CS647 Modern Storage Systems

- Compaction techniques

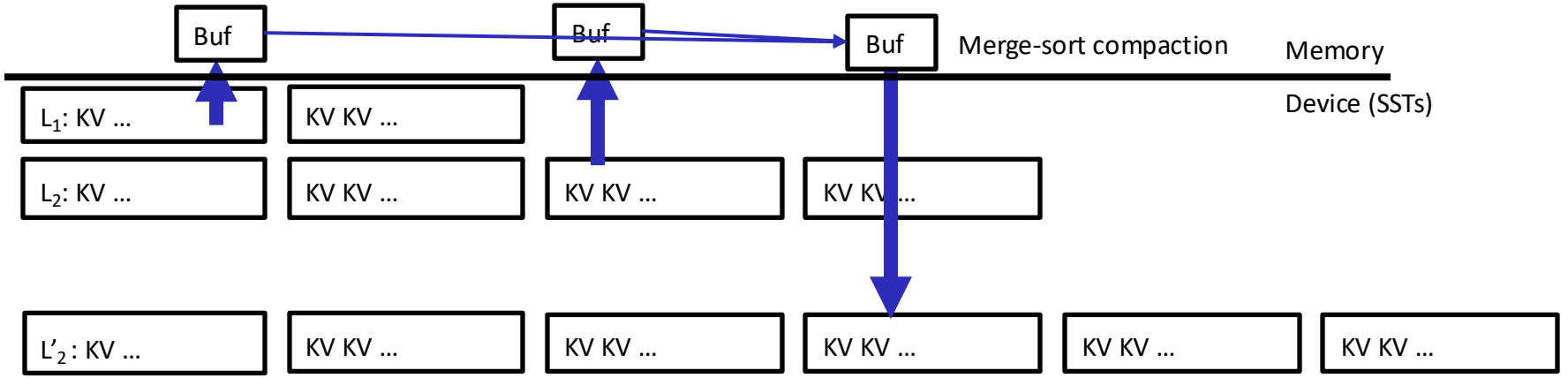
Angelos Bilas, bilas@ics.forth.gr

CS647 Modern Storage Systems, Fall 2023
(Modern Key-Value Store Design and Implementation)
Angelos Bilas, bilas@csd.uoc.gr
Computer Science Department, University of Crete

Outline

- Leveling compaction
- Tiering compaction (size tiering)
- Adaptive compaction
- Incremental compaction
- Date-tiered compaction
- KV separation
- Hybrid KV placement
- Partitions, Levels, SSTs (and Files)

Leveling Compaction ("traditional")

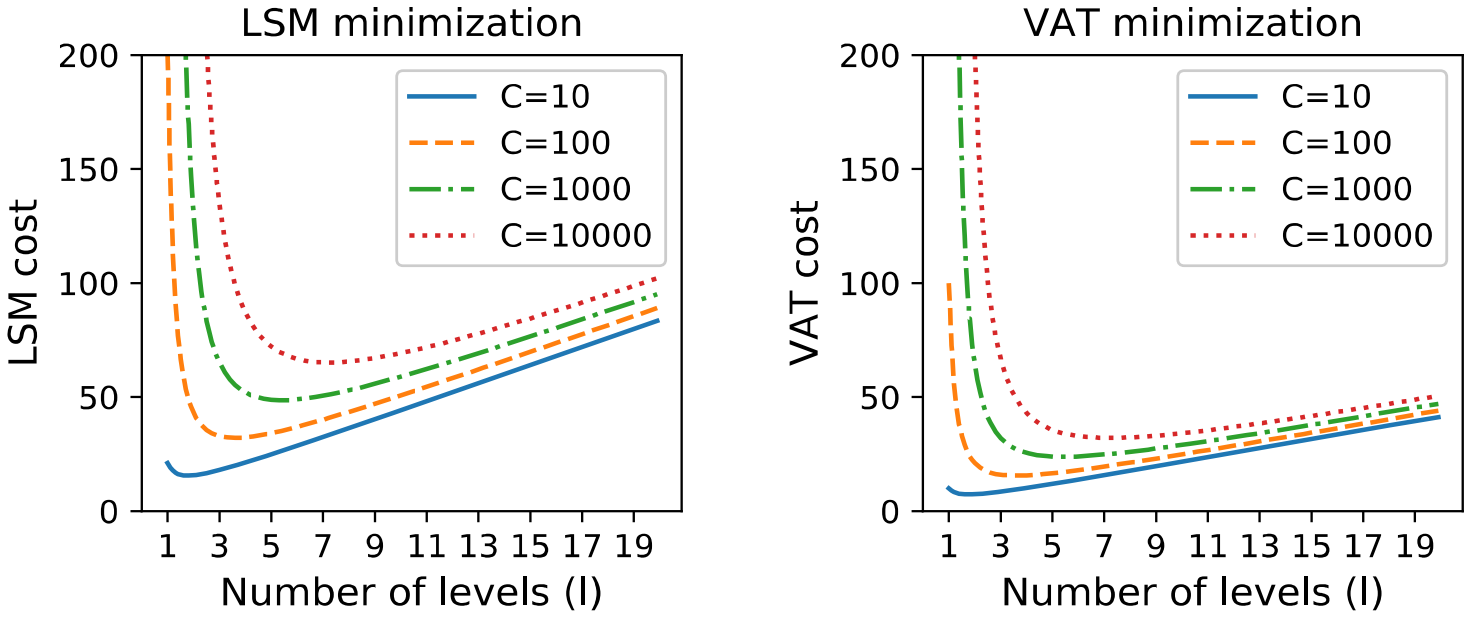


- 1. SSTs are sorted
  - => Read each SST once exactly
- 2. Read one key from each input SST
  - Add the smaller key to the output SST (and its value)
  - If they are the same, add the one from the smaller level (more recent)
- 3. SSTs have max/fixed size (e.g. 64 MB)
  - => Even if you read whole SSTs, it requires limited buffering (3 x SST size)
- 4. Compaction scans all SSTs for the two levels
  - => Can build an index on top if need be
- 5. Output SSTs are always full, except for the last

Optimal f, #levels for fixed $C=S_l/S_0$

[VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores. Nikos Batsaras et al. February 2020. arXIV 2003.00103. arxiv.org/abs/2003.00103v1]

$$\text{I/O amplification} = \frac{total\ device\ I/O}{application\ I/O}$$

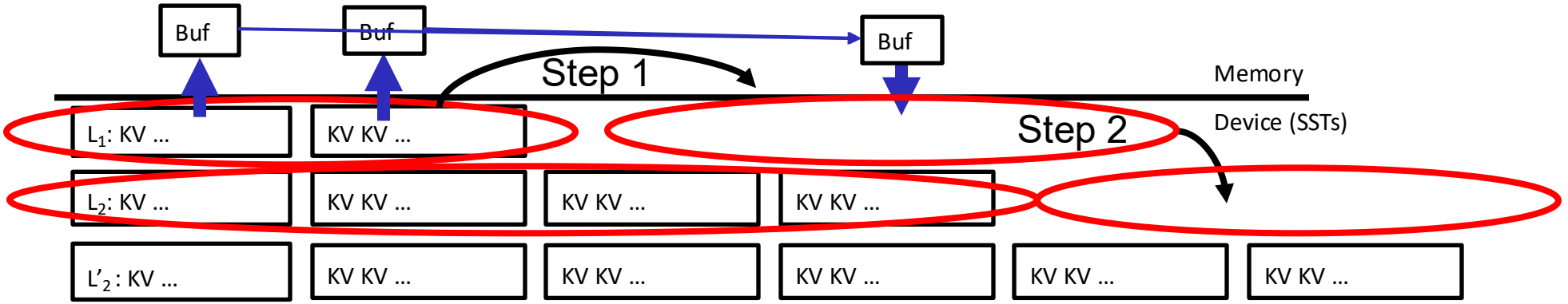


(a) LSM (b) VAT, a = r = 1

Figure 6: (a) Amplification vs. number of levels as a result of minimizing the LSM cost for various $C = \frac{\text{Workload size} = S_l}{\text{DRAM size} = S_0}$ ratios. (b) Amplification vs. number of levels as a result of minimizing VAT cost for various C ratios.

Tiering (Size-Tiered) Compaction (f files per level)

P. Raju et al. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. SOSP'17. https://doi.org/10.1145/3132747.3132765

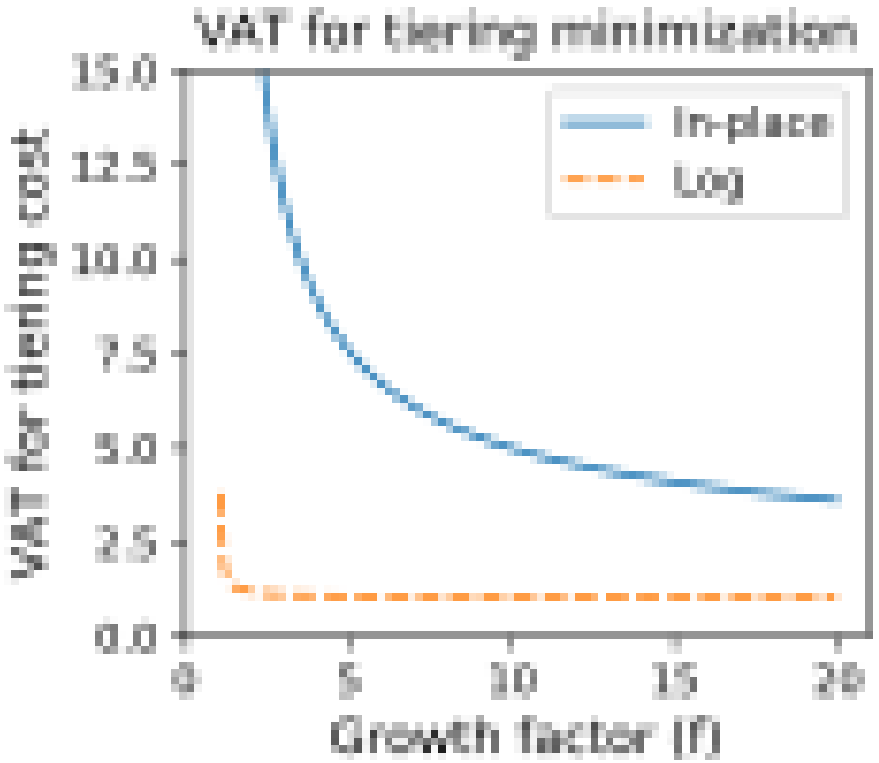


- Step 1: Merge f files in L_i in a single larger file
  - Each level is a set of f files
  - Each file can contain a key once - same key can repeat in all f files
  - Merge-sort these f sets to apply updates and reclaim space
  - **Merge-sort compaction only within L_i ➔ less work than leveling**
- Step 2: Add new file to L_{i+1} (**no merge**)
  - Files get larger from level to level - each level has at most f files
  - No read/write of data from L_{i+1} during tiering compaction
  - **A key may appear in multiple files in L_{i+1} ➔ up to f copies of a key**

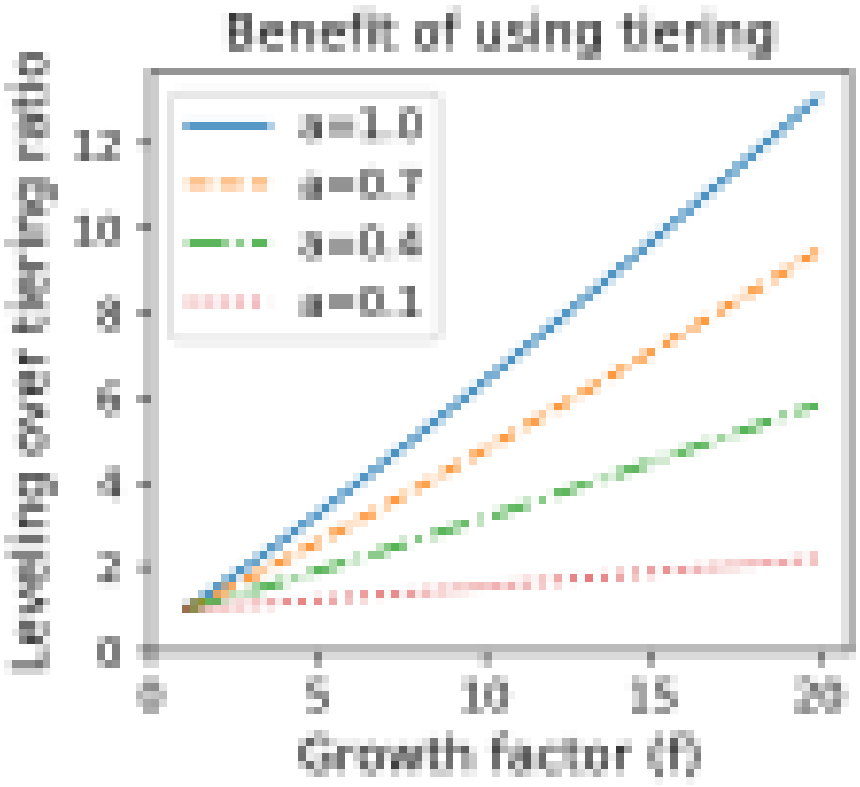
Benefit of Tiering over Leveling

[VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores. Nikos Batsaras et al. February 2020. arXIV 2003.00103. arxiv.org/abs/2003.00103v1]

- p=k/v size ratio, consider case a=1 (ignore rest of values)
- p=0.01 ➔ values are 100 times larger than keys
- log=kv separation, in-place=no kv separation



(a) $p = 1\%, r = 1, C = 1000$



(b) $p = 1\%, r = 1, C = 1000$

Tiering vs. Leveling

- How does tiering compare to leveling?
  - **Amplification:** Significantly lower (10x ?) with tiering and improves as f grows
  - **Space amplification:** Ln is f times larger than necessary when many updates
    - Because the last level could contain multiple values for the same key (updates)
    - A lot of waste (techniques to reclaim this?)
    - But no-one does tiering in the last level
  - **Reads:** f times worse without bloom filters
    - But similar with bloom filters and index in memory

Reads with Leveling	IO	Reads with Tiering → f times slower	IO
In each level a key appears at most once	-	In each level, a key may appear f times	-
Find single SST that may contain the key (use guards + index)	0	Find **f SSTs** that may contain the key (use guards + index)	0
Read the SST bloom filter	0-1	**Read** the SST bloom filter	0-f
Read the SST index	0-1	**Read** the SST index	0-f
Read the data block for the key	1	**Read** the data block for the key	f
If yes, return key, if not move to next level	N	If yes, return key, if not move to next level	N
Leveling Total per level	0-3	**Tiering Total per level**	0-3f

  - **Scans:** f times worse
    - Simple bloom filters do not help: Need to work with keys that do not exist in store
  - **Tail latency:** Can we perform incremental compaction with tiering?
- So: Leveling or Tiering?

Adaptive Leveling-Tiering Compaction

- Per full level
  - Niv Dayan et al. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. SIGMOD'18. https://doi.org/10.1145/3183713.3196927
  - Some levels use tiering, some levels use leveling
  - All lower levels use tiering and delay leveling until last level
    - If bloom filters work, then no impact of tiering for all levels on reads
    - Except for scans
    - Compaction to last level cannot be incremental, otherwise amplification too high (why?)
      - Each L_{n-1} SST will overlap with full last level (if uniform, no skew) - disaster
- Partially within a level
  - Part of the level is ”leveled” and part is “tiered”
  - E.g. allow some limited tiering **as excess space**
  - Not sure if this is actually used today

L_0: How to exceed memory size?

- L_0 affects the number of levels (and amplification)
  - The larger the better
  - But memory (DRAM) not as "plenty"
- Different approach
  - Place L_0 on disk as well and make it relatively large (size S_0)
  - Then "in-memory component" is just a (set of) buffers (aka memtables)
    - These buffers are overlapping (not a "proper" L_0)
  - Each memtable can have any (convenient) size (M)
    - Typically, a few memtables of a few MB each
  - What if $M << S_0$?
- Problem
  - Cannot merge M in L_0 because this will incur high amplification
- Solution
  - Make L_0 a tiered level → Each memtable is simply flashed to L_0
  - L_0 becomes a queue of overlapping/flashed memtables
  - L_0 requires tiering compaction → keeps amplification low
  - However: impact on reads – need to search all L_0 components
    - Significant impact on tail latency as well
  - To mitigate impact on reads, use a small number of components in L_0
    - More than f, but let's say less than 2-3 x f?
- This allow us to decouple memory used by LSM from # levels
  - Can have few(er) levels for the same memory size -- to some extent

Role of Incremental Compaction

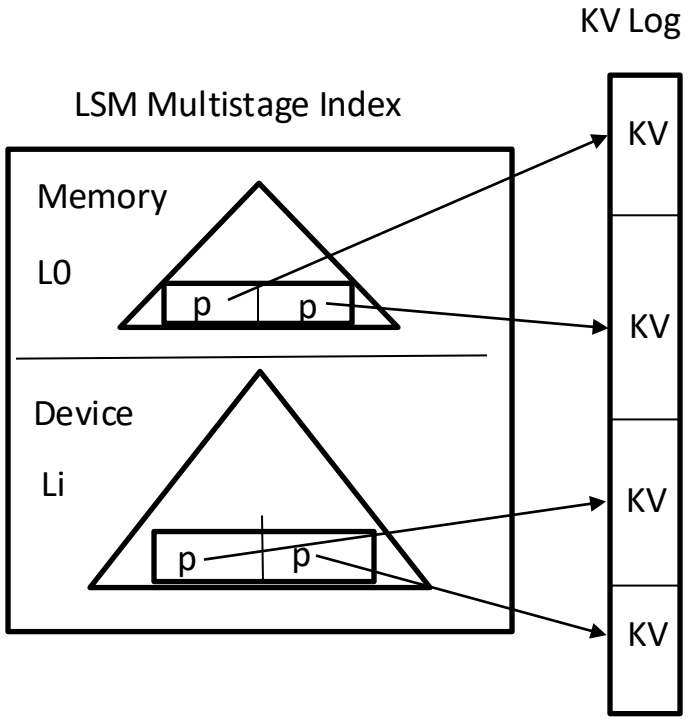
- Compaction is heavy
- **+Blocks inserts when you run out of memory in L_0 -> very long tail latency**
  - An insert may need to wait for L_0 to spill over to L_1
  - If L_1 is full you need to wait for L_1 to spill over to L_2, and so on …
  - In the worst case, wait for L_{N-1} to spill over to L_N
  - This can be 10s-100s of Gbytes of data (per region) - several seconds/minutes
  - Not acceptable for user-facing workloads
- Two sub-problems
- (a) Amount of work during compaction -> Reduce via incremental compaction
- Technique 1 for incremental compaction: Use SSTs (RocksDB approach)
  - Reduce the unit of work per chain
  - Move 1 SST from n-1 to n, then 1 SST from n-2 to n-1, etc, until 1 SST moves from memory to L1
  - Longest chain remains the same
  - Need to compact a few SSTs for each level of the chain (practically constant)
  - IOs for C x N x SSTs, at 1 Gby/s maybe ~100ms - 1s
  - **How can we reduce this unit of work further to 100μs? (open)**
- Technique 2 for incremental compaction: Schedule full-leveled compactions (bLSM approach)
  - Ensure compactions are scheduled to frequently create a bit of empty space in L0
    - Instead of creating a lot of empty space infrequently
  - Start all compactions (for all levels)
  - Use the scheduler to stop/start threads in a manner that no compaction has to block
  - If you choose a small enough unit to start/stop threads
  - You never need to block for more than N threads (1 per level), each doing a bit of work
  - Not simple/certain: **Careful when scheduling so that you never bring in a level keys that overlap with remaining keys**
- (b) I/O and thread scheduling
  - This is a different problem from (a) (and Tecnique 2 above).
  - Ensure I/O from appications are not delayed unnecessarily from other, background I/Os
  - Various papers and recent work

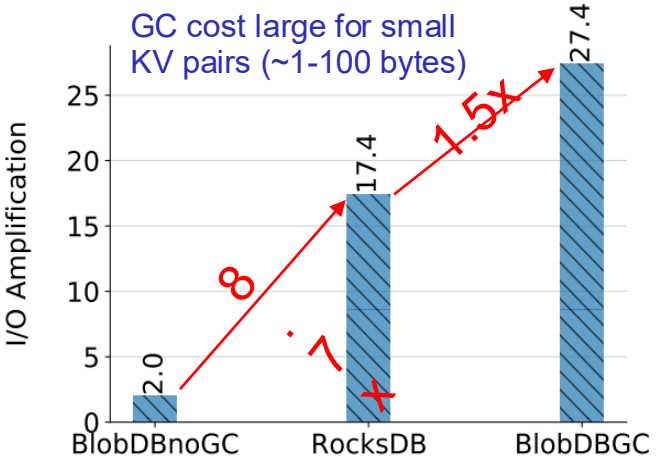
Date-tiered Compaction

- [Option in various systems, e.g. Cassandra]
- Good for time series: Reduce amplification + read Ios
  - Compared to (plain) tiering
  - Key is timestamp, kv pairs arrive mostly in order
  - Reads are performed with timestamp
  - Deletes are based on time-to-live
- Think of tiering (size-tiered) compaction
  - Compacts SSTs of the same size, but they may contain any timestamps
  - Merge-sort within level, plus f IOs per level for reads
- Instead: Compact strictly based on age
  - Same process as tiering, but use different rules for picking SSTs to merge
  - In Li, each SST has a range of timestamps
  - Create a larger SST that covers a larger range of timestamps from all SSTs
  - "Append" operation **-> not much cost** (unlike tiering that merge-sorts)
  - Move to next level, no cost (same as tiering)
  - Each level contains larger SSTs with older timestamps that are generally non-overlapping (or small overlaps) **-> reads require 1 IO per level (not f)**

KV Separation

- Reduce I/O amplification
  - At the cost of increased I/O randomness
- Places KV pairs in log (value log)
- Maintains LSM index only for keys
- **Reduces write amplification by 10x**
  - One more IO to find value
- Major drawback in KV separation
- Excessive overhead for GC in value log



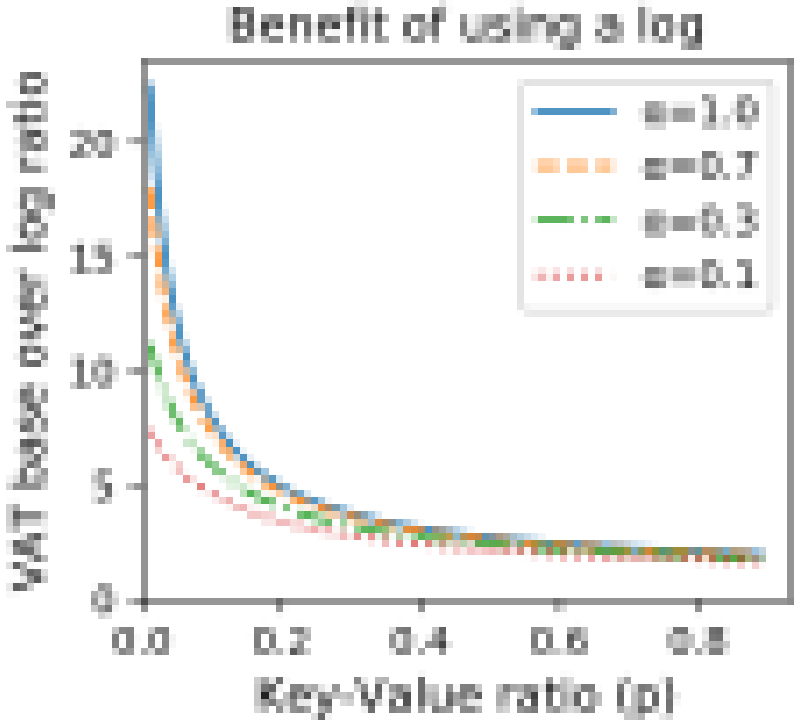


- C. Lai et al. Atlas: Baidu's key-value storage system for cloud data. IEEE MSST'15. https://doi.org/10.1109/MSST.2015.7208288
- L. Lu et al. WiscKey: Separating Keys from Values in SSD-Conscious Storage. ACM FAST'16. https://doi.org/10.1145/3033273
- A. Papagiannis et al. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. USENIX ATC'16. https://doi.org/10.1145/3267809.3267824

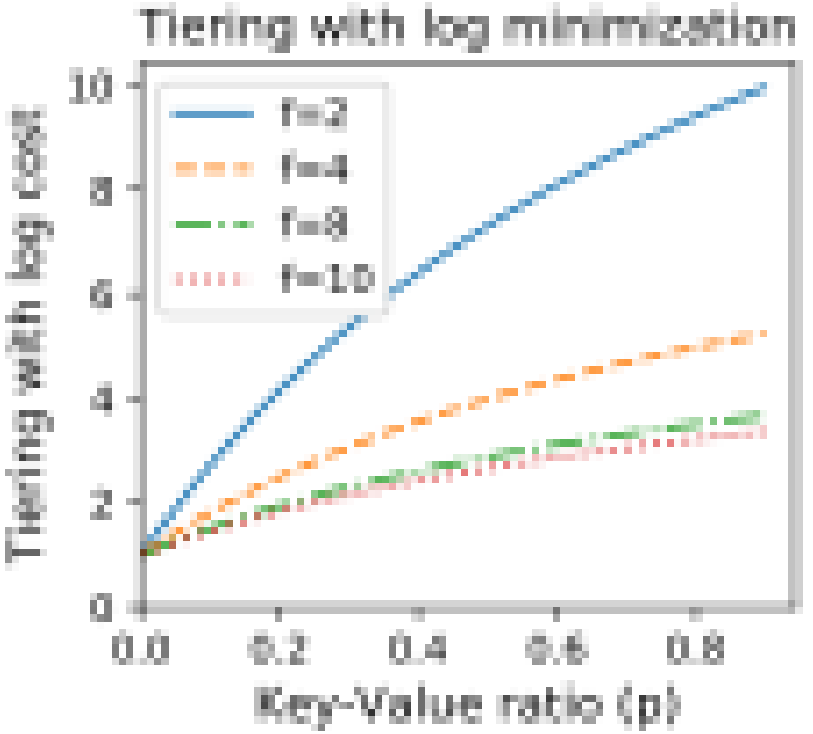
Benefit of KV Separation

[VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores. Nikos Batsaras et al. February 2020. arXIV 2003.00103. arxiv.org/abs/2003.00103v1]

- Ratio of I/O amplification
- Consider the case a=1
- p=k/v size



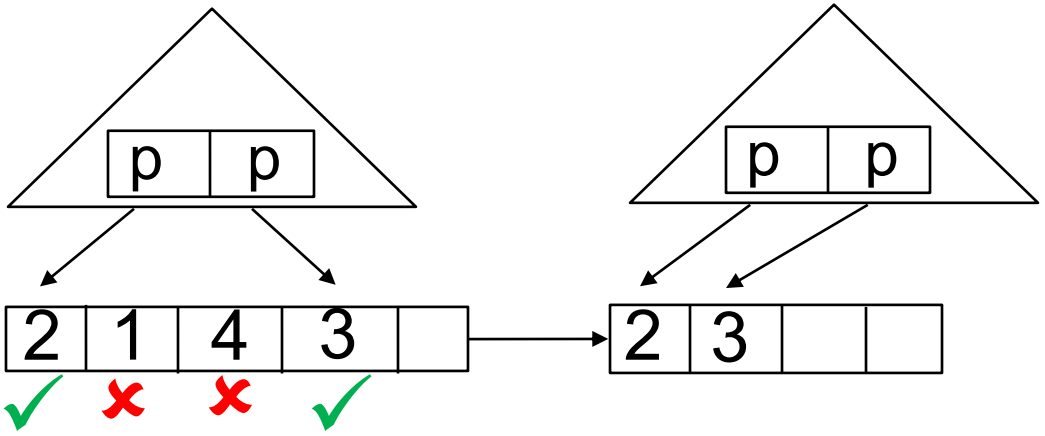
(a) $f = 10, C = 1000$



(b) $r = 1 \quad C = 1000$

Value Log Garbage Collection

- Deletes, Updates result in stale values
- Stale values need to be garbage collected
  - To avoid running out of space on device
- Two main overheads for GC
  - Identify valid KV pairs → requires lookup for each key in the LSM index
  - Re-append valid KV pairs in log → requires update in LSM index

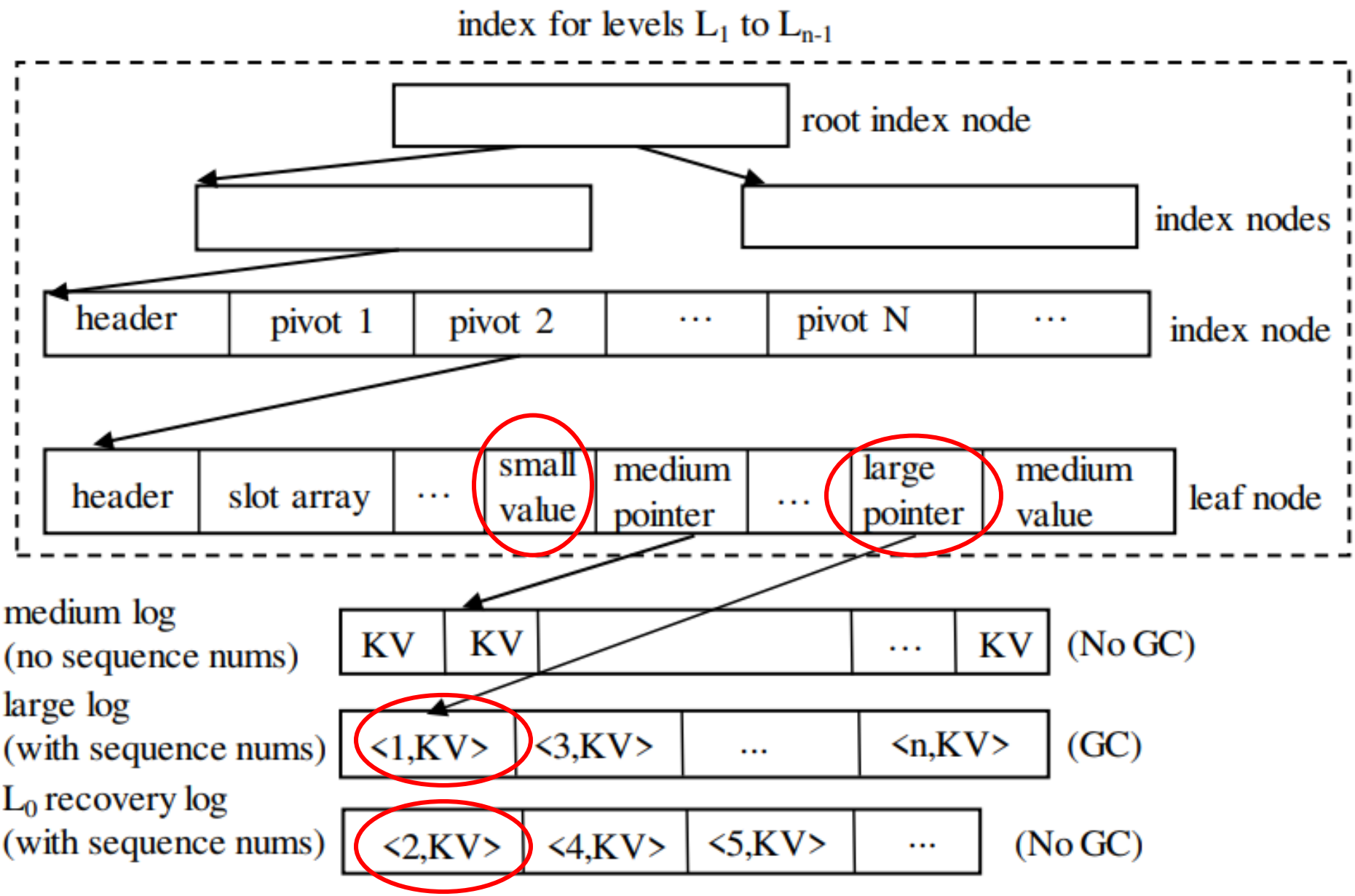


Hybrid KV Placement

G. Xanthakis et al. Parallax: Hybrid Key-Value Placement in LSM-based Key-Value Stores. ACM SoCC '21. https://doi.org/10.1145/3472883.3487012

- Categorize KV pairs based on size
- Small KV pairs (1 – 100 B) → Store always in-place
  - No benefits from KV separation
  - Avoids garbage collection completely
  - Stores them in place in the LSM index
- Large KV pairs → Store always in a log
  - Up to 10x less amplification
  - Uses garbage collection to reclaim space
  - Writes them in a separate log
  - Stores a pointer in the LSM index
- Medium KV pairs → Deferred Compaction

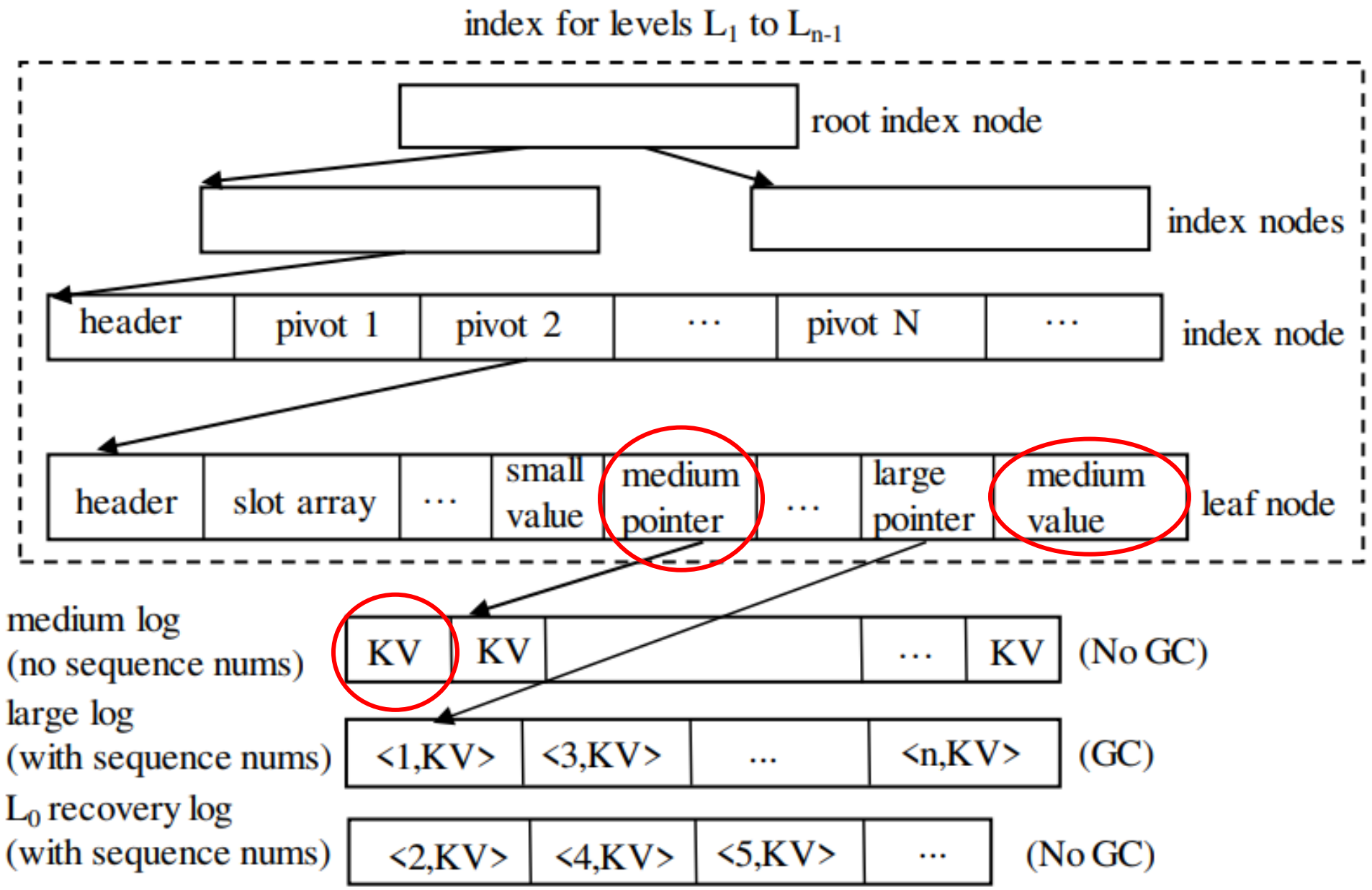
Parallax: Hybrid KV Placement



Deferred Compaction for Medium KVs

- Medium KV pairs (100 – 1024 B)
- Place in a "medium" log and
  - Store a pointer in the LSM index for the first N-1 levels
  - Defers placement in the medium log at $L_0 \rightarrow L_1$ compaction
- Keeps medium KV pairs in sorted runs for efficient in-place merge at level L_i
- Result
  - Small values always in place – compactions
  - Large values, always in log – not too expensive for GC
  - Medium values: in log until last level, then in place
- What if more levels are created over time?
  - Medium values [probably] continue in place

Deferred Compaction for Medium KVs

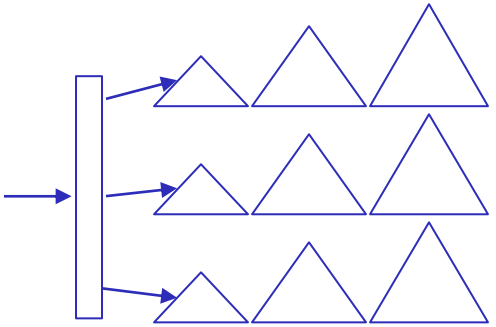


Recovery in Parallax

- KV pairs may change category over time
  - E.g. due to updates
- Multiple logs -> no ordering of operations
- Use sequence numbers to maintain order
  - Increasing monotonic counter stored with each KV pair
- Small and medium KVs
  - **Need** to store them in L_0 recovery log
- Large KVs
  - Large log also serves recovery purposes
  - These are not written in L_0 recovery log

Partitions (Regions), Levels, SSTs

[1] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. The VLDB Journal, 2007. https://dl.acm.org/doi/10.1007/s00778-005-0171-7



- Partitions
  - Divide the key address space in partitions (–oo, K1], … (Ki-1, Ki], …, [Kn, +oo)
  - Each partition is an LSM of fixed size
  - Split (merge) to balance partitions
- Partitions can help in two ways
  - (a) Concurrency (independent accesses)
  - (b) Key skew: e.g. If updates are concentrated within a single partition, reorganize less
  - Disadvantages?
- What is a file? Options:
  - 1: Single file per partition -> Partitioned Exponential Files [1]
  - 2: Single file per level per partition -> Leveled LSM (basically no incremental compaction)
  - 3: Single file per SST per level per partition -> SST-based LSM (incremental compaction)
- Each file (be it partition, level, or SST) contains
  - Data, index, bloom filters
  - Important (at least until recently) for ensuring sequential accesses
  - Part of metadata assumed to be pinned/cached in memory
  - **File organization relatively complex (variable size items, prefixes, hashes, etc.)**
- How necessary is to pack data and metadata in a single file?
  - Index: unclear answer, if SSTs become smaller (why small SSTs?)
  - Bloom filters: per level or per SST or combination and what operations required?
- What is the role of partitions, levels, SSTs?
  - Partitions, levels more clear (by now) – Generally, few partitions and few levels
  - SST? -> primarily incremental compaction (maybe robustness and portability too)

Optimal way to reorganize data?

- So far we assume KV store has levels with growing size by factor f
  - L_0 fits in memory (compulsory)
  - In some sense, we limit the LSM structure and reorganization possibilities
- What if we do not assume anything about levels and their size and how merge occurs?
  - What is the optimal way to reorganize (merge) data?
  - Optimal = minimum work (per byte inserted)
- Explanation and exploration in
  - Qizhong Mao, Steven Jacobs, Waleed Amjad, Vagelis Hristidis, Vassilis J. Tsotras, and Neal E. Young. 2021. Comparison and evaluation of state-of-the-art LSM merge policies. The VLDB Journal 30, 3 (May 2021), 361–378. https://doi.org/10.1007/s00778-020-00638-1

Reading

- Review
  - Slides + pointers, Your class notes
- Read
  - Any of the papers used as references in the slides
- Next class
  - Paper discussion