

# CS 7270: Advanced Database Systems Fall 2025

## Lecture 12

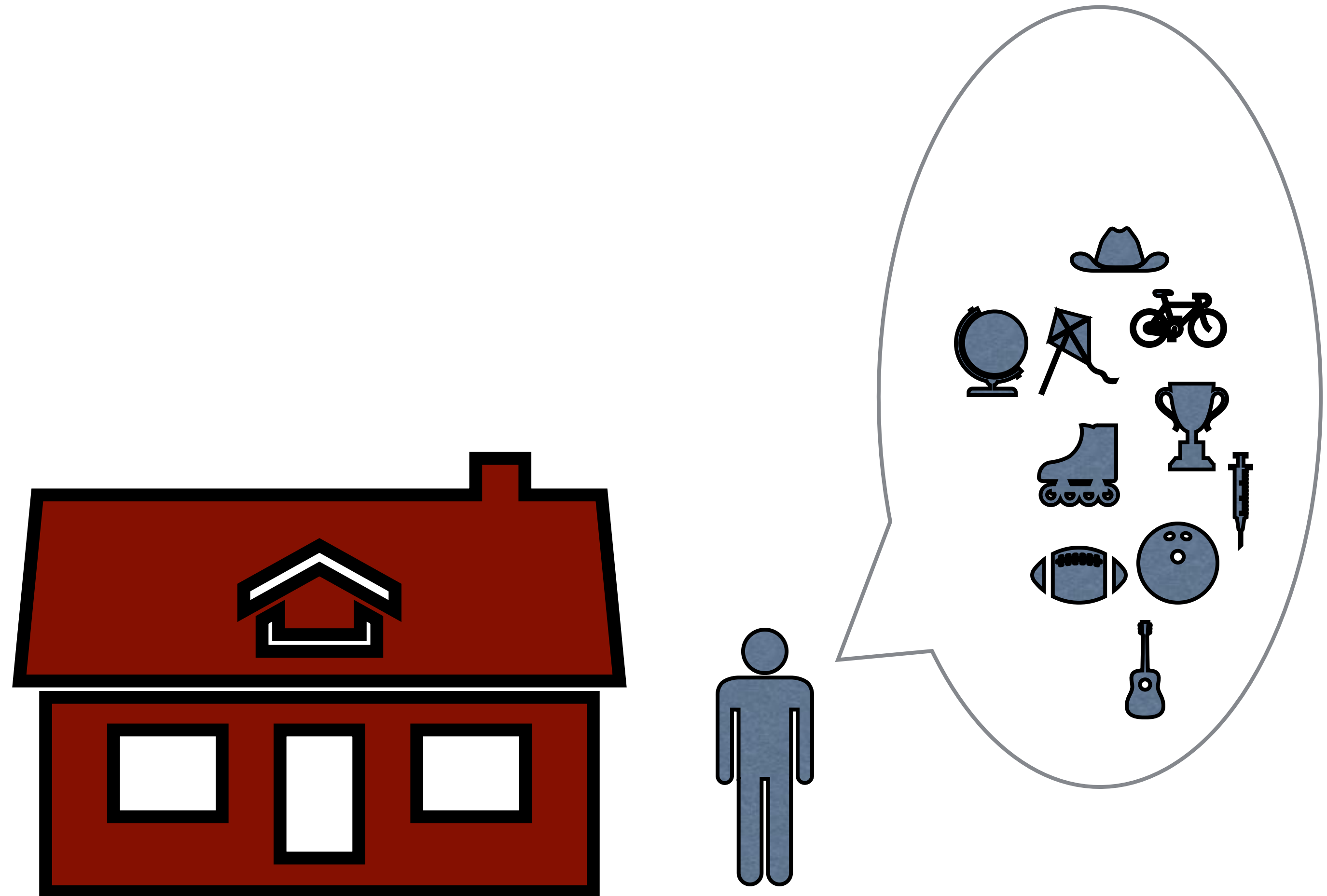
# Log-Structured Merge (LSM) Trees

Prashant Pandey

[p.pandey@northeastern.edu](mailto:p.pandey@northeastern.edu)

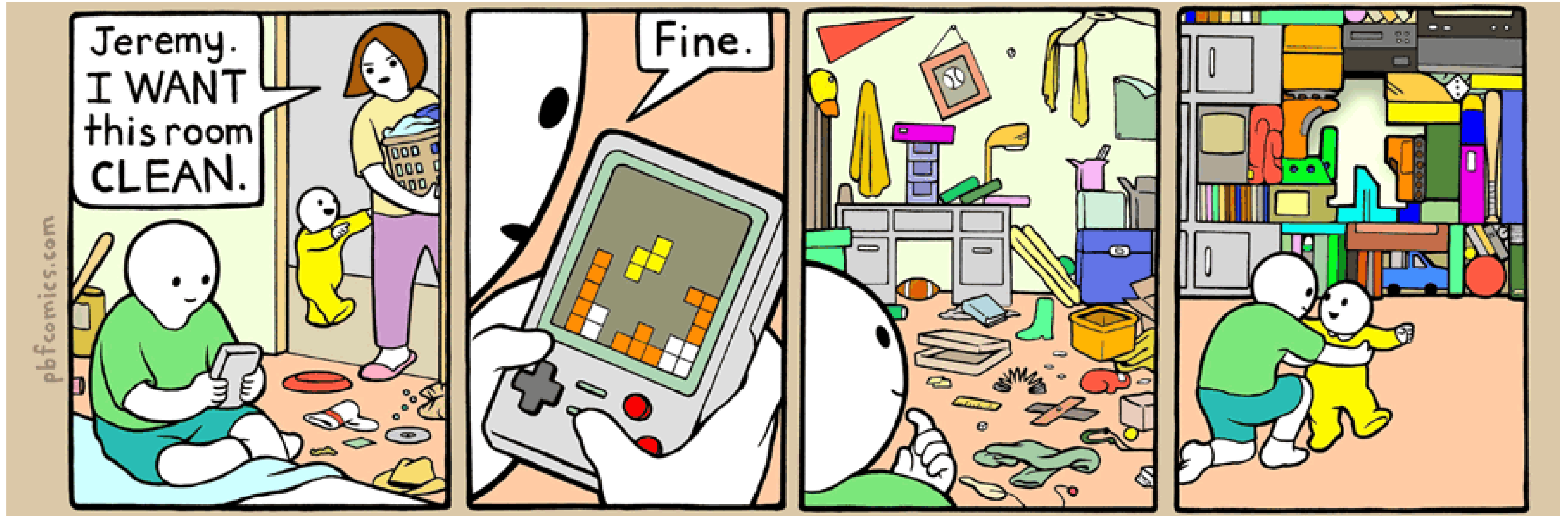
Acknowledgement: Slides taken from Prof. Willam Janen, Williams

# How Should I Organize My Stuff (Data)?



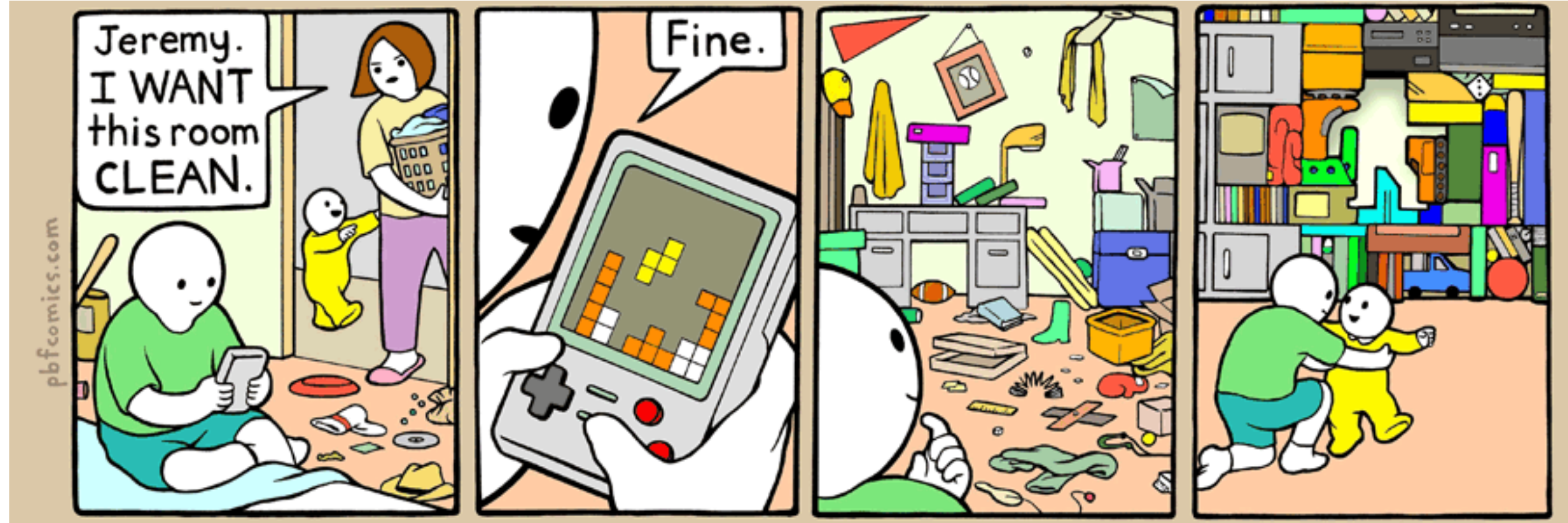
# How Should I Organize My Data?

**Different people approach the problem differently...**



<https://pbfcomics.com/comics/game-boy/>

# How Should I Organize My Data?



"Logging"

"Indexing"

# How Should I Organize My Data?

Logging

Indexing

Inserting

Append at  
end of log

Insert at leaf  
(traverse root-  
to-leaf path)

Searching

Scan through  
entire log

Locate in leaf  
(traverse root-  
to-leaf path)

# How Should I Organize My Data?



# Are We Forced to Choose?

**It appears we have a tradeoff between insertion and searching**

- **B-trees have**
  - ▶ fast searches:  $O(\log_B N)$  is the optimal search cost
  - ▶ slow inserts
- **Logging has**
  - ▶ fast insertions
  - ▶ slow searches: cannot get worse than exhaustive scan

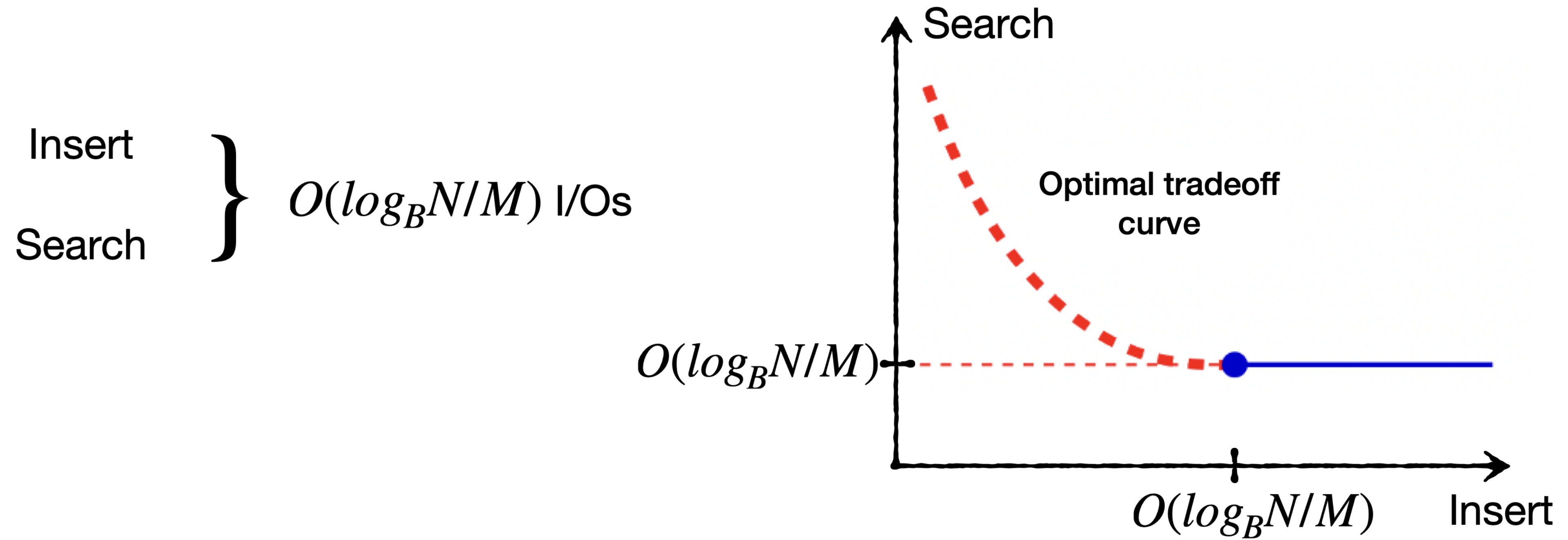
# Goal: Data Structural Search for Optimality

**B-tree searches are optimal**

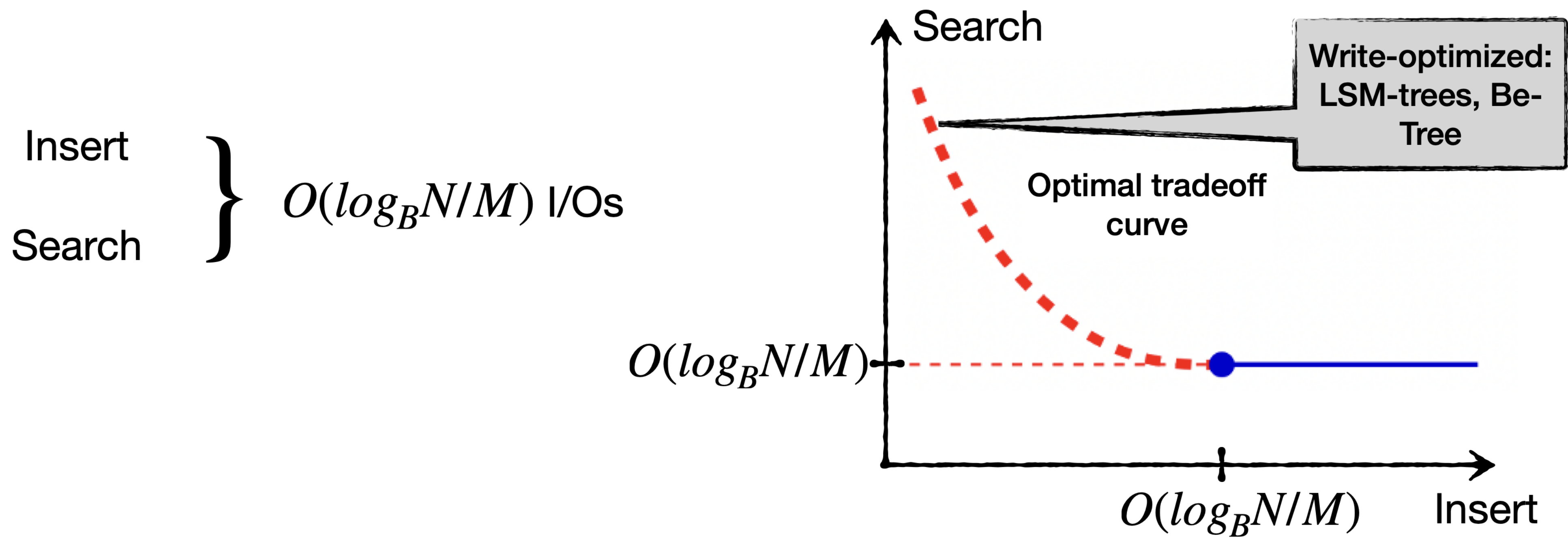
**B-tree updates are not**

- We want a data structure with inserts that beat B-tree inserts without sacrificing on queries

> This is the promise of **write-optimization**



**B-trees are asymptotically optimal for point operations [BF03]**



**B-trees are asymptotically optimal for point operations [BF03]**

# Log-Structured Merge Trees

**Data structure proposed by O'Neil, Cheng, and Gawlick in 1996**

- Uses write-optimized techniques to significantly speed up inserts

**Hundreds of papers on LSM-trees (innovating and using)**

**To get some intuition for the data structure, let's break it down**

Log-structured • Merge • Tree

# Log-Structured Merge Trees

## Log-structured

- All data is written sequentially, regardless of logical ordering

Merge • Tree

# Log-Structured Merge Trees

## Log-structured

- All data is written sequentially, regardless of logical ordering

## Merge

- As data evolves, sequentially written runs of key-value pairs are merged
  - ▶ Runs of data are indexed for efficient lookup
  - ▶ Merges happen only after much new data is accumulated

Tree

# Log-Structured Merge Trees

## Log-structured

- All data is written sequentially, regardless of logical ordering

## Merge

- As data evolves, sequentially written runs of key-value pairs are merged
  - ▶ Runs of data are indexed for efficient lookup
  - ▶ Merges happen only after much new data is accumulated

## Tree

- The hierarchy of key-value pair runs form a tree
  - ▶ Searches start at the root, progress downwards

# Log-Structured Merge Trees

**Start with [O'Neil 96], then describe LevelDB**

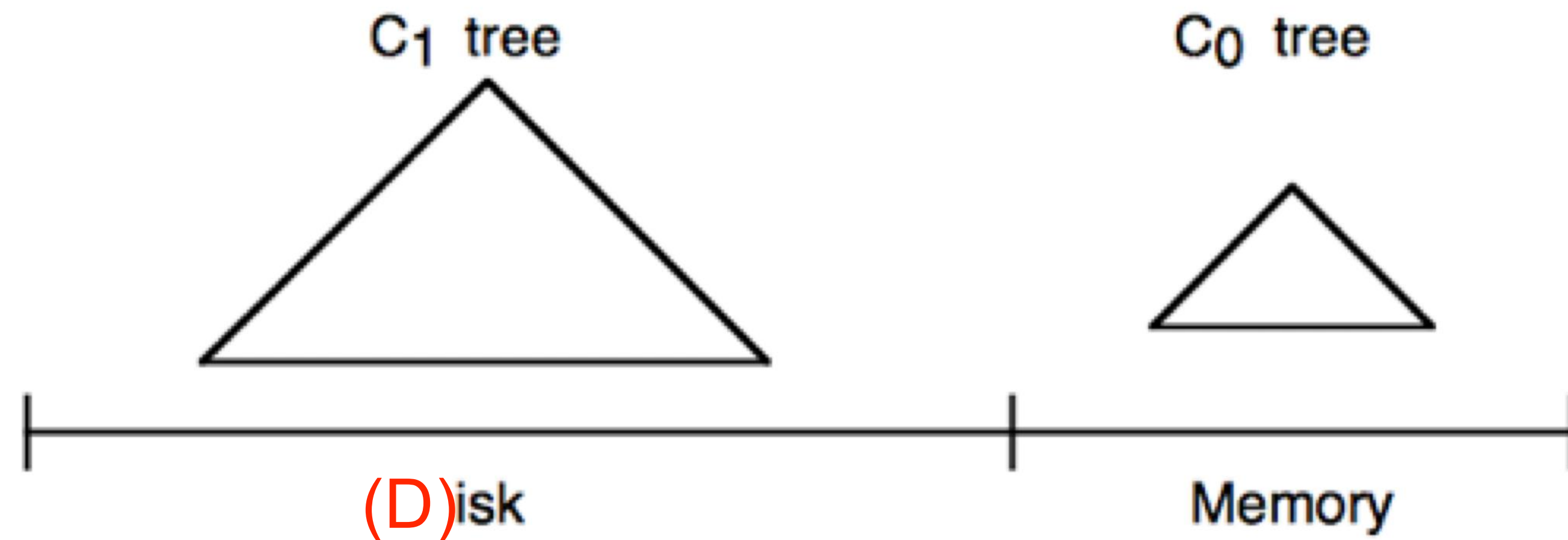
**We will discuss:**

- **Compaction** strategies
- Notable “tweaks” to the data structure
- Commonly cited drawbacks
- Potential applications

[O'Neil, Cheng, Gawlick '96]

**An LSM-tree comprises a hierarchy of trees of increasing size**

- All data inserted into in-memory tree ( $C_0$ )
- Larger on disk trees ( $C_{i>0}$ ) hold data that does not fit into memory



**Figure 2.1.** Schematic picture of an LSM-tree of two components

[O'Neil, Cheng, Gawlick '96]

When a tree exceeds its size limit, its data is **merged** and rewritten

- Higher level is always merged into next lower level ( $C_i$  merged with  $C_{i+1}$ )
  - Merging always proceeds top down

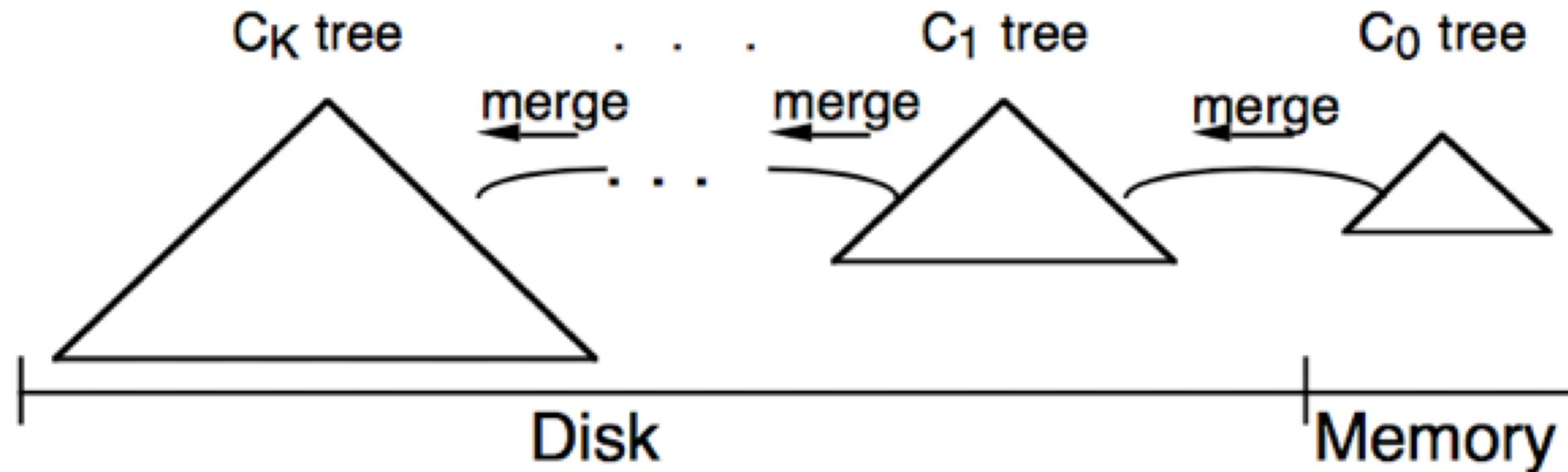


Figure 3.1. An LSM-tree of  $K+1$  components

# [O'Neil, Cheng, Gawlick '96]

- Recall mergesort from data structures/algorithms
  - We can efficiently merge two sorted structures in linear time using iterators
- When merging two levels, newer key-value pair versions replace older (GC)
  - LSM-tree **invariant**: newest version of any key-value pair is version nearest to top of LSM-tree

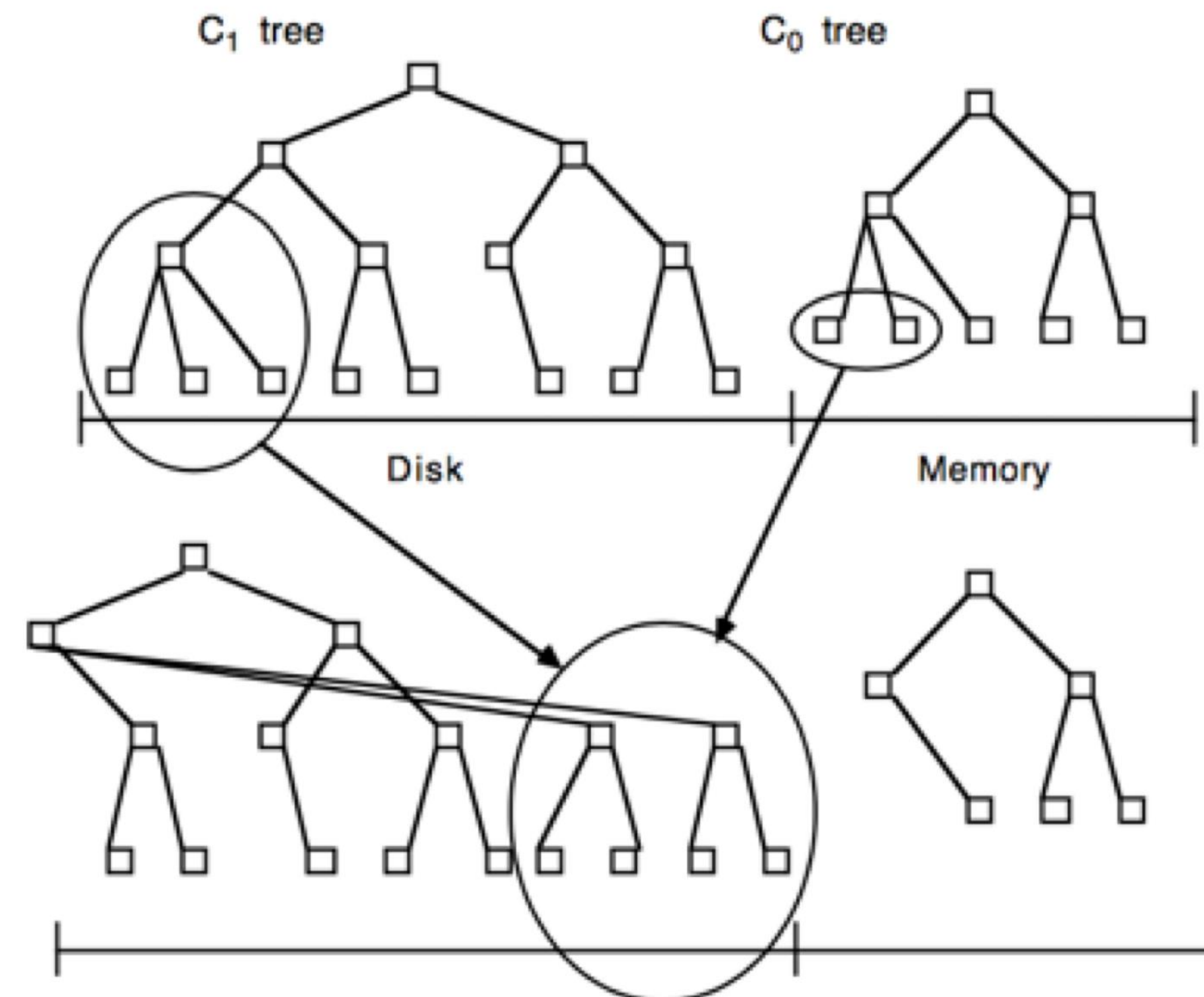


Figure 2.2. Conceptual picture of rolling merge steps, with result written back to disk

# LSM-trees implement the dictionary interface

## Maintain a set of key-value pairs (kv pairs)

- Support the following operations (at minimum):
  - ▶  $\text{insert}(k, v)$  - insert a new kv pair, (possibly) replacing old value
  - ▶  $\text{delete}(k)$  - remove all values associated with key  $k$
  - ▶  $(k, v) = \text{query}(k)$  - return latest value  $v$  associated with key  $k$
  - ▶  $\{(k_1, v_1), (k_2, v_2), \dots, (k_j, v_j)\} = \text{query}(k_i, k_l)$  - return all key-value pairs in the range from  $k_i$  to  $k_l$

> **Question:** How do we implement each of these operations?

# Insert(k)

## **We insert the key-value pair into the in-memory level, $C_0$**

- Don't care about lower levels, as long as newest version is one closest to top
- But if an old version of kv-pair exists in the top level, we must replace it
- If inserting into  $C_0$  causes  $C_0$  to exceed its size limit, compact (merge)

> Inserts are fast! Only touch  $C_0$  in common case.

# Delete( $k$ )

We insert a **tombstone** into the in-memory level,  $C_0$

- A tombstone is a “logical delete” of all key-value pairs with key  $k$ 
  - ▶ When we merge a tombstone with a key-value pair, we delete the key-value pair
  - ▶ When we merge a tombstone with a tombstone, just keep one copy
  - ▶ When can we delete a tombstone?
    - ▶ At the lowest level
    - ▶ When merging a *newer* key-value pair with key  $k$

> Deletes are fast! Only touch  $C_0$ .

# Query( $k$ )

## Begin our search in the in-memory level, $C_0$

- Continue until:
  - ▶ We find a key-value pair with key  $k$  (return that value)
  - ▶ We find a tombstone with key  $k$  (return “not found”)
  - ▶ We reach the lowest level and fail-to-find (return “not found”)

> Searches traverse (worst case) every level in the LSM-tree

# Query( $k_j, k_l$ )

## We must search *every* level, $C_0 \dots C_n$

- Return all keys in range, taking care to:
  - ▶ Return newest  $(k_i, v_i)$  where  $k_j < k_i < k_l$  such that there are no tombstones with key  $k_i$  that are newer than  $(k_i, v_i)$
  - ▶ Common strategy is to create an iterator for each level and use merge-esque logic

> Range queries must scan every level in the LSM-tree (although not all ranges in every level)

# LevelDB

Google's Open Source *LSM-tree-ish* KV-store

# Some Definitions

## LevelDB consists of a hierarchy of **SSTables**

- An SSTable is a sorted set of key-value pairs (Sorted Strings Table)
  - ▶ Typical SSTable size is 2MiB

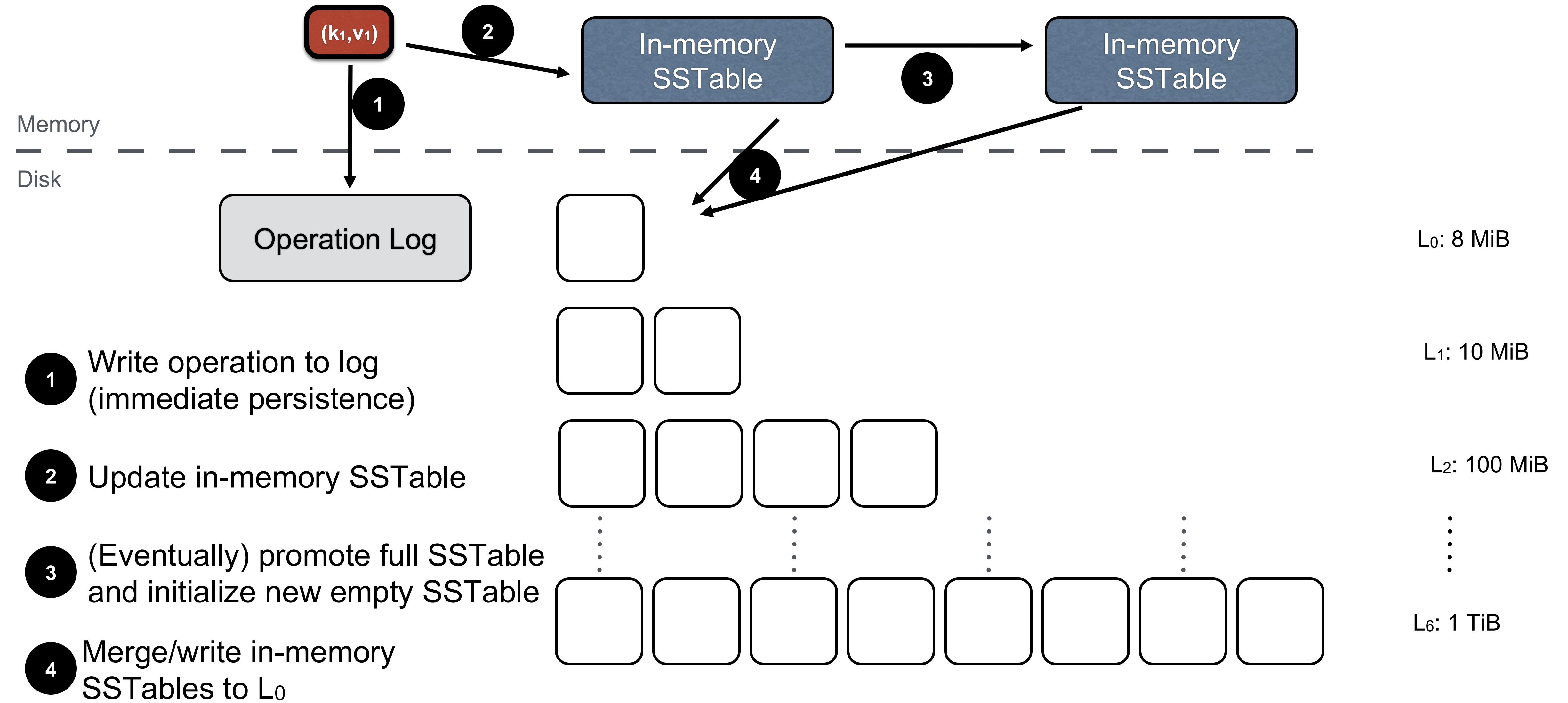
## The **growth factor** describes how the size of each level scales

- Let  $F$  be the growth factor (fanout)
- Let  $M$  be the size of the first level (e.g., 10MiB)
- Then the  $i^{\text{th}}$  level,  $C_i$  has size  $F^i M$

## The **spine** stores metadata about each level

- $\{\mathbf{key}_i, \mathbf{offset}_i\}$  for all SSTables in a level (plus other metadata TBD)
- Spine cached for fast searches of a given level
  - ▶ (if too big, a B-tree can be used to hold the spine for optimal searches)

# LevelDB Example



# Compaction

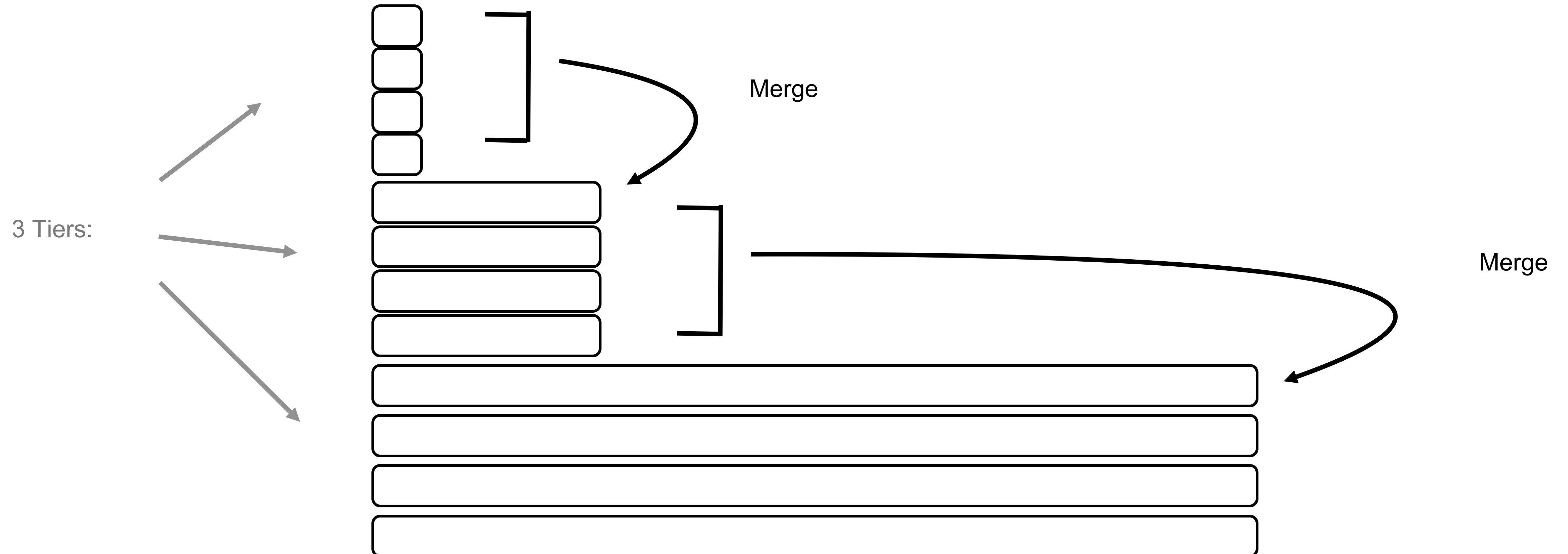
## How do we manage the levels of our LSM?

- **Ideal data management strategy would:**
  - ▶ Write all data sequentially for fast inserts
  - ▶ Keep all data sorted for fast searches
  - ▶ Minimize the number of levels we must search per query (low read amplification)
  - ▶ Minimize the number of times we write each key-value pair (low write amplification)
- **Good luck balancing so many competing interests in a single policy!**
  - ▶ ... but let's talk about some common approaches

# Compaction Strategies

## Option 1: Size-tiered

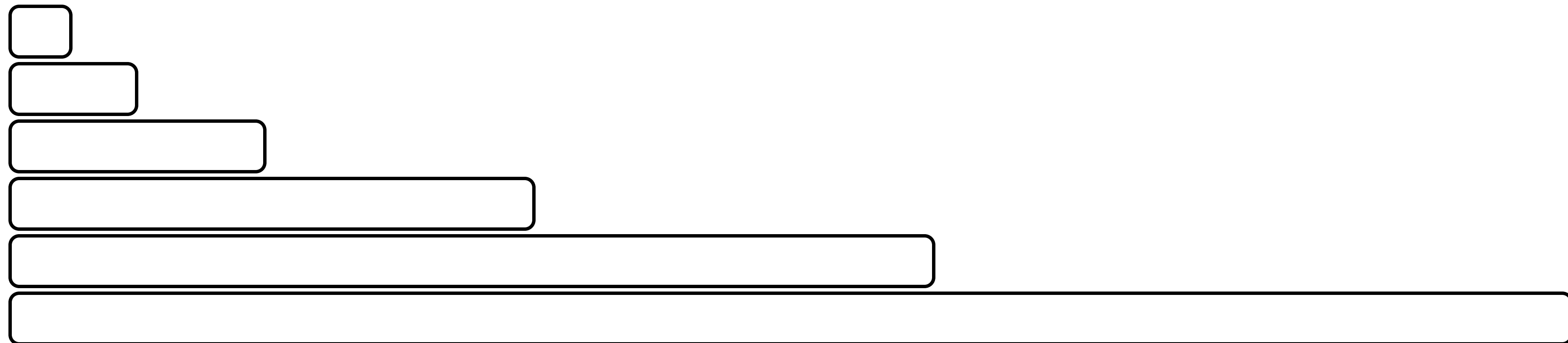
- Each “tier” is a collection SSTables with similar sizes
- When we compact, we merge some number of SSTables with the same size to create an SSTable in the next tier



# Compaction Strategies

## Option 2: Level-tiered

- All SSTables are fixed size
- Each level is a collection SSTables with **non-overlapping** key ranges
- To compact, pick SSTable(s) from  $L_i$  and merge them with SSTable(s) in  $L_{i+1}$ 
  - ▶ Rewrite merged SSTables into  $L_{i+1}$  (redistributing key ranges if necessary)
  - ▶ Possibly continue (cascading merge) of  $L_{i+1}$  to  $L_{i+2}$
  - ▶ Several ways to choose candidate SSTables for merge (e.g., round-robin or ChooseBest)
  - ▶ Possibly add invariants to our LSM to control merging (e.g., an SSTable at  $L_{i+1}$  can cover at most X SSTables at  $L_{i+1}$ )



(Note: This picture shows the aggregate size of individual levels, not the size of individual SSTables in a level.)

# LSM-tree Problems?

## We write a lot of data during compaction

- Not all data is new
  - ▶ We may rewrite a key-value pair to the same level multiple times
- How might we save extra writes?
  - ▶ VT-trees [Shetty FAST '13]: if a long run of kv-pairs would be rewritten unchanged to the next level, instead write a pointer
- Problems with VT-trees?
  - ▶ Fragmentation
    - ▶ Scanning a level might mean jumping up and down the tree, following pointers

> There is a tension between locality and rewriting

# LSM-tree Problems?

## We write a lot of data during compaction

- Not all data written during a compaction is new data at that level
    - ▶ We may rewrite a key-value pair to the same level multiple times
  - How might we save extra writes?
    - ▶ Fragmented LSM-Tree [Raju SOSP '17]: each level can contain up to  $F$  fragments
    - ▶ Fragments can be appended to a level without merging with SSTables in that level
      - ▶ Saves the work of doing a “merge” until there is enough work to justify the I/Os
  - Problems with fragments?
    - ▶ Fragments can have overlapping key ranges, so may need to search through multiple fragments
    - ▶ Need to be careful about returning newest values
- > Again, we see a tension between locality and rewriting

# LSM-tree Problems?

## We read a lot of data during searches

- We may need to search every level of our LSM-tree
  - ▶ Caching the spine & binary search both help (SSTables are sorted), but still many I/Os in worst case
- How might we save extra reads?
  - ▶ Bloom filters!
  - ▶ By adding a Bloom filter, we only search if the data exists in that level (or false positive)
  - ▶ Bloom filters for large data sets can fit into memory, so approximately  $1+e$  I/Os per query
- Problems with Bloom filters?
  - ▶ Do they help with range queries?
    - ▶ Not really...

# Thought Questions

## How might you design:

- an LSM-tree for an SSD?
- an LSM-tree for a HDD?
  - ▶ how would your designs be different?
    - ▶ Different concerns (e.g., wear leveling & endurance, parallelism, gap between sequential and random I/O)

## Should we store the data inside the index, or separating the data from the index (clustered vs. declustered index)

- How might you design a system that separates keys from values?
  - ▶ Wisckey [Lu FAST 16]: Store keys in LSM-tree, values in a log
- What are the advantages/disadvantages?
  - ▶ Can fit most of the LSM-tree (keys) in memory -> 1 I/O per search
  - ▶ Need to GC your value log, just like LFS

# Final Thoughts

## **LSM-trees are a write-optimized data structure:**

- Many updates are batched and committed in a sequential I/O

## **Although we may need to search for data in multiple levels, we can avoid unnecessary I/Os with additional metadata**

- Bloom filters help avoid unnecessary searches in a given level
- Metadata in “spine” helps to target searches within a level

## **I/O amplification is one of the biggest challenges for LSM-trees**

- **Leveled-design causes read amplification**
  - Searches may require I/Os at each level in worst case
- **Compaction causes write amplification**
  - Different compaction strategies favor write vs. read performance