Building Fast Recoverable Persistent Data Structures

Haosen Wen*, Wentao Cai*,
Mingzhe Du, Louis Jenkins, Benjamin Valpey
and Michael L. Scott

University of Rochester

March 10, NVMW '21

*Equal contributions

Background

- Non-volatile memory (NVM) offers the possibility of keeping pointer-rich data structures across program runs and even crashes:
    - Correct persistence order is needed for crash consistency
    - Volatile caches mean that stores may reach memory out of program order; explicit write-back and fence instructions are necessary
    - Durable linearizability [Izraelevitz et al., DISC'16] necessitates high latency in every operation—ops must persist before returning
    - *Buffered* durable linearizability might reduce this latency, but all known implementations are ad-hoc

Montage

- First general-purpose system for buffered durably linearizable data structures
- Excellent performance, makes good use of NVM by:
  - Persisting periodically (every 1 – 10ms, or whenever sync() is called) rather than per-operation
  - Persisting only abstract data

Persistence Order: Durable Linearizability

- Durable Linearizability[Izraelevitz et al., DISC'16] :
    - Intuitive correctness criterion: operations persist before return
    - Enforced by writes-back (for persistence) and fences (for ordering) on *every* happens-before relationship on persistent data
    - Significant overhead

Buffered Durable Linearizability

- Buffered Durable Linearizability [Izraelevitz et al., DISC'16] :
  - After a crash, drop not-fully-persisted suffix of the history
  - Just make sure if O_1 happens before O_2 and O_2 is persisted, O_1 must be persisted
  - Agrees with persistency models of databases and file systems
- Reduces the overhead of persistence ordering
  - Avoid the need to write back and fence each op before returning & on each happens-before relationship

Montage: Periodic Persistence

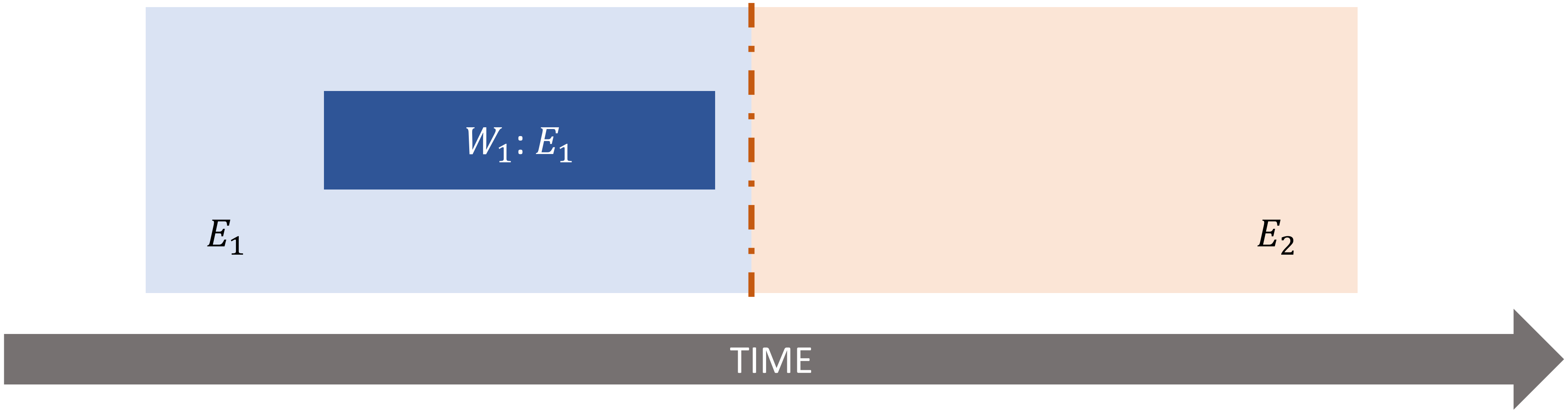
- Inspired by Dalí[Nawab et al., DISC'17], Montage implements **buffered** durable linerizability by dividing time into *epochs*, and

$$\text{epoch}(O_1) < \text{epoch}(O_2) \Rightarrow \neg(O_2 \prec_{hb} O_1)$$

  - Each operation is marked with *one* epoch
  - Operations in the same epoch persist together, atomically

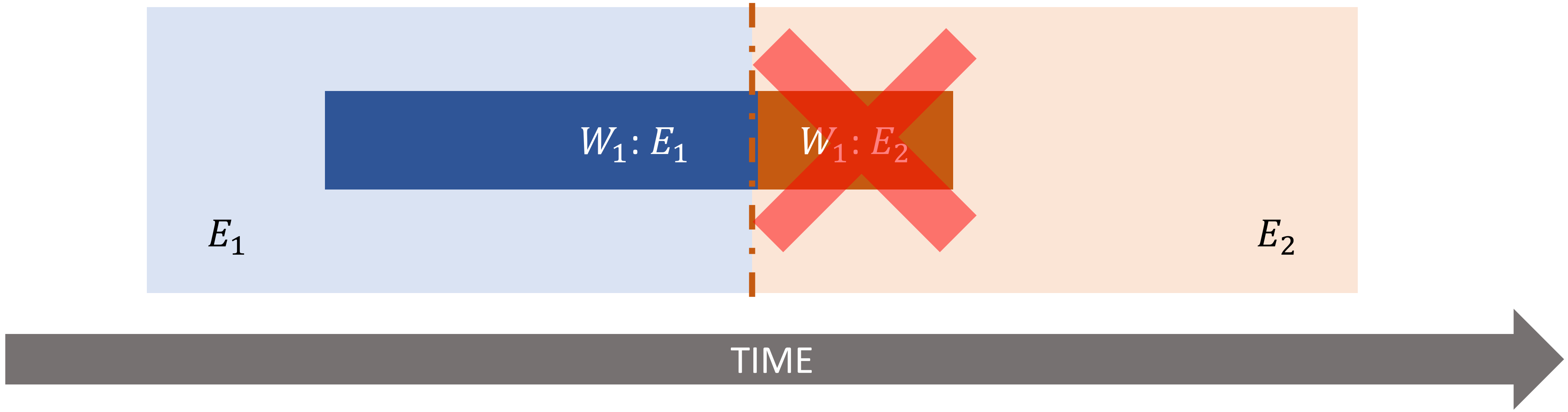
Montage: Periodic Persistence

- Design:
  - Write operations are assigned epoch numbers
    - All writes of an operation are marked with the same epoch



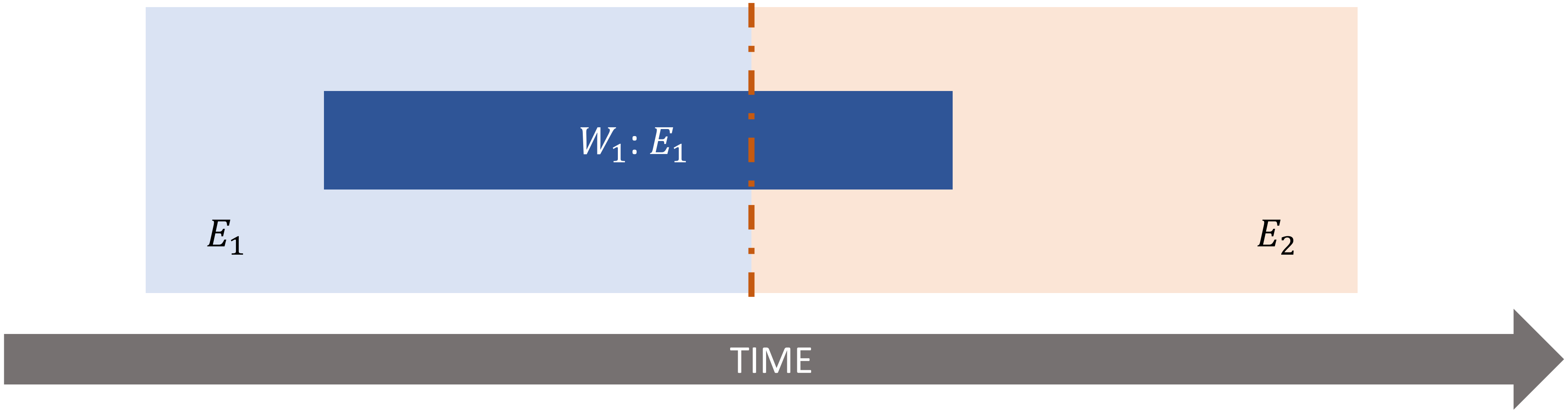
Montage: Periodic Persistence

- Design:
  - Write operations are assigned epoch numbers
    - All writes of an operation are marked with the same epoch



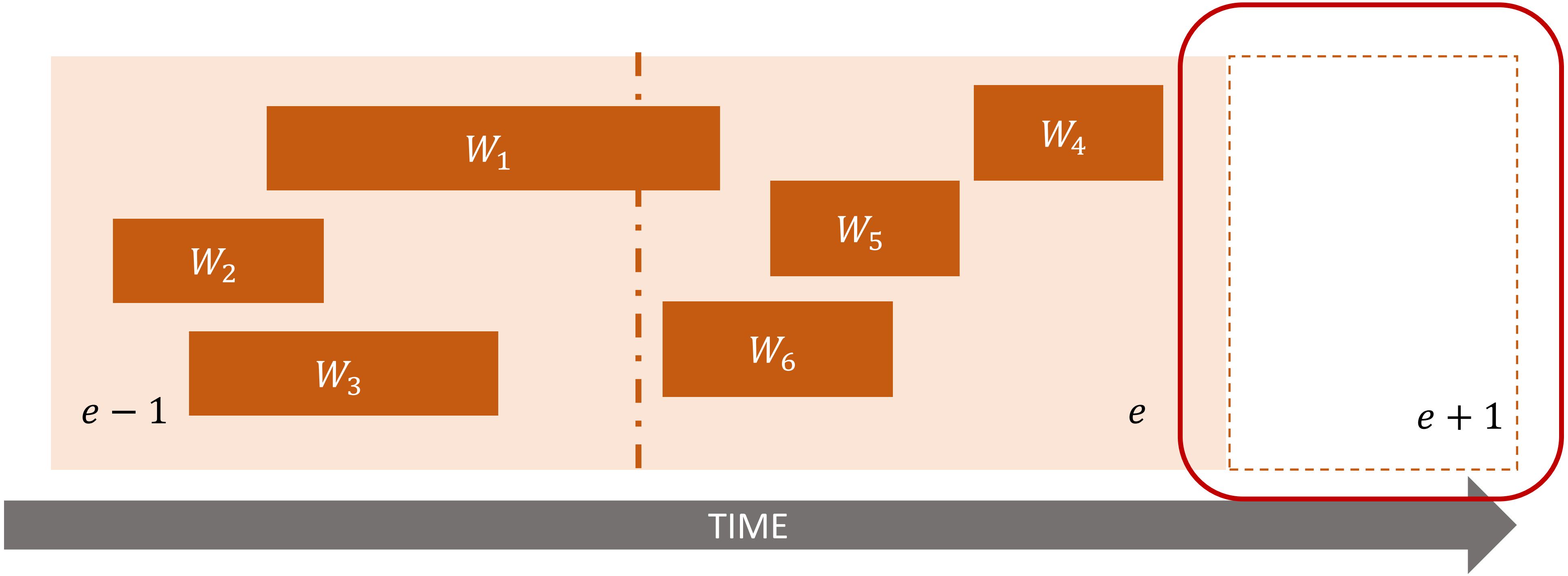
Montage: Periodic Persistence

- Design:
  - Write operations are assigned epoch numbers
    - All writes of an operation are marked with the same epoch



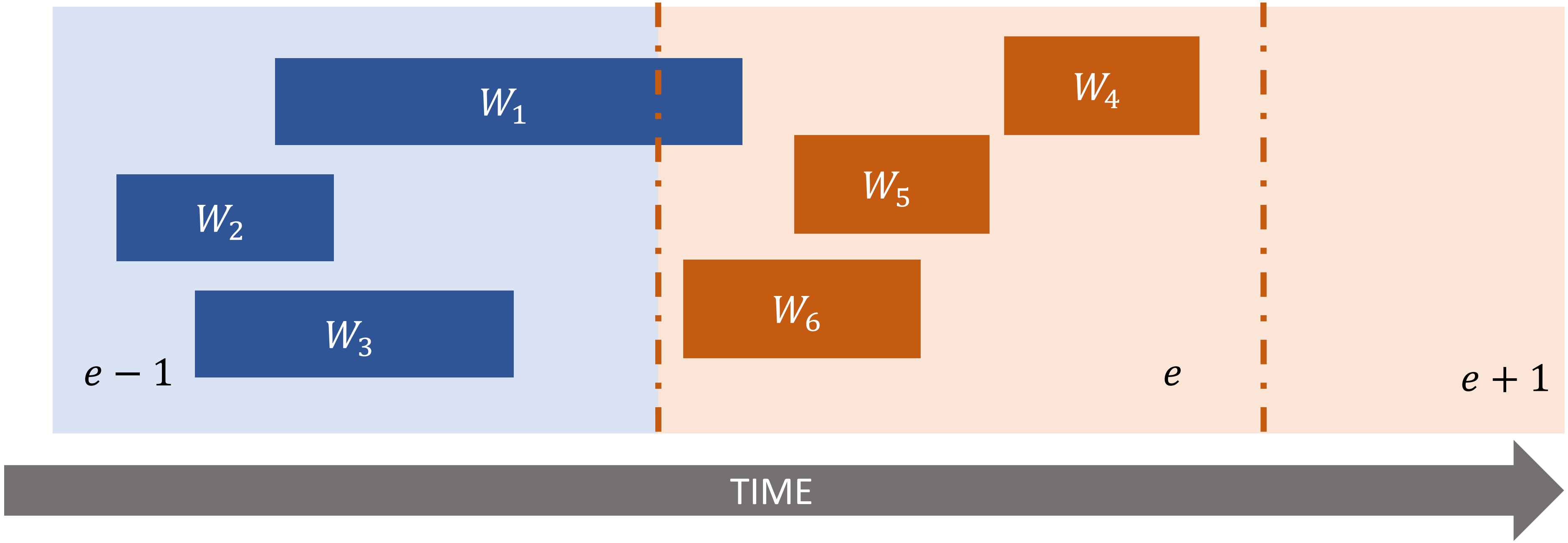
Montage: Periodic Persistence

- Design:
  - Before $e \rightarrow e+1$, operations in $e-1$ are finished and persisted



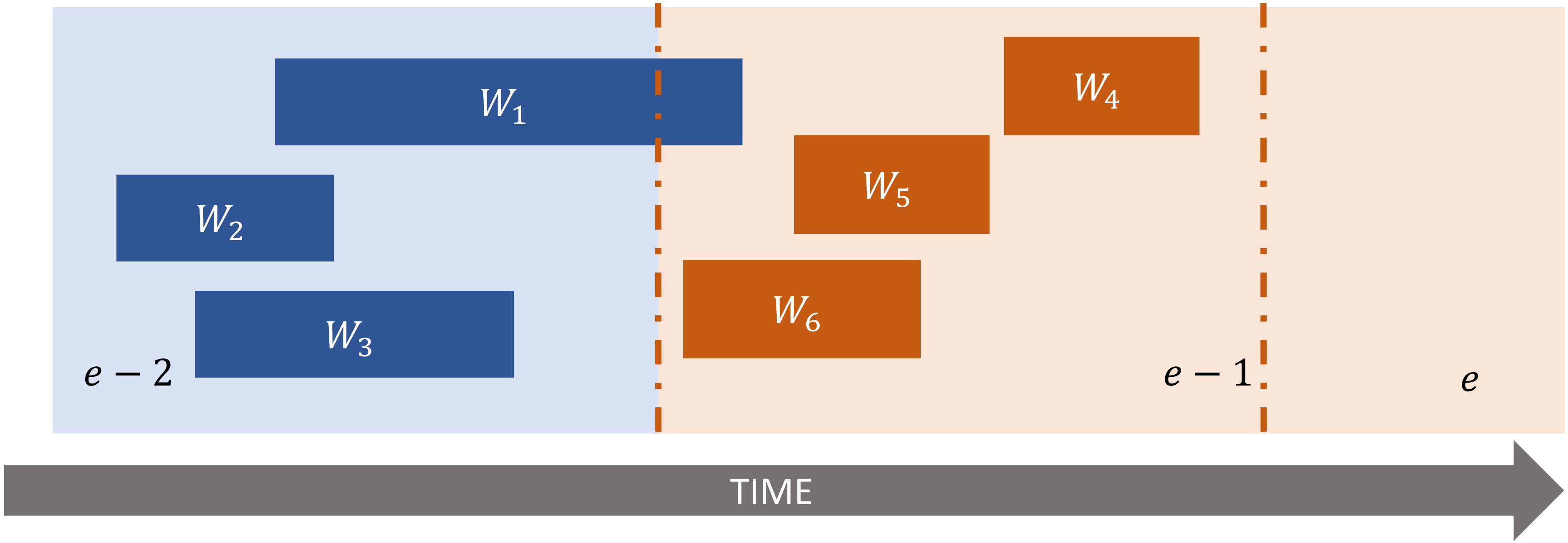
Montage: Periodic Persistence

- Design:
  - Before $e \rightarrow e+1$, operations in $e-1$ are finished and persisted



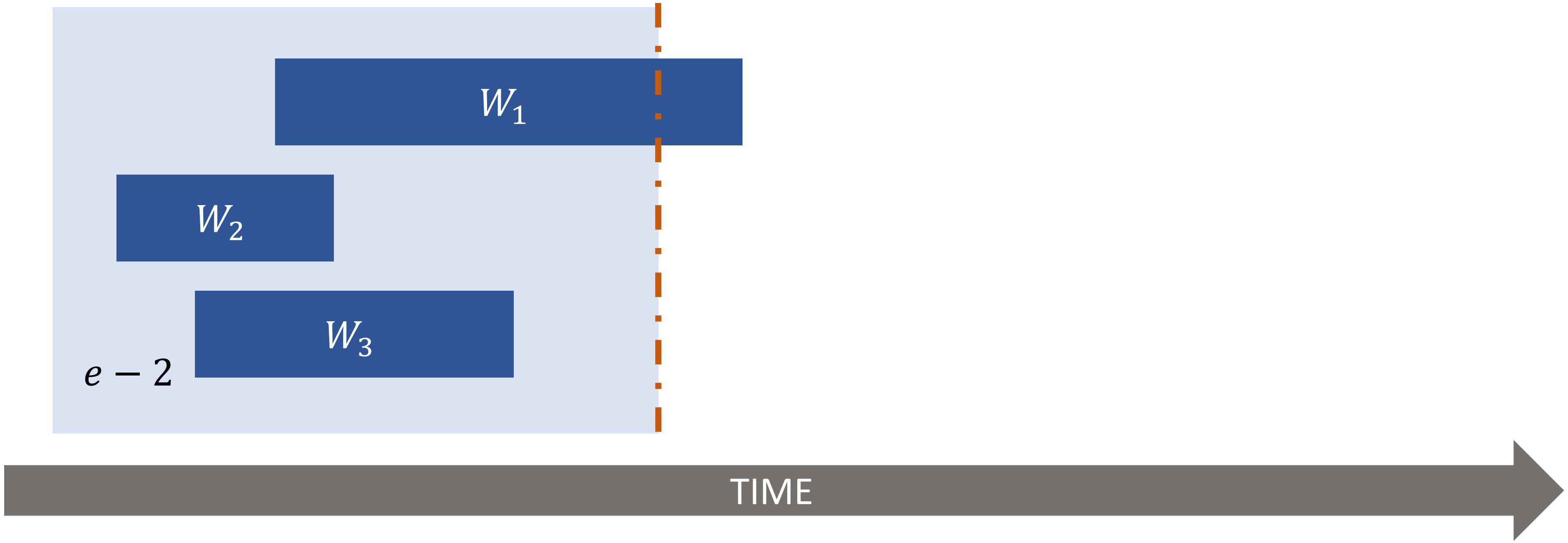
Montage: Periodic Persistence

- Design:
  - If we crash in e, all operations in $e-1$ and e are discarded
    - The boundary between $e-2$ and $e-1$ is chosen as the consistent cut
    - No in-place updates of blocks from old epochs – copy to preserve history



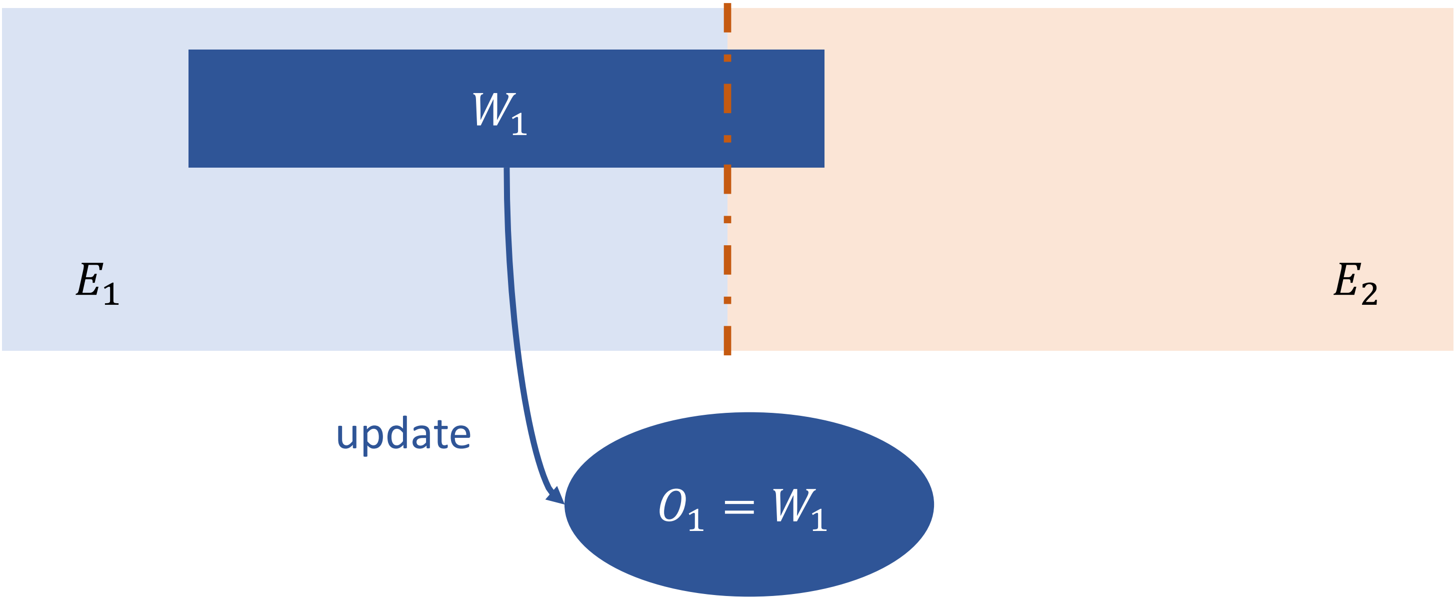
Montage: Periodic Persistence

- Design:
  - If we crash in e, all operations in $e-1$ and e are discarded
    - The boundary between $e-2$ and $e-1$ is chosen as the consistent cut
    - No in-place updates of blocks from old epochs – copy to preserve history



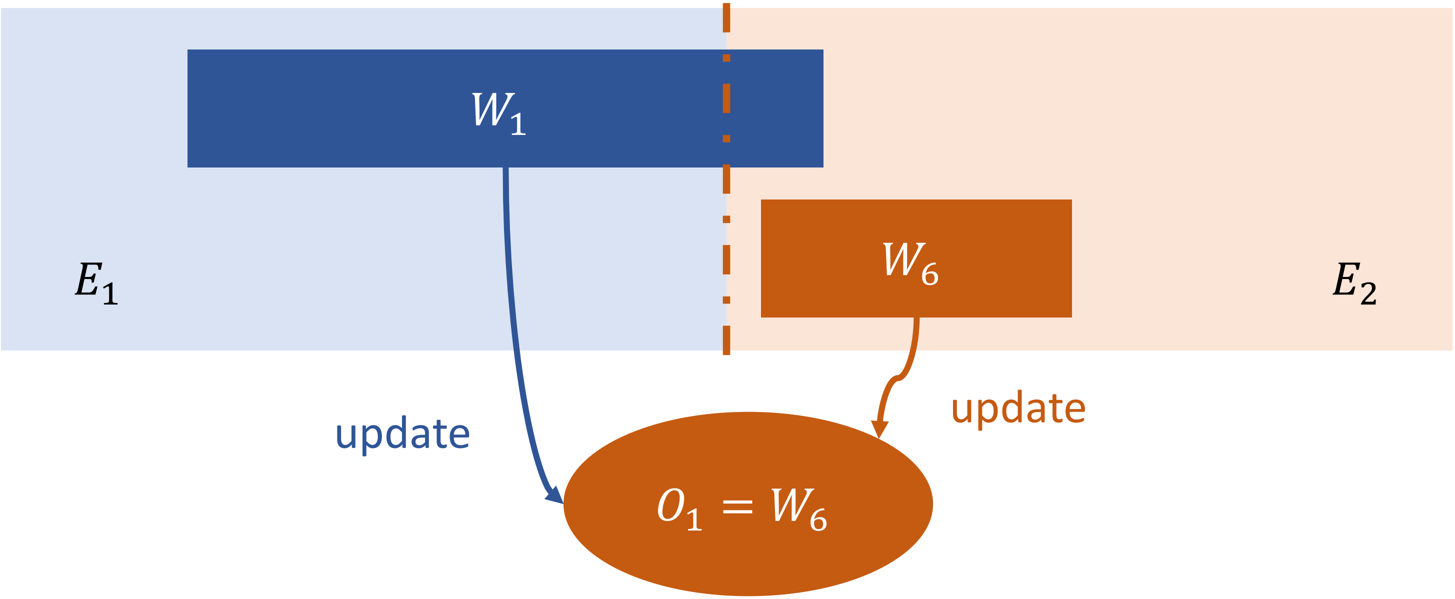
Montage: Periodic Persistence

- Design:
  - If we crash in e, all operations in $e-1$ and e are discarded
    - The boundary between $e-2$ and $e-1$ is chosen as the consistent cut
    - No in-place updates of blocks from old epochs – copy to preserve history



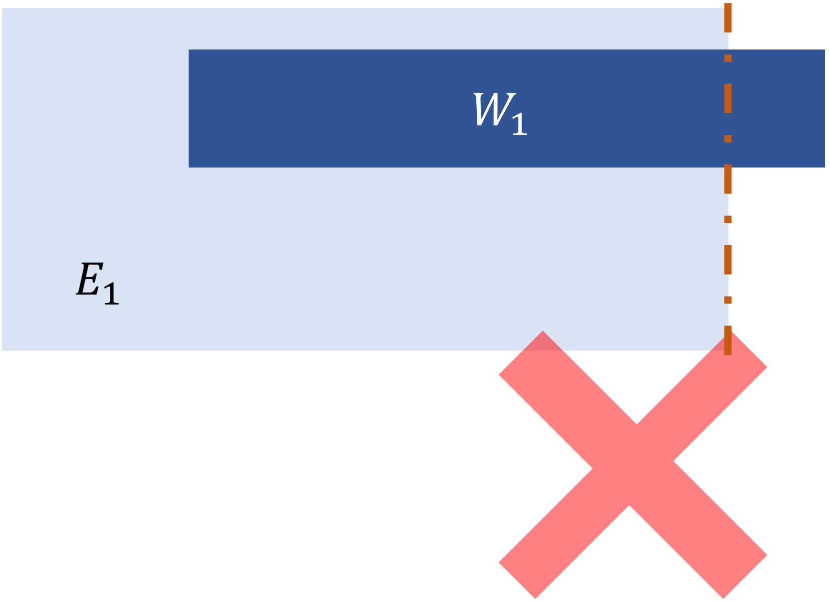
Montage: Periodic Persistence

- Design:
  - If we crash in e, all operations in $e-1$ and e are discarded
    - The boundary between $e-2$ and $e-1$ is chosen as the consistent cut
    - No in-place updates of blocks from old epochs – copy to preserve history



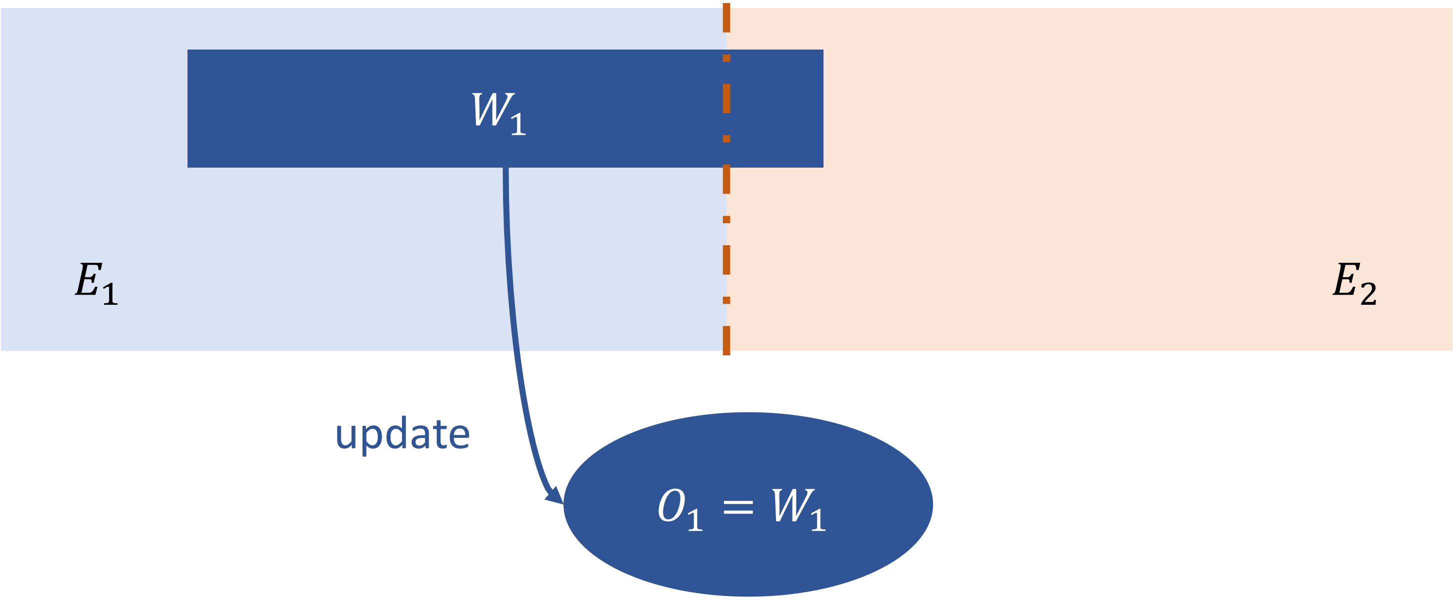
Montage: Periodic Persistence

- Design:
  - If we crash in e, all operations in $e-1$ and e are discarded
    - The boundary between $e-2$ and $e-1$ is chosen as the consistent cut
    - No in-place updates of blocks from old epochs – copy to preserve history



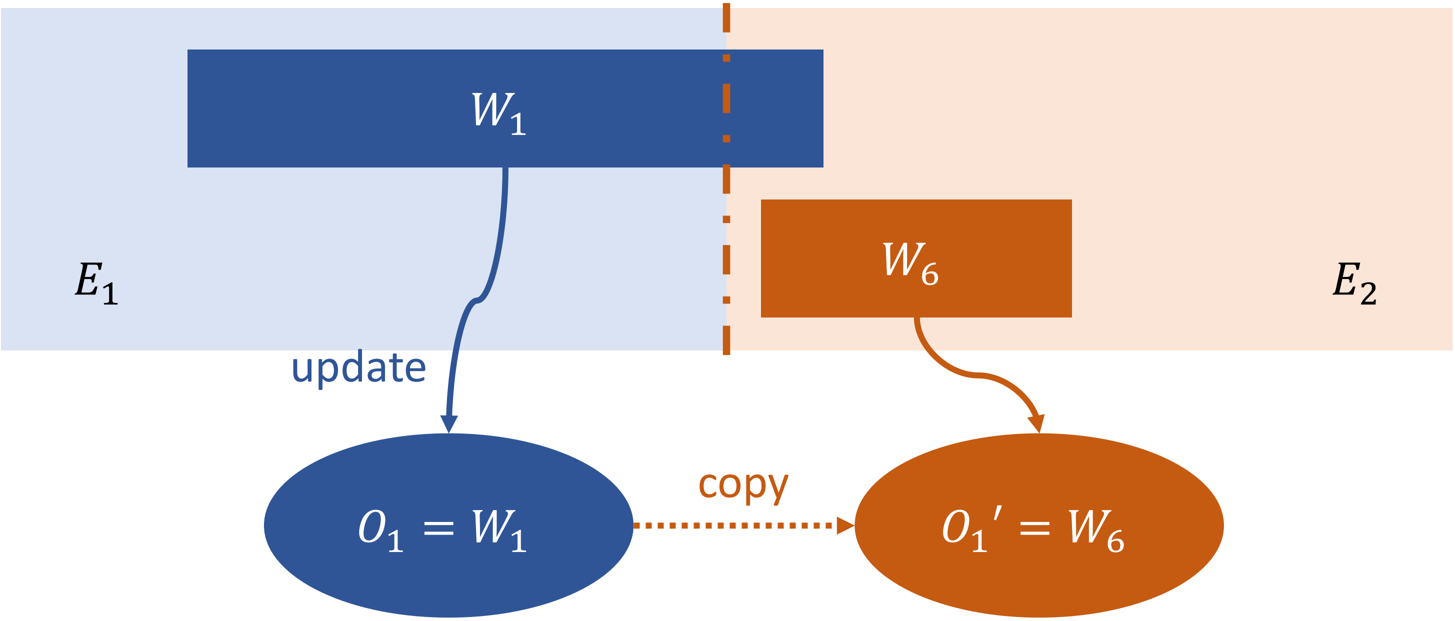
Montage: Periodic Persistence

- Design:
  - If we crash in e, all operations in $e-1$ and e are discarded
    - The boundary between $e-2$ and $e-1$ is chosen as the consistent cut
    - No in-place updates of blocks from old epochs – copy to preserve history



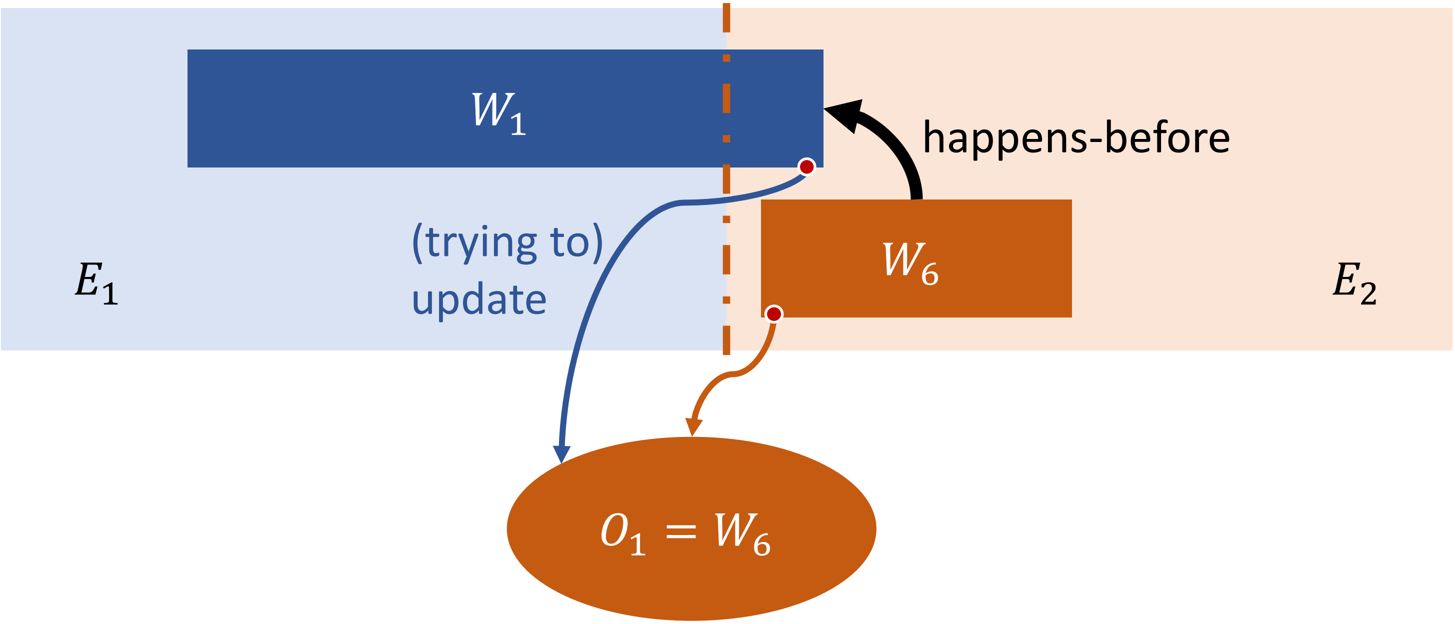
Montage: Periodic Persistence

- Design:
  - If we crash in e, all operations in $e-1$ and e are discarded
    - The boundary between $e-2$ and $e-1$ is chosen as the consistent cut
    - No in-place updates of blocks from old epochs – copy to preserve history



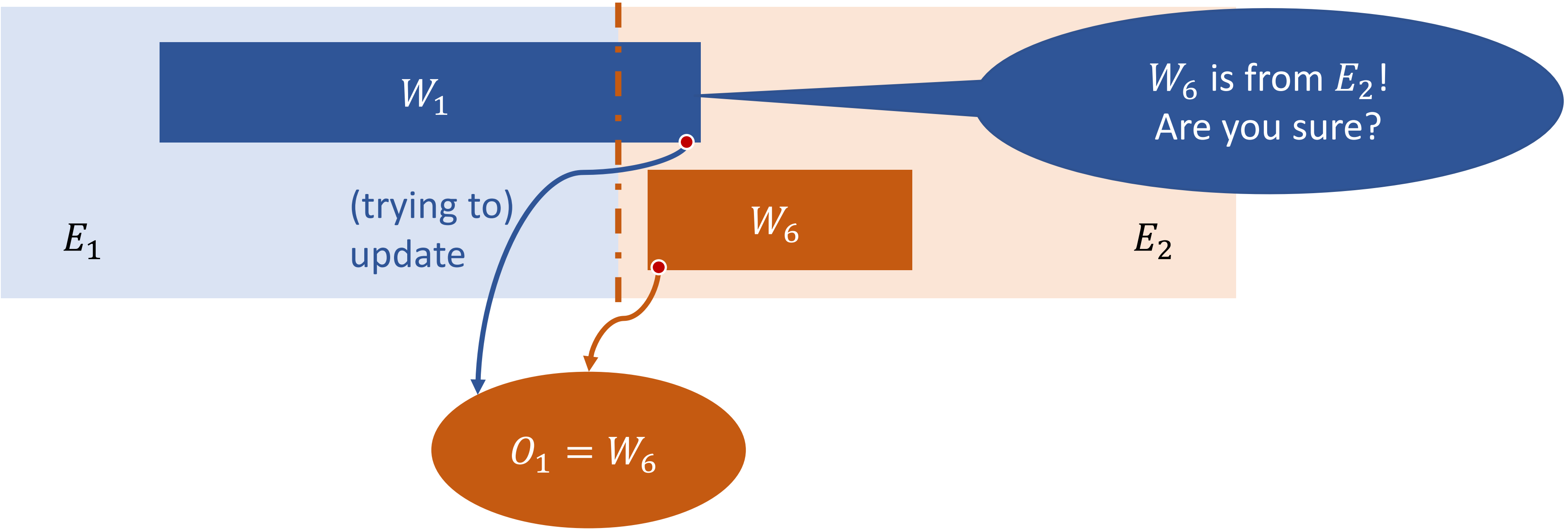
Montage: Periodic Persistence

- Design:
    - Data structure must ensure each operation linearizes in the epoch of its writes
        - Operation in e seeing blocks from $e' > e$ suggests there *might* be a problem. Montage (optionally) raises an exception to help



Montage: Periodic Persistence

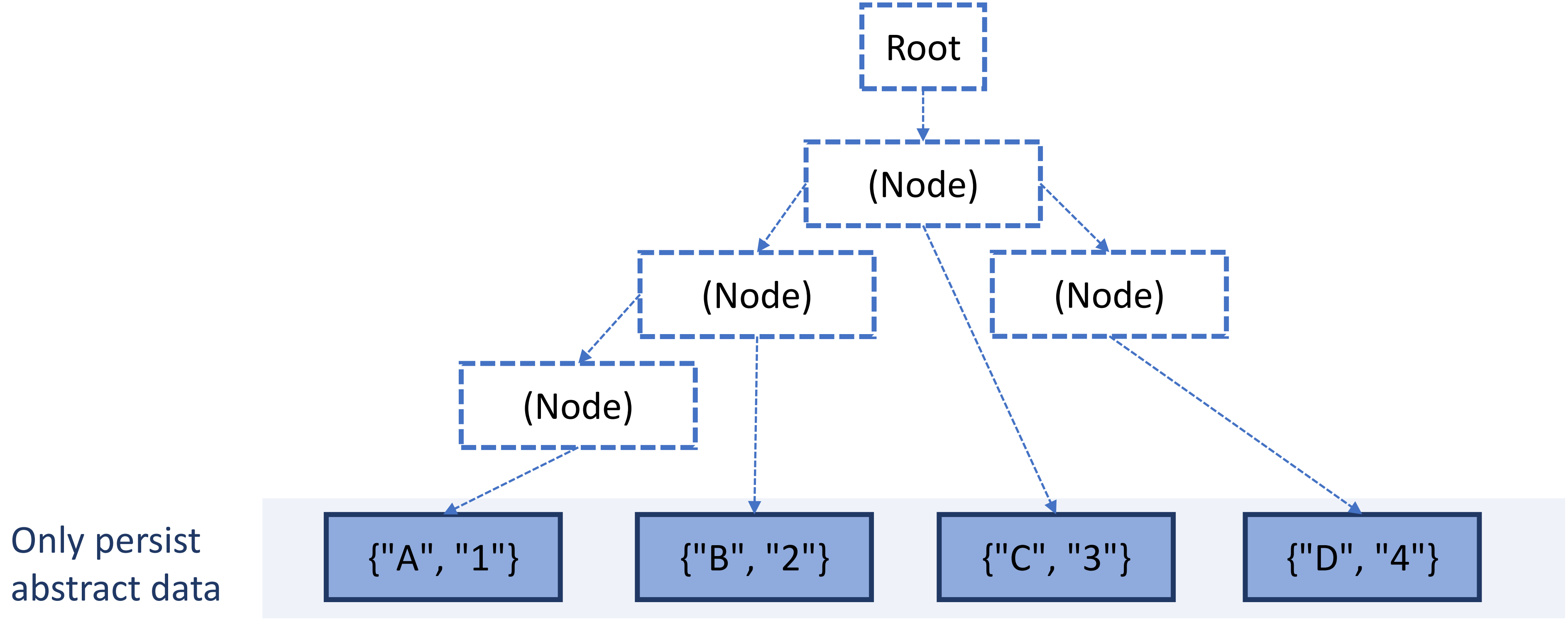
- Design:
  - Data structure must ensure each operation linearizes in the epoch of its writes
    - Operation in e seeing blocks from $e' > e$ suggests there *might* be a problem. Montage (optionally) raises an exception to help



Montage: Persisting Abstract Data Only

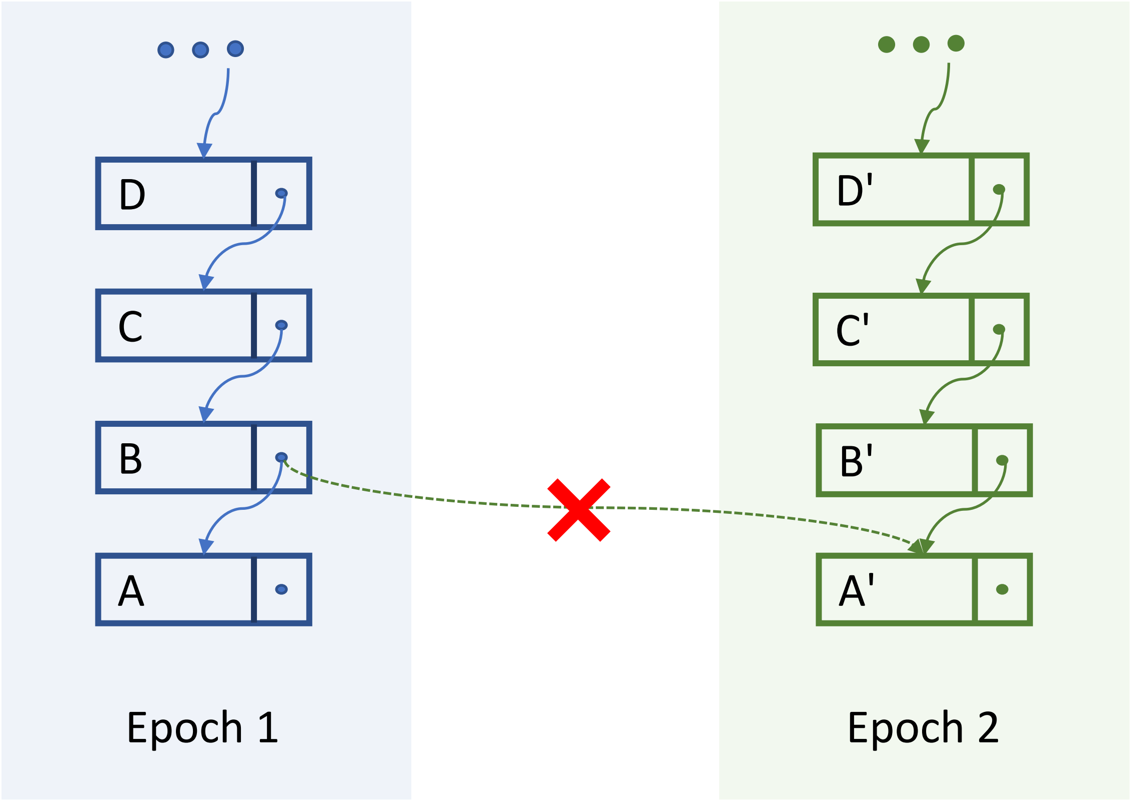
- Inspired by NV-Tree[Yang et al., FAST'15] , FPTree[Oukid et al., SIGMOD'16], Ralloc[Cai et al., ISMM'20], and Pronto[Memaripour et al., ASPLOS'20], among others, data structures can be rebuilt from abstract data after crash
  - Sets/maps: keys (and values)
  - Queues: values and order
  - Graphs: vertices and edges
- Abstract data may take the majority of a data structure's memory usage
- Can always persist more than abstract data for faster recovery

Montage Persistent Mapping



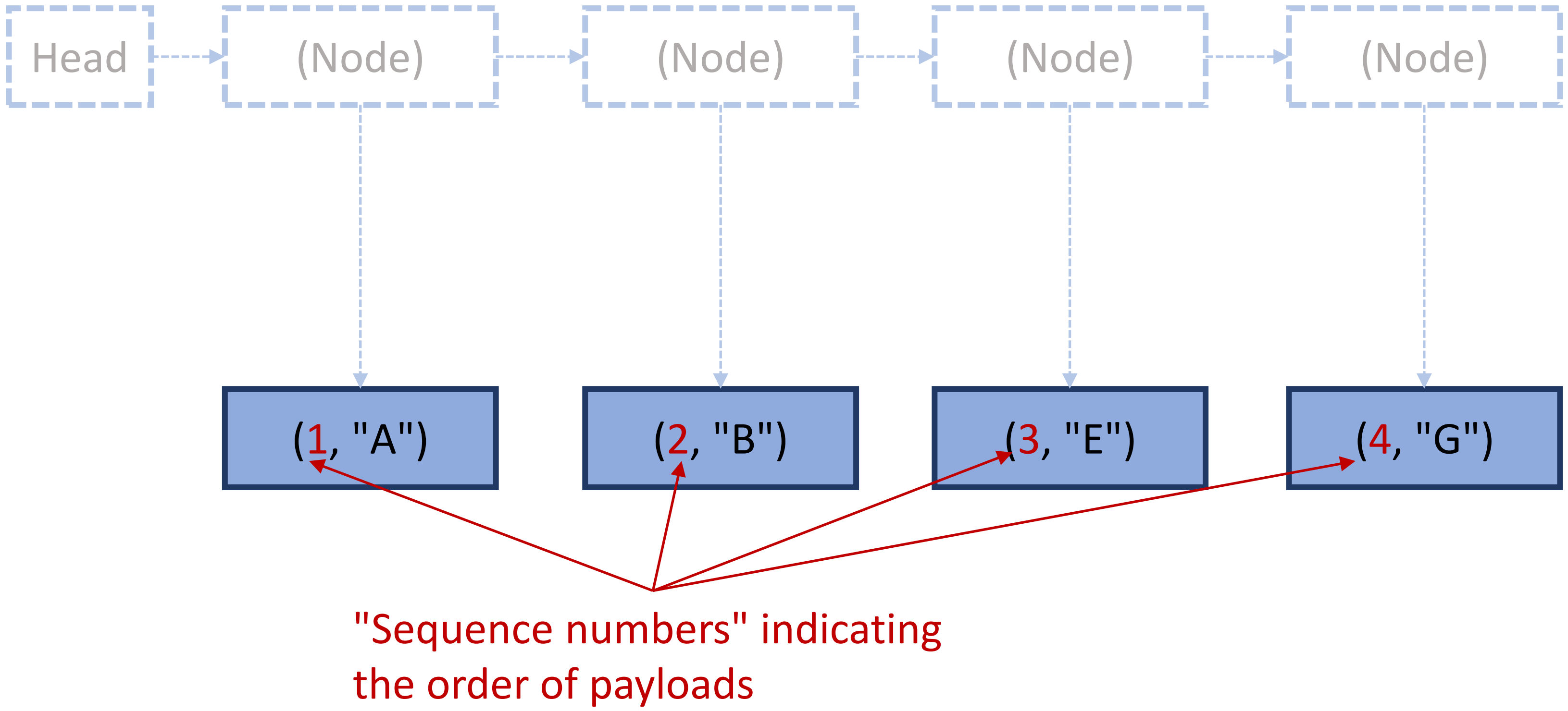
Montage: Avoid Persistent Long Chains

- Avoid long persistent chains consist of pointers in data structures like **queues** and **graphs**, since an update of one node in a new epoch can propagate to the beginning of the chain due to copying



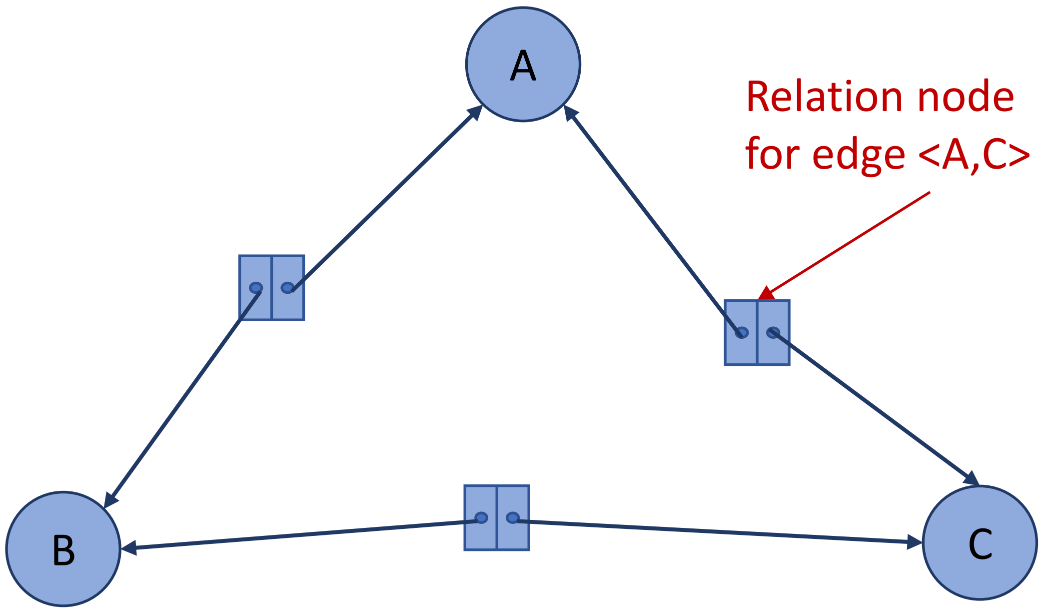
Prevent Persistent Chain in Queue

- Use "sequence numbers" in queue nodes instead of pointers



Prevent Persistent Chain in Graph

- Use "relation nodes" in place of pointers in general graphs to keep chain length ≤ 2



Nonblocking Data Structures

- Operation O in e needs to linearize before advance $e \rightarrow e + 1$
  - The advance may happen before O's linearization point
  - If so, O's linearization point needs to "fail" in old epoch
- All possible linearization points must be epoch-verified
  - Visible readers: double-wide counted CAS in all accesses
  - Invisible readers: double-compare-single-swap (DCSS) or HTM in updates
- Nonblocking epoch advance in progress

Montage: Implementation

- Use Ralloc[Cai et al., ISMM'20] as NVM allocator
- Montage provides (C++) API to:
  - track reads and writes ((de-)allocations, updates) from/to persistent payloads.
  - get the boundaries of each operation to ensure writes are marked with the same epoch for an operation
- Persisting writes, buffering reclamations:
  - clwb right after each write messes up cache locality on current machines, while buffering unbounded writes brings overhead and stretches epochs
  - Bounded buffers for to-be-persisted writes
  - Reclamations must be buffered for 2 epochs – cannot be undone after crash
  - Only need those containers for 4 epochs: reuse containers from 3 epochs ago

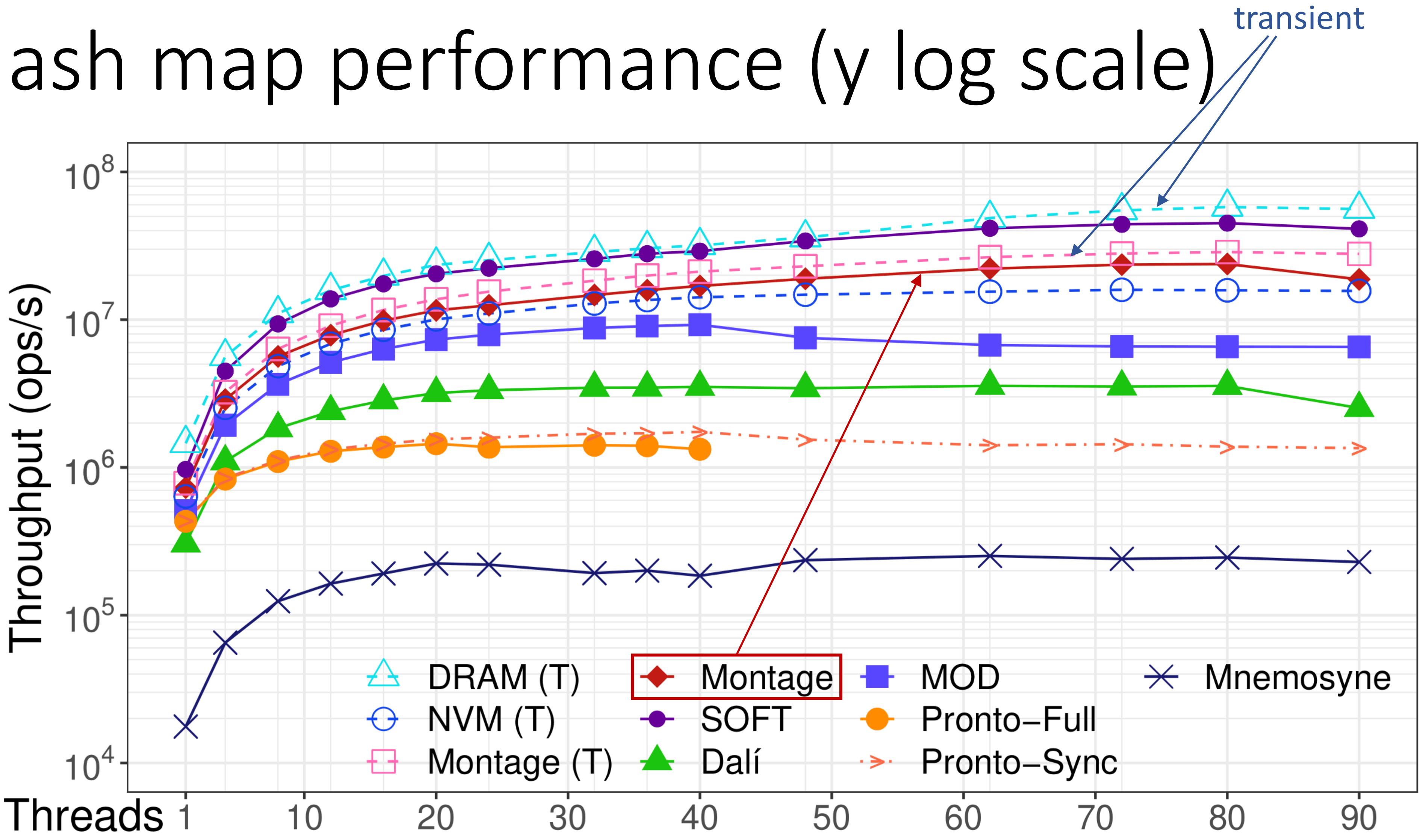
Montage: Implementation

- Epoch advances and sync()
  - Epoch advances every 1 – 10 ms, automatically
  - sync() blocks until all returned operations persists
  - Epoch e gets persisted in $e + 2$, so sync() asks epoch to advance twice immediately
  - A background epoch advancer thread, before advancing to $e + 1$:
    - Reclamation for $e - 2$
    - Writes-back for $e - 1$
    - sfence
    - Advance epoch
    - Repeat until all sync() goals are met
  - Background thread eases the burden of worker threads

Experimental Setup

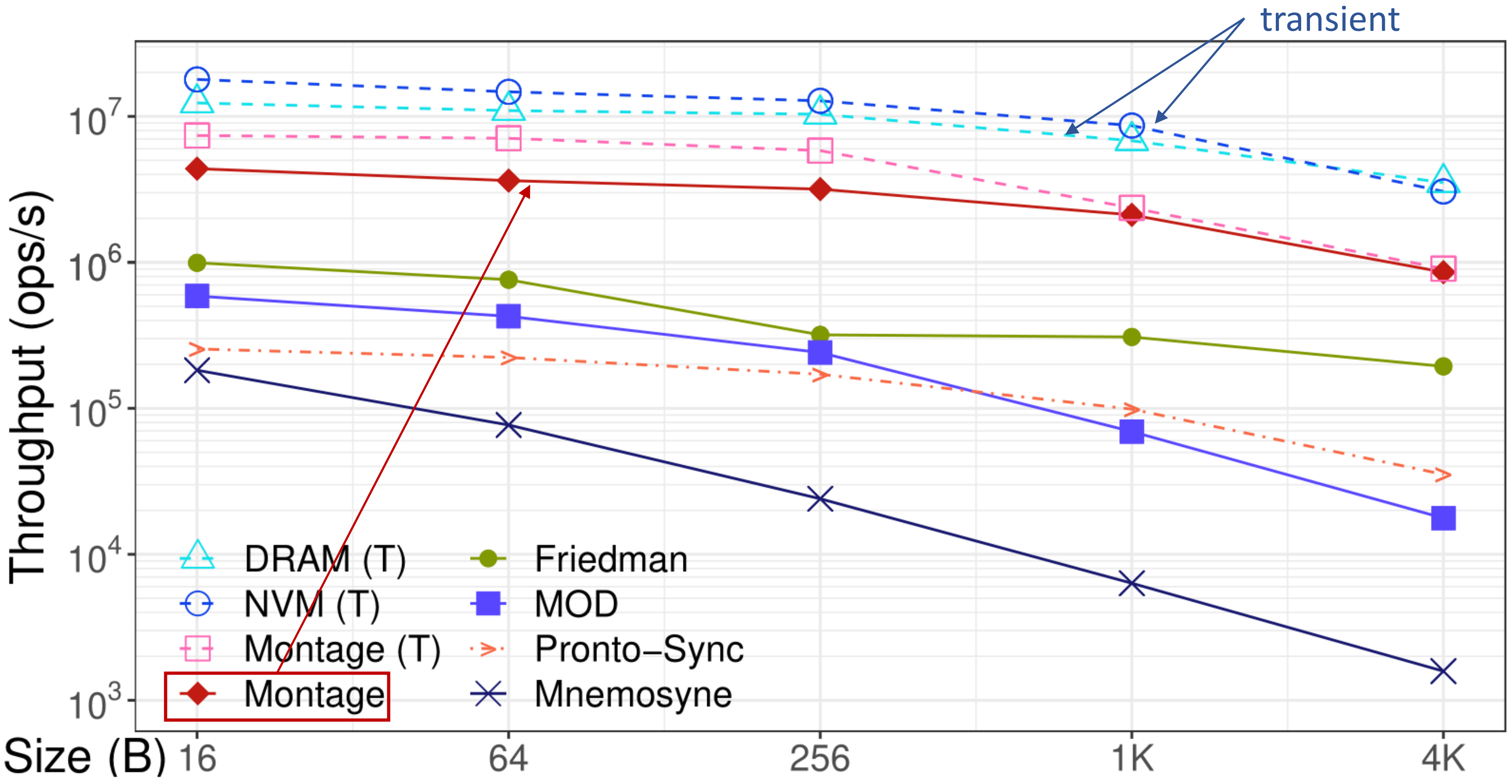
- 2x Intel Xeon Gold 6230 processors, 80 physical threads in total
- `128GB*12` NVM in 2 sockets, `mmaped` in DAX mode, interleaved using `dm-stripe` with 2MB chunk size
- Threads first goes into cores one socket, then hyperthreads in the same socket, then cross socket

Hash map performance (y log scale)



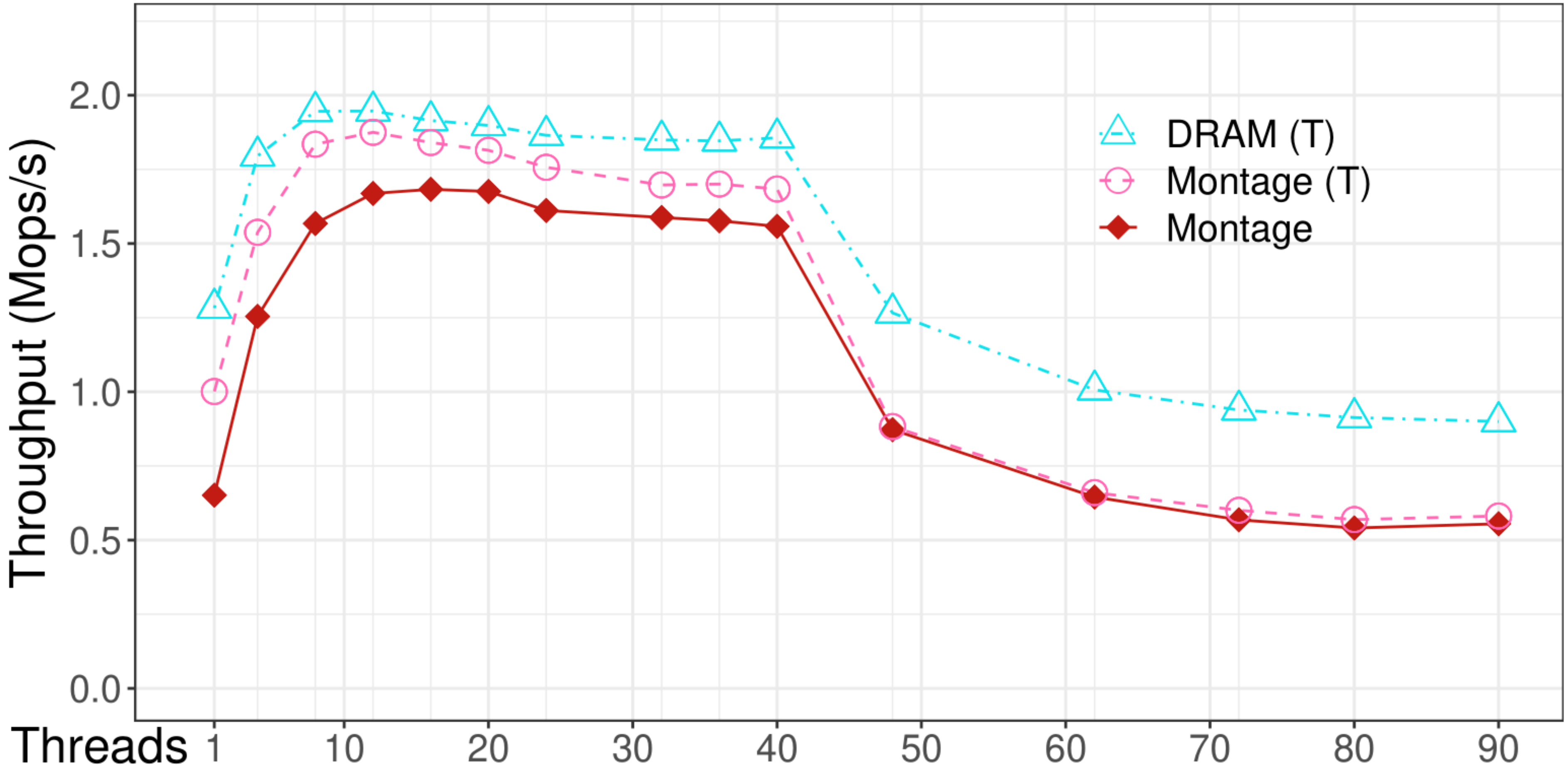
Hash Maps: 90% lookups, 10% updates

Queue Performance (1 thread, log scale)



Queues: 50% enqueue, 50% dequeue

Memcached Performance (linear scale)



Memcached: YCSB-A

Conclusion

- Montage reduces the persistence overhead of recoverable data structures by:
  - Reducing cost of persist ordering
  - Reducing the amount of persistent data
- Suitable for both lock-based and nonblocking data structures
- Unprecedented performance
- Ongoing: nonblocking epoch advance
- Future work: Atomic composition of operations on multiple data structures
- Full version: https://arxiv.org/abs/2009.13701
- BA in DISC'21: https://drops.dagstuhl.de/opus/volltexte/2020/13130/
- Artifact: https://github.com/urcs-sync/Montage