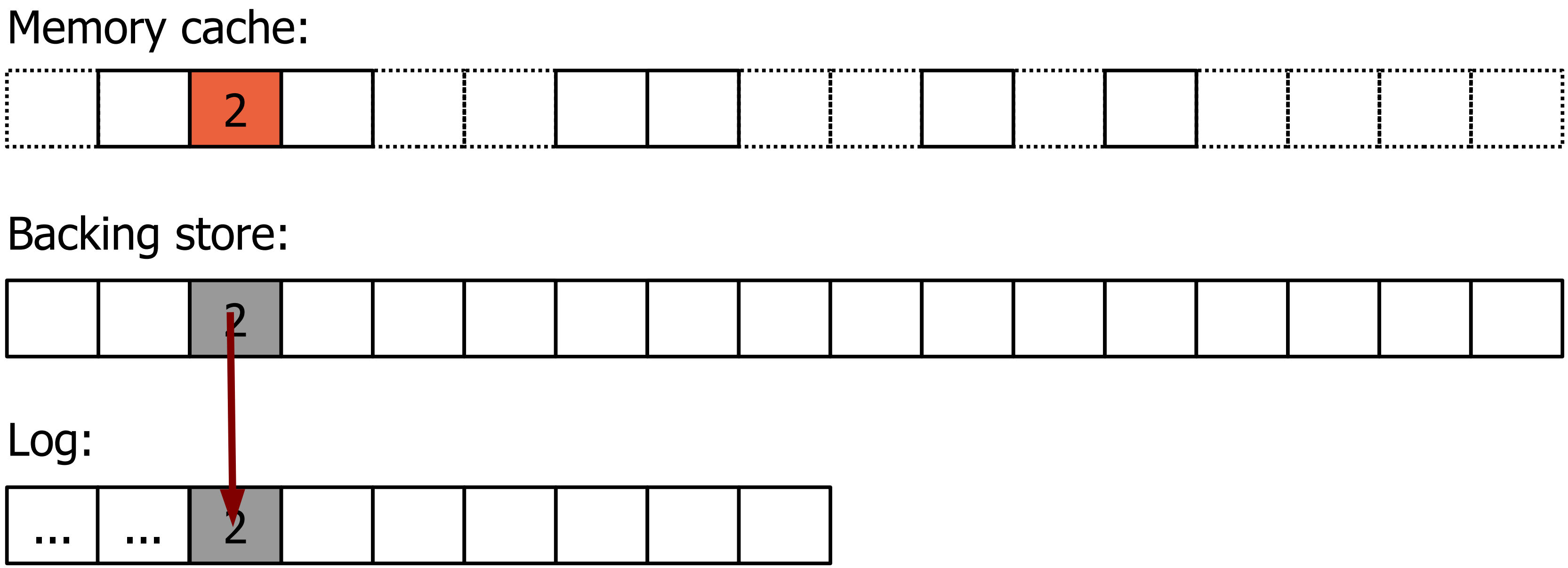
- The good:
  - Modifications can be written to backing store before entering commit (steal)
    - Lock granularity can be smaller than a block
    - Much easier checkpoints
    - Assumes only that the log is large enough to hold all modifications
- The bad:
  - Modifications must be written to backing store before entering commit (force)

Undo-Redo log

- Third method: Un-Do and Re-Do log
- Based on using both previous methods simultaneously

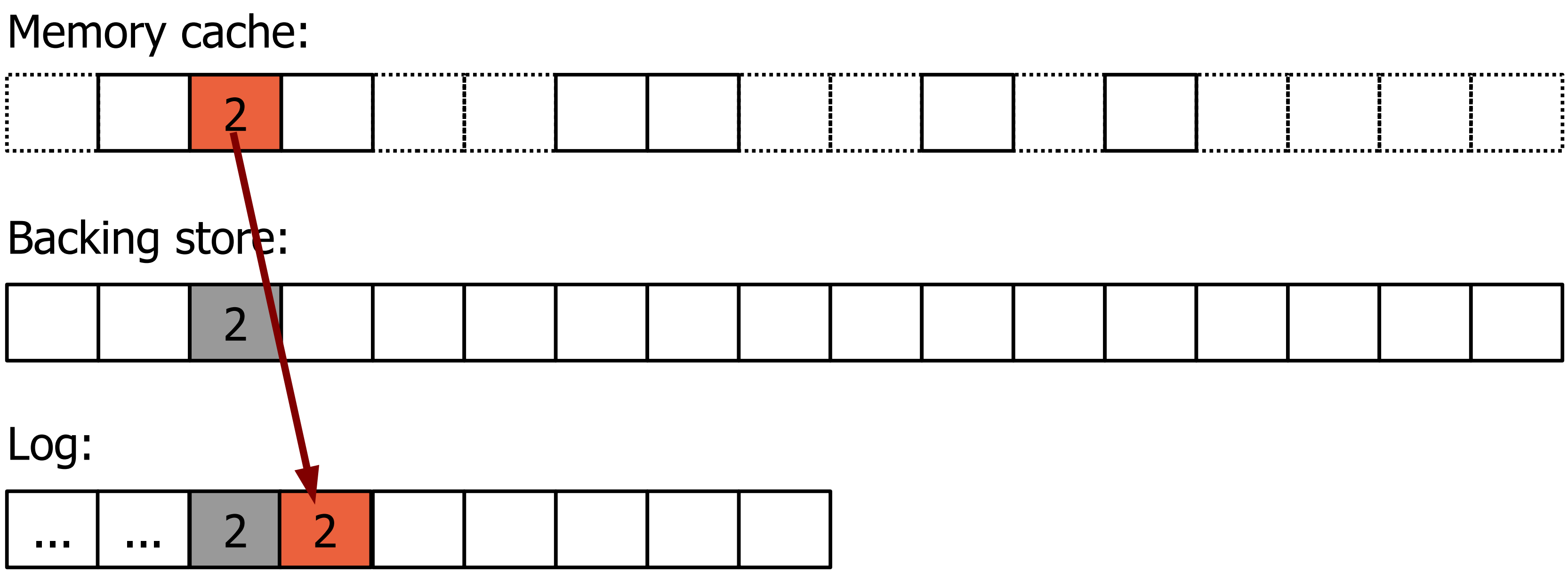
Undo-redo log

- Blocks can be changed after having been copied to the log (undo):



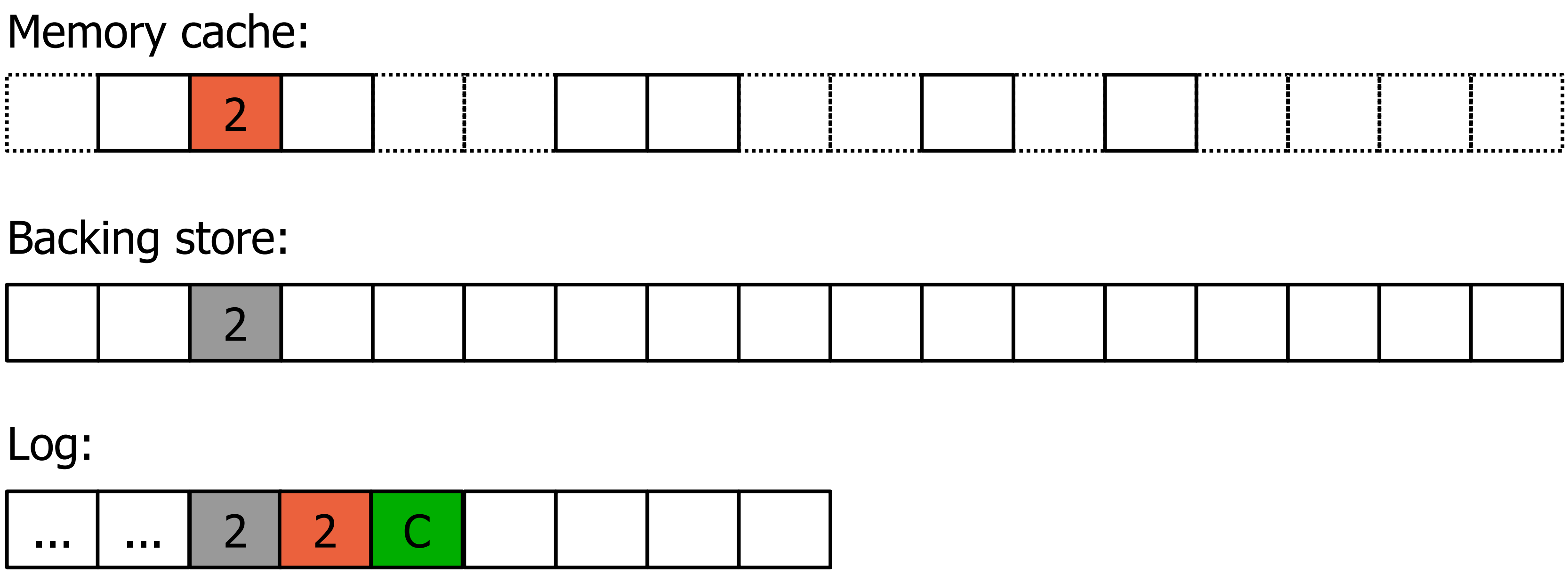
Undo-redo log

- Before commit, new values are also copied to the log (redo):



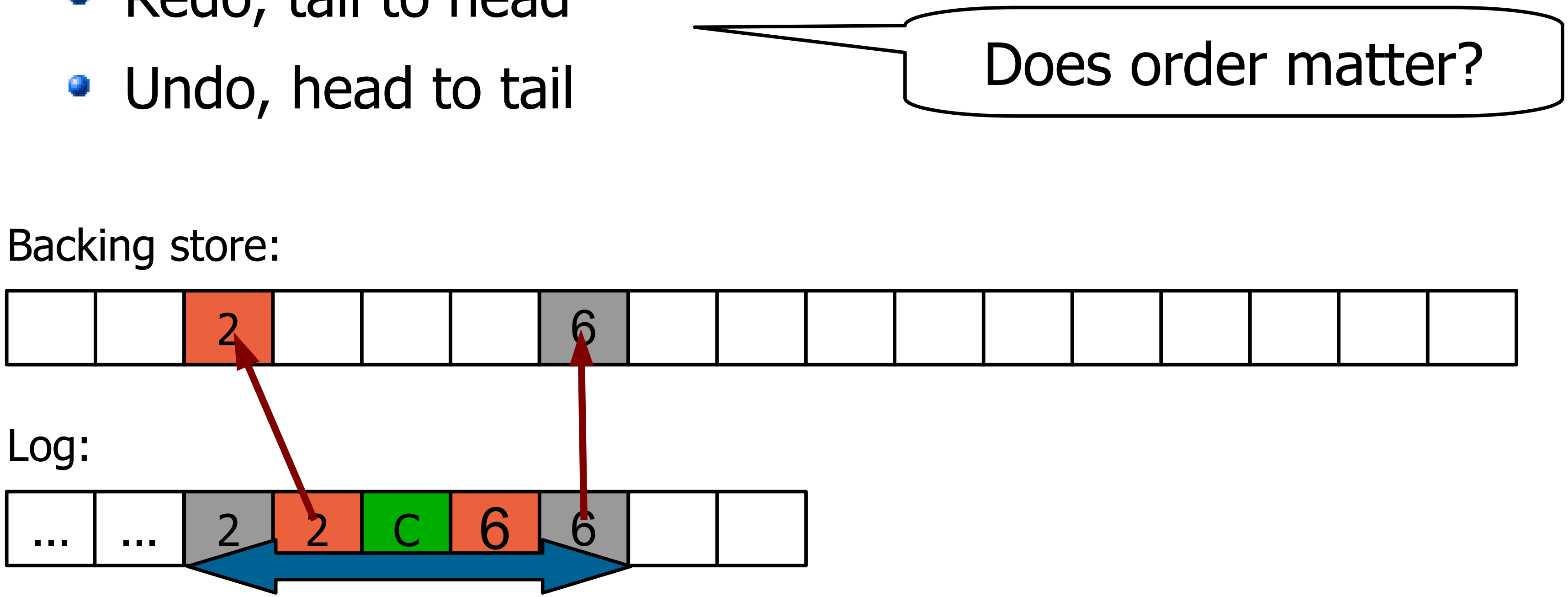
Undo-redo log

- A commit marker is inserted in the log:

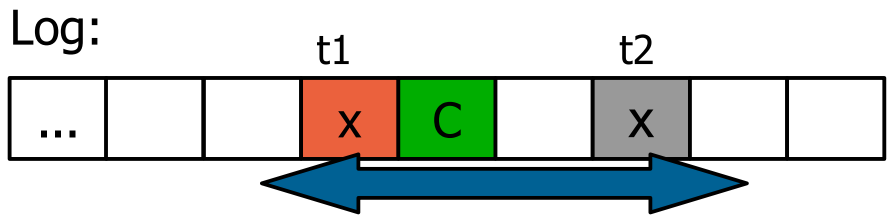


Undo-redo log

- Do both recovery procedures:
  - Redo, tail to head
  - Undo, head to tail



Undo-redo log



- Order matters only if:
  - Unfinished t2 updates same item as finished t1
  - t2 reads x before t1 writes x
- Means that:
  - t1 and t2 would be concurrent transactions
  - Both would have a lock on x!

Undo-redo log summary

- If all log entries contain <u>both</u> undo and redo data:
  - A block can be always be copied to backing store (<u>steal</u>)
  - A transaction can always commit immediately (<u>no force</u>)

Case study: PostgreSQL

- Log method:
  - Redo log (a.k.a. WAL)
  - Undo log implicit in previous versions (updates never overwrite!)
- Log format:
  - Logical for most operations
  - Physical, the first time each block is modified after a checkpoint
- Background writer and periodical checkpoints
- On-line archiving and backup (a.k.a. PITR)