# CS647 Modern Storage Systems

- $B^{\varepsilon}$-trees

Giorgos Saloustros, gesalous@ics.forth.gr

CS647 Modern Storage Systems, Fall 2022
(Modern Key-Value Store Design and Implementation)
Angelos Bilas, bilas@csd.uoc.gr
Computer Science Department, University of Crete

# Relevant Historical Timeline

- 1996 LSM
  - Publication of the log structured merge tree
- 1997 Tiering
  - Incremental Organization for Data Recording and Warehousing
- 2003 $B^{\varepsilon}$-tree
  - Buffered, write-optimized trees
- 2006 Partitioned exponential files
  - Full leveling but at small chunks
- 2012 RocksDB LSM incremental compaction
  - https://github.com/facebook/rocksdb/wiki/Leveled-Compaction

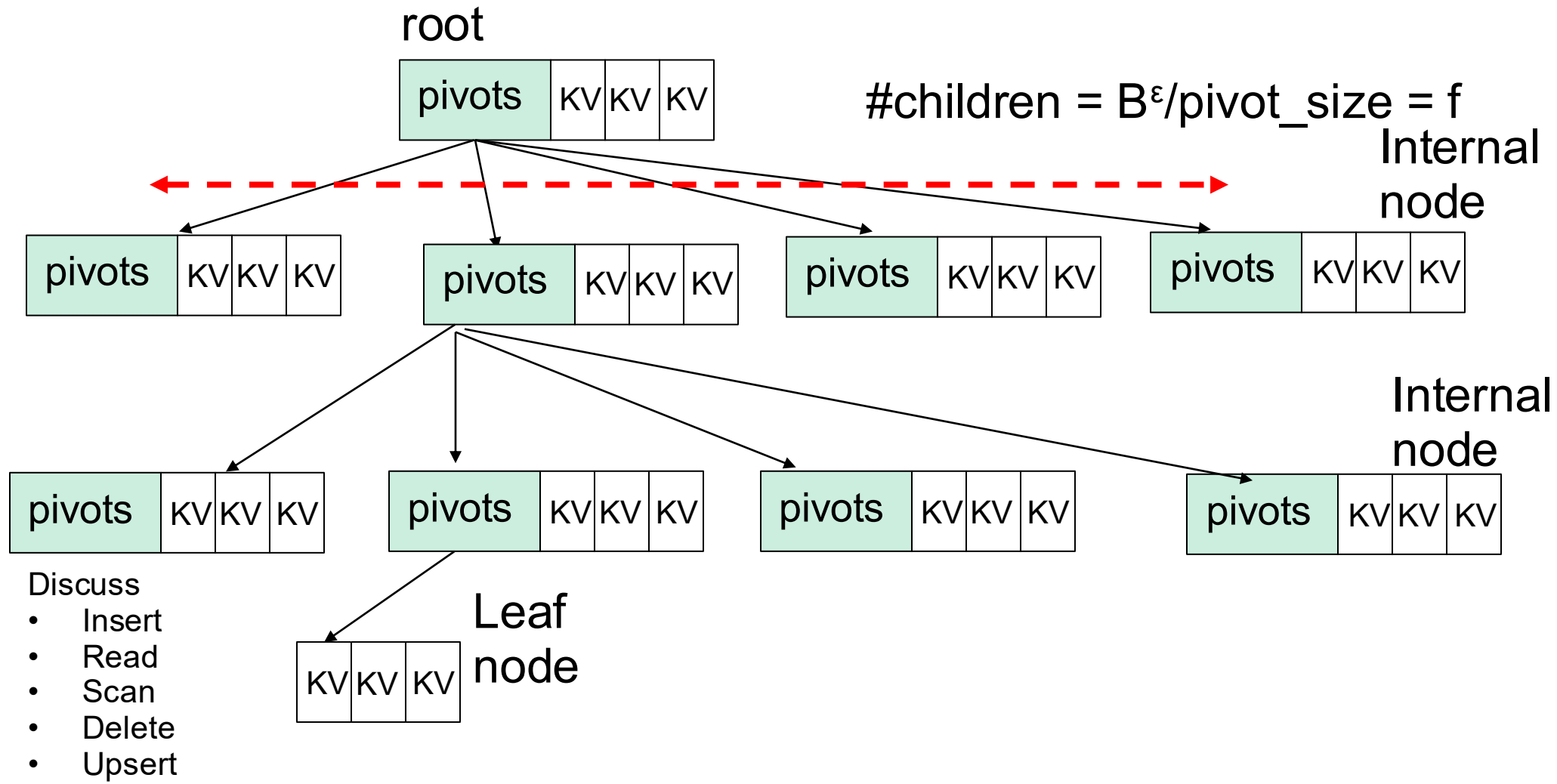
# $B^{\varepsilon}$-Trees

[M. Bender et al. An Introduction to Bε-trees and Write-Optimization. Usenix ;login:, October 2015, Vol. 40, No. 5. https://www.usenix.org/system/files/login/articles/login_oct15_05_bender.pdf]
[A. Papagiannis et al. 2016. Tucana: design and implementation of a fast and efficient scale-up key-value store. USENIX ATC '16. https://www.usenix.org/system/files/conference/atc16/atc16_paper-papagiannis.pdf]

- B-Tree variant that uses buffering to improve inserts
- Operations: Insert, Update, Delete, Scan
  - Supports merge operations of RocksDB (called upserts in [1])
- Similar complexity as B-Tree for point, range queries
- Buffered inserts – much better than B+Tree
- No compactions – small unsorted buffers, full index
- Better CPU overhead and I/O amplification
- Worse randomness and size of I/O operations

# Anatomy of a Bε-tree

- B+Tree where each node contains KV pairs
- ε is the portion of each node used for pivots (rest for KV pairs)
- ”Buffered” structure

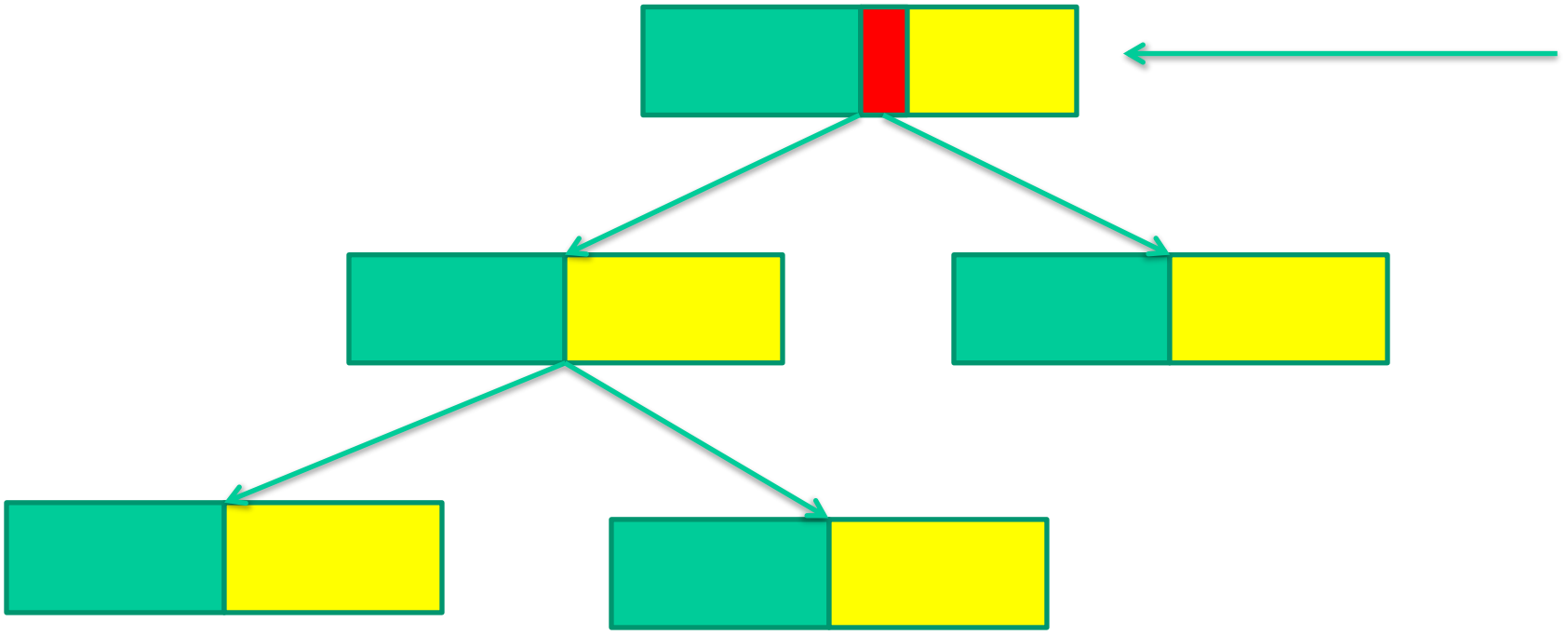


# Upserts

- Update operations are expensive
  - Read – Modify – Write
- Use compaction to perform blind updates—no read
  - Similar to merge operators of RocksDB
- Upsert are placed in a buffer as with other operations
- Upsert message consists of
  - Target Key
  - Callback function, f
  - Function arguments
- Callback is evaluated whenever the message is applied

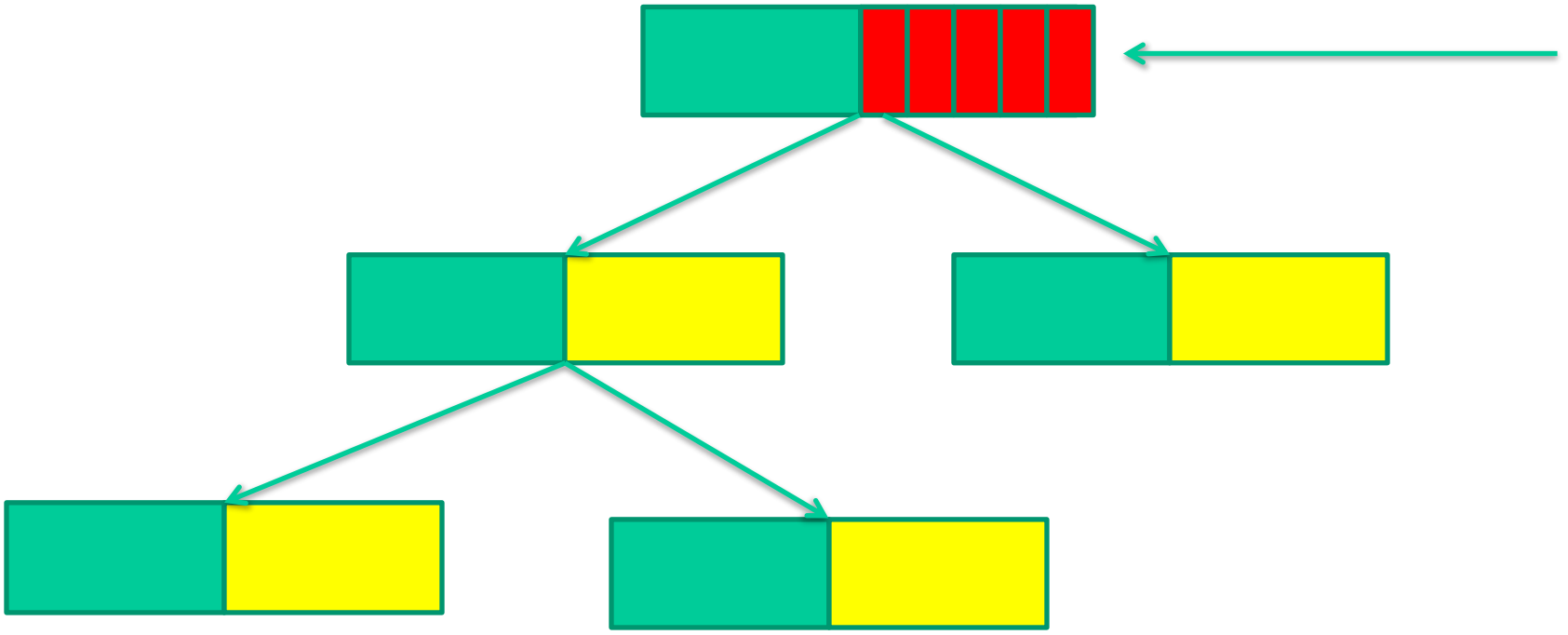
# $B^{\varepsilon}$-Tree Compaction

- Each internal node has a persistent buffer
- Buffers "log" multiple inserts and use one I/O to device



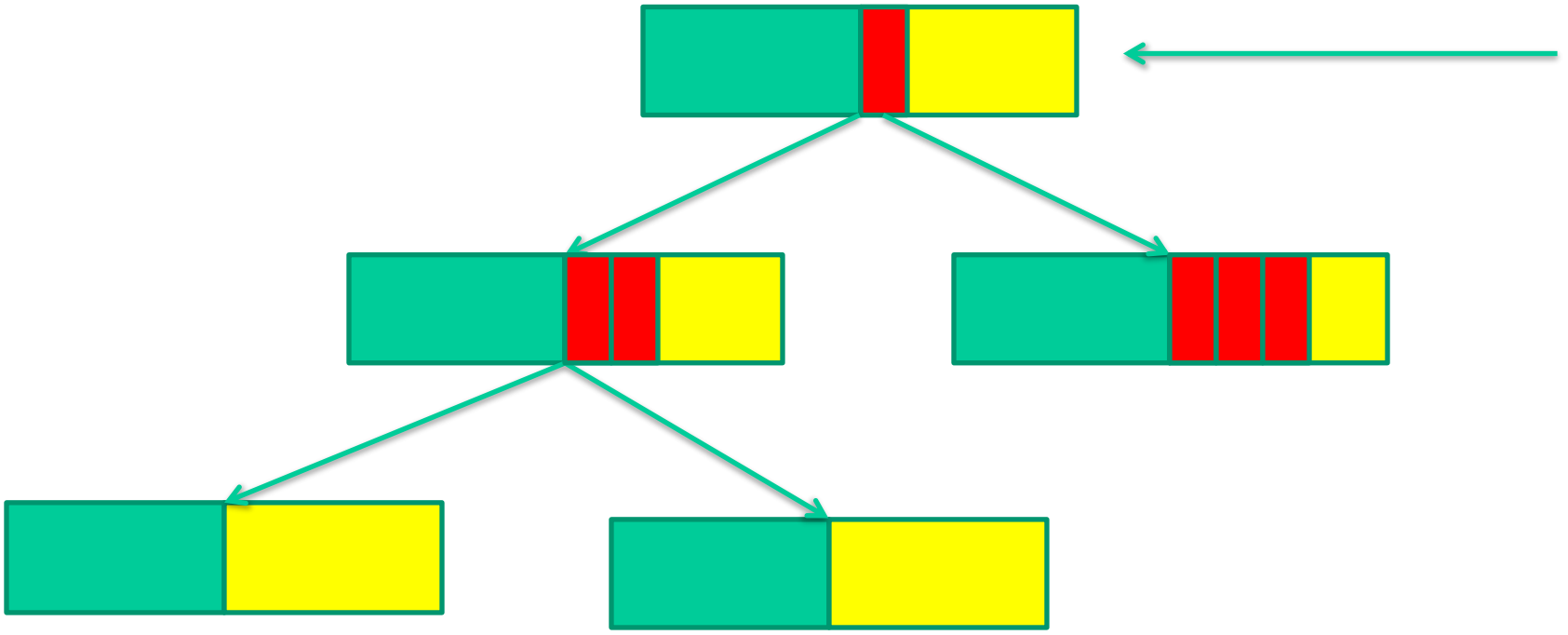
# $B^{\varepsilon}$-Tree Compaction

- Each internal node has a persistent buffer
- Buffers "log" multiple inserts and use one I/O to device
- Each level holds xf data compared to the previous one



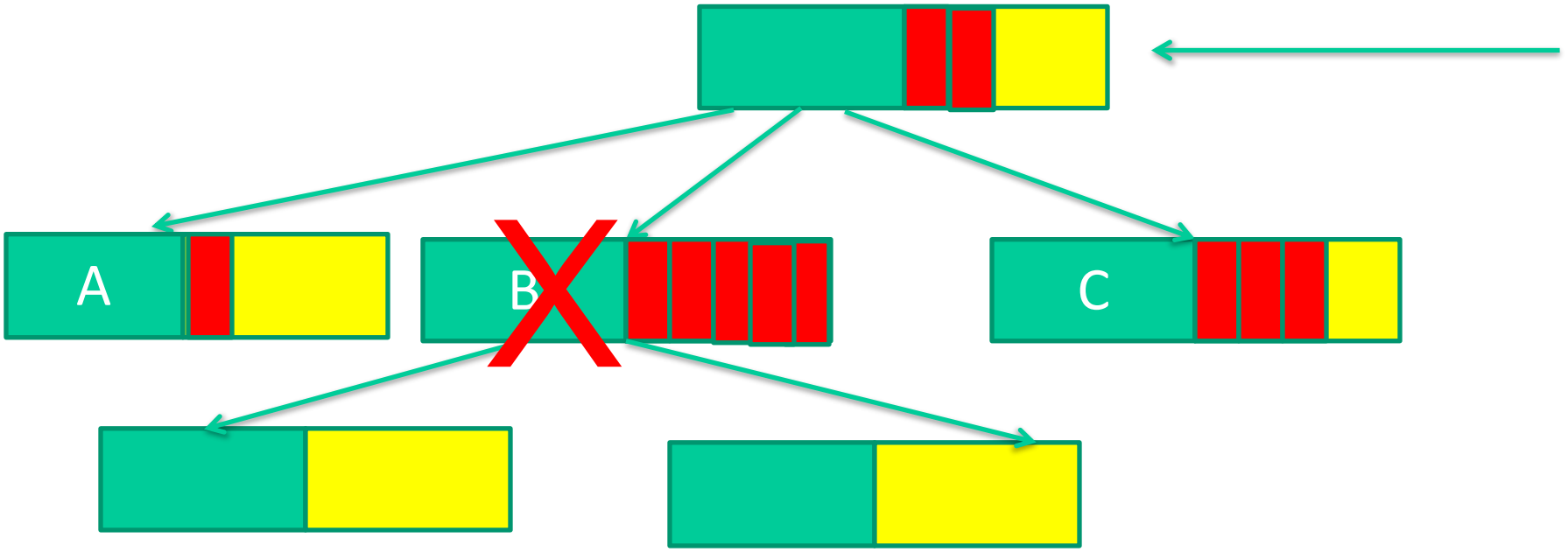
# $B^{\varepsilon}$-Tree Compaction

- Each internal node has a persistent buffer
- Buffers "log" multiple inserts and use one I/O to device



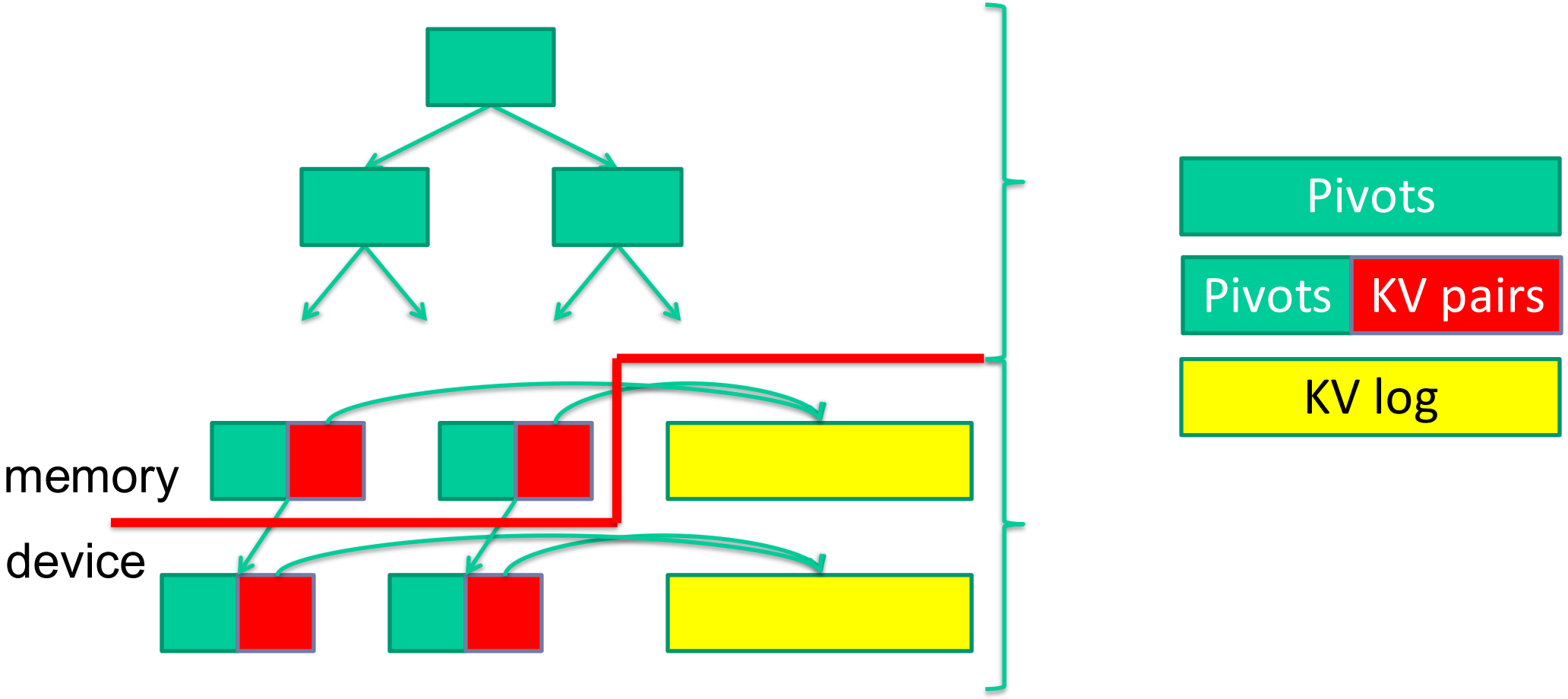
# $B^{\varepsilon}$-Tree Compaction

- We do not need to empty a buffer when it becomes full, but we need to maintain the f amortization factor
- Do we reorganize siblings after emptying a node (e.g. A B C)?




- B becomes empty after compaction
- Do we repartition the interval covered by A, missing B, C?
- Merge (missing B) with either A or C and let the intervals to be created anew (How does LSM behave in this case?)

# Tucana B$^{\varepsilon}$-Tree: Buffer only once in memory

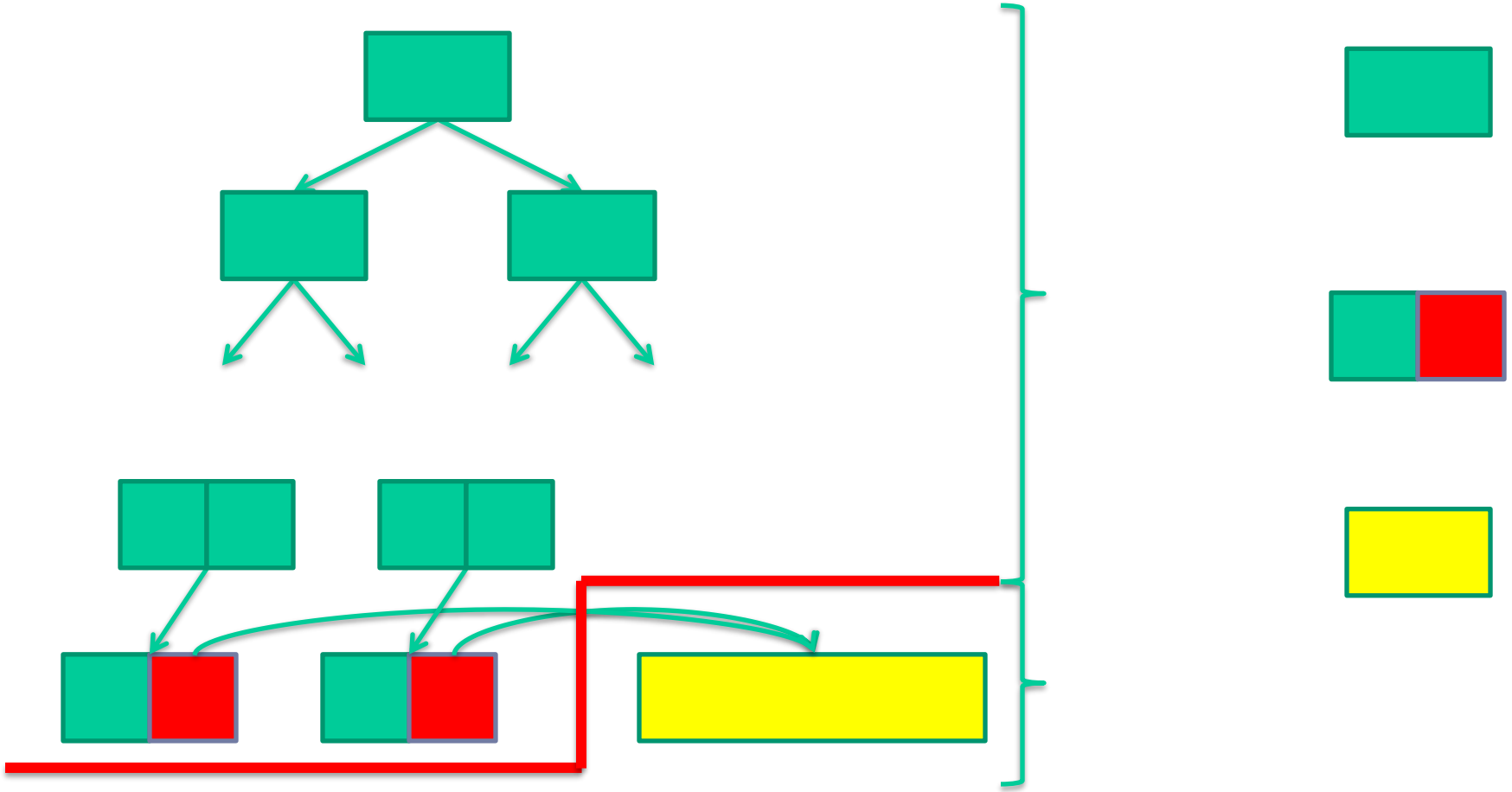


# Buffered Node Organization

- Searching buffered nodes requires accessing keys on device
- Tucana uses two optimizations for buffered nodes
- 1) Include key prefixes (fixed size)
  - Eliminates 65%-75% of I/Os for keys in all queries
- 2) Include hashes for full keys (fixed size)
  - Eliminates 98% of I/Os for keys in point queries

# Tucana B$^{\varepsilon}$-Tree: Assume all internal levels but BINs fit in memory

If we assume full index fits in memory (however, index still persistent)



# DRAM Caching – Device I/O

- Key-value stores use a user-space DRAM cache
  - Avoids system calls for hits - Explicit kernel I/O for misses
- However, hits incur overhead in user-space
  - **Both** index+data in **every** traversal – Not important for HDDs
- Alternative: DRAM caching via mmap

# DRAM caching via mmap: hits at no cost

- Use multiple regions/containers per device
  - Each region contains allocator + multiple indexes
- mmap each region directly to memory
  - Same layout of metadata + data on device and in memory
- Hits via mapped virtual addresses incur no overhead
- Misses do not require de/serialization of index
- mmap introduces new challenges

# mmap: Misses = page faults, fetches, evictions

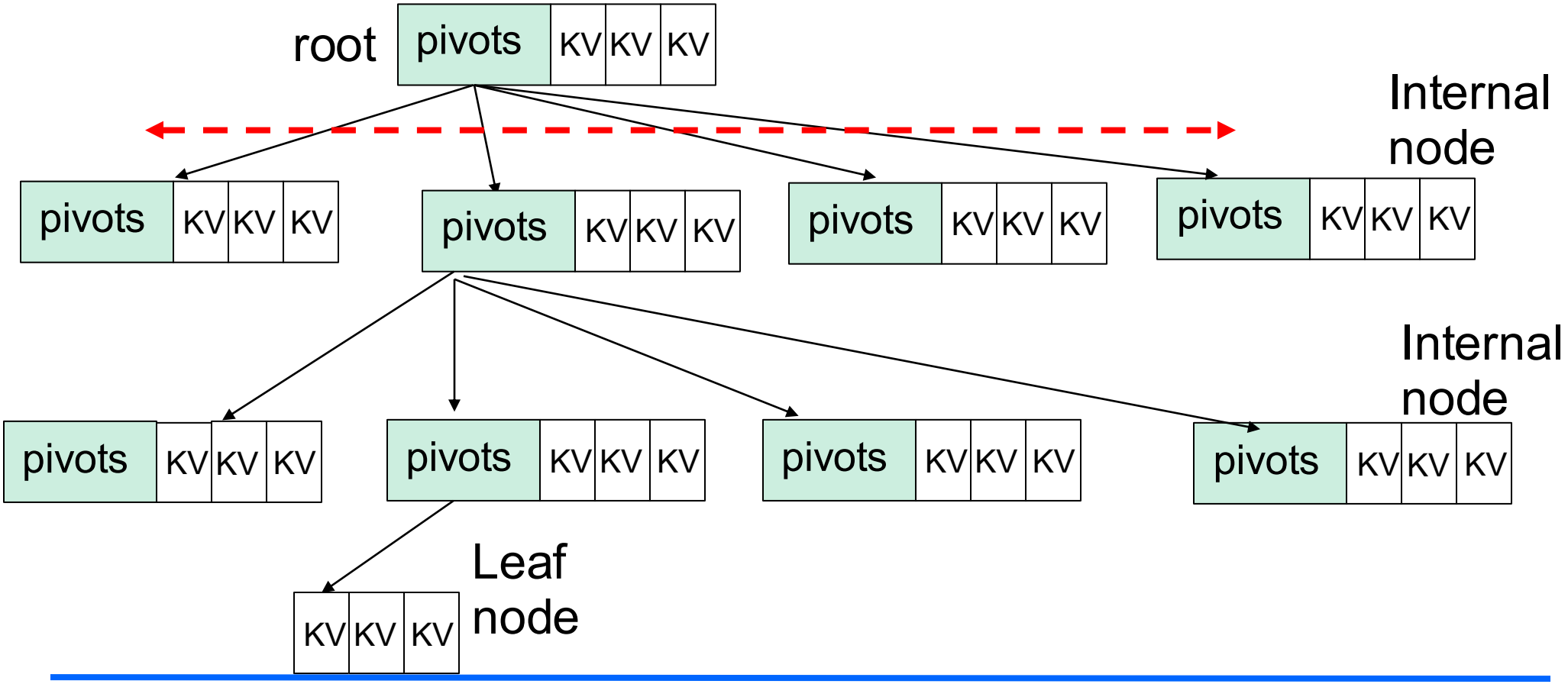
- (1) We can improve inserts
  - Inserts with mmio require a read-before-write I/O
  - We insert only on newly allocated pages
  - We detect and eliminate fetches to newly allocated pages
  - Requires a kernel module with access to allocator metadata
- (2) Still, no control over size, timing of I/Os + evictions
  - We use mmap hints to disable prefetching
  - Should examine these in detail in future work

# Persistence and Recovery

- Typical for HDDs: Write-Ahead-Logging (WAL)
  - Sequential I/O and ability to batch I/Os – both good
  - However, double writes – first to log, then in-place
  - Incurs overhead for log management during recovery
- Alternative: Copy-On-Write (CoW)
  - Instantaneous recovery and amenable to versioning
  - Write-anywhere approach and modify pointers atomically
  - Single write, however, more random I/O

# Discuss

- Should $B^{\varepsilon}$ buffers in internal nodes be small or large?
- Can buffers be shared in each level for better memory use?
- What are the differences between $B^{\varepsilon}$ and LSM?



# Reading

- An Introduction to Bε-trees and Write-Optimization. M. A. Bender et al. ;login: issue: October 2015. [https://www.usenix.org/publications/login/oct15/bender]
- A. Papagiannis et al. 2016. Tucana: design and implementation of a fast and efficient scale-up key-value store. USENIX ATC ‘16. [https://www.usenix.org/system/files/conference/atc16/atc16_paper-papagiannis.pdf]
- A. Conway. 2020. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. USENIX ATC'20. [https://dl.acm.org/doi/pdf/10.5555/3489146.3489150]
- BetrFS FAQ, https://www.betrfs.org/faq.html