Chucky: A Succinct Cuckoo Filter for LSM-Tree

Niv Dayan, Moshe Twitto
Pliops

ABSTRACT

Modern key-value stores typically rely on an LSM-tree in storage (SSD) to handle writes and Bloom filters in memory (DRAM) to optimize reads. With ongoing advances in SSD technology shrinking the performance gap between storage and memory devices, the Bloom filters are now emerging as a performance bottleneck.

We propose Chucky, a new design that replaces the multiple Bloom filters by a single Cuckoo filter that maps each data entry to an auxiliary address of its location within the LSM-tree. We show that while such a design entails fewer memory accesses than with Bloom filters, its false positive rate off the bat is higher. The reason is that the auxiliary addresses occupy bits that would otherwise be used as parts of the Cuckoo filter's fingerprints. To address this, we harness techniques from information theory to succinctly encode the auxiliary addresses so that the fingerprints can stay large. As a result, Chucky achieves the best of both worlds: a modest access cost and a low false positive rate at the same time.

ACM Reference Format:
Niv Dayan, Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree . In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457273

1 INTRODUCTION

Modern KV-stores rely on an LSM-tree to persist data in storage. An LSM-tree buffers new data in memory, flushes the buffer to storage as a sorted run whenever it fills up, and compacts runs across a logarithmic number of levels [79]. To optimize application point queries, there is an in-memory Bloom filter [11] for each run to rule out runs that do not contain a target entry. Such designs are used in OLTP [45], HTAP [71], social graphs [74], FTL design [12, 27], data series [61–63], blockchain [34], stream-processing [21], etc.

Problem 1: Changing Storage Media. LSM-tree was originally designed for HDDs, which are 5-6 orders of magnitude slower than DRAM memory chips. The advent of SSDs, however, has shrunk the performance gap between storage and memory to 2-3 orders of magnitude [10, 30, 37]. Today, a memory I/O takes $\approx$ 100 ns while a read I/O on, say, an Intel Optane SSD takes $\approx$ 10 μs. Hence, memory access is no longer negligible relative to storage access. For LSM-trees, especially modern designs with tens to hundreds of runs [59, 76, 85, 86, 98], querying a Bloom filter at $\approx$ 100 ns for each run can approach and even exceed the latency of the SSD I/O(s) that fetch the target entry from storage.

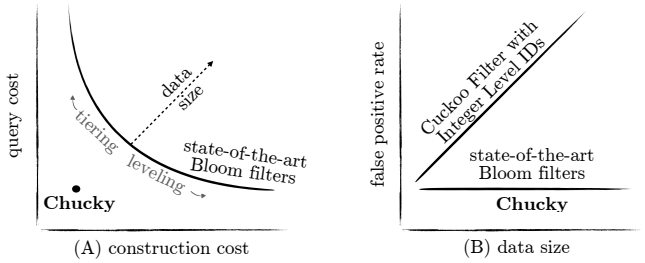


Figure 1: Existing filters for LSM-tree can not scale query cost, construction cost and the false positive rate all at once.

Problem 2: Workload Skew. KV-stores typically maintain an in-memory block cache to store frequently accessed data blocks and thereby optimize for skew, which is often common [41]. Querying a block cache still requires to first traverse potentially all the Bloom filters to identify the run that contains the target entry. As there are no storage I/Os in this case, the cost of traversing the Bloom filters becomes even more dominant.

Problem 3: Read vs. Write Contention. To mitigate the Bloom filters' query cost, one can tune the LSM-tree to merge more frequently so that there are fewer runs and thus fewer Bloom filters to access. This, however, increases the Bloom filters' construction costs. A Bloom filter is immutable and has to be rebuilt from scratch during each compaction. Compacting more frequently, therefore, entails rebuilding Bloom filters more frequently. It was recently reported that Bloom filter construction can amount to over 70% of performance overheads on the write path [58]. Thus, the access and construction costs of Bloom filters contend with each other in an LSM-tree context, as depicted conceptually in Figure 1 Part (A).

Problem 4: Scalability with Data Size. As the data size grows, more Bloom filters need to be queried and constructed across more LSM-tree levels. Hence, the overheads of querying and constructing Bloom filters grow too. As shown in Figure 1 Part (A), this causes the read vs. write trade-off curve to move outwards and leave applications with worse trade-offs to choose between. With data growing exponentially across many modern applications, the outcome is rapidly deteriorating performance.

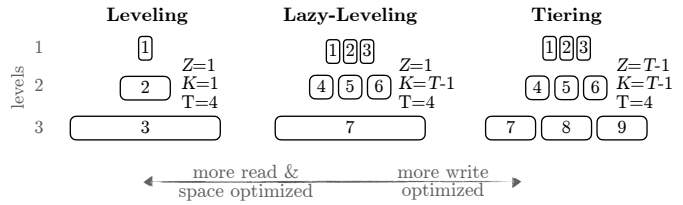
Research Question. Can we devise a replacement for Bloom filters that exhibits more robust and scalable performance for LSM-tree with respect to (1) storage media, (2) workload skew, (3) LSM-tree tuning, and (4) data size?

Cuckoo Filter: The Promise. Over the past decade, a new family of data structures emerged as an alternative to Bloom filters. These structures operate by storing a small hash digest called a fingerprint for every entry's key within a compact hash table. They include Quotient filter [9, 81], Cuckoo filter [39] and others [15, 44, 80, 96].

In this paper, we replace an LSM-tree's multiple Bloom filters by a Cuckoo filter variant that maps each data entry to both a

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457273



Term	Definition
N	largest level size (entries)
P	buffer size (entries)
T	LSM-tree size ratio
L	number of LSM-tree levels
Z	number of sub-levels at largest level
K	number of sub-levels at each of Levels 1 to L-1
Ai	number of sub-levels at Level i
A	number of sub-levels in the whole LSM-tree
M	filtering memory budget (bits/entry)

Figure 2: LSM-tree variants and terms used to describe them throughout the paper. Each rectangle represents a sub-level.

fingerprint and to an auxiliary *Level ID* (LID). A LID is a small string of bits that identifies where a given entry resides within the LSM-tree. It points to the level (or part of a level) that needs to be searched during a point read for a matching fingerprint within the Cuckoo filter. The promise is to reduce filtering memory I/Os to a small and constant number, an improvement over Bloom filters.

Scaling the False Positive Rate. Despite the promise of this approach, we also identify a challenge: keeping the false positive rate (FPR) low and stable. The FPR is inversely related to the fingerprint size. Under a constrained memory budget, the LIDs occupy bits that would otherwise be used as parts of the fingerprints. Worse, as the data grows, the number of levels in the LSM-tree grows too. This requires increasing the LID size (in bits) to still be able to identify every level (or part thereof). As a result, the LIDs "steal" more bits from the fingerprints as data grows. This causes the FPR to increase over time. The outcome is more storage I/Os, which harm performance. We illustrate the challenge conceptually in Figure 1 Part (B) using the curve labeled Cuckoo Filter with Integer Level IDs.

Insight: Level ID Compressibility. Our core insight in this paper is that, fortunately, LIDs are extremely compressible. The reason is the LSM-tree's exponential growth; most entries reside at larger levels. Hence, the distribution of LIDs within the Cuckoo filter is heavily skewed: most LIDs correspond to the largest level while exponentially fewer correspond to smaller levels. We can therefore encode the LIDs of larger levels with fewer bits than the LIDs of smaller levels to minimize the average LID size. The saved bits can be assigned to the fingerprints to keep them large.

Chucky. We present **C**hucky: **Hu**ffman **C**oded **Ke**y-Value Store, a new design that scales memory I/Os, memory footprint, and the false positive rate at the same time. It achieves this by replacing the Bloom filters by a Cuckoo filter with succinctly encoded (i.e., compressed) LIDs. We explore in detail the design space for LID compression, and we identify and tackle the resulting challenges: (1) how to align fingerprints and compressed LIDs within the Cuckoo filter's buckets, and (2) how to en/decode LIDs efficiently. We use the saved bits to keep the fingerprints large and thus to keep the FPR low and stable as the data grows.

Contributions. Our contributions are as follows.

(1) We show that the Bloom filters of LSM-tree are emerging as a memory I/O bottleneck as SSDs evolve and get faster.
(2) We replace the Bloom filters by a Cuckoo filter that maps data entries to their locations within the LSM-tree.
(3) We show that level IDs are extremely compressible. We study how to to efficiently en/decode them using Huffman coding.
(4) We show how to align compressed level IDs and fingerprints within Cuckoo filter buckets to ensure good space utilization.
(5) We show experimentally that Chucky scales memory bandwidth, memory footprint, and the FPR at the same time.

2 STATE-OF-THE-ART: LSM & BLOOM

LSM-tree consists of multiple levels of exponentially increasing capacities. Level 0 is an in-memory buffer (a.k.a memtable), typically implemented as a skip list or a hash table. All other levels are in storage. The application inserts key-value entries into the buffer. When the buffer fills up, it gets flushed to storage.

Merge Policy. An LSM-tree's merge policy dictates which data to merge in storage and when. While merge policies can be formalized in different ways, we adopt the Dostoevsky framework [28] as it generalizes several well-known policies. The number of levels L is $\lceil \log_T (N/P) \rceil$, where N is the number of entries at the largest level, P is the buffer size, and T is the capacity ratio between any two adjacent levels ($T \geq 2$). Each level consists of one or more sub-levels, where a sub-level is a placeholder for one chunk of sorted data. At the largest level, there are Z sub-levels ($1 \leq Z < T$). At each of the smaller levels, there are K sub-levels ($1 \leq K < T$). Figure 2 shows how sub-levels are numbered for different configurations of the parameters Z and K. The capacity at Level i is $P \cdot T^i$ entries, and it is equally divided among the sub-levels at Level i. Equation 1 denotes A_i as the number of sub-levels at Level i and A as the overall number of sub-levels. The number of levels L and the number of sub-levels A both grow with the data size.

$$A_i = \begin{cases} K & \text{for } 1 \leq i < L \\ Z & \text{else} \end{cases} \qquad A = \sum_{i=1}^{L} A_i = (L-1) \cdot K + Z \quad (1)$$

There is zero or one run at each sub-level. A run comprises key-value entries sorted by key. Different runs may have overlapping key ranges. Each run may further be divided into smaller files with non-overlapping key ranges called Sorted String Tables (SSTs), though we will use run as the unit of data in the paper.

When all sub-levels at Level i have reached capacity, their constituent runs get merged into the highest sub-level at Level $i+1$ that is below capacity. If there is already a run at this target sub-level, it is included in the merge. Hence, the j^{th} youngest run at Level i is always at sub-level number $(i-1) \cdot K + j$.

The parameters K and Z can be co-tuned to assume different trade-offs. Figure 2 shows how to tune them to assume three merge policies: (1) leveling, best for range reads, (2) tiering, best for writes, and (3) lazy leveling, best for point reads. The size ratio T can be tweaked to fine-tune these trade-offs, though Figure 2 fixes it to four. When the size ratio T is set to two, its lowest possible setting, the three merge policies behave identically. As we increase T with each policy, their behaviors diverge. With a vision towards navigable systems that can learn and adapt across a wide design space to optimize for different workloads [5, 6, 47, 52–57], we design Chucky to span this entire wide compaction design space.

Updates & Deletes. Updates and deletes are performed out-of-place by inserting a key-value entry with the updated value into

	Leveling	Lazy-Leveling	Tiering
application query	$O(L)$	$O(L \cdot T)$	$O(L \cdot T)$
application update	$O(L \cdot T)$	$O(L + T)$	$O(L)$

Table 1: Blocked Bloom filters' memory I/O complexities.

the buffer (for a delete, the value is a tombstone). Whenever runs that contain entries with the same key are merged, older versions are discarded and only the newest version is kept. To always find the most recent version of an entry, a query traverses the runs from youngest to oldest (from smaller to larger levels, and from lower to higher sub-levels within a level). It terminates when it finds the first entry with a matching key. If this entry's value is a tombstone, the query returns a negative result.

Fence Pointers. For each run, there are fence pointers in memory that comprise the min/max key at every data block. They allow queries to binary search for the relevant data block that contains a given key in $\approx \log(N)$ memory I/Os so that this block can be retrieved cheaply with one storage I/O.

Bloom Filters. For each run in the LSM-tree, there is an in-memory Bloom filter (BF), a space-efficient probabilistic data structure used to test whether a key is a member of a set [11]. A BF is an array of bits with h hash functions. Each key is mapped using these hash functions to h random bits, setting them from 0 to 1 or keeping them set to 1. Checking for the existence of a key requires examining its h bits. If any are set to 0, we have a *negative*. If all are set to 1, we have either a *true* or a *false* positive. The false positive probability (FPP) is $2^{-M \cdot \ln(2)}$, where M is the number of bits per entry. As we increase M, the probability of hash collisions decreases and so the FPP drops. A BF does not support range reads or deletes. The lack of delete support means that a new BF has to be built from scratch for every new run as a result of compaction.

A BF entails h memory I/Os for an insertion or for a query to an existing key. For a query to a non-existing key, it entails on average two memory I/Os since $\approx$ 50% of the bits are set to zero and so the expected number of bits checked before incurring a zero is two.

Blocked Bloom Filters. To optimize memory I/Os, Blocked Bloom filter has been proposed as an array of contiguous BFs, each the size of a cache line [66, 84]. A key is inserted by first hashing it to one of the constituent BFs and then inserting the key into it. This entails only one memory I/O for any insertion or query. The trade-off is a slight FPP increase. RocksDB has adopted blocked BFs. We use standard and blocked BFs as baselines in this paper, and we focus more on blocked BFs as they are the tougher competition.

Memory I/O Analysis. With blocked BFs, the overall cost of a point query is $(L-1) \cdot K + Z$ memory I/Os, one for each sub-level of the LSM-tree. The cost of an insertion/update/delete, on the other hand, is the same as the LSM-tree's write-amplification: $\approx \frac{T-1}{K+1} \cdot (L-1) + \frac{T-1}{Z+1}$ with Dostoevsky. The reason is that every compaction that an entry participates in leads to one BF insertion, which costs one memory I/O. Table 1 summarizes these costs for each of the merge policies. It shows that query cost over the BFs can be significant, especially with tiering and lazy leveling. Moreover, the BF's query and construction costs both increase with respect to the number of levels L and thus with the data size. Finally, there is an inverse relationship between the BFs' query and construction costs: the greedier we set the LSM-tree's merge policy to be (i.e., by fine-tuning the the parameters T, K and Z), query cost decreases as there are fewer BFs while construction cost increases as the BFs get rebuilt more frequently. Can we better scale these costs with respect to the data size while also alleviating their read/write contention?

False Positive Rate Analysis. We define the false positive rate (FPR) as the sum of FPPs across all filters. The FPR expresses the average number of I/Os due to false positives that occur per point query over the whole LSM-tree. Equation 2 expresses the FPR for most KV-stores, which assign the same number of bits per entry to all their BFs. This approach, however, was recently deemed sub-optimal. The optimal approach is to reassign $\approx$ 1 bit per entry from the largest level and to use it to assign linearly more bits per entry to filters at smaller levels [25, 26]. While this increases the largest level's FPP, it exponentially decreases the FPPs at smaller levels such that the overall FPR is lower, as expressed in Equation 3 [28].

$$FPR_{uniform} = 2^{-M \cdot \ln(2)} \cdot (K \cdot (L-1) + Z) \tag{2}$$

$$FPR_{optimal} = 2^{-M \cdot \ln(2)} \cdot Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \tag{3}$$

Equation 3 states that with the optimal approach, the relationship between memory and FPR is independent of the number of levels and thus of data size, unlike Equation 2. The reason is that as the LSM-tree grows, smaller levels are assigned exponentially lower FPRs thus causing the sum of FPRs to converge. It is imperative that any replacement we devise for the LSM-tree's Bloom filters either matches or improves on the FPR expressed in Equation 3.

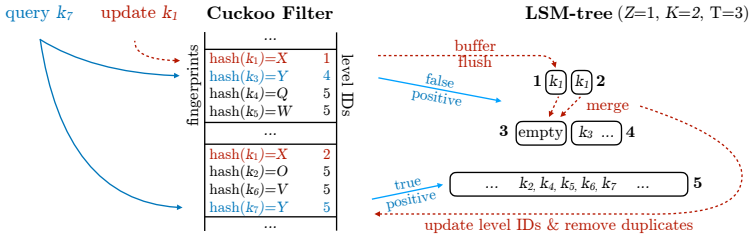
3 PROMISE: LSM-TREE & CUCKOO FILTER

Cuckoo filter [39] (CF) is one of several data structures [9, 15, 44, 81, 96] that recently emerged as alternatives to Bloom filters. In their core, these structures all employ a compact hash table that stores fingerprints of keys, where a fingerprint is a string of F bits derived by hashing a key. CF comprises an array of buckets, each with S slots for fingerprints. During insertion, an entry with key k is hashed to two bucket addresses b_1 and b_2 using Equation 4. A fingerprint of key k is inserted into whichever bucket has space. If both buckets are full, however, some fingerprint from one of the two buckets is randomly chosen and swapped to its alternative bucket to clear space. By virtue of using the xor operator, the right-hand side of Equation 4 allows to always compute an entry's alternative bucket using the fingerprint and current bucket address without the original key. The swapping process continues recursively either until a free slot is found for all fingerprints or until a swapping threshold is reached, at which point the insertion fails.

$$b_1 = hash(k) \qquad b_2 = b_1 \oplus hash(k's\ fingerprint) \tag{4}$$

We employ a CF with S set to four slots per bucket through the paper. Such a tuning can reach 95% space utilization with high probability without incurring insertion failures and with only 1-2 amortized swaps per insertion. The false positive rate is $\approx 2 \cdot S \cdot 2^{-F}$, where F is the fingerprint size in bits. Querying entails at most two memory I/Os as each entry is in one of two buckets.

Promise. Cuckoo filter supports storing updatable auxiliary data for each entry alongside its fingerprint. We propose to replace the LSM-tree's multiple Bloom filters with a CF that maps each data entry not only to a fingerprint but also to a Level ID (LID), mapping to the sub-level that contains the entry. The promise is to allow finding the run that contains a given entry with at most two memory I/Os, far more cheaply than with blocked Bloom filters.



Term	Definition
B	CF bucket size (bits)
F	fingerprint size (bits)
D	level ID size (bits)
S	slots per CF bucket
p_i	fraction of data at Level i
f_j	probability of Level ID j
H	entropy of level ID probability distribution
H_{comb}	entropy of combination probability distribution
C	the set of all level ID combinations
C_{freq}	the set of most probable level ID combinations

Figure 3: Chucky uses a Cuckoo filter to map entries in the LSM-tree, and it keeps this mapping up-to-date during compactions.

The FPR, meanwhile, is given in Equation 5 by subtracting the LID size D from the memory budget M. The challenge is keeping D small so that the FPR also stays small.

$$FPR \approx 2 \cdot S \cdot 2^{-F} = 2 \cdot S \cdot 2^{-M+D} \tag{5}$$

Case-Study. SlimDB [86] is the first system to replace an LSM-tree's Bloom filters by a Cuckoo filter with LIDs. It therefore provides an interesting case-study. SlimDB encodes LIDs as fixed-length integers. Each LID therefore comprises at least $\log_2(A)$ bits to identify all runs uniquely, where A is the total number of sub-levels from Equation 1. By plugging in $\log_2(A)$ as D in Equation 5 and substituting for A, the FPR simplifies into Equation 6, which indicates that the FPR increases with the number of levels L. The reason is that the LIDs take away bits from the fingerprints as the data grows to be able to identify more sub-levels uniquely. Is it possible to keep the FPR lower and stable as the data size grows?

$$FPR_{binary} \approx 2 \cdot S \cdot 2^{-M} \cdot (K \cdot (L-1) + Z) \tag{6}$$

Another issue with SlimDB is that it accesses storage before each application write to check if the entry in question exists. If so, it modifies the entry's LID within the CF to reflect updated entry's location. This entails a substantial overhead on the write path. Can we keep the LIDs up-to-date without read-before-write operations?

4 CHUCKY

Chucky is a novel LSM-tree filter that simultaneously scales memory bandwidth, memory footprint, and the false positive rate (FPR). It achieves this by replacing the Bloom filters by a Cuckoo filter (CF) variant that uses level IDs to map each entry to its sub-level within the LSM-tree. It further innovates along two directions.

Opportunistic Maintenance. Chucky keeps the level IDs within the CF up-to-date opportunistically during merge operations at no additional storage I/O cost. We discuss this in Section 4.1.

Level ID Compression. Chucky compresses level IDs to prevent their size from increasing and stealing bits from the fingerprints as the data grows. Thus, Chucky keeps the FPR low. We show how to compress level IDs in Section 4.2. We identify and address the implications of compressed level IDs on bucket alignment and computational efficiency in Sections 4.3 and 4.4, respectively. Section 4.5 covers miscellaneous design considerations .

4.1 Integration with LSM-Tree

Figure 3 illustrates Chucky's architecture. For every data entry in the LSM-tree, there is one CF entry consisting of a fingerprint and a level ID (LID) that maps the entry's current sub-level number. Figure 3 also illustrates Chucky's query and update workflows with solid blue arrows and dashed red arrows, respectively.

Querying. Chucky processes a query by accessing both CF buckets that the key maps to (using Equation 4). For all matching fingerprints within these two buckets, it searches the corresponding runs from youngest to oldest. In Figure 3, for example, the application queries for key k_7. Chucky maps this key to two buckets, both of which have one entry with a matching fingerprint Y. One maps to Sub-Level 4 and one to Sub-Level 5. The query searches Sub-Level 4 first as it contains younger data, but it incurs a false positive. It then searches Sub-Level 5, where it successfully finds the target entry. Since a query accesses two CF buckets, the overhead is two memory I/Os, irrespective of the data size or merge policy.

Inserting New Data. Whenever the memtable gets flushed to storage, Chucky adds a CF entry for each key in the batch (including for tombstones). The overhead is approximately two memory I/Os per entry as entries may be swapped across buckets to clear space. For example, consider entry k_1 in Figure 3, for which there is originally one version at Sub-Level 2. A new version of this entry is then flushed into Sub-Level 1. Chucky adds a new mapping entry for the new version while still keeping the older version's mapping in the CF. This is in contrast to SlimDB, which would issue a storage I/O to check if an entry with key k_1 exists and if so update the existing mapping entry's LID in the CF. Hence, Chucky removes SlimDB's read-before-write operation. The outcome is better performance. The trade-off is that Chucky has to map obsolete entries in its CF until compaction, differently from SlimDB but similarly to Bloom Filters, for which different versions of the same entry also take up space across multiple filters until compaction. A problem that can arise with Chucky is that CF buckets can overflow if too many obsolete versions of the same entry exist, as their mapping entries all get placed in the same pair of CF buckets. We handle such overflows in Section 4.5 through extension buckets.

Maintenance. As an entry moves into a new sub-level during a compaction, Chucky updates its LID in the CF as it is brought into memory to be merged. For every obsolete entry identified during compaction, on the other hand, Chucky finds and removes the corresponding mapping entry from the CF. In Figure 3, for example, compaction is triggered, merging the two runs at Sub-Levels 1 and 2 into one run at Sub-Level 3. During this operation, the older version of key k_1 from Sub-Level 2 is removed from the CF. For the newer version of k_1, the LID is updated from 1 to 3 to reflect the new sub-level. This approach does not involve any additional storage I/Os on top of the ones that are already issued for compaction. The memory access cost is 1.5 I/Os on average to find the target entry across the two possible CF buckets that may contain it.

Interestingly, an entry's LID does not need to be updated when the entry stays at the same sub-level after compaction. For example, suppose Sub-Levels 3 and 4 in Figure 3 now get merged with Sub-Level 5, and the resulting run stays at Sub-Level 5 as it does not

	Leveling	Lazy-Leveling	Tiering
application query	$O(1)$	$O(1)$	$O(1)$
application update	$O(L)$	$O(L)$	$O(L)$

Table 2: Chucky's memory I/O complexities.

exceed its capacity. In this case, the LIDs of entries k_2, k_4, k_5, k_6 and k_7 stay the same. By contrast, with Bloom filters we would have to rebuild the filter at Sub-Level 5 and reinsert each of these entries at a significant memory I/O cost. Hence, a LID is updated at most L times, once for each time it moves into a new level. The overall overhead is therefore at most $\approx 1.5 \cdot L$ amortized memory I/Os per application insert/update/delete, irrespective of the merge policy.

Memory I/O Complexities. Table 2 summarizes the memory I/O complexities of Chucky. Relative to the BFs in Table 1, Chucky's core advantage is reducing query cost to a small constant that is independent of the data size and of the merge policy. Chucky also reduces the update cost complexities for leveling and lazy leveling, thus eliminating the dependence of update cost on the merge policy. In practice, for tiering and lazy leveling, Chucky's update cost of $\approx 1.5 \cdot L$ memory I/Os per entry may be slightly more expensive than with blocked BFs. This, however, is counterbalanced by the substantial point read cost reduction.

Interplay with CPU Caching. For workloads with *point skew*, whereby the same data entries are repeatedly read by the application, Chucky can accommodate a larger working set within the CPU caches. The reason is that only two CF buckets need cached for any frequently accessed entry. With blocked BFs, however, at most A filters need to be cached for such an entry. On the other hand, for workloads with *areal skew*, whereby entries in the same temporal or spatial area are more likely to be read, BFs may better lend themselves to CPU caching as they are more granular temporally (across sub-levels) and spatially (across SSTs within a sub-level). In terms of update cost, BFs for smaller runs of the LSM-tree may in practice fit in the CPU caches and thus entail fewer memory I/Os than predicted in Table 1. The effects of CPU caching are subtle. While workloads favoring BFs may be envisioned, Chucky gives better worst-case guarantees and thus renders performance more robust across all cases and especially as the data grows.

4.2 Compressing Level IDs

This section establishes theoretical bounds on the compressibility of LIDs and explores their encoding design space in detail.

LID Probability Distribution. Equation 7 denotes p_i as the fraction of the LSM-tree's overall capacity at Level i. The expression on the left is more accurate but quickly converges to the expression on the right as the number of levels grows. As expected, capacities of smaller levels are exponentially decreasing.

$$p_i = \frac{T-1}{T^{L-i}} \cdot \frac{T^{L-1}}{T^L-1} \qquad \lim_{L\to\infty} p_i = \frac{T-1}{T} \cdot \frac{1}{T^{L-i}}, \tag{7}$$

Equation 8 denotes f_j as the fraction of the LSM-tree's capacity at Sub-Level j, which is a part of Level $\lceil j/K \rceil$. Equation 8 is derived by dividing the level's capacity $p_{\lceil j/K \rceil}$ (from Eq. 7) by the number of sub-levels $A_{\lceil j/K \rceil}$ at that level (from Eq. 1). For example, in Figure 3 Sub-Level 5 is at Level $\lceil 5/2 \rceil = 3$, and so it comprises a fraction of $f_3/A_3 \approx 0.62$ of the overall LSM-tree's capacity.

$$f_j = \frac{p_{\lceil j/K \rceil}}{A_{\lceil j/K \rceil}} \tag{8}$$

Let us assume that all sub-levels of the LSM-tree are filled up to capacity. Let us also assume that the average data entry size is the same in different runs. Under these two assumptions, Equation 8 gives the probability that a randomly selected LID from the Cuckoo filter corresponds to Sub-Level j. In other words, Equation 8 becomes a probability distribution of the LIDs within the CF.

The assumption that all sub-levels are full reflects the case where memory pressure is highest. To optimize memory footprint for the worst-case, we maintain this assumption for the rest of Section 4.

Entropy. Equation 9 derives the Shannon entropy of the LID probability distribution, which represents the average number of bits needed to represent a LID after maximal compression. We derive it by stating the definition of entropy on the left-hand side, plugging in Equations 8 for f_j, taking the number of sub-levels A and thus the data size to infinity, and simplifying. Interestingly, the entropy converges with respect to the number of sub-levels A and hence with the data size. The intuition is that the exponential decrease in LID probabilities for smaller levels trumps the fact that LIDs at smaller levels would require more bits to represent uniquely.

$$H = \lim_{A\to\infty} \sum_{j=1}^{A} -f_j \cdot \log_2(f_j) = \log_2\left(Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1}\right) \tag{9}$$

Chucky's FPR Lower Bound. By plugging in the entropy H in Equation 9 as the LID size D in Equation 5, we obtain an optimistic FPR approximation for Chucky in Equation 10. This is the lowest we may expect the FPR to be by virtue of compressing each LIDs as much as possible and assigning all remaining bits to the fingerprints. We observe that this bound is asymptotically lower with respect to data size than the FPR for a CF with integer encoded LIDs in Eq. 6. It is also asymptotically lower than the FPR upper bound for uniformly allocated Bloom filters in Eq. 2. Finally, in comparison to the FPR upper bound with optimally allocated Bloom filters in Equation 3, we observe that while Equation 10 has a higher multiplicative constant of $2 \cdot S$, the FPR decreases more quickly with respect to memory (i.e., $\propto 2^{-M}$ as opposed to $\propto 2^{-M \cdot \ln(2)}$). This implies that for a high enough memory budget ($M \geq 10$ bits per entry), Chucky should be able to beat state-of-the-art Bloom filters in terms of FPR. This theoretical finding reaffirms our approach.

$$FPR_{chucky} \gtrsim 2 \cdot S \cdot 2^{-M} \cdot Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \tag{10}$$

Huffman Coding. Chucky uses Huffman coding [46] to compress LIDs in practice. As input, the Huffman encoder takes the LIDs and their probability distribution (i.e., Eq. 8) for a particular LSM-tree configuration (i.e., of T, K, Z and L). As output, it returns a code to represent each LID, where LIDs with a higher probability are assigned shorter codes. It does this by creating a binary tree from the LIDs by connecting the least probable LIDs first as sub-trees. A LID's ultimate code length corresponds to its depth in the resulting tree. Figure 4 illustrates an example for an LSM-tree with labeled LIDs and their probabilities from Eq. 8. For example, LID 6 contains a fraction of $5/124 \approx 4\%$ of the LSM-tree's capacity, and therefore, when all sub-levels are full, $\approx 4\%$ of all entries in the Cuckoo filter have a LID of 6. The Huffman encoder creates the tree shown alongside for this LSM-tree instance. The code for a given LID is derived by concatenating the tree's edge labels on the path

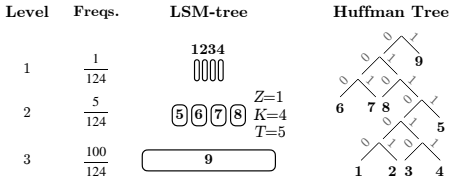


Figure 4: A Huffman tree encodes each LID uniquely such that the average code length is minimized.

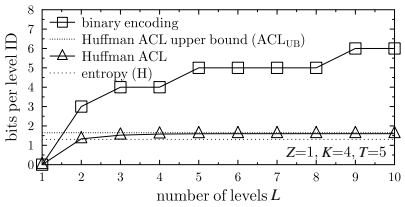


Figure 5: Compression causes the LIDs' average code length to converge with respect to data size.

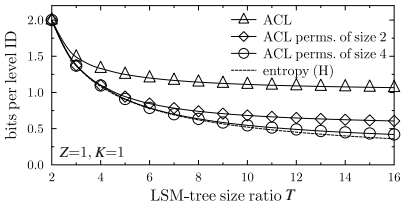


Figure 6: The average code length approaches the entropy as larger permutations of LIDs are used.

from the tree's root to the given LID's leaf node. For instance, the codes for LIDs 4 and 9 are 011011 and 1, respectively.

Decodability. With Huffman coding, no code is a prefix of another code [46]. This property allows for unique decoding of an input bit stream by traversing the Huffman tree starting at the root until we reach a leaf, outputting the LID at the given leaf, and then restarting at the root. For example, the input bit stream 11001 gets uniquely decoded into LIDs 9, 9 and 7 based on the Huffman tree in Figure 4. This property allows us to uniquely decode all LIDs within a bucket without the need for delimiting symbols.

Average Code Length. We measure the encoded LIDs' size using their average code length (ACL), defined as $\sum_{j=1}^{A} l_j \cdot f_j$, where l_j is the code length assigned to LID j. For example, this equation computes 1.52 bits for the Huffman tree in Figure 4. This is a saving of 62% relative to integer encoding, which would require four bits to represent each of the nine LIDs uniquely.

Memory Footprint Analysis. It is well-known in information theory that an upper bound on a Huffman code's ACL is the entropy plus one [46]. The intuition for adding one is that each code length must be rounded up to an integer number of bits. We express this as $ACL \leq H + 1$, where H is the entropy from Eq. 9. We therefore expect the ACL to also converge and become independent of the data size, similarly to Eq. 9. We verify this in Figure 5 by increasing the number of levels for the example in Figure 4 and plotting the Huffman ACL, which indeed converges (in contrast to integer-encoded LIDs). The reason is that while runs at smaller levels are assigned longer codes, they are exponentially less probable, so the smaller codes of runs at larger levels dominate the ACL.

Tight ACL Upper Bound. Huffman coding is known to be optimal in that it minimizes the ACL [46]. However, the precise ACL is difficult to analyze because the Huffman tree structure is difficult to predict from the onset. Instead, we can derive an even tighter upper bound on the ACL than $H + 1$ by assuming a less generic coding method and observing that the Huffman ACL will be at least as short. Let us represent each LID using (1) a unary encoded prefix of length $L - i + 1$ bits to represent Level i followed by (2) a truncated binary encoding suffix of length $\approx \log_2(A_i)$ to represent each of the A_i sub-levels at Level i uniquely. This is effectively a Golomb encoding [43]. We derive this encoding's average length in Equation 11 as ACL_{UB} and illustrate it in Figure 5 as a reasonably tight upper bound of the Huffman ACL.

$$ACL_{UB} = \lim_{L\to\infty} \sum_{i=1}^{L} p_i \cdot \left(L - i + 1 + \log_2(A_i)\right) = \frac{T}{T-1} + \log_2(K^{\frac{1}{T}} \cdot Z^{\frac{T-1}{T}}) \quad (11)$$

Proximity to Entropy. Figure 5 also plots the entropy of the LID probability distribution from Eq. 9. As shown, there is a gap between the Huffman ACL and the entropy. Figure 6 shows that as we increase the LSM-tree's size ratio T, the gap between the ACL and the entropy grows; the ACL approaches one while the entropy tends towards zero. The reason is that a larger size ratio increases the skew of the LID probability distribution by pushing a higher proportion of the data to larger levels. With more skew, each LID carries less information, leading to a lower entropy and thus higher compressibility. However, each LID requires at least one bit to represent with a code, and so the ACL cannot drop below one. Hence, we cannot harness the increase in compressibility.

Level ID Permutations. As there are multiple LIDs at each CF bucket, we can push the compression barrier of one bit per LID by encoding multiple LIDs collectively. The Huffman Tree labeled Perms. Figure 7 gives a toy example of how to encode two LIDs at a time as permutations, obtained by feeding every possible permutation of size S (two in this case) along with its probability (the product of the constituent LIDs' probabilities) into a Huffman encoder. As shown, the ACL now drops below one bit by virtue of representing the most probable permutation with fewer bits than the number of LIDs within it. Interestingly, Figure 6 shows that as we increase the number of collectively encoded LIDs within a permutation, the ACL approaches the entropy.

Level ID Combinations. To push compression even further, we can encode combinations as opposed to permutations of LIDs. A combination, unlike a permutation, disregards information about the ordering of entries. As there are fewer possible combinations than permutations of LIDs within a CF bucket ($\binom{S+A-1}{S}$ as opposed to A^S), we need fewer bits on average to represent them.

The probability distribution of LID combinations is multinomial. For n independent trials, each leading to a success for one of k categories, with each category having a fixed success probability, the multinomial distribution gives the probability of any particular combination of successes across the various categories. In our case, the number of trials is the number of slots S per CF bucket, the different categories are the A LIDs, and the success probabilities are given by the LID probability distribution in Equation 8.

Now, let us denote $c(j)$ as the number of occurrences of LID j within a combination c. Equation 12 gives c_{prob} as the probability of combination c using the multinomial distribution. By feeding all combinations and their probabilities into a Huffman encoder for the example in Figure 7, we obtain the Huffman tree titled Combs, where the combination 12 replaces the two prior permutations 12 and 21. For this combination, we have $S = 2$, $c(1) = 1$ and $c(2) = 1$, and so its probability is $2! \cdot (1/11) \cdot (10/11) = 20/121$.

$$c_{prob} = S! \cdot \prod_{j=1}^{A} \frac{f_j^{c(j)}}{c(j)!} \quad (12)$$

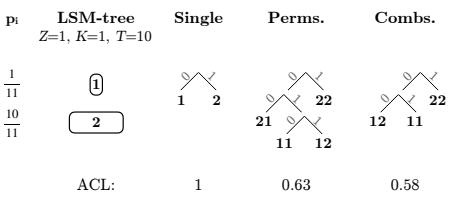


Figure 7: Encoding level IDs as permutations or combinations allows reducing the average code length below one.

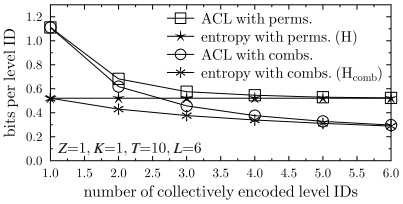


Figure 8: Encoding level IDs as large combinations maximizes compressibility.

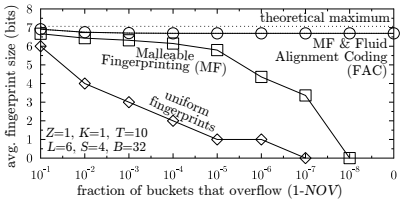


Figure 9: Chucky resolves the contention between fingerprint size and bucket overflows via FM and FAC.

Combinations Analysis. The fact that combinations exclude information about ordering causes a reduction in entropy. Equation 13 derives H_{comb} as the entropy of the LID combinations distribution by using the multinomial distribution's entropy function and plugging in S, A, and Eq. 9. Figure 8 shows that as we increase the combination size, H_{comb} drops relative to H as it eliminates more ordering information. This leads to an increase in compressibility.

$$H_{comb} = H - \frac{1}{S} \cdot \left(\log_2(S!) - \sum_{i=1}^{A} \sum_{j=0}^{S} \binom{S}{j} \cdot f_i^j \cdot (1-f_i)^{S-j} \cdot \log_2(j!) \right) \quad (13)$$

The ACL with combinations is $\sum_{c \in C}(l_c \cdot c_{prob})/S$ where C is the set of all combinations and l_c is the code length for Combination c (we divide by S to express the ACL per LID rather than per bucket). We observe that the combinations ACL is strictly lower than the permutations ACL in Figure 8, and that it converges with the combinations entropy as we increase the number of collectively encoded LIDs. In the rest of the paper, we continue with encoded combinations as they achieve the best compression.

Bucket Structure. Each CF bucket in Chucky commences with one combination code followed by S fingerprints. Since the combination code excludes information about ordering, the fingerprints within the bucket are sorted based on their LIDs in order to be able to infer which fingerprint corresponds to which LID.

4.3 Aligning Level ID Codes with Fingerprints

Since LIDs are variable-length due to compression, aligning them along with fingerprints within CF buckets becomes a challenge. We depict this challenge in Figure 10 Part (A) with sixteen-bit CF buckets that need to store one combination code for two entries along with two five-bit fingerprints (FPs). This example is based on the LSM-tree instance in Figure 4 except we now encode combinations rather than every LID individually. The term $l_{x,y}$ in the figure is the code length assigned to a bucket with LIDs x and y. We observe that some codes and fingerprints perfectly align within a bucket (Row I). However, others exhibit underflows (Row II), meaning some bits at the end of the bucket are unused. Still other buckets exhibit overflows (Rows III and IV), meaning the cumulative length of the code and the fingerprints exceeds the bucket's size. Underflows occur at buckets with more probable LIDs (belonging to larger levels) as a result of having shorter combination codes. They are undesirable as they waste bits that could have otherwise been used for having larger fingerprints. On the other hand, overflows occur in buckets with less probable LIDs (belonging to smaller levels) as a result of having longer combination codes. They are undesirable as they require storing the rest of the bucket contents elsewhere. This can result in poorer memory utilization and higher access costs.

Figure 10 Part (A) implies that there is a contention between the propensity of buckets to overflow and the fingerprint size. While decreasing the fingerprint size alleviates overflows in some buckets, it results in a higher false positive rate for the filter as a whole. We substantiate this contention in Figure 9 with the curve labeled *uniform fingerprints*. The x-axis measures the fraction of overflowing CF buckets while the y-axis measures the fingerprint size. Is it possible to eliminate this contention so as to guarantee few overflows and large fingerprints at the same time?

Malleable Fingerprinting (MF). To enable better alignment of codes and fingerprints, we introduce MF. The goal is to counterbalance the fact that entries from larger levels tend to have smaller combination code lengths and to use the spare space in the bucket for having longer fingerprints. Thus, MF assigns entries at smaller levels of the LSM-tree shorter fingerprints. As they get merged into larger levels, however, they get assigned longer fingerprints.

The question with MF is how to choose a fingerprint length for each level so as to carefully control the balance between fingerprint lengths and bucket overflows. We frame this as an constrained optimization problem, where the objective is to maximize the average fingerprint length, $\sum_{i=1}^{L} FP_i \cdot p_i$, and where FP_i be an integer denoting the length of fingerprints of entries at Level i. The problem is defined for $2^B > \binom{S+A-1}{S}$, meaning the bucket size B has to be at least large enough to identify all combinations uniquely. We constrain the problem using a parameter NOV for the fraction of **n**on-**ov**erflowing buckets we wish to guarantee (ideally at least 0.9999). We use this parameter to define C_{freq} as a subset of C that contains only the most probable LID combinations in C such that their cumulative probabilities fall just above NOV[1]. We define Equation 14 as a constraint requiring that for all $c \in C_{freq}$, the code length (denoted l_c) plus the cumulative fingerprint length (denoted c_{FP}) do not exceed the number of bits B in the bucket[2].

$$\forall c \in C_{freq}: \quad c_{FP} + l_c \leq B \quad (14)$$

While optimization problems involving integers are known to be difficult to solve, we exploit the particular structure of our problem with an effective hill-climbing approach shown in Algorithm 1. The algorithm initializes all fingerprint lengths to zero. It then increases larger levels' fingerprint sizes as much as possible, moving to a next smaller level if the overflow constraint in Equation 14 is violated.

[1]Formally, C_{freq} is defined such that $\min_{c \in C_{freq}} c_{prob} \geq \max_{c \notin C_{freq}} c_{prob}$ and $NOV \leq \sum_{c \in C_{freq}} c_{prob} \leq NOV + \min_{c \in C_{freq}} c_{prob}$.

[2]More concretely, for a combination c let $c(j)$ denote the number of occurrences of the j^{th} LID. Then c's cumulative fingerprint length is $c_{FP} = \sum_{j=1}^{A} FP_{\lceil j/K \rceil} \cdot c(j)$. The term $FP_{\lceil j/K \rceil}$ is the fingerprint size set to the j^{th} LID, which is at Level $\lceil j/K \rceil$.

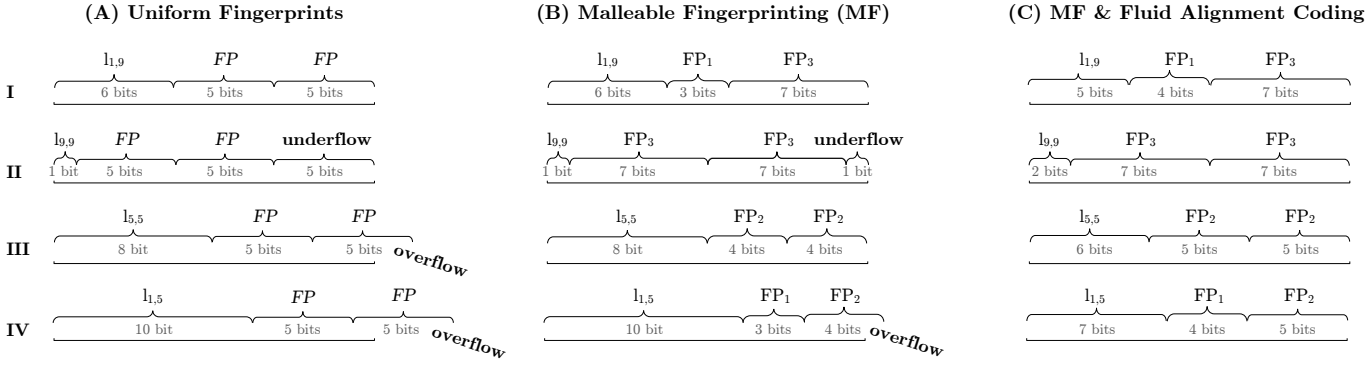


Figure 10: Storing compressed level ID codes with uniformly sized fingerprints leads to poor bucket alignment (Part A). We solve this problem using Malleable Fingerprinting (Part B) and Fluid Alignment Coding (Part C).

for $i \leftarrow 1$ **to** L **by** 1 **do** $FP_i \leftarrow FP_{min}$ **end**
$FP_{max} = M - 1$
for $i \leftarrow L$ **to** 1 **by** -1 **do**
  for $b \leftarrow FP_{min} + 1$ **to** FP_{max} **by** 1 **do**
    temp $\leftarrow FP_i$
    $FP_i \leftarrow b$
    if *overflow constraint is violated* **then**
      $FP_i \leftarrow$ temp
      FP_{max} = temp

Algorithm 1: Maximizing the average fingerprint size by hill-climbing.

The rationale for lengthening larger levels' fingerprints first is that their entries are more common in the CF. Hence, the algorithm follows the steepest ascent with respect to maximizing the objective function. Figure 10 Part (B) gives an example of how MF allows for entries from larger levels to have longer fingerprints (Row II) while at the same time eliminating some overflows (Row III). The result is a better balance between overflows and average fingerprint size as shown in Figure 9.

Since MF assigns different fingerprint lengths to different versions of the same entry across different levels, a problem arises whereby the Cuckoo filter can map these different versions of the same entry to more than two CF buckets. The reason is that Equation 4 relies on an entry's fingerprint to compute the alternative bucket location, and so different fingerprint lengths would lead to different bucket addresses. We resolve this by ensuring that all fingerprints comprise at least FP_{min} bits, and we adapt the CF to determine an entry's alternative bucket based on its first FP_{min} bits. This forces all versions of the same entry to reside in the same pair of CF buckets. While this constraint slightly reduces the average fingerprint size given by Algorithm 1, it provides a lower FPR variance as no entries are assigned very small fingerprints. In accordance with the original CF paper [39], we set FP_{min} to five bits to ensure that an entry's two buckets are independent enough to achieve a 95% space utilization.

Fluid Alignment Coding (FAC). Figure 10 Part (B) illustrates that even with MF, underflows and overflows still occur (Rows II and IV, respectively). To further mitigate them, we introduce FAC. FAC exploits a well-known trade-off that the smaller some codes are assigned within a Huffman code, the longer other codes must be for all codes to remain uniquely decodable. This trade-off is embodied in the Kraft-McMillan inequality [64, 75], which states that for a given set of code lengths $\mathcal{L}$, all codes can be uniquely decodable if $1 \geq \sum_{l \in \mathcal{L}} 2^{-l}$. The intuition is that code lengths are set from a budget amounting to 1, and that smaller codes consume a higher proportion of this budget.

To exploit this trade-off, FAC assigns longer codes to occupy the underflowing bits for the most probable bucket combinations. As a result, the codes for less probable bucket combinations can be made shorter. This creates more space in less probable buckets, which is exploited to reduce overflows and to increase fingerprint sizes for smaller levels. Figure 10 Part (C) illustrates this idea. The combination in Row II, which is the most common in the system, is now assigned a longer code by one bit to remove the underflow. This allows reducing the code lengths for all other combinations, which in turn allows setting longer fingerprints to entries at Levels 1 and 2 as well as to eliminate the bucket overflow in Row IV.

We implement FAC on top of MF as follows. First, we replace the previous constraint in Equation 14 by a new constraint, given in Equation 15. Expressed in terms of the Kraft-McMillan inequality, it ensures that the fingerprint sizes stay short enough such that it is still possible to construct non-overflowing buckets with uniquely decodable codes for all combinations in C_{freq}. It also ensures that all bucket combinations not in C_{freq} can be uniquely identified using unique codes that are at most the size of a bucket B.

$$1 \geq \sum_{c \in C} \begin{cases} 2^{-(B - c_{FP})}, & \text{for } c \in C_{freq} \\ 2^{-B}, & \text{else} \end{cases} \tag{15}$$

Equation 15 does not rely on knowing Huffman codes in advance (i.e., as Equation 14 does). Thus, we run the Huffman encoder after rather than before finding the fingerprint lengths using Algorithm 1. Furthermore, we run the Huffman encoder only on combinations in C_{freq} while setting the probability input for a combination c as $2^{-(B - c_{FP})}$ as opposed to using its multinomial probability (in Equation 12) as before. This causes the Huffman encoder to generate codes that exactly fill up the leftover bits $B - c_{FP}$. For all combinations not in C_{freq}, we set uniformly sized binary codes of size B bits, which consist of a common prefix in the Huffman tree and a unique suffix. Hence, we can decode both sets uniquely.

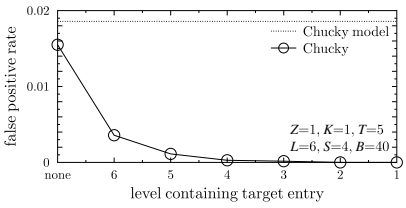


Figure 11: The FPR decreases exponentially as newer entries are accessed by queries.

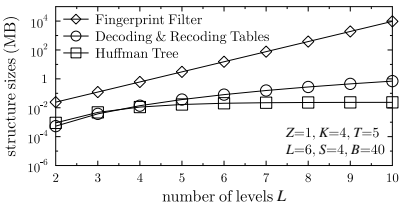


Figure 12: The Huffman tree size converges while the de/recoding table sizes grow slowly with data size.

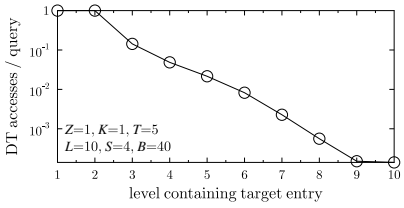


Figure 13: The decoding table access cost increases slowly with data size until flattening at one memory I/O.

The horizontal curve in Figure 9 shows that MF and FAC eliminate the contention between overflows and fingerprint size when applied together; fingerprints stay long and overflows stay rare at the same time. The trade-off is that the average code length becomes slightly longer than before. The reason is that by occupying the underflowing bits of the most probable combination codes, FAC makes the ACL at least S bits long ($\geq$ 1 bit per entry). This means that achieving good bucket alignment requires sacrificing some space. Figure 9 measures this sacrifice as the gap between the curve for MF & FAC and the curve labeled theoretical maximum, obtained by subtracting the entropy (from Eq. 13) from the memory budget M. It stands as $\approx 1/2$ bit per entry for our example, a modest price. We use MF and FAC by default for the rest of the paper.

Construction Time. The run-time complexity of Algorithm 1 is $O((L + M - M_{min}) \cdot |C|)$, where $L + M - M_{min}$ is the number of iterations and $|C|$ is the cost of evaluating the constraint in Equation 15. In addition, the time complexity of the Huffman encoder is $O(|C_{freq}| \cdot \log_2(|C_{freq}|))$. To express these bounds more loosely but in closed form, note that $|C_{freq}| \leq |C| = \binom{A+S-1}{S} < (A + S - 1)^S \cdot (\frac{e}{S})^S < A^S$. This workflow is seldom invoked, only when number of LSM-tree levels changes, and it can be performed offline. Its run-time is therefore practical. Each of the points in Figure 9 takes a fraction of a second to generate.

False Positive Rate (FPR). Chucky's FPR is tricky to precisely analyze because the fingerprints have variable sizes that are not known in advance. Instead, we conservatively approximate the FPR to still allow reasoning about system behavior. We use the ACL upper bound ACL_{UB} from Equation 11 to slightly overestimate the average combination code length per entry with FAC. By plugging in ACL_{UB} for D in Equation 5, we obtain Equation 16, for which the interpretation is the expected number of false positives for a query to a non-existing key.

$$FPR_{chucky} \approx 2 \cdot S \cdot 2^{-M} \cdot 2^{\frac{T}{T-1}} \cdot K^{\frac{1}{T}} \cdot Z^{\frac{T-1}{T}} \quad (16)$$

Figure 11 compares Equation 16 to Chucky's actual FPR. The x-axis varies the level whereon the target entry resides, where 6 is the ID of the largest level and 'none' means the target entry does not exist. The y-axis measures the average number of false positives incurred per query. The FPR drops exponentially when the target entry is at smaller levels. The reason is that a point read accesses the relevant levels from smallest to largest (to be able to find the most recent version of the entry), and it terminates once it finds the first matching entry. As a result, exponentially fewer false positives take place as there are, on average, exponentially fewer entries within the target two buckets that correspond to even smaller levels than where the target entry resides. The figure demonstrates that Equation 16 predicts within reason the case where the target entry does not exist, and that it provides a reliable upper bound in all cases where the entry does exist.

4.4 Optimizing Decoding & Recoding

We now discuss how to efficiently decode and recode combinations codes during application reads and writes.

Cached Huffman Tree. A Huffman code is typically decoded one bit at a time by traversing the Huffman tree from the root to a leaf node. A possible problem is that traversing it can require as much as one memory I/O per node visited. This cost grows with the data size as the Huffman tree becomes deeper when there are more levels. To restrict this cost, we observe that the bucket combination distribution in Equation 12 is heavy-tailed. Hence, it is feasible to keep a small Huffman Tree in the CPU caches to allow to quickly decode only the most common combination codes. Hence, we only construct a Huffman tree for the most common LID combinations within the set C_{freq}, and we set the parameter NOV to 0.9999 so that the set C_{freq} comprises 99.99% of the most common combinations within the CF. Figure 12 shows that the corresponding Huffman tree's size converges with respect to the data size. The reason is that the probability of a given bucket combination (in Eq. 12) is convergent with respect to the number of levels, and so any set whose size is defined in terms of its constituent members' cumulative probabilities is also convergent in size with respect to the number of levels. This property ensures that the Huffman tree does not exceed the CPU cache size as the data grows.

Decoding Table (DT). In addition to the Huffman tree, there is a Decoding Table in main memory to allow decoding combinations codes not in C_{freq}. To ensure fast decoding speed for DT, we exploit the property from in the last subsection that all bucket combinations not in C_{freq} have uniformly sized codes. Hence, we structure DT as an array whereby index i contains the LIDs that code i corresponds to. This guarantees decoding speed in one memory I/O. Figure 12 measures the DT size as we increase the number of levels on the x-axis (each DT entry is eight bytes). As DT contains $\approx |C| = \binom{A+S-1}{S}$ entries, its size grows slowly with the data size and stays smaller than 1MB even for a large LSM-tree instance with ten levels.

When point queries target data at smaller levels, the DT is more likely to be accessed. Figure 13 varies the level whereon the target entry resides and measures on the log-scale y-axis the average number DT accesses per query. The reason for the increase in access cost for entries at smaller levels is that a bucket that has at least one LID corresponding to a smaller level is less likely to be in

the set C_{freq} and hence in the cached Huffman tree. However, this overhead eventually flattens and therefore stays modest even in the worst-case. Overall, compared to a plain Cuckoo, which accesses on average 1.5 buckets before finding a matching fingerprint, Chucky always accesses two buckets and potentially also the decoding table, leading to three memory I/Os.

Overflow Hash Table. To handle bucket overflows, we use a small hash table to map from an overflowing bucket's ID to the corresponding fingerprints. Its size is $\approx (1 - NOV) = 10^{-4}$ of the CF size. It is accessed seldom, i.e., only for infrequent bucket combinations, and it supports access in $O(1)$ memory I/O.

Recoding Table (RT). To find the correct code for a given combination of LIDs while handling application writes, we employ a Recoding Table, implemented as a fast static hash table. It costs at most O(1) memory I/O to access and its size scales the same as the Decoding Table in Figure 12. Note that the most frequent RT entries are in the CPU caches during run-time and thus cost no memory I/Os to access.

Space Summary. Figure 12 illustrates the CF size as we increase the number of LSM-tree levels. All auxiliary data structures are comparatively small and therefore not space bottlenecks.

4.5 Additional Design Considerations

This section discusses additional design considerations.

Sizing & Resizing. When Chucky reaches capacity, it needs to be resized to accommodate new data. Since a CF needs to access the base data in order to be resized, we exploit the fact that merge operations into the largest level of the LSM-tree pass over the entire dataset. We use this opportunity to also build a new and larger instance of Chucky.

Partitioning. Since Cuckoo filter relies on the xor operator to locate an entry's alternative bucket, the number of buckets must be a power of two. This can waste up to 50% of the allotted memory, especially whenever LSM-tree's capacity just crosses a power of two. To keep memory better-utilized, Vacuum filter [96] proposes partitioning a CF into multiple independent CFs, each of which is a power of two, but where the overall number of CFs is flexible. Each key is mapped to one of the constituent CFs using a hash modulo operation (similarly to blocked Bloom filters). In this way, capacity becomes adjustable by varying the number of CFs. While Chucky does not support this yet, it is an important future step for memory-sensitive applications.

Empty CF Slots. We represent empty fingerprint slots using a reserved all-zero fingerprint coupled with the most frequent LID to minimize the corresponding combination code length.

Entry Overflows. Since a CF maps multiple versions of the same entry from different LSM-tree runs into the same pair of CF buckets, a bucket overflow can take place if there are more than $2 \cdot S$ versions of a given entry. Some filters address this problem using embedded fingerprint counters (e.g., Counting Quotient Filter [81]). Chucky, however, uses an additional hash table (AHT), which maps from bucket IDs to the overflowing entries. With insertion-heavy workloads, AHT stays empty. Even with update-heavy workloads, AHT stays small since LSM-tree by design limits space-amplification and thus the average number of versions per entry (e.g., at most $\frac{T}{T-1} \leq 2$ with Leveling or Lazy Leveling). We check AHT for every full CF bucket encountered during a query or update thus adding to them at most $O(1)$ additional memory access.

Persistence. For each run, Chucky persists the fingerprints of all entries in storage. During recovery, it reads only the fingerprints from storage and thus avoids a full scan over the data. It inserts each fingerprint along with its LID into a brand new CF at a practically constant amortized memory I/O cost per entry. In this way, recovery is efficient in terms of both storage and memory I/Os.

Range Reads. Similarly to mainstream KV-stores [3, 4, 38], Chucky processes a range read by accessing the relevant key range at each run without using the cuckoo filter. Range reads are therefore not directly affected by this work. Note, however, that applications with many range reads often opt for a leveled LSM-tree, whereon Bloom filters constitute high construction overheads. Chucky can indirectly improve performance for such applications by improving write throughput and thus system performance as a whole.

Batch Updates. Chucky can support batch updates by (1) atomically inserting a batch into the WAL and the memtable, (2) inserting all entries in the batch into the CF, (3) asynchronously flushing the memtable to storage when it is full, and finally (4) atomically removing the memtable from the read path.

5 EVALUATION

We now show experimentally that Chucky renders memory and storage bandwidth more robust than with existing designs.

Baselines. We use our own LSM-tree implementation, designed based on Dostoevsky [28]. We added as baselines blocked [84] and non-blocked BFs with uniform false positive rates (FPRs) to represent design decisions in RocksDB [38] and Cassandra [3], respectively. We also support optimal FPRs [25]. We implemented Chucky as described in Section 4. We support a version of Chucky with uncompressed level IDs to loosely represent SlimDB [86].

Setup. The default setup consists of a Lazy-Leveled LSM-tree with a 1MB buffer, a size ratio of five, and with six levels amounting to $\approx$ 16GB of data. Each entry is 64B. There is a 1GB block cache, and the database block size is 4KB. Chucky uses ten bits per entry and 5% over-provisioned space. Hence, all BF baselines are assigned a factor of $1/0.95$ more memory to equalize memory across the baselines. Every point in the figures is an average of three experimental trials. We use a uniform workload distribution to represent worst-case performance and a Zipfian distribution to create skew and illuminate performance properties when the most frequently accessed data is in the block cache. To account for filter resizing overheads, any experiment that measures write cost commences with an LSM-tree state whereby all levels but the largest are empty. We then fill them up with writes until a major compaction into the largest level occurs, leading to filter resizing. In Figure 14 Parts (A) to (D), we evaluate filter performance in isolation from other parts of the system (e.g., memtable, storage I/Os, block cache, block index). We focus on end-to-end performance in Parts (E) to (H).

Platform. Our machine has 32GB DDR memory, Xeon E3-1505M v5 with four 2.8 GHz cores and 8MB L3 caches. It runs Ubuntu 18.04 LTS and is connected to a 750GB Intel Optane SSD DC P4800X.

Memory I/O Scalability. Figure 14 Part (A) compares read/write latency with Chucky against blocked and non-blocked BFs (both with optimal FPRs) with a uniform workload as the data grows.

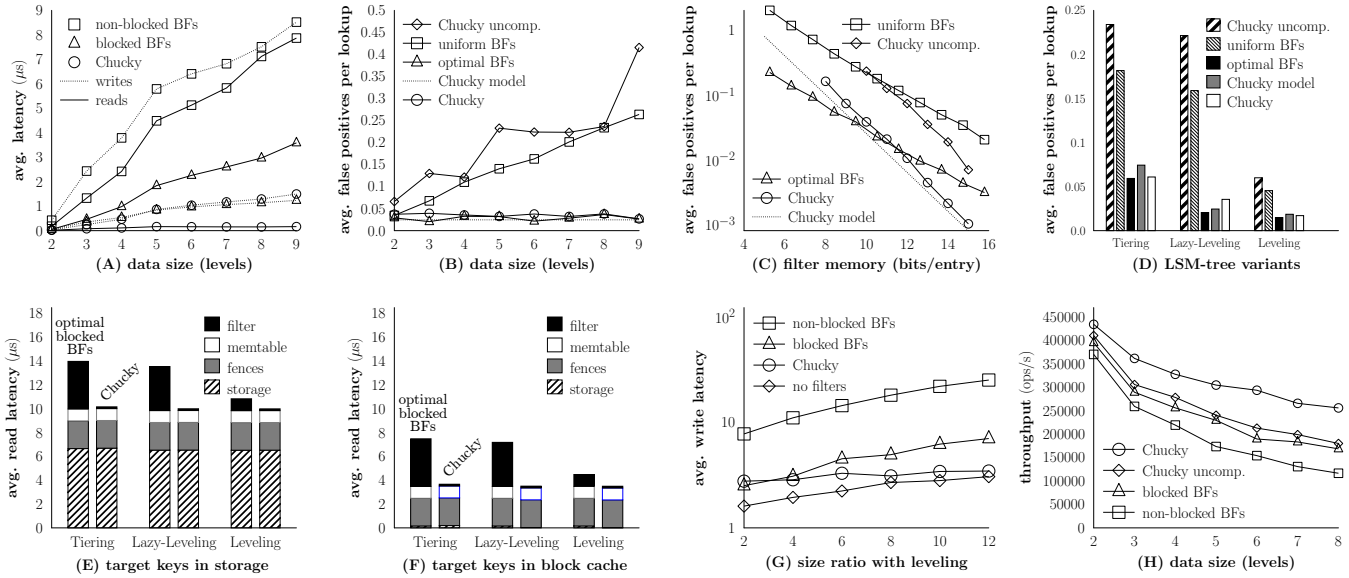


Figure 14: Chucky scales memory I/Os with data size (A) and for any LSM-tree variant (B). At the same time, Chucky's false positive rate scales with data sizes (C) with memory footprint (D), and for any LSM-tree variant (E). Hence, it improves read latency when target data is in fast storage (F) or cached (G), resulting in more scalable throughput (H).

Write latency is measured by dividing the overall time spent on filter maintenance by the number of writes issued by the application. Read latency is measured just before a full merge operation (when there are the most runs in the system) to highlight worst-case performance. Non-blocked BFs exhibit the fastest growing latency as they require multiple memory I/Os per filter across a growing number of filters. With blocked BFs, read/write latency grows more slowly as they require at most one memory I/O per read or write. Chucky's write latency also grows slowly with data as there are more levels across which to update LIDs. Crucially, Chucky is the only baseline that keeps read latency low with data size as each read requires a constant number of memory I/Os.

FPR Scalability. Figure 14 Part (B) compares the FPR for Chucky with both compressed and uncompressed LIDs to blocked BFs with both uniform and optimal space allocation. As we increase the data size, the FPR of Chucky with uncompressed LIDs increases since the LIDs grow and steal bits from the fingerprints. With uniform BFs, the FPR also grows with data size as there are more filters across which false positives can take place. In contrast, with optimal BFs, smaller levels are assigned exponentially lower FPRs, and so the sum of FPRs converges to a constant that's independent of the number of levels. Similarly, Chucky's FPR stays constant as the data grows since the average LID code length converges, thus allowing most fingerprints to stay large. The figure also includes the FPR model of Chucky from Equation 16 to show that, here too, it gives a reasonable approximation of the FPR in practice.

Figure 14 Part (C) shows that Chucky requires at least eight bits per entry to work (i.e., for codes and minimum fingerprint sizes). However, with eleven bits per entry and above Chucky offers better memory/FPR trade-offs than all BF variants. The reason is that BFs are known to exhibit suboptimal space use, which effectively reduces the memory budget by a factor of ln(2). Thus, Chucky scales the FPR better with respect to memory. Part (D) show that these results hold for any LSM-tree variant. Overall, Parts (B) to (D) show that Chucky is at least on par with optimal BFs with respect to scaling the FPR vs. memory budget trade-off.

Data in Storage vs. Memory. Figure 14 Parts (E) and (F) measure end-to-end read latency with uniform and Zipfian (with parameter $s = 1$) workloads, respectively. Read latency is broken in four components, including storage I/Os, fence pointers, memtable, and filter search. In Part (F), relevant data is most often in storage and so storage I/Os dominates read cost. Since our SSD is fast, however, the BFs probes still impose a significant latency overhead that Chucky is able to eliminate. In Part (F), on the other hand, the workload is skewed, meaning that target data is most often in the block cache. In this case, the BFs become a bottleneck as they must be searched before the relevant block in the cache can be identified. Chucky alleviates this bottleneck thus significantly improving read latency.

End-to-End Write Cost. Figure 14 Part (G) highlight's Chucky's ability to keep filter construction overheads low as we increase merge greediness (e.g., to optimize for range reads). We start with a leveled LSM-tree with size ratio two on the left-hand side and increase it along the x-axis. The y-axis measures end-to-end write cost, derived by dividing the overall time spent processing these updates by the number of updates issued by the application. As we increase the size ratio, write cost across all baselines increases since there is more overlap between runs at adjacent levels, and so more data needs to be rewritten on average during each merge operation. With relatively lower merge greediness (i.e., to the left of the figure), Chucky and blocked Bloom filters have similar construction overheads. However, as we increase the size ratio, end-to-end write costs with blocked Bloom filters increase more rapidly. The reason is that Bloom filters must be constructed from scratch during each merge operation, and so their construction overheads are proportional

to the LSM-tree's merge overheads. On the other hand, Chucky's performance draws nearer to the curve with disabled filters. The reason is that it only updates an entry's LID as an entry moves from one level to the next, and with larger size ratios there are fewer levels in the LSM-tree. Hence, under greedy merge policies, which are commonly used to optimize for range reads (e.g., the default setting in RocksDB), Chucky tangibly improves end-to-end writes.

Throughput Scalability. Figure 14 Part (H) shows how throughput scales as we increase the data size for a workload consisting of 95% Zipfian reads and 5% Zipfian writes (modeled after Workload B in YCSB [23]). Since writes are skewed, newer updates rapidly replace older entries during compaction at smaller levels and so major compaction and filter resizing do not take place. The BF baselines do not scale well as they issue memory I/Os across a growing number of BFs. Chucky with uncompressed LIDs also exhibits deteriorating performance as its FPR grows and leads to more storage I/Os. Chucky with compressed LIDs also exhibits deteriorating performance, mostly because the of the growing cost of the binary search across the fence pointers. However, Chucky provides better throughput with data size than all baselines because it scales the filter's FPR and memory I/Os at the same time.

6 RELATED WORK

LSM-Tree Performance. With LSM-tree being adapted as a storage engine across many systems (e.g., Cassandra [3], HBase [4], AsterixDB [2], RocksDB [35, 38, 74]), there is significant interest in optimizing LSM-tree performance [48, 70]. Most designs to date focus on managing compaction overheads, for example by separating values from keys [18, 68], partitioning runs into files and merging based on maximal file intersections [7, 92, 94], keeping hot entries in the buffer [7], making the buffer more dense [14] or concurrent [42], scheduling carefully to prevent tail latency [8, 69, 91], using customized or dedicated hardware [1, 45, 95, 97, 101], and by controlling delete persistence [90].

Another strain of work uses lazier compaction policies [76, 85, 86, 98, 99], which result in more runs and thus more BFs across which false positives and memory I/Os take place. Several works show how to implement lazier merge policies while still keeping the FPR modest, but they still incur many memory I/Os across many BFs [25, 28, 29, 49–51]. SlimDB [86] shows how to reduce memory I/Os using a Cuckoo filter, but its memory footprint does not scale well as discussed in Section 3. In contrast, we show how to scale the FPR, memory I/Os and memory footprint at the same time (for any merge policy including lazy ones) by replacing the Bloom filters by a Cuckoo filter with compressed level IDs.

Fingerprint Filters. While we build Chucky on top of Cuckoo filter [39] for its design simplicity, there exist many other fingerprint filter designs with nuanced properties. Many strive for better cache locality by using linear probing [9, 81], biasing Cuckoo insertions to one bucket [15], or by ensuring both candidate buckets are physically close [96]. Vacuum filter offers better memory utilization by allowing the filter size to not be a power of two [96]. Some designs allow to delay resizing the filter by chaining overflow filters [22, 96] or by sacrificing fingerprint bits [81]. Xor filter supports a better FPR in exchange for higher construction time [44, 80]. Other designs prevent overflows due to duplicate insertions using internal counters [81]. Morton filter [15] maps entries to variable sized "slots" within larger fixed-sized "blocks" and can therefore accommodate variable-sized entries more gracefully. Integrating Chucky with these filters to harness their properties can make for intriguing future work.

Range Filters. Recent LSM-tree designs use a range filter for each run [73, 100], which can save storage I/Os for range reads but require more memory I/Os to access and construct. Applying design elements from Chucky to create a unified range filter over a whole LSM-tree to reduce memory I/Os is an intriguing future direction.

Learned Fence Pointers. Recent work attempts to reduce the fence pointers' memory I/O overheads through learned indexes [24] by extrapolating an entry's location within a run based on the data's key distribution. Such work can complement Chucky by addressing the fence pointers, which become the next memory I/O bottleneck once Chucky is applied.

Learning from Negative Queries. Recent filtering approaches have been devised to learn from commonly issued negative queries (to non-existing keys) to reduce the false positive rate [32, 65, 78]. Integrating such techniques with Chucky is an intriguing direction.

Bloom Filters (BF). Numerous BF variants have been proposed [16, 72, 93], which enable counting [13, 40, 89], compressibility [77], vectoriziation [83], deletes for some but not all entries [88], efficient hashing [33, 60] and cache locality [17, 31, 66, 67, 84]. Bloomier filter allows to associate values with keys but is unable to compress values and is more complicated than fingerprint filters [19, 20].

Entropy Coding. Aside to Huffman coding, there exist other methods for compressing alphabets based on the probability distribution of the constituent symbols: Arithmetic Coding [82, 87] and Asymmetric Numeral Systems [36]. These methods do not require the use of auxiliary structures for encoding or decoding symbols. Harnessing such techniques to eliminate Chucky's auxiliary structures (i.e., the Huffman Tree, the Decoding Table, and the Recoding table) is an interesting future direction.

7 CONCLUSION

This paper that shows that as the performance gap between SSDs and memory devices is shrinking, the Bloom filters of LSM-tree are becoming a memory access bottleneck. We therefore propose Chucky, a filter for LSM-tree that requires fewer memory I/Os to query and maintain than Bloom filters. Chucky uses a Cuckoo filter in memory to map all entries to their locations within the LSM-tree, and it compresses this location information to keep the false positive rate low and stable. Thus, Chucky achieves the best of all worlds: fewer memory I/Os and a low and stable false positive rate, all for the same memory budget.

8 ACKNOWLEDGMENT

We thank the anonymous Reviewers for their invaluable feedback. We also thank the entire Pliops team for facilitating this work.

REFERENCES

[1] Ahmad, M. Y., and Kemme, B. Compaction management in distributed key-value datastores. *PVLDB 8*, 8 (2015), 850–861.

[2] Alsubaiee, S., Altowim, Y., Altwaijry, H., Behm, A., Borkar, V. R., Bu, Y., Carey, M. J., Cetindil, I., Cheelangi, M., Faraaz, K., Gabrielova, E., Grover, R., Heilbron, Z., Kim, Y.-S., Li, C., Li, G., Ok, J. M., Onose, N., Pirzadeh, P.,

Tsotras, V. J., Vernica, R., Wen, J., and Westmann, T. AsterixDB: A Scalable, Open Source BDMS. *PVLDB 7*, 14 (2014), 1905–1916.
[3] Apache. Cassandra. *http://cassandra.apache.org*.
[4] Apache. HBase. *http://hbase.apache.org/*.
[5] Athanassoulis, M., and Idreos, S. Design Tradeoffs of Data Access Methods. *SIGMOD* (2016).
[6] Athanassoulis, M., Kester, M. S., Maas, L. M., Stoica, R., Idreos, S., Ailamaki, A., and Callaghan, M. Designing Access Methods: The RUM Conjecture. *EDBT* (2016).
[7] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. *USENIX ATC* (2017).
[8] Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., and Didona, D. {SILK}: Preventing latency spikes in log-structured merge key-value stores. In *USENIX ATC* (2019).
[9] Bender, M. A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B. C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R. P., and Zadok, E. Don't Thrash: How to Cache Your Hash on Flash. *PVLDB 5*, 11 (2012), 1627–1637.
[10] Bjørling, M., Bonnet, P., Bouganim, L., and Dayan, N. The Necessary Death of the Block Device Interface. *CIDR* (2013).
[11] Bloom, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM 13*, 7 (1970), 422–426.
[12] Bonnet, P., and Dayan, N. Solid-state storage device flash translation layer, 2017. US Patent App. 15/056,381.
[13] Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., and Varghese, G. An improved construction for counting bloom filters. In *European Symposium on Algorithms* (2006).
[14] Bortnikov, E., Braginsky, A., Hillel, E., Keidar, I., and Sheffi, G. Accordion: Better Memory Organization for LSM Key-Value Stores. *PVLDB 11*, 12 (2018), 1863–1875.
[15] Breslow, A. D., and Jayasena, N. S. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. In *VLDB* (2018).
[16] Broder, A. Z., and Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. *Internet Mathematics 1* (2002), 636–646.
[17] Canim, M., Mihaila, G. A., Bhattacharjee, B., Ross, K. A., and Lang, C. A. SSD Bufferpool Extensions for Database Systems. *PVLDB 3*, 1-2 (2010), 1435–1446.
[18] Chan, H. H. W., Li, Y., Lee, P. P. C., and Xu, Y. HashKV: Enabling Efficient Updates in KV Storage via Hashing. *ATC* (2018).
[19] Charles, D., and Chellapilla, K. Bloomier filters: A second look. In *European Symposium on Algorithms* (2008).
[20] Chazelle, B., Kilian, J., Rubinfeld, R., and Tal, A. The bloomier filter: an efficient data structure for static support lookup tables. In *Symposium on Discrete Algorithms* (2004).
[21] Chen, G. J., Wiener, J. L., Iyer, S., Jaiswal, A., Lei, R., Simha, N., Wang, W., Wilfong, K., Williamson, T., and Yilmaz, S. Realtime data processing at facebook. In *SIGMOD* (2016).
[22] Chen, H., Liao, L., Jin, H., and Wu, J. The dynamic cuckoo filter. In *IEEE ICNP* (2017).
[23] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. *SoCC* (2010).
[24] Dai, Y., Xu, Y., Ganesan, A., Alagappan, R., Kroth, B., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. From wisckey to bourbon: A learned index for log-structured merge trees. In *USENIX OSDI* (2020).
[25] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal Navigable Key-Value Store. *SIGMOD* (2017).
[26] Dayan, N., Athanassoulis, M., and Idreos, S. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *TODS 43*, 4 (2018), 16:1–16:48.
[27] Dayan, N., Bonnet, P., and Idreos, S. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. *SIGMOD* (2016).
[28] Dayan, N., and Idreos, S. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. *SIGMOD* (2018).
[29] Dayan, N., and Idreos, S. The log-structured merge-bush & the wacky continuum. In *SIGMOD* (2019).
[30] Dayan, N., Svendsen, M. K., Bjorling, M., Bonnet, P., and Bouganim, L. Eagletree: exploring the design space of ssd-based algorithms. *VLDB* (2013).
[31] Debnath, B., Sengupta, S., Li, J., Lilja, D. J., and Du, D. H. Bloomflash: Bloom filter on flash-based storage. In *ICDCS* (2011).
[32] Deeds, K., Hentschel, B., and Idreos, S. Stacked filters: learning to filter by structure. *PVLDB* (2020).
[33] Dillinger, P. C., and Manolios, P. Bloom Filters in Probabilistic Verification. *Formal Methods in Computer-Aided Design* (2004).
[34] Dinh, T. T. A., Wang, J., Chen, G., Liu, R., Ooi, B. C., and Tan, K.-L. Blockbench: A framework for analyzing private blockchains. In *SIGMOD* (2017).
[35] Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., and Strum, M. Optimizing Space Amplification in RocksDB. *CIDR* (2017).
[36] Duda, J., Tahboub, K., Gadgil, N. J., and Delp, E. J. The use of asymmetric numeral systems as an accurate replacement for huffman coding. In *Picture Coding Symposium (PCS)* (2015).
[37] Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A., and Katti, S. Reducing dram footprint with nvm in facebook. In *EuroSys* (2018).
[38] Facebook. RocksDB. *https://github.com/facebook/rocksdb*.
[39] Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. Cuckoo Filter: Practically Better Than Bloom. *CoNEXT* (2014).
[40] Fan, L., Cao, P., Almeida, J., and Broder, A. Z. Summary cache: A scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking 8*, 3 (2000), 281–293.
[41] Gilad, E., Bortnikov, E., Braginsky, A., Gottesman, Y., Hillel, E., Keidar, I., Moscovici, N., and Shahout, R. Evendb: Optimizing key-value storage for spatial locality. In *EuroSys* (2020).
[42] Golan-Gueta, G., Bortnikov, E., Hillel, E., and Keidar, I. Scaling Concurrent Log-Structured Data Stores. *EuroSys* (2015).
[43] Golomb, S. Run-length encodings. *IEEE transactions on information theory* (1966).
[44] Graf, T. M., and Lemire, D. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics* (2020).
[45] Huang, G., Cheng, X., Wang, J., Wang, Y., He, D., Zhang, T., Li, F., Wang, S., Cao, W., and Li, Q. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *SIGMOD* (2019).
[46] Huffman, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.
[47] Idreos, S., Athanassoulis, M., Dayan, N., Guo, D., Kester, M. S., Maas, L., and Zoumpatianos, K. Past and future steps for adaptive storage data systems: From shallow to deep adaptivity. In *Real-Time Business Intelligence and Analytics*. Springer, 2015.
[48] Idreos, S., and Callaghan, M. Key-value storage engines. In *SIGMOD* (2020).
[49] Idreos, S., and Dayan, N. File management with log-structured merge bush, 2020. US Patent App. 16/963,411.
[50] Idreos, S., and Dayan, N. Key-value stores with optimized merge policies and optimized lsm-tree structures, 2020. US Patent App. 16/963,411.
[51] Idreos, S., Dayan, N., and Athanassoulis, M. Optimized navigable key-value store, 2020. US Patent App. 16/433,075.
[52] Idreos, S., Dayan, N., Qin, W., Akmanalp, M., Hilgard, S., Ross, A., Lennon, J., Jain, V., Gupta, H., Li, D., et al. Learning key-value store design. *arXiv preprint arXiv:1907.05443* (2019).
[53] Idreos, S., Dayan, N., Qin, W., Akmanalp, M., Hilgard, S., Ross, A., Lennon, J., Jain, V., Gupta, H., Li, D., and Zhu, Z. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR* (2019).
[54] Idreos, S., and Kraska, T. From auto-tuning one size fits all to self-designed and learned data-intensive systems. In *SIGMOD* (2019).
[55] Idreos, S., Zoumpatianos, K., Athanassoulis, M., Dayan, N., Hentschel, B., Kester, M. S., Guo, D., Maas, L. M., Qin, W., Wasay, A., and Sun, Y. The Periodic Table of Data Structures. *IEEE DEBULL 41*, 3 (2018), 64–75.
[56] Idreos, S., Zoumpatianos, K., Chatterjee, S., Qin, W., Wasay, A., Hentschel, B., Kester, M., Dayan, N., Guo, D., Kang, M., et al. Learning data structure alchemy. *IEEE DEBULL* (2019).
[57] Idreos, S., Zoumpatianos, K., Hentschel, B., Kester, M. S., and Guo, D. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. *SIGMOD* (2018).
[58] Im, J., Bae, J., Chung, C., Lee, S., et al. Pink: High-speed in-storage key-value store with bounded tails. In *USENIX ATC* (2020).
[59] Jagadish, H. V., Narayan, P. P. S., Seshadri, S., Sudarshan, S., and Kanneganti, R. Incremental Organization for Data Recording and Warehousing. *VLDB* (1997).
[60] Kirsch, A., and Mitzenmacher, M. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms 33*, 2 (2008), 187–218.
[61] Kondylakis, H., Dayan, N., Zoumpatianos, K., and Palpanas, T. Coconut: A scalable bottom-up approach for building data series indexes. *VLDB 11*, 6 (2018), 677–690.
[62] Kondylakis, H., Dayan, N., Zoumpatianos, K., and Palpanas, T. Coconut palm: Static and streaming data series exploration now in your palm. In *SIGMOD* (2019).
[63] Kondylakis, H., Dayan, N., Zoumpatianos, K., and Palpanas, T. Coconut: sortable summarizations for scalable indexes over static and streaming data series. *VLDBJ* (2019).
[64] Kraft, L. G. *A device for quantizing, grouping, and coding amplitude-modulated pulses*. PhD thesis, MIT, 1949.
[65] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The Case for Learned Index Structures. *SIGMOD* (2018).
[66] Lang, H., Neumann, T., Kemper, A., and Boncz, P. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. In *VLDB* (2019).
[67] Lu, G., Debnath, B., and Du, D. H. C. A Forest-structured Bloom Filter with flash memory. *MSST* (2011).

[68] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. *FAST* (2016).

[69] Luo, C., and Carey, M. J. On performance stability in lsm-based storage systems. In *VLDB* (2019).

[70] Luo, C., and Carey, M. J. Lsm-based storage techniques: a survey. *The VLDB Journal* (2020).

[71] Luo, C., Tözün, P., Tian, Y., Barber, R., Raman, V., and Sidle, R. Umzi: Unified multi-zone indexing for large-scale htap. In *EDBT* (2019).

[72] Luo, L., Guo, D., Ma, R. T., Rottenstreich, O., and Luo, X. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Commun. Surv. Tutor.* (2018).

[73] Luo, S., Chatterjee, S., Ketsetsidis, R., Dayan, N., Qin, W., and Idreos, S. Rosetta: A robust space-time optimized range filter for key-value stores. In *SIGMOD* (2020).

[74] Matsunobu, Y., Dong, S., and Lee, H. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *VLDB* (2020).

[75] McMillan, B. Two inequalities implied by unique decipherability. *IRE Transactions on Information Theory* (1956).

[76] Mei, F., Cao, Q., Jiang, H., and Li, J. Sifrdb: A unified solution for write-optimized key-value stores in large datacenter. In *ACM SOCC* (2018).

[77] Mitzenmacher, M. Compressed bloom filters. *IEEE/ACM Transactions on Networking* (2002).

[78] Mitzenmacher, M., Pontarelli, S., and Reviriego, P. Adaptive cuckoo filters. In *SIAM ALENEX* (2018).

[79] O'Neil, P. E., Cheng, E., Gawlick, D., and O'Neil, E. J. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[80] Pagh, A., Pagh, R., and Rao, S. S. An optimal bloom filter replacement. In *SODA* (2005).

[81] Pandey, P., Bender, M. A., Johnson, R., and Patro, R. A general-purpose counting filter: Making every bit count. In *SIGMOD* (2017).

[82] Pasco, R. C. *Source coding algorithms for fast data compression.* PhD thesis, Stanford University CA, 1976.

[83] Polychroniou, O., and Ross, K. A. Vectorized Bloom filters for advanced SIMD processors. *DAMON* (2014).

[84] Putze, F., Sanders, P., and Singler, J. Cache-, hash-, and space-efficient bloom filters. *Journal of Experimental Algorithmics (JEA)* (2010).

[85] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. *SOSP* (2017).

[86] Ren, K., Zheng, Q., Arulraj, J., and Gibson, G. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *PVLDB 10*, 13 (2017), 2037–2048.

[87] Rissanen, J., and Langdon, G. G. Arithmetic coding. *IBM Journal of research and development* (1979).

[88] Rothenberg, C. E., Macapuna, C., Verdi, F., and Magalhaes, M. The deletable Bloom filter: a new member of the Bloom family. *IEEE Communications Letters 14*, 6 (jun 2010), 557–559.

[89] Rottenstreich, O., Kanizo, Y., and Keslassy, I. The variable-increment counting bloom filter. *IEEE/ACM Transactions on Networking* (2013).

[90] Sarkar, S., Papon, T. I., Staratzis, D., and Athanassoulis, M. Lethe: A tunable delete-aware lsm engine. In *SIGMOD* (2020).

[91] Sears, R., and Ramakrishnan, R. bLSM: A General Purpose Log Structured Merge Tree. *SIGMOD* (2012).

[92] Shetty, P., Spillane, R. P., Malpani, R., Andrews, B., Seyster, J., and Zadok, E. Building Workload-Independent Storage with VT-trees. *FAST* (2013).

[93] Tarkoma, S., Rothenberg, C. E., and Lagerspetz, E. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys & Tutorials 14*, 1 (2012), 131–155.

[94] Thonangi, R., and Yang, J. On Log-Structured Merge for Solid-State Drives. *ICDE* (2017).

[95] Vinçon, T., Hardock, S., Riegger, C., Oppermann, J., Koch, A., and Petrov, I. Noftl-kv: Tackling write-amplification on kv-stores with native storage management. In *EDBT* (2018).

[96] Wang, M., Zhou, M., Shi, S., and Qian, C. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. In *VLDB* (2019).

[97] Wang, P., Sun, G., Jiang, S., Ouyang, J., Lin, S., Zhang, C., and Cong, J. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. *EuroSys* (2014).

[98] Wu, X., Xu, Y., Shao, Z., and Jiang, S. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. *USENIX ATC* (2015).

[99] Yao, T., Wan, J., Huang, P., He, X., Wu, F., and Xie, C. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *TOS 13*, 4 (2017), 29:1–29:28.

[100] Zhang, H., Lim, H., Leis, V., Andersen, D. G., Kaminsky, M., Keeton, K., and Pavlo, A. SuRF: Practical Range Query Filtering with Fast Succinct Tries. *SIGMOD* (2018).

[101] Zhang, T., Wang, J., Cheng, X., Xu, H., Yu, N., Huang, G., Zhang, T., He, D., Li, F., Cao, W., et al. Fpga-accelerated compactions for lsm-based key-value store. In *USENIX FAST* (2020).