CSE 350: Advanced Data Structures and Indexes (Spring 2026)

Lecture 17: Log-Structured Merge Tree

3/31/2026 & 4/2/2026

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Write-heavy workload

- Consider a possible instant messaging app
  - Central database serving as the message store
    - Keyed by (session_id, timestamp)
  - Lots of active sessions (e.g., > 100K messages/second)
    - High volume of writes & requires fast insertion
  - Recent messages needs to be synced to client side
    - Lots of range search on recent insertions based on keys
      - E.g., session_id = 100 AND timestamp >= 1000
    - Also need to support efficient (but less often) history searches
      - E.g., session_id = 100 AND timestamp BETWEEN 10 AND 50

Does (external memory) B+-Tree work?

- $O(\log N)$ lookup and insertion
  - Also need to deal with concurrency (which is not easy to be correct and efficient)

- Lookup is usually ok for range searches
  - Followed by leaf level scans anyway (often more I/Os in leaf level scans)

- However, it's hard, if not impossible, to make B-Tree insertion to keep up
  - Each insertion is at least one leaf read + one leaf write
    - If split happens, up to roughly 3h I/Os.

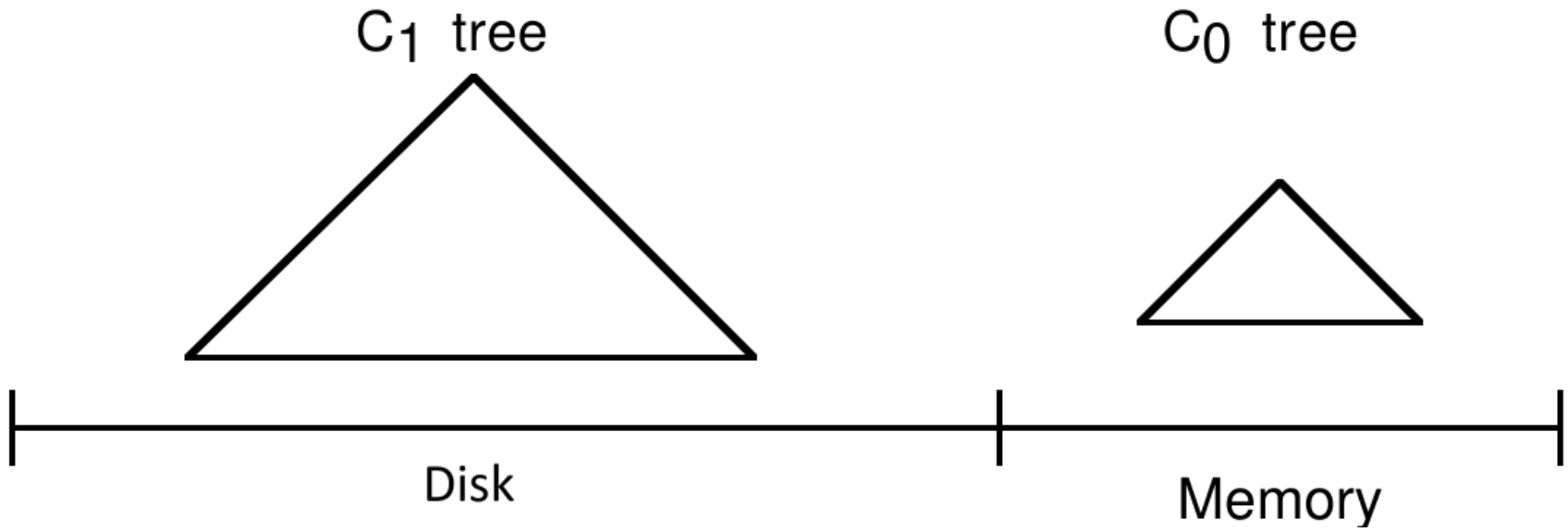
How do we optimize for write?

Want: < 1 I/Os per insertion

- Pure in-memory index?
  - Possible option, but costly, less scalable, long recovery time
- Hybrid EM + in-memory index
  - Writes into bounded in-memory index
    - Persistency can be provided by a Write-ahead Log (sequential log flushed for a batch of writes)
  - Merges in-memory index into on-disk index over time (merge + bulk loading)
    - Write I/O is delayed and amortized
    - Avoids random I/Os for regular tree insertion

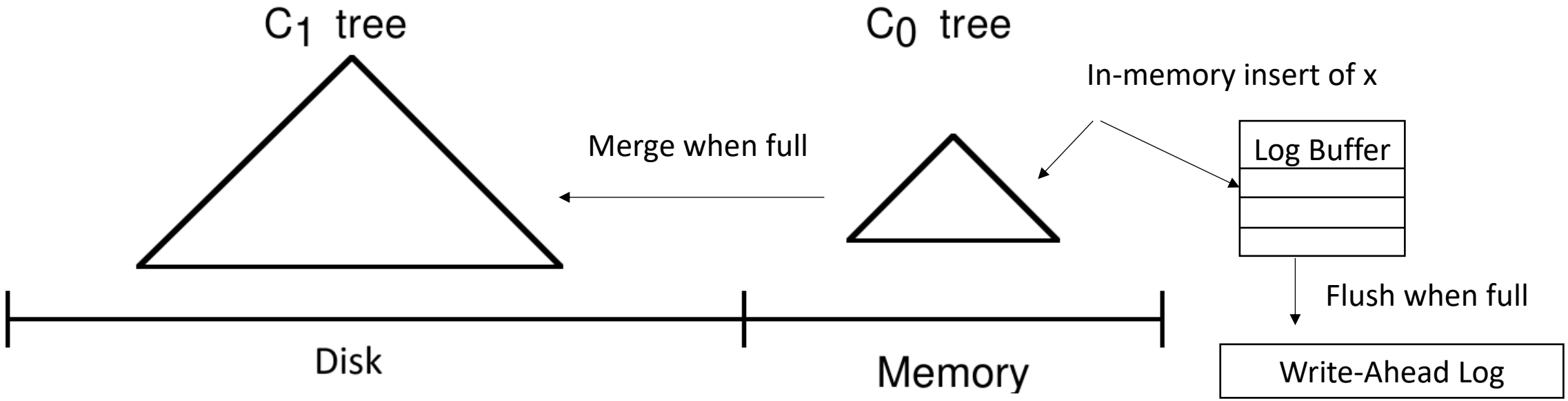
A Two-Component Log-Structured Merge Tree

- O'Neil et al., The log-structured merge-tree (LSM-tree), In Acta Informatica, 1996.
  - C1 tree is fully packed (100% fill in almost all nodes)
  - Uses batching for merging process (which they call multi-page write block)



Insertion

- Amortized 1/B I/O in most cases



Rolling Merge

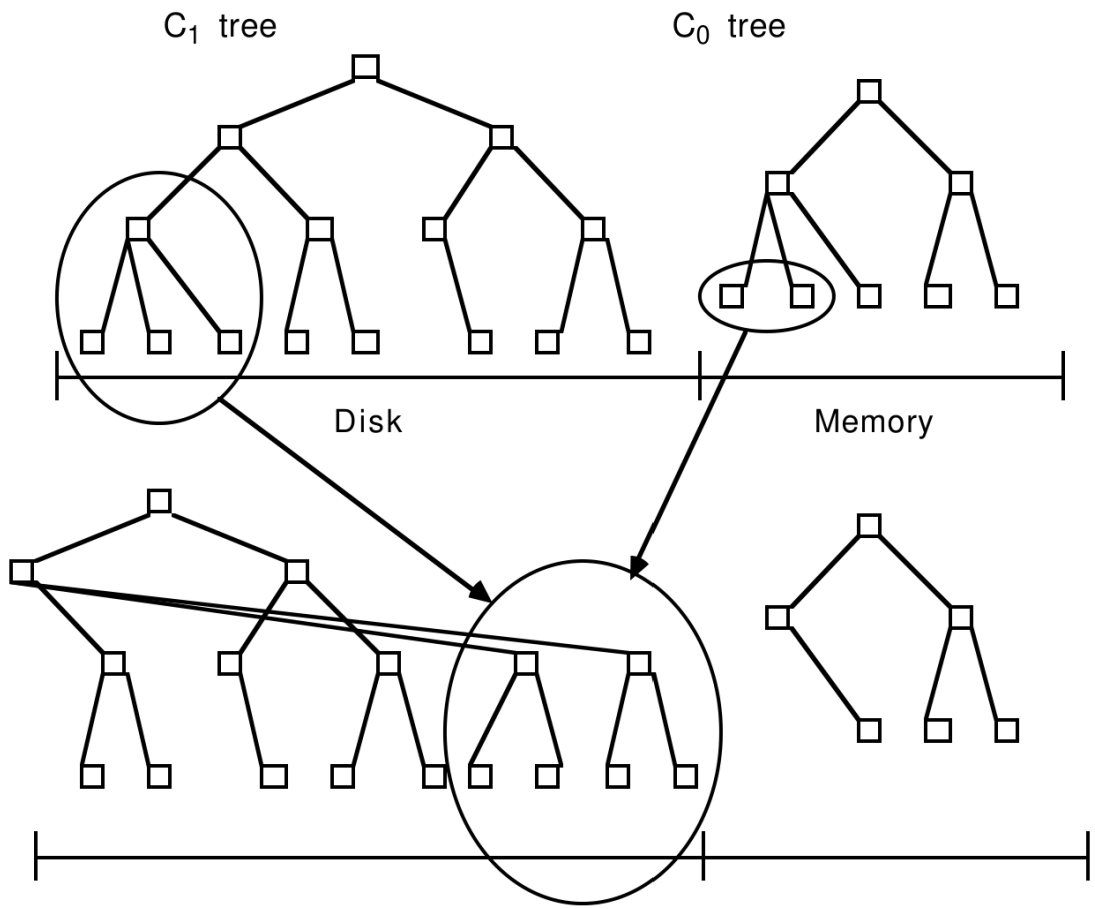
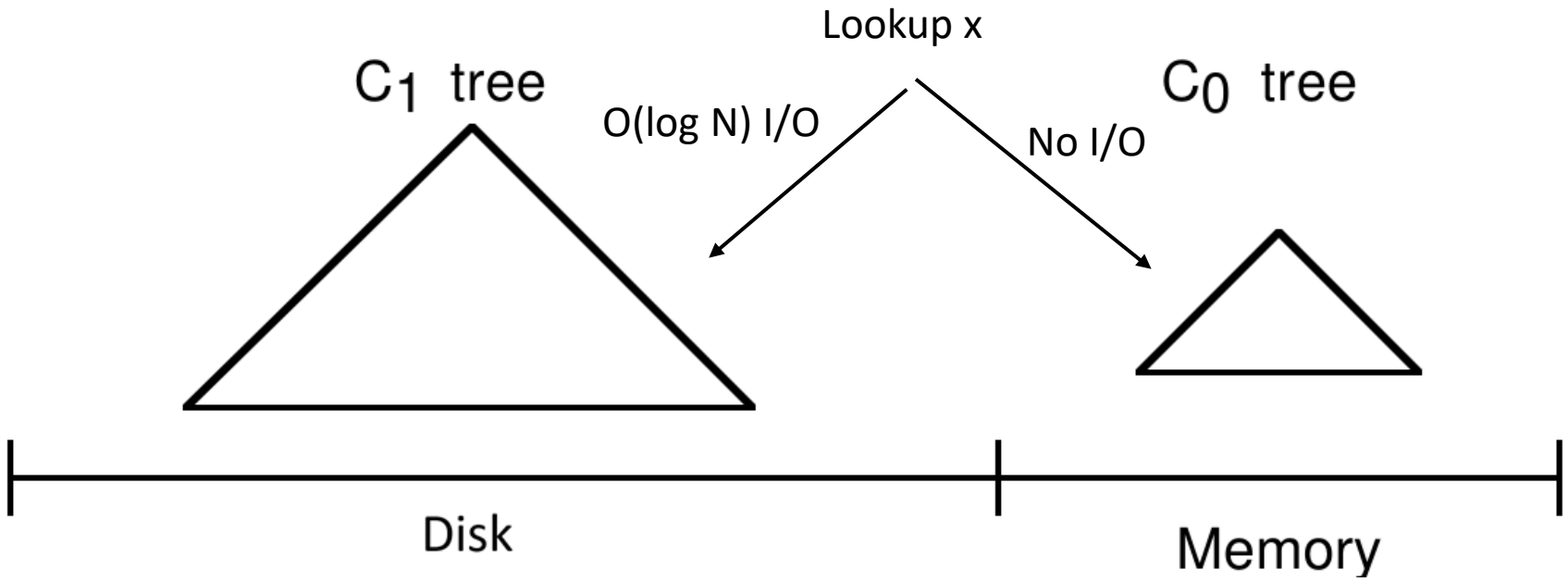


Figure 2.2. Conceptual picture of rolling merge steps, with result written back to disk

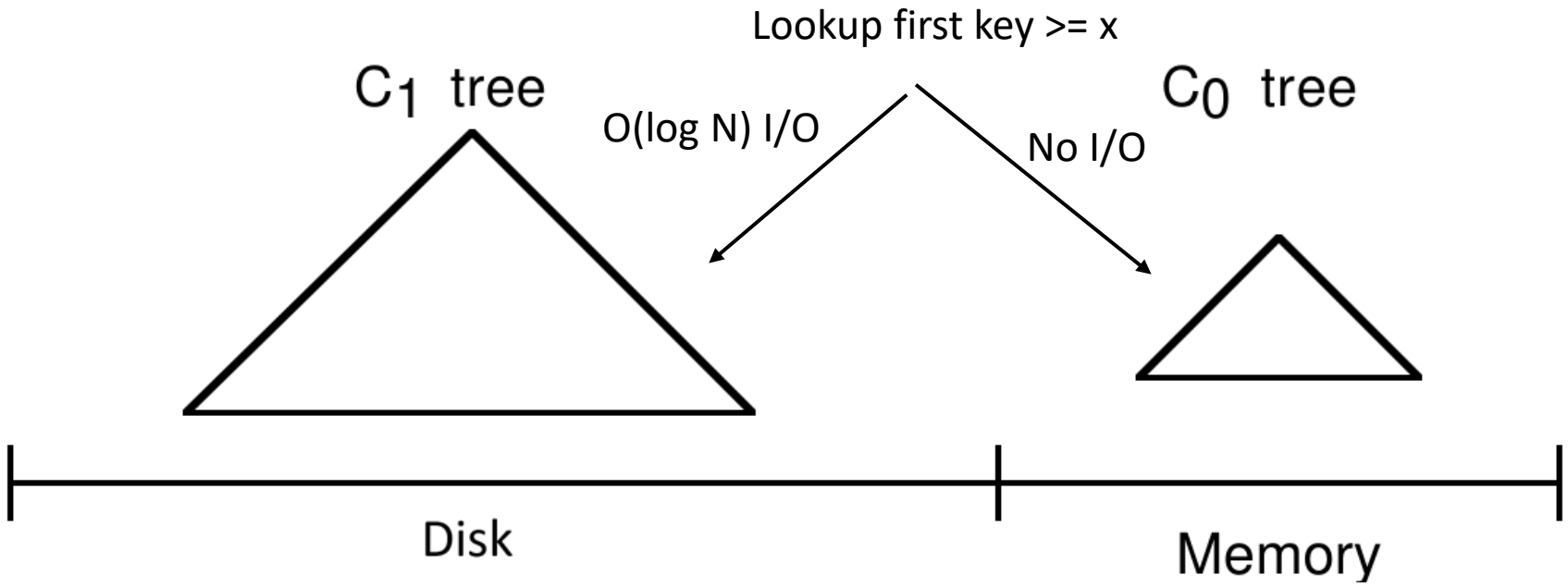
Lookup

- Search for key = x



Range search

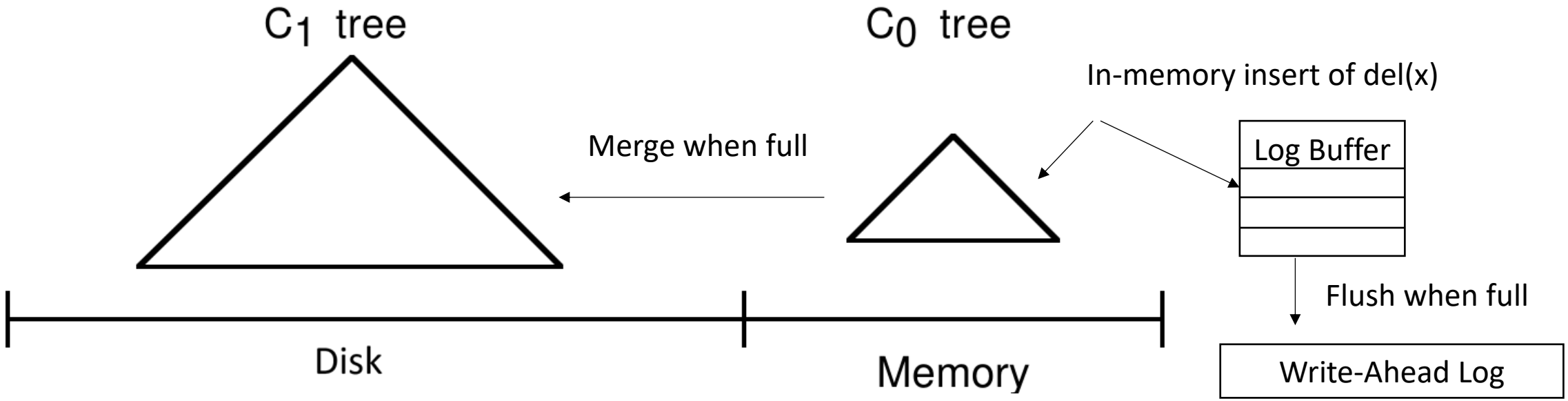
- Search for key in [x, y]



Scan + merge: O(n/B) I/Os – assuming there are n matches and each page stores B records

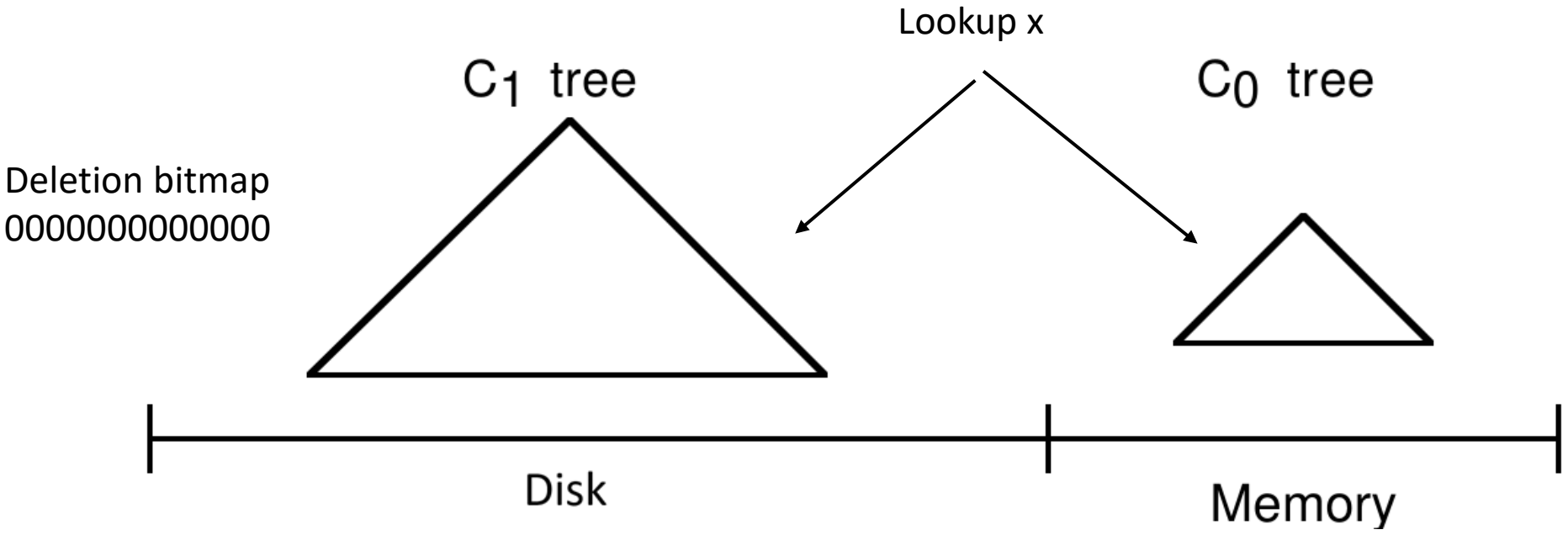
Deletion

- How to handle deletion efficiently?
  - Insert a negative record (aka tombstone)
- Delete x
- Complicates lookups



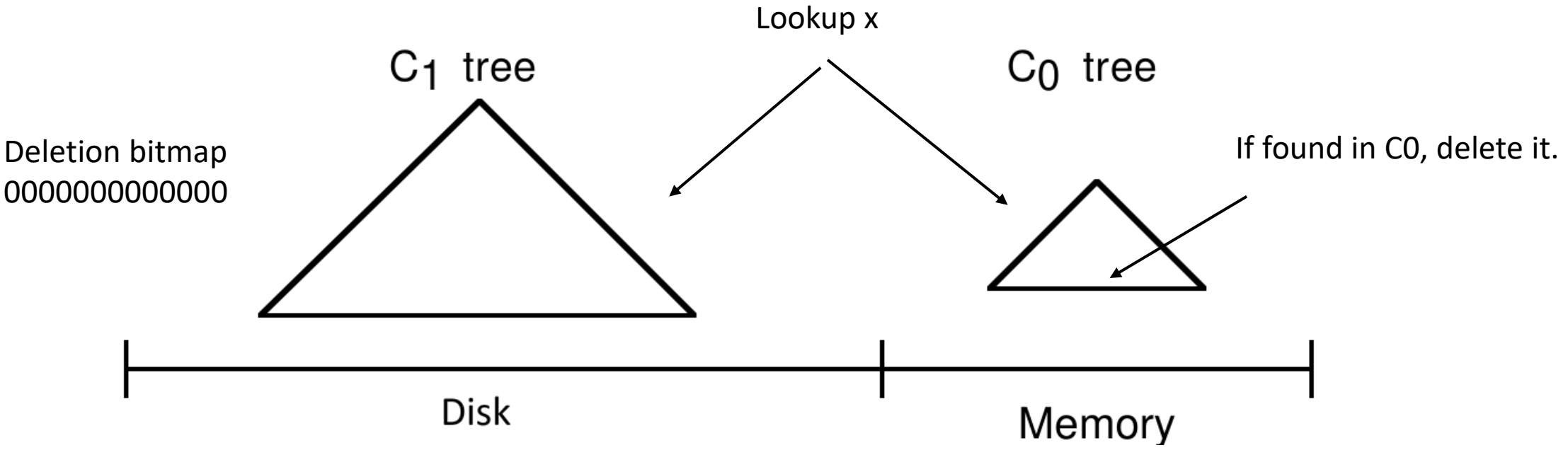
Alternative way of deletion: tagging

- Maintain a deletion bit for each record in C1
- Delete x
- Lookup ignores anything marked for deletion



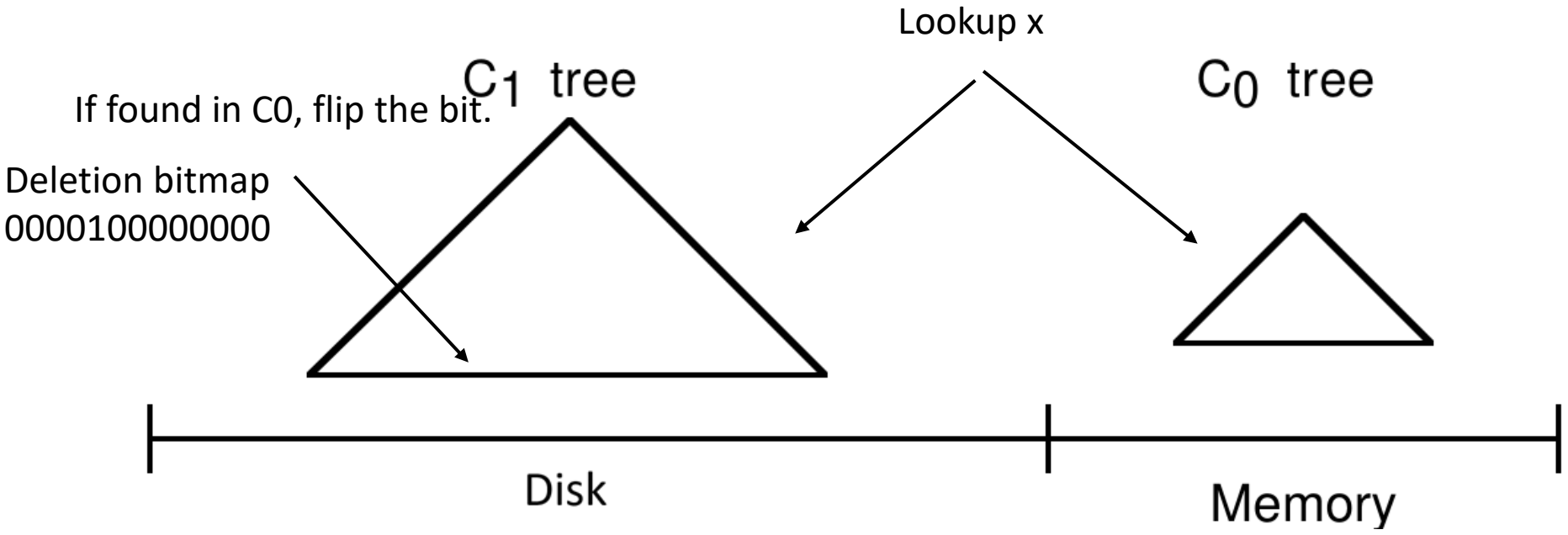
Alternative way of deletion: tagging

- Maintain a deletion bit for each record in C1
- Delete x
- Lookup ignores anything marked for deletion



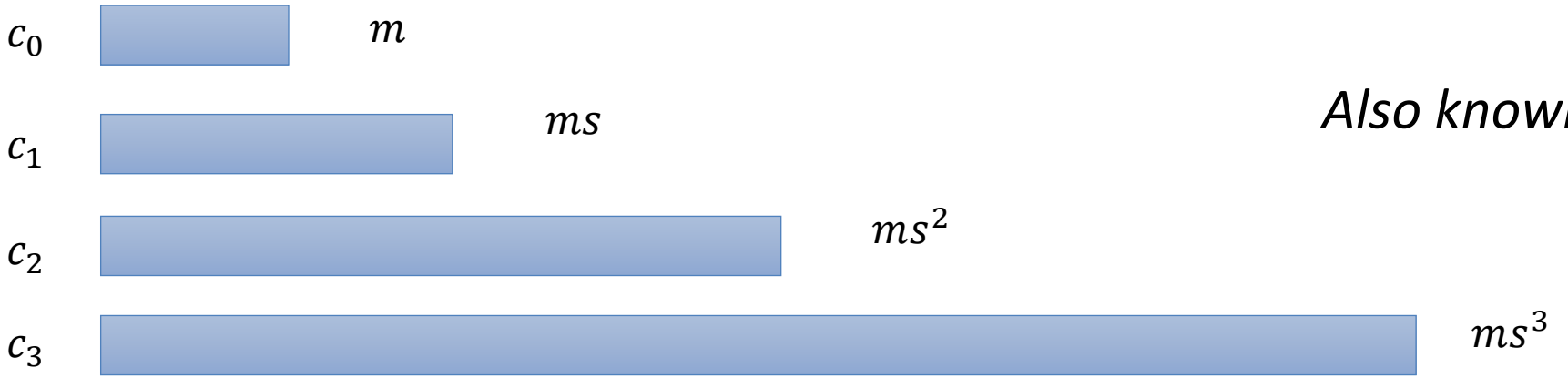
Alternative way of deletion: tagging

- Maintain a deletion bit for each record in C1
- Delete x
- Lookup ignores anything marked for deletion



Multi-Component Log-Structured Merge Tree

- Levels $c_0, c_1, \ldots, c_k$
  - For some size ratio s, $\forall i \geq 1, \frac{|c_i|}{|c_{i-1}|} = s$



Also known as leveling

How many levels?

$$m + ms + \cdots + ms^k = \frac{s^{k+1} - 1}{s - 1} m = N \quad \Rightarrow \quad k + 1 = \left\lceil \log_s \left(\frac{N(s-1)}{m} + 1 \right) \right\rceil$$

Alternative multi-component log structured merge tree

- Leveling requires rolling merge to immediate execute once a level is filled up
  - Can cause blocking
- How about allowing up to T of c_i trees?

Also known as tiering

How many levels?

$$m + Tm + \cdots + Tms^k = m + \frac{s^{k+1} - 1}{s - 1} mT = N \quad \Rightarrow \quad k + 1 = \left\lceil \log_s \left(\frac{(N - m)(s - 1)}{mT} + 1 \right) \right\rceil$$

Lookup becomes expensive

- Need to search all levels

- Can we skip some trees?
  - Maintain membership testing structures
    - Hash set is too heavy-weight
    - Approximate hashing-based sketches
      - e.g., Less hashing, same performance: Building a better Bloom filter | Random Structures & Algorithms
    - What about ranges? Need range filters:
      - E.g., Diva: Dynamic Range Filter for Var-Length Keys and Queries | Proceedings of the VLDB Endowment

Modern LSM

- Examples include RocksDB & LevelDB

- C0 – memtable (lock-free skiplists)
- C1, C2 … -- SStable (Sorted String tables)
  - Small sorted string arrays, segmented into fixed size
  - Fetched from disk for binary search
  - Maintain ranges over each SSTable
    - Can also build B-tree over the ranges

- Not the only design – depends on optimization target
  - SSD wearing levels?
  - Write amplification?
  - Range lookup efficiency?
  - ….