CS647 Modern Storage Systems

- Multi-level key-value stores
- Log-structured Merge Tree (LSM)
  - Levels, growth, compaction, SSTs, index, device space, bloom
- Discussion: How do KV stores relate to other stores?

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

What is the fastest way to perform writes?

- Append only
  - Create a single file
  - Always write at the end of the file
- Batch writes to avoid 1 I/O per request
- Put all applications in a single file
- All writes will operate at device write throughput
  - Scales with number of devices
- There is nothing better we can do
- Problem: Reads extremely slow → search full file
  - Forget about scans (many random operations per scan)
  - This is still used for write only data (e.g. logs/archives)

What is the best way to do reads?

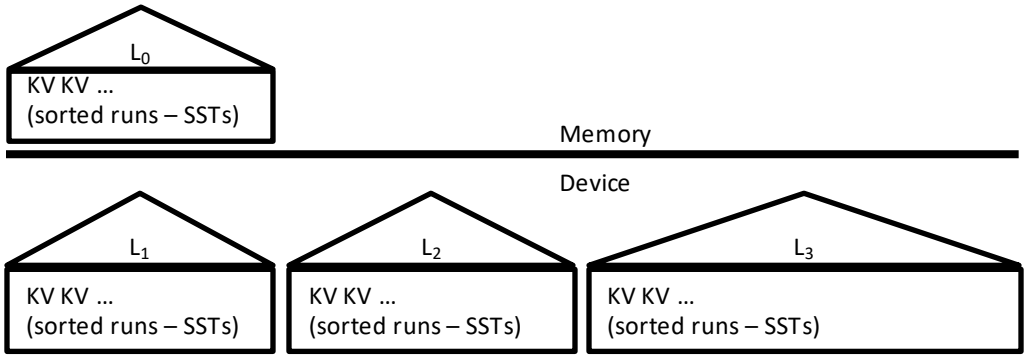
- Assume a very large address space (big data)
- Assume random reads (not necessarily, but likely)
  - Especially if you consider many users/applications
  - "Blender" effect: even if you optimize one application
  - Cannot achieve this for dynamically co-located applications
  - Fundamental aspect of consolidation in datacenters/cloud
- Some type of tree index (B+-Tree)
  - If you assume index fits in memory (questionable)
  - 1 I/O per random read operation
- Problem -- Writes become slow
  - They need to update the index for persistence/recovery
  - Several random I/Os and write amplification for small writes
  - This is what filesystems (and many other storage systems)
  - Possibly, wasted space on device (will come back to this)

Multi-level Key-Value Stores

- An interesting solution
  - Initial goal: bridge characteristics of different storage devices
- Always write in an append-only fashion
  - Fastest writes ➔ fastest ingest to the system
  - Irrespective of users/apps/patterns
  - Scales with device throughput
- In the background reorganize data and (re)build index
  - How? – This is the core of multi-level key-value stores
- Always read/scan using an index
  - Fast reads too (with caveats)

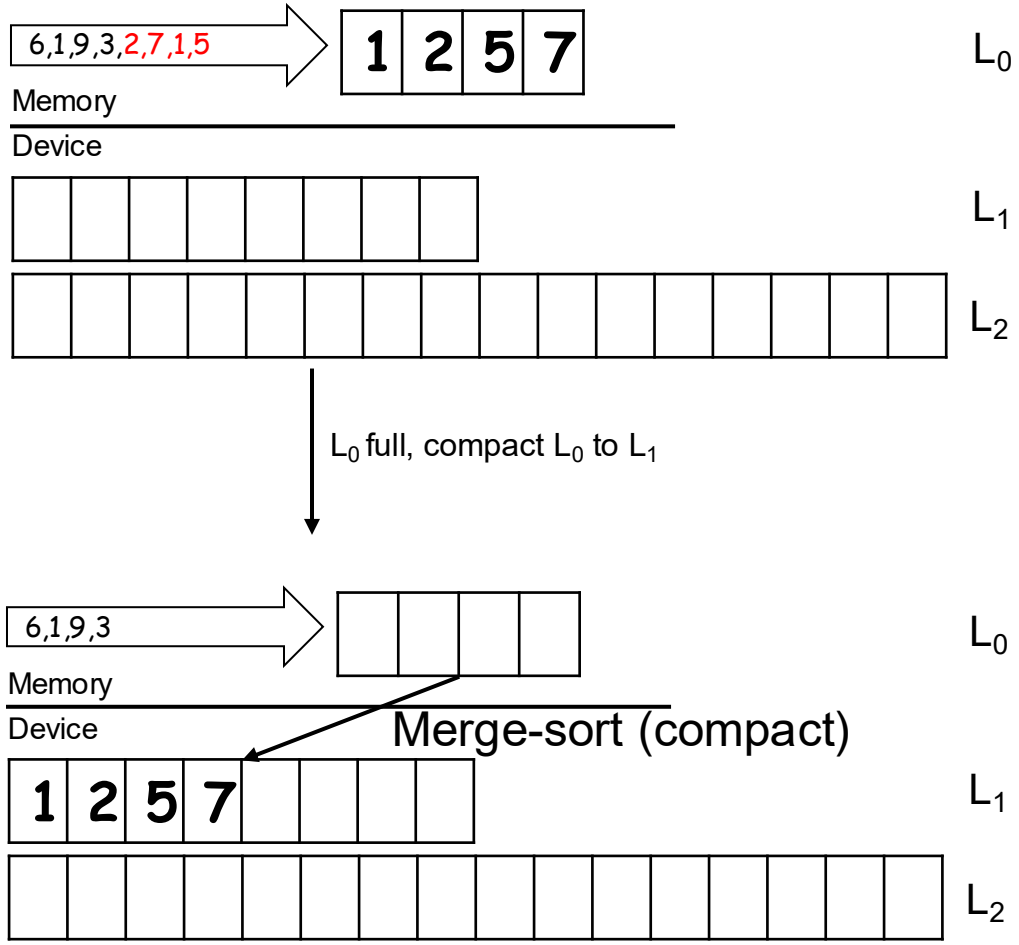
Log-structured Merge Tree (LSM)

P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. 1996. The log-structured merge-tree (LSM-tree). Acta Inf. 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

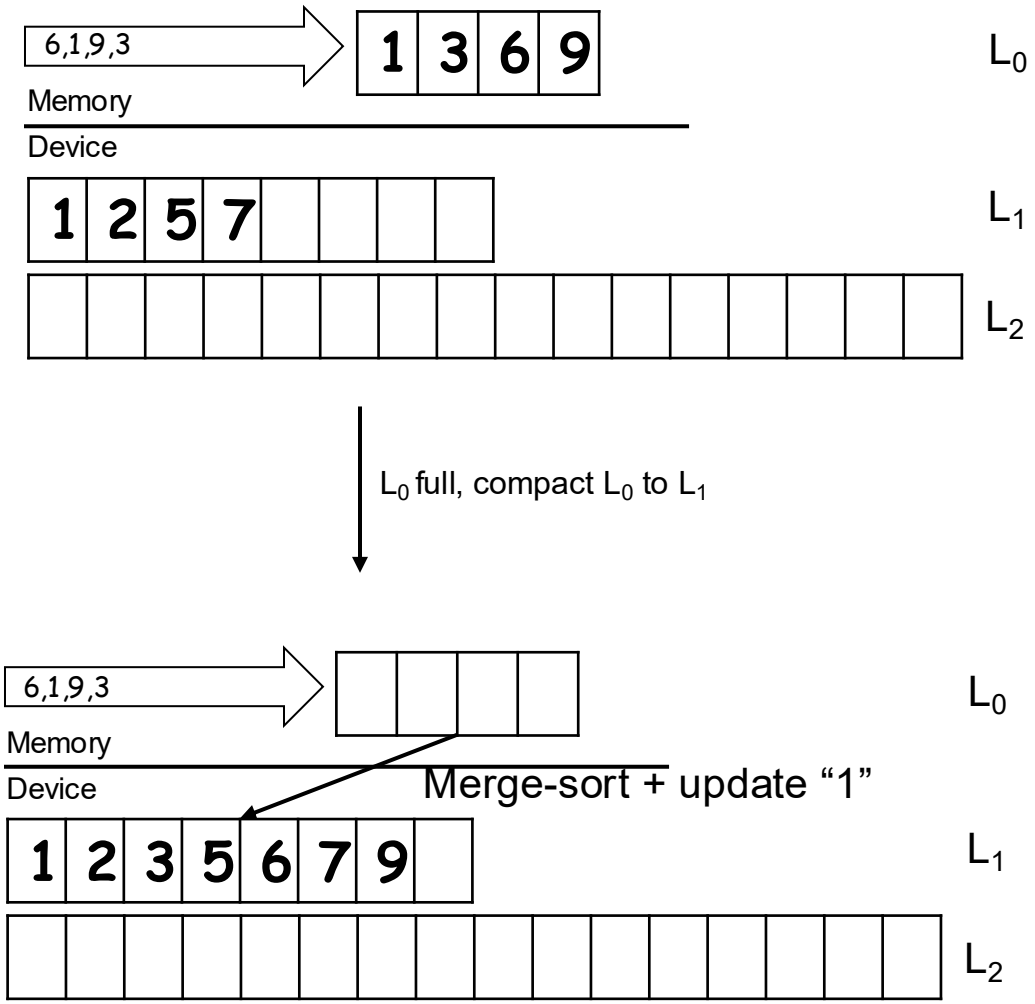


- To reorganize data use multiple levels
  - Each level is f times (e.g. 10) larger than previous
  - Each level is sorted
  - Each value appears only once in each level
  - L_0 is always fully in memory (has to fit in memory)
- Data **travel** from level to level
  - Always from L_0 to L_{max}
  - Newer values/updates always in lower levels
- When do you reorganize
  - Only when a level is full
  - Take the full level and "merge" it with the next one
  - **This is "compaction"**
    because in the process you overwrite (compact) old values and cleanup space
- Each level will "see" about f incoming compactions before it becomes full
  - No keys are inserted to a level during its compaction to the next level
  - This is **full leveling** compaction (no overlapping compactions across levels)

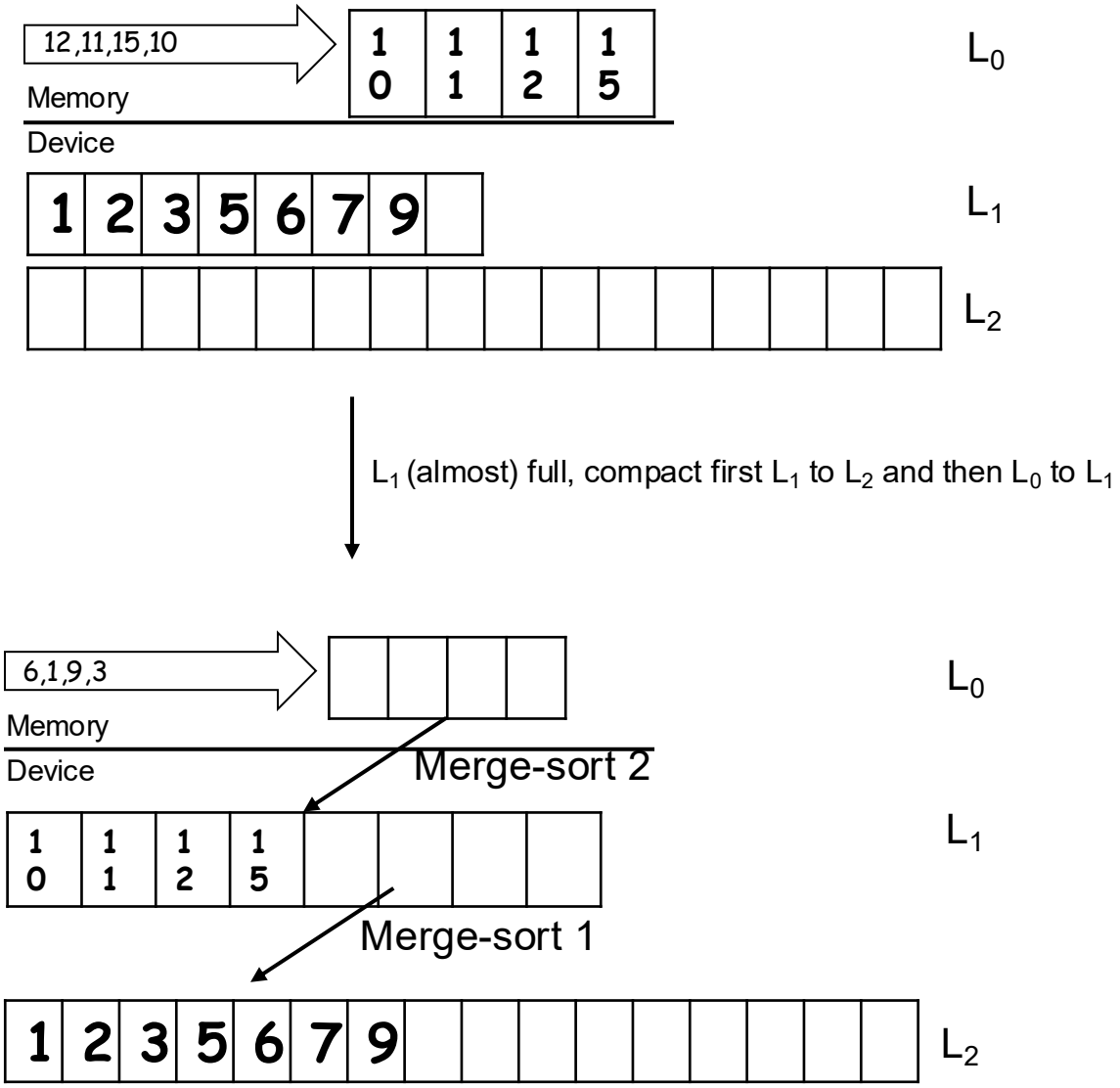
Example (L_0= 4 keys, f=2)



Example (L_0= 4 keys, f=2)

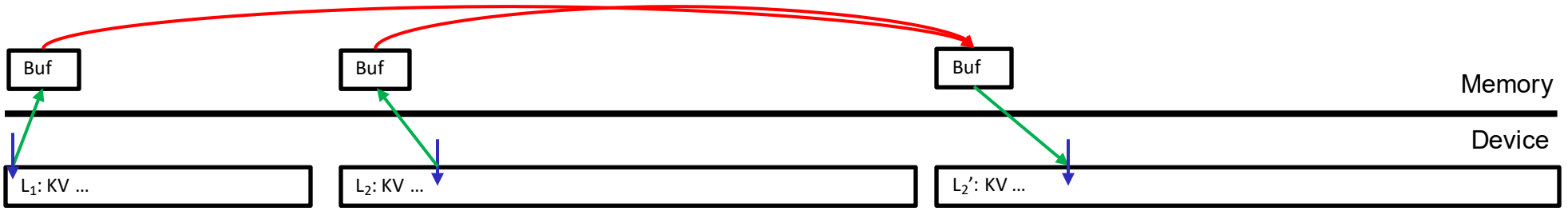


Example (L_0= 4 keys, f=2)



LSM Compaction = Re-organize Data

P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. 1996. The log-structured merge-tree (LSM-tree). Acta Inf. 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

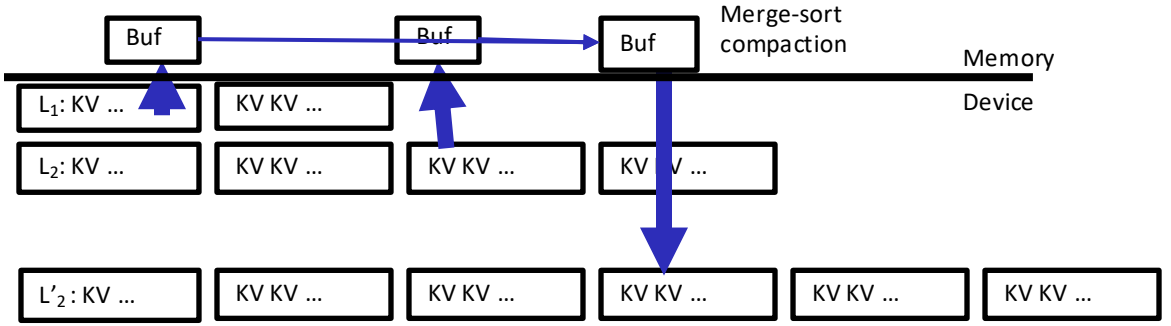


- Why?
  - (a) apply updates and free space
  - (b) reduce number of (larger) levels and re-create index/sort (or similar)
  - Name "compaction" from (a), as it collects old/stale values (GC)
  - How important is (b)? Consider only inserts (no updates)
- Compaction = Take two files L1, L2 and produce L2' = Merge-sort
  - L2' file: larger, sorted (+index), each key once
  - **Single pass** over L1, L2, L2', using little memory (1 read each level, 1 write for output level)
  - Buffer enough to max device throughput for read/write (several KB or a few MB)
  - As you merge-sort create new index (to avoid binary search on dev for reads)
  - Heavy: read/write amortized but involves a lot of data
  - Reads/writes can be parallelized – memory merge is a question (today single thread per compaction)
- Levels grow in size only via compactions
  - Always large IOs – no writes to a KV store need to be small
  - Compact to the next level only when a level is full (full leveling)
  - First merge with empty next level, second with larger next level, etc. (f times)
- What if L_0 becomes full? - Same process
- What if the last level fills up? - Create a new last level, f times in size

LSM-Tree – Level growth factor f

- Assume size of L_0 in mem=S_0 → Max/nominal L_i size is $S_i = S_0 * f^i$
  - S0 = 1GB (in memory), S1=10GB, S2=100GB, S3=1TB, S4=10TB
- Why do levels need to grow in size by f? ← IMPORTANT
  - (a) To reduce number of locations for each key → f cannot be too small
    - If f=1 then one key could be located in many locations (levels) on the device
    - Reads would become slower
  - (b) To amortize write cost during compaction → f cannot be too large
    - If f too big, then we could have a single level (L0 in memory and L1 on device)
    - When we spill L0 to L1, as expensive as B+Tree (no/little write amortization)
- What is a reasonable value for f?
  - Optimal f ~=4, typically today f=8-10 (more later in the course)
  - Can f be variable?
    - Not clear today - We do not know what implications this would have
    - Systems may allow some "slack" as levels grow
    - i.e., no need to be exactly f times the previous one

Single File per Level vs. SSTs



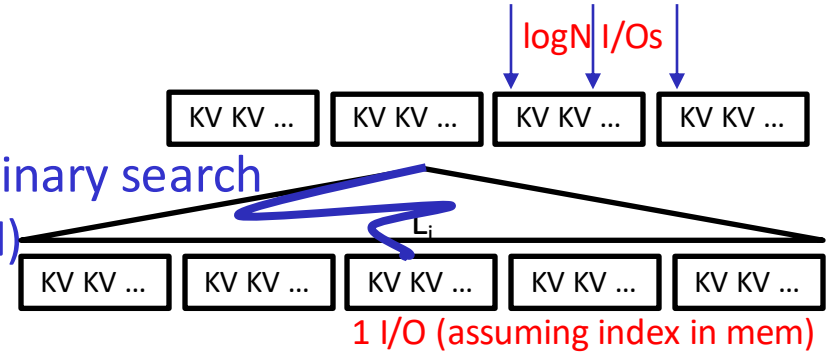
- So far, each level is a single file (full leveling)
  - When compaction starts, it needs to finish before ”accepting” new kv pairs
  - You can stop and restart, but not new kv pairs can be added
- Alternative: Sorted String Tables (SSTs)
  - Each level is a set of fixed size files - a few MBs per SST
  - Could also use a fixed number of variable-size SSTs per level (by f)
- Not much changes for compaction (for now)
  - Start reading L1, L2: SSTs are sorted => Need 1 SST from each level
  - Still one small buffer for each level -- Merge buffers in mem, write to L2’
  - If any buffer is empty, read next portion of SST
- SSTs will become important later (incremental compaction)

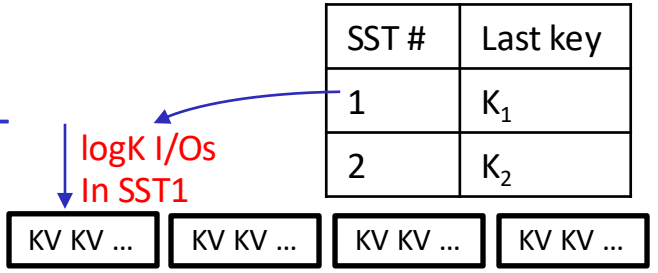
Compaction: Full or incremental

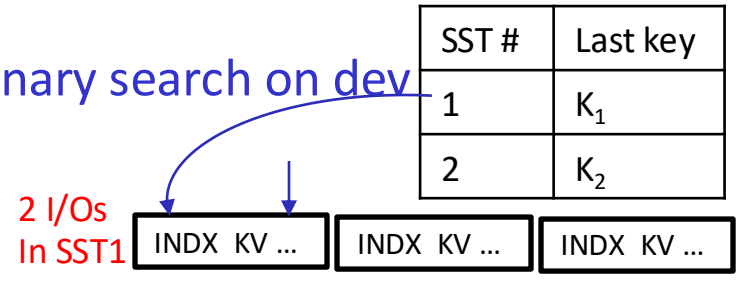
- Heavy process
  - IO (read and write), CPU, memory traversals
- Why do we really need this?
  - (a) Apply updates and reclaim space
  - (b) Increase level size so each key in few places on device
- Various optimizations but two main forms
  - For this discussion define "compaction = the process of emptying a full level"
- Full leveling: Compactions cannot overlap if they have a common level
  - Start compacting L_i - Until L_i becomes empty you cannot compact L_{i-1} to L_i
    - Cannot start emptying a level and continue filling it up
  - Compactions may still overlap, if there are no common levels
  - Full leveling can be done with single-file levels or SSTs
- Incremental: Compactions can overlap even with common levels
  - If you need to compact data to L_i and there is no space
  - Create some by compacting a single SST from L_i to L_{i+1}
  - Essentially, a chain of small compactions from L_n to L_0 to create space for new kv pairs
  - Easier to implement when using SSTs (with single file, cannot insert new kv pairs easily)
  - Limited forms can be done also with single-file levels (complicated file allocation)
- Incremental compaction has various and significant implications
  - Amplification, latency, …, we will discuss later

How do we index each level?

- Goal: 1 I/O per read (per level?)
- (1) No index
  - Use sorted levels/SSTs to find a value with binary search
  - No metadata, but more I/Os on device (logN)
- (2) B+-Tree, e.g. original LSM paper
  - Used as index in many storage systems (filesystems, caches, databases, etc.)
  - A tree, where nodes and leaves are adjusted to device block size
- (3) Guards
  - For each SST, keep the last key (guard) in a table in memory
  - Guard table: guaranteed to fit in memory
    - 64MB SSTs => 16 SSTs / GB => 16K SSTs / TB => 16K guards
    - If each guard ~20B => 320 KB of memory / TB of storage
  - When you need to find a key, check guard table for SST
  - Then perform binary search in SST (index within SST)
- Can combine, e.g. RocksDB
  - Use guards to find SST (upper part of full index)
  - Place the lower part of index in the SST – avoid binary search on dev
- **More on indexes later in the course !**







KV-store space on the device

- Let's define
  - Static capacity KV store: A KV store with a fixed number of pre-allocated levels
  - Dynamic capacity KV store: A KV store that grows level by level as data get inserted, up to available dev space
  - Consider a dynamic capacity KV store with N levels
- **How much space on the device is reserved but not used at any point in time?**
- First, let's define when the KV store is considered full
  - When we cannot add any new KV pairs, regardless of whether it contains unique kv pairs or updates
  - Basically, no space left = full (but space is used by both unique kv pairs or updates)
  - Could define in more elaborate manner (unique keys only), but probably not practical
  - Essentially, assume that intermediate levels do not matter because they are (relatively) small
- We only need to consider how we allocate space for the last/new level
- (1) When creating L_{n+1} reserve S_{n+1} space for it
  - Simple → Leveling (and incremental) compaction in this case need not change
  - BUT until L_{n+1} becomes full, it will be half empty on average, so only 50% of the space is used
  - This is also about 50% of the full KV store space on device (at f=10) since the rest of the levels are small
  - So KV store always reserves about 2x the space it uses (2x number of devices, etc.)
  - Can we do better?
- (2) When creating L_{n+1}, allocate S_n space at each compaction to L_{n+1}
  - Leveling compaction (and incremental) still works
  - Wasted space is only S_n at a time (as L_{n+1} grows) so:
    - 1st compaction to L_{n+1}: kv store has size about S_n and wasted space is S_n, so about ½
    - 2nd compaction: kv store is about $2S_n$ and wasted is S_n, so about 1/3, etc.
  - Assuming f=10 then wasted space is about (1/2+1/3+…+1/11)/10 = (55991/27720)/10 = ~20%
  - Can we do better?
- (3) When creating L_{n+1}, allocate space for one SST at L_{n+1}
  - We compact one SST from N to N+1, and so on, from 0 to 1, then level 0 has an empty SST and we can insert
  - ok, but it is incremental compaction; **it overlaps compactions at levels** – not full leveling compaction
- **Wasted device space: Full leveling compaction → about 20% vs. Incremental → a few SSTs**

Dynamic Capacity Adaptation (DCA)

[1Siying Dong et al. Optimizing Space Amplification in RocksDB. CIDR'2017. https://www.cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf]

- RocksDB does (3) → Problem is that
  - Each new last level does not maintain ratio f with its previous level
- **Dynamic Capacity Adaptation (DCA)** in RocksDB[1]
  - When all levels are full, instead of creating a partial new level n+1
    - Rename the current last level to +1
    - Same thing for all previous levels
    - Create a new, empty L_0
  - We merely allocate one new SST L_0 (or the first level on the device)
  - However, the nominal (max) size for each level has now increased
  - As we compact data in each level, we need to allocate a new SST
    - And compact/reorganize as usual
  - When all levels reach their max size, repeat
- DCA is applicable to full leveling as well
  - At the point where you need to create the new level
  - Rename level N to N+1 and so on
  - When compacting level i to i+1, allocate S_i space at a time
  - For compaction to last level, allocate S_{n-1} (max waste ~20% at f=10)

Bloom filters: Avoid reading all levels

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

- A read requires looking up all levels, one by one
- Can we do better? Bloom filters!
  - Filters for checking set membership
  - As you insert keys in a set L, build a filter
  - At any time, answer question "is key K part of this set L?"
- Baseline for exact membership
  - Use a hash Table T of h cells each of key size (b bits)
  - When inserting a key k, generate a hash h1 and
    - If cell h1 is empty add the key (b bits)
    - If cell h1 is full generate another hash h2 and go to cell h2 - continue until you find an empty cell to store the key
  - When checking for membership, generate h1, h2, … in the same manner
    - If the key is found in a cell, then key is present in the set – if you find an empty cell, then key is not present
  - Modification: Use less than b bits per tag (smaller tags than keys)
- Bloom filter
  - Do not use arrays with cells but a bitmap of N bits (5-10 bits per key expected to be in the set)
  - Each key is mapped to d bits with d hash functions (d=4 seems to be a good number)
  - When inserting set the corresponding bits to 1
  - When checking membership, if d bits are 1, then key is present otherwise not
  - ➔ **Few bits per item and few hashes (d) for good accuracy – NO is an exact answer, but YES may be wrong**
    - Table for 50K items, d=4, N=~400KBits (8 bits per item), >87% accuracy
  - Cannot delete elements from set – need to rebuild (other filters exist)
- Keep a bloom filter in memory per level, check if key present, avoid I/O to level
  - A level with 10M keys (~1GByte data?) requires between ~1 MByte bloom filter
  - 10B keys (~1TByte data?) require ~1GByte bloom filter
  - False positives due to wrong YES answers (never false negatives)
- When L is merged to L+1, add the keys to the L+1 filter and delete the filter of L
  - Today, bloom filters per level and per SST, with different accuracy – other techniques as well

TABLE I. Summary of Expected Performance of Hyphenation Application of Hash Coding Using Method 2 for Various Values of Allowable Fraction of Errors

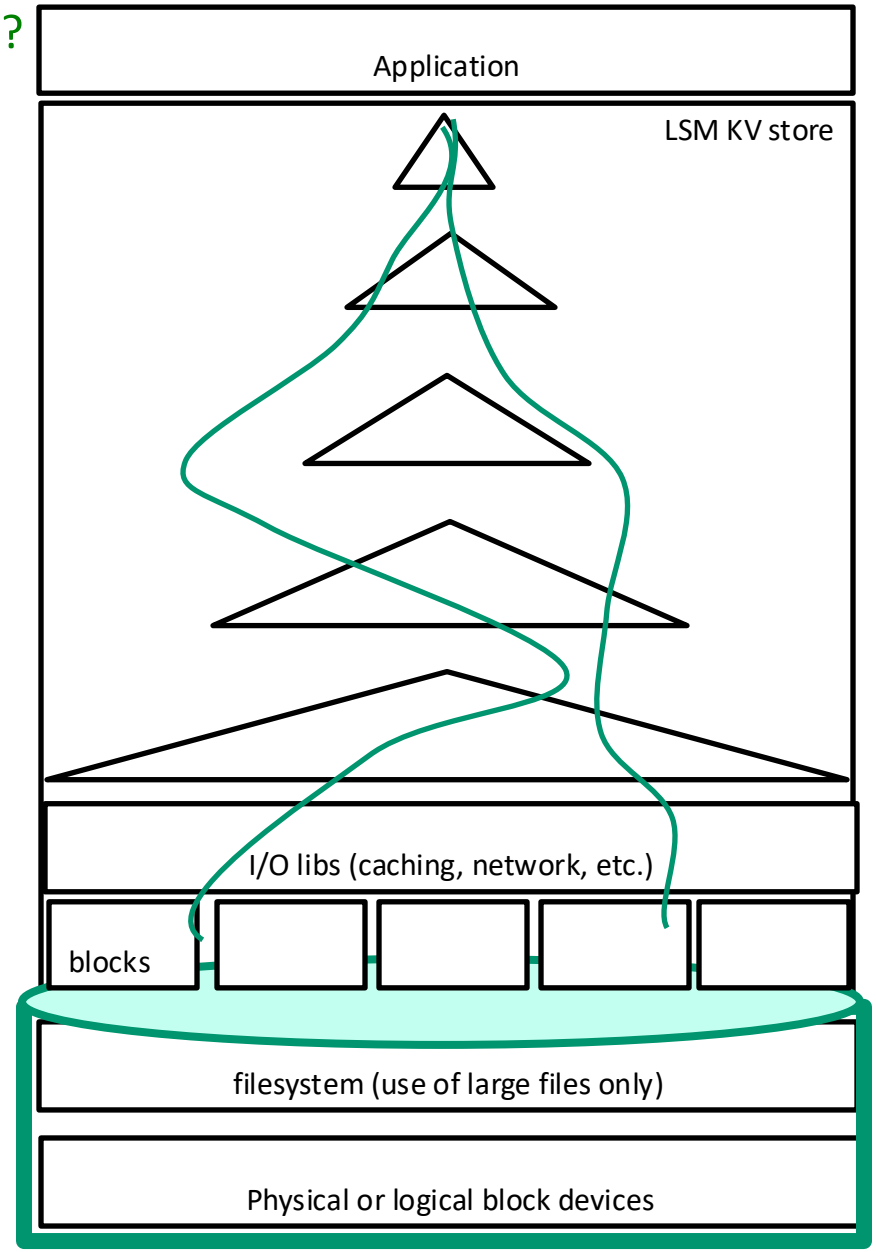
P = Allowable Fraction of Errors	N = Size of Hash Area (Bits)	Disk Accesses Saved
$\frac{1}{2}$	72,800	45.0%
$\frac{1}{4}$	145,600	67.5%
$\frac{1}{8}$	218,400	78.7%
$\frac{1}{16}$	291,200	84.4%
$\frac{1}{32}$	364,000	87.2%
$\frac{1}{64}$	509,800	88.5%

Bloom Filters

- Probabilistic data structure which answers
  - False: item not present
  - True: present with high probability
    - Depends on the bits per key used
- E.g., initial state Bloom Filter size 12 bits, 3 hash functions
  - {0,0,0,0,0,0,0,0,0,0,0,0}
- Add “apple” in set
  - Bloom_filter[hash_func_1(“apple”)%12] = 1
  - Bloom_filter[hash_func_2(“apple”)%12] = 1
  - Bloom_filter[hash_func_3(“apple”)%12] = 1
  - {0,1,0,0,0,0,0,1,0,0,0,1}
- “Is x in set”?
  - Compute three hash functions for x
  - If the bits in the position are set to 1, x is in the set

Discuss: storage stack then and now (L01)

- What changes in the storage stack with KV stores?
- Then
  - Application
  - I/O libs (cache, network)
  - OS (block + FS)
  - Device (+ firmware)
- Now
  - Application
  - Key-value store (KVS)
  - I/O libs (cache, network)
  - OS (block + FS)
  - Device (+ firmware)
- Read/Scan
  - Reads look for a specific key
  - Scan start key may not exist
  - Scans need to look at ALL levels
- Application performance?
  - Inserts/Writes: 1 amortized I/O per insert
  - Gets/Reads: 1 random I/O (bloom + index in mem)
  - Constant compaction → Significant cost
    - Can we not compact? E.g. if no updates?
    - Can we reduce compaction overhead?
  - Tail latency?



LSM: Optimal growth factor and #levels

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

- LSM amplification is sum of
  - Level amplification (better for larger f → fewer levels)
  - Merge amplification (better for smaller f → more amortization)
- What is the optimal growth size f?
- Definitions
  - Ki, Vi: Size of all keys, values
  - Si: capacity of level i (Ki+Vi)
  - l: number of levels on storage device
  - f: growth factor
  - H: Required throughput pages/s
  - R: device throughput (sustained)
  - Sp: page size for device to achieve R
- LSM: contribution of each level to compaction traffic is
  - Each level is read and written twice
  - Except last last level

$$H = \frac{R}{S_p}((2f_1 + 2) + \ldots + (2f_l + 1)) \qquad (18)$$

$$H = \frac{R}{S_p}(2(f_1 + f_2 + \ldots + f_l) + 2l - 1)$$

$$= \frac{R}{S_p}(2\sum_{i=1}^{l} f_i + 2l - 1)$$

- Optimal case is when H is minimum

$$\min H = \min(2\sum_{i=1}^{l} f_i + 2l - 1) \qquad (20)$$

- Trivially, will lead to S0=Sl, l=0, or Sl=0
  - Need to minimize H under some constraint

- (a) When data size S_l is constant
  - Represent data size Sl as multiple of mem
  - Then data size is Sl=C*S0
  - Then constraint becomes: constant C=Sl/S0
  - "Try to find minimum H (best f) when all alternatives can hold the same data Sl"
  - Alternatives may have fewer or more updates but last level needs to hold same data

$$\min_{\frac{S_l}{S_0}=C} H = \min_{\frac{S_l}{S_0}=C}(2\sum_{i=1}^{l} f_i + 2l - 1) \qquad (21)$$

  - Result f=~4
- (b) When data+update size is constant
  - Data+update size $C=S_0+\ldots+S_l$ = device space
  - "Try to find minimum H (best f) when all alternatives use the same total device space"
  - Alternatives may divide this space differently between data (Sl) and updates

$$\min_{S_0+\ldots+S_l=C} H = \min_{S_0+\ldots+S_l=C}(2\sum_{i=1}^{l} f_i + 2l - 1) \qquad (22)$$

  - Result f=~10
  - This is what RocksDB uses by default
- LSM includes both (a) and (b) -- VAT paper includes all step of the derivations

LSM: Optimal growth factor and #levels

- Constant memory $C = S_l/S_0$
  - $\Rightarrow C = f^l \Rightarrow l = \log_f C$

$$\min_{\frac{S_l}{S_0}=C} H = \min_{\frac{S_l}{S_0}=C} (2lf + 2l - 1) = \min_{\frac{S_l}{S_0}=C} (2l \cdot \sqrt[l]{C} + 2l - 1)$$

- To minimize solve for derivative = 0

$$\frac{d}{dl}(2l \cdot \sqrt[l]{C} + 2l - 1) = 0 \Rightarrow 2\sqrt[l]{C} - \frac{2\sqrt[l]{C} \cdot \log_e C}{l} + 2 = 0$$

$$\Rightarrow \sqrt[l]{C}\left(\frac{\log_e C}{l} - 1\right) = 1 \Rightarrow (e^{\log_e C})^{\frac{1}{l}}\left(\frac{\log_e C}{l} - 1\right) = 1$$

$$\Rightarrow e^{\frac{\log_e C}{l}}\left(\frac{\log_e C}{l} - 1\right) = 1 \xrightarrow{\frac{\log_e C}{l} = x} e^x(x-1) = 1$$

$$\Rightarrow (x-1)e^x e^{-1} = e^{-1} \Rightarrow (x-1)e^{x-1} = \frac{1}{e}$$

- With Lambert's function W, by definition $W(xe^x)=x$

$$W((x-1)e^{x-1}) = W(\frac{1}{e}) \Rightarrow x - 1 = W(\frac{1}{e}) \xrightarrow{x = \frac{\log_e C}{l}} \Rightarrow$$

$$\frac{\log_e C}{l} - 1 = W(\frac{1}{e}) \Rightarrow l = \frac{\log_e C}{W(\frac{1}{e}) + 1}$$

- Therefore, optimal f is

$$f = \sqrt[l]{C} = C^{\frac{1}{l}} = C^{\frac{1}{\frac{\log_e C}{W(\frac{1}{e})+1}}} = C^{\frac{W(\frac{1}{e})+1}{\log_e C}}$$

$$= (C^{\frac{1}{\log_e C}})^{W(\frac{1}{e})+1} = (e^{\log_e C \frac{1}{\log_e C}})^{W(\frac{1}{e})+1}$$

$$= e^{W(\frac{1}{e})+1} \qquad (24)$$

- Given that W(1/e) is about 0.5, we can get optimal **f = $e^{3/2}$ = ~4.8**
  - Simpler solution

$$\min_{\prod_{i=1}^{l} f_i = C} (2lf + 2l - 1) \approx \min_{\prod_{i=1}^{l} f_i = C} (lf)$$

  - Optimal $l = \log_e C \Rightarrow e^l = C$. Since $f^l = C \Rightarrow f = e$ ~= 2.718

- Constant device space $C = S_0 + \ldots + S_l$

$$\min_{S_0 + \ldots + S_l = C} H = \min_{S_0 + \ldots + S_l = C} (2\sum_{i=1}^{l} f_i + 2l - 1) \approx \min_{\sum_{i=0}^{l} S_i = C} \sum_{i=1}^{l} f_i$$

- Minimize with Lagrange multipliers
  - Masters thesis of Nikos Batsaras and arxiv TR

$$f_{l-2} = f_{l-1} + \frac{1}{f_{l-1}}$$

Similarly:

$$f_{l-3} = f_{l-2} + \frac{1}{f_{l-1}} \cdot \frac{1}{f_{l-2}}$$

$$\ldots$$

$$f_1 = f_2 + \frac{1}{f_{l-1}} \cdots \frac{1}{f_2}$$

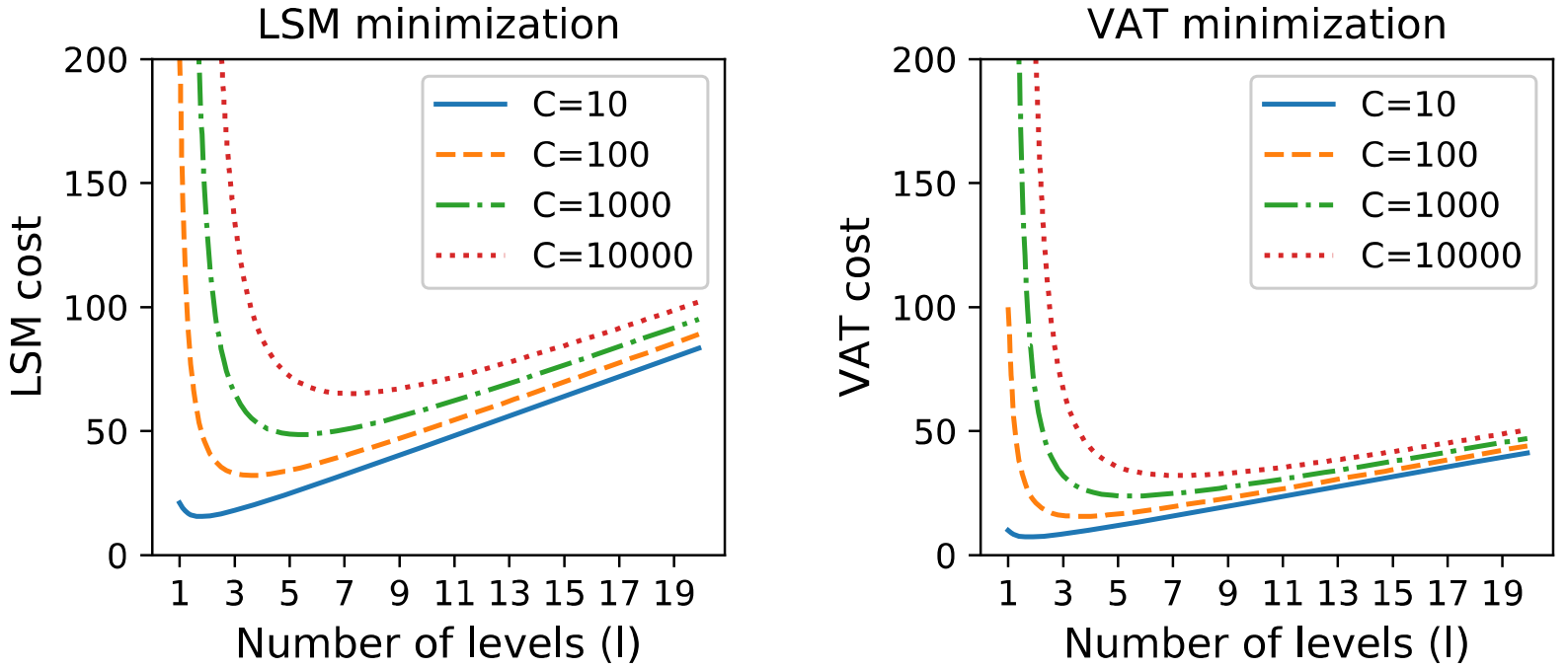
- E.g. l=5
  - f1=11.0998, f2=11.0991, f3=11.0909, f4=11, f5=10
  - For all practical purposes all equal to f5=10
- Even with total size assumption, for all practical purposes, optimal growth factor can be considered as constant across levels

$$\frac{d}{dl}\left(l \cdot \sqrt[l]{C}\right) = \sqrt[l]{C} + l\left(\log_e C \cdot \sqrt[l]{C} \cdot (-\frac{1}{l^2})\right)$$

$$= \sqrt[l]{C}\left(1 - \frac{\log_e C}{l}\right) = 0 \Rightarrow l = \log_e C \qquad (25)$$

LSM: Optimal growth factor and #levels

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



(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.

Reading

- Review
  - Slides + pointers, Your class notes
- Read
  - Sections 1,2,4,5 of P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. 1996. The log-structured merge-tree (LSM-tree). Acta Inf. 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048
- Next class
  - Paper discussion