Chucky: Huffman Coded Key-Value Store

ABSTRACT

Modern persistent key-value stores rely on an LSM-tree in storage (SSD) to optimize writes and Bloom filters in memory (DRAM) to optimize reads. In this work, we show that the Bloom filters are emerging as a performance bottleneck. First, the costs of probing and constructing them grow with data size. Second, their costs are becoming more pronounced as storage access on modern SSDs is becoming less expensive relative to memory access.

Recently, a new slew of data structures has emerged as an alternative to Bloom filters. They work by storing a fingerprint for every data entry within a compact hash table. We call them Fingerprint Filters (FFs). In this paper, we show how to scale an LSM-tree's memory bandwidth by replacing its Bloom filters with an FF that's augmented with every entry's location within the LSM-tree. However, we show that this new design does not immediately scale in terms of its false positive rate (FPR). This is due to the auxiliary location information, which grows superlinarly with the data size, thus taking away bits from the fingerprints. By harnessing information theory and compression techniques, we show how to scale the size of this location information to keep the FPR small as the data grows. In this way, we show how to achieve the best of both worlds: scalable memory *and* storage bandwidth at the same time.

ACM Reference Format:
. 2021. Chucky: Huffman Coded Key-Value Store. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern key-value stores (KV-stores) rely on an LSM-tree to persist data in storage. An LSM-tree optimizes application writes by buffering them in memory, flushing the buffer to storage as a sorted run whenever it fills up, and sort-merging runs across a logarithmic number of levels [53]. To optimize application point reads, there is an in-memory Bloom filter [8] for each run to allow ruling out runs that do not contain a target entry. Such designs are used in OLTP [33], HTAP [44], social graphs [48], block-chain [24], etc.

Memory I/O: A Creeping Bottleneck. LSM-tree was originally designed for hard disk drives, which are 5-6 orders of magnitude slower than DRAM memory chips. The advent of SSDs, however, has shrunk the performance difference between storage and memory access to 2-3 orders of magnitude [26]. Today, a memory I/O takes $\approx$ 100 ns while an SSD read I/O (e.g., on Intel Optane) takes $\approx 10\ \mu$s. Hence, memory access is no longer negligible relative to storage access. This is driving a need to revise database architectures to eliminate creeping memory bandwidth bottlenecks [47].

Conference'17, July 2017, Washington, DC, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

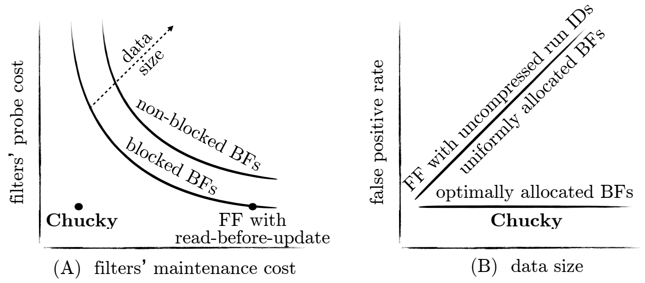


Figure 1: Existing LSM-tree filter designs do not scale probe cost, maintenance cost and false positives at the same time.

The Problem with Bloom Filters. In this work, we show that the Bloom filters (BFs) in modern KV-stores are emerging as a memory I/O bottleneck. Consider first LSM-tree designs that optimize for writes by merging lazily and thus having tens to hundreds of runs present in the system [36, 50, 59, 60, 71]. For such designs, probing each BF at a cost of $\approx$ 100 ns can approach and even exceed the latency of the SSD I/O(s), which fetch the target entry from storage.

Moreover, many application workloads exhibit skew such that the most sought after entries reside in a block cache in memory [30]. Locating an entry in this cache still entails traversing potentially all the BFs to first identify the run (and then the data block within it) that contains the target entry. As there are no storage I/Os in this case, memory I/Os for probing the BFs become the bottleneck.

A naive solution is tuning the LSM-tree to merge more greedily so that there are fewer runs and thus fewer BFs to probe. However, increasing merge greediness also increases storage write I/Os, which can outweigh the benefit of saving BF probes. Moreover, tuning up merge greediness also increases the BFs' construction cost, which can account for over 70% of merge overheads [35]. A BF is immutable and has to be rebuilt from scratch during every merge operation. Thus, merging runs more greedily also entails reconstructing BFs more greedily. We illustrate this contention conceptually in Figure 1 (A) with two types of BFs (explained more in Section 2). The takeaway is that BFs make it impossible to mitigate probe cost and construction cost at the same time. In fact, both of these costs grow with data size thus leaving the application with worsening trade-offs to choose from.

Fingerprint Filters: The Promise. Over the past decade, a new family of data structures has emerged as an alternative to Bloom filter. They operate by storing a fingerprint for every data item in a compact hash table. We refer to them as Fingerprint Filters (FFs), and they include Quotient filter [7, 55], Cuckoo filter [28] and others [11, 32, 54, 69]. Their promise is that they support storing auxiliary and updatable information for each entry alongside its fingerprint. Hence, it is possible to replace an LSM-tree's multiple BFs by one updatable FF that maps each data entry to a fingerprint and an auxiliary *run ID* [60]. During an application read, the fingerprints match or dismiss the target key while the run IDs direct which run to search for every fingerprint match. Hence, an FF requires far fewer memory I/Os than BFs to find a given entry's run.

Challenges: Scaling False Positives & Updates. Despite this promise, we identify two challenges in harnessing an FF for LSM-tree filtering. First, the false positive rate (FPR) for an FF does not scale well. This is due to the run IDs, which must grow with data size to identify more runs uniquely. Assuming a fixed memory budget, the run IDs must "steal" bits from the fingerprints as data grows. This increases the FPR and results in more storage I/Os. We illustrate the problem conceptually in Figure 1 Part (B), which shows that unlike state-of-the-art BF approaches (explained further in Section 2), the FPR for an FF increases with data size.

A second challenge is efficiently keeping the run IDs within the FF up-to-date. A possible solution is to issue a read to storage to check if an entry exists before the write and, if so, to update its run ID [60]. However, the extra read I/O to storage is expensive. In summary, existing designs based on FF do not scale storage I/Os, while designs based on BFs do not scale memory I/Os.

Insights. To scale false positives, our insight is that the run IDs are extremely compressible. The reason is that their distribution is approximately geometric, meaning that entries with run IDs of larger levels are exponentially more common than entries with run IDs of smaller levels. This allows encoding larger runs with fewer bits and smaller runs with more bits. The saved space can be dedicated to the fingerprints to keep them large as the data grows.

For scaling updates, our insight is that the run IDs can be opportunistically updated during merge operations while the target entries are brought to memory. Hence, we can keep the run IDs up-to-date without introducing any additional storage I/Os.

Chucky. We present Chucky: **Hu**ffman **C**oded **Key**-Value Store. Chucky's novelty is the ability to scale memory *and* storage I/Os at the same time. It achieves this by replacing the BFs by a single FF with compressed run IDs that are updated during merge operations. We explore in detail the design space for run ID compression using Huffman coding, and we identify and address the resulting challenges: (1) how to align fingerprints and compressed run IDs within the FF's buckets, and (2) how to en/decode run IDs efficiently. We use the bits saved through compression to keep the fingerprints large and to thereby guarantee a scalable false positive rate as the data grows. Chucky's underlying techniques can fit with any FF, and we specifically show how to tailor them to Cuckoo filter.

Contributions. Our contributions are as follows.

- We show that BFs in LSM-based KV-stores are emerging as a memory bandwidth bottleneck for both reads and writes.
- We show how to replace the BFs by an FF with auxiliary run IDs that are kept up-to-date opportunistically while merging.
- We show that run IDs are extremely compressible, and we study how to minimize their size using Huffman coding.
- We show how to align compressed run IDs and fingerprints within FF buckets to achieve good space utilization.
- We show how to encode and decode run IDs efficiently.
- We show how to integrate Chucky with Cuckoo filter.
- We show experimentally that Chucky scales in terms of memory I/Os *and* storage I/Os at the same time.

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

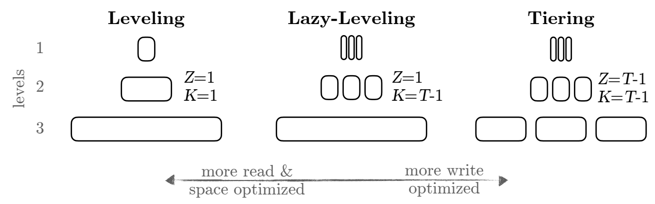
LSM-Tree. LSM-tree consists of multiple levels of exponentially increasing capacities. Level 0 is an in-memory buffer and all other levels are in storage. The application inserts key-value pairs into the buffer. When the buffer reaches capacity, its contents get flushed as a sorted array called a *run* into Level 1 in storage. Whenever a given Level i reaches capacity, its runs get merged into Level $i + 1$. To merge runs, their entries are brought from storage to memory to be sort-merged and then written back to storage as a new run. The number of levels L is $\approx \log_T(N)$, where T is the capacity ratio between any two adjacent levels and N is the ratio between the overall data size and the in-memory buffer's size. Figure 2 lists terms used to describe LSM-tree throughout the paper.

Updates and deletes are performed out-of-place by inserting a key-value entry with the updated value into the buffer (for a delete, the value is a tombstone). Whenever two runs get merged while containing two entries with the same key, the older entry is discarded as the newer entry supersedes it. In order to always find the most recent version of an entry, an application read traverses the runs from youngest to oldest across the levels and terminates when it finds the first entry with a matching key. If its value is a tombstone, the read returns a negative result to the application. For every run in storage, there is an array of fence pointers in memory that contains the min/max key at every data block and thereby allows finding the relevant block within a run with one storage I/O.

Wide Design Space. The LSM-tree design space spans many variants that favor different application workloads [43]. The most common two are Leveling [53] and Tiering [36] (used by default in RocksDB and Cassandra, respectively). We illustrate them in Figure 2. With Leveling, merging is performed greedily within each level (i.e., as soon as a new run comes in). As a result, there is at most one run per level and every entry gets merged on average $\approx T/2$ times within each level. With Tiering, merging is performed lazily within each level (i.e., only when the level fills up). As a result, there are at most $\approx T$ runs per level and every entry gets merged once across each of the levels. Leveling is more read and space optimized while Tiering is more write-optimized. The size ratio T can be varied to fine-tune this trade-off [43]. Figure 2 also illustrates Lazy-Leveling, a hybrid that uses Leveling at the largest level and Tiering at all smaller levels to offer favorable trade-offs in-between [20] (i.e., for space-sensitive write-heavy applications with mostly point reads). The recent Dostoevsky framework [20] generalizes these three variants using two parameters: (1) a threshold Z for the number of runs at the *largest level* before a merge is triggered, and (2) a threshold K for the number of runs at each of the *smaller levels* before a merge is triggered. Figure 2 shows how to set these parameters to assume each of the three designs. Equation 1 denotes A_i as the maximum number of runs at Level i and A as the maximum number of runs in the system with respect to these parameters.

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

We build Chucky on top of Dostoevsky to be able to span multiple LSM-tree variants that can accommodate diverse workloads.



Term	Definition
L	number of LSM-tree levels
A	number of LSM-tree runs
Ai	number of LSM-tree runs at Level i
T	LSM-tree size ratio
N	data size / buffer size
Z	merge triggering threshold for largest level
K	merge triggering threshold for smaller levels
M	filtering memory budget (bits/entry)

Figure 2: LSM-tree variants and terms used to describe LSM-tree throughout the paper.

	Leveling	Lazy-Leveling	Tiering
probe cost	$O(L)$	$O(L \cdot T)$	$O(L \cdot T)$
construction cost	$O(L \cdot T)$	$O(L + T)$	$O(L)$

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

	Leveling	Lazy-Leveling	Tiering
Uniform	$O(2^{-M \cdot ln(2)} \cdot L)$	$O(2^{-M \cdot ln(2)} \cdot L \cdot T)$	$O(2^{-M \cdot ln(2)} \cdot L \cdot T)$
Optimal	$O(2^{-M \cdot ln(2)})$	$O(2^{-M \cdot ln(2)})$	$O(2^{-M \cdot ln(2)} \cdot T)$

Table 2: Bloom Filters' false positive rate complexities.

Fixing Merge Assumptions. While some designs such as HBase and Cassandra merge entire runs at a time, others such as RocksDB partition each run into multiple files called Sorted String Tables (SSTs) and merge at the granularity of SSTs. This grants finer control of how merge overheads are scheduled in space and time, though it increases write-amplification [67]. For ease of exposition, we discuss merging as though it occurs at the granularity of runs, though this work is also applicable to designs that rely on SSTs for merging. We use RocksDB's *dynamic level size adaptation* technique [25] that sets the capacities of Levels 1 to $L-1$ based on the number of entries at the largest level in order to restrict storage space-amplification. We assume preemptive merging [20], whereby we detect when Levels 1 to i are near capacity and merge their runs all at once as opposed to having the merge recursively trickling across the levels and resulting in more write-amplification.

Bloom filters. Each run in the LSM-tree has a corresponding in-memory Bloom filter (BF) [8], which is a space-efficient probabilistic data structure used to test whether a key is a member of a set[1]. A BF is an array of bits with h hash functions. For every key inserted, we map it using each of the 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 *false positive*. The false positive rate (FPR) is $2^{-M \cdot \ln(2)}$, where M is the number of bits per entry. As we increase M, the probability of bit collisions decreases and so the FPR drops. In KV-stores in industry (e.g., RocksDB), the number of bits per entry is typically set to ten.

A BF does not support deletes (i.e., by resetting bits back to 0) as this could lead to false negatives. For this reason, a new BF is created from scratch for a new run as a result of a merge.

A BF entails h memory I/Os for an insertion as well as for a positive query. For a negative query, 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 [39, 58]. 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 FPR increase. RocksDB recently switched from standard to blocked BFs. We use both approaches as baselines in this paper, and we focus more on blocked BFs as they are the tougher competition.

[1]All Bloom filters are persisted in storage to be recoverable in case of system failure.

Memory I/Os. For an LSM-tree with blocked BFs, an application query costs at most $O(K \cdot (L-1) + Z)$ memory I/Os (i.e., one to the filter of each run). On the other hand, an application update costs $O(T/K \cdot (L-1) + T/Z)$ amortized memory I/Os (the average number of times an entry gets merged and thus inserted into a new BF). Table 1 summarizes these costs for each of the LSM-tree variants. We observe that both cost metrics increase with respect to the number of levels L and thus with the data size. Second, we observe an inverse relationship between these metrics: the greedier we set the LSM-tree's merging to be (i.e., either by changing merge policy or by fine-tuning the size ratio), probe cost decreases as there are fewer BFs while construction cost increases as the BFs get rebuilt more greedily. Hence, it is impossible to improve on one of these metrics without degrading the other. Figure 1 Part (A) conceptually illustrate this relationship. Can we scale these metrics while also eliminating the contention between them?

Memory Allocation. KV-stores in industry set a uniform number of bits per entry to BFs at all levels. This approach, however, was recently identified as sub-optimal. The optimal approach is to reallocate $\approx$ 1 bit per entry from the largest level and to use it to assign linearly more bits per entry to filters at smaller levels [19, 20]. While this slightly increases the largest level's FPR, it exponentially decreases the FPRs at smaller levels such that the overall sum of FPRs is smaller. Equations 2 and 3 express the FPR with both approaches [20].

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

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

The intuition for Equation 2 is that as the data grows, the FPR increases as there are more runs and thus more BFs across which false positives can occur. On the other hand, 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. The reason is that as the LSM-tree grows, smaller levels are assigned exponentially smaller FPRs thus causing the sum of FPRs to converge. We summarize the corresponding FPR complexities in Table 2 and visualize them conceptually in Figure 1 Part (B). While our primary goal is to improve on the BFs' memory bandwidth, we must also at least match the FPR scalability with the optimal BF approach to be competitive across all performance fronts.

3 PROMISE: LSM & FINGERPRINT FILTER

Fingerprint Filters (FFs) are a family of data structures that have recently emerged as an alternative to Bloom filter. At its core, an FF is a compact hash table that stores fingerprints of keys, where a fingerprint is a string of F bits derived by hashing a key. To test for set membership, FF hashes a key in question to a bucket and compares its fingerprint to all fingerprints in the bucket. If there is a match, we have a positive. An FF cannot return a false negative, and it returns a false positive with a probability of at least 2^{-F}. The fingerprint size F controls a trade-off between accuracy and space. The various FFs that have been proposed differ in their collision resolution methods, which swap entries across buckets to resolve collisions. For example, Cuckoo filter [28] uses a variant of Cuckoo hashing while Quotient filter [7, 55] uses a variant of linear probing.

Promise. While different collision resolution methods give different FFs nuanced performance and space properties, all FFs to date share a common set of desirable properties with respect to our problem. First, they support queries and updates in practically constant time for a similar memory footprint to Bloom filters. Second, unlike Bloom filters, FFs support storing updatable auxiliary data for each entry alongside its fingerprint. These capabilities allows to replace an LSM-tree's multiple Bloom filters with a single FF that maps from data entries to the runs in which they reside in the LSM-tree. Such a design promises to allow finding an entry's target run with a small and constant number of memory I/Os, unlike Bloom filters which require at least one memory I/O across numerous filters.

Challenges. Despite this promise, two challenges arise with this approach. The first is how to keep the run IDs up-to-date as entries get merged across the LSM-tree. The second is how to keep the size of the run IDs modest as the data size grows.

Case-Study. The recent SlimDB system [60] is the first to integrate LSM-tree with an FF. As such, it provides an interesting case-study and baseline with respect to meeting the above two challenges.

To keep the run IDs within the FF up-to-date, SlimDB performs a read I/O to storage for each application update to check if the entry exists and if so to update its run ID within the FF. This involves a substantial overhead in terms of storage I/Os, specifically for applications that perform blind writes. Can we maintain the run IDs up-to-date without using additional storage I/Os?

Second, SlimDB represents the run IDs using binary encoding. Each run ID therefore comprises $\lceil \log_2(K \cdot (L-1) + Z) \rceil$ bits to identify all runs uniquely[2]. Hence, more bits are needed as the number of levels L grows. This is not a problem for SlimDB as it is designed for systems with a less constrained memory budget[3]. In contrast, we focus on applications with a tighter budget of M bits per entry, where M is a non-increasing small constant. Under this constraint, Equation 4 denotes the FPR over a single entry with respect to the number of bits per entry M and the run ID size D.

$$FPR > 2^{-F} = 2^{-M+D} \tag{4}$$

By plugging the run ID size for D, the lower bound simplifies to $2^{-M} \cdot \lceil K \cdot (L-1) + Z \rceil$, meaning the FPR increases with the number of levels as the run IDs steal bits from the fingerprints. Is it possible to better scale the FPR and thus storage I/Os as the data grows?

[2]SlimDB uses the Tiered merge strategy whereby $K \approx T$ and $Z \approx T$, and so our analysis here is more generic.

[3]In fact, SlimDB uses additional memory to prevent false positives altogether by storing the full keys of colliding fingerprints in memory. SlimDB also proposes a novel fence pointers format, which is orthogonal to our purposes in this paper.

4 CHUCKY

Chucky is an LSM-based KV-store that scales memory *and* storage I/Os at the same time. It achieves this by replacing the Bloom filters with a fingerprint filter and innovating along two areas.

Opportunistic Updates. Chucky keeps the run IDs within the FF up-to-date opportunistically during merge operations at no additional storage I/O cost. Moreover, it allows run IDs to be inherited across merge operations to obviate FF updates and thereby reduce memory I/Os. In this way, Chucky both scales and decouples the costs of updating and querying the FF, as shown in Figure 1 Part (A). We discuss this concretely in Section 4.1.

Run ID Compression. Chucky compresses run IDs to prevent their size from increasing and taking bits from the fingerprints as the data grows. Thus, Chucky scales the FPR and thereby storage I/Os as shown in Figure 1 Part (B). We show how to compress run IDs in Section 4.2. We identify and address the implications of compressed run IDs on bucket alignment and decompression efficiency in Sections 4.3 and 4.4, respectively.

Abstract Model. For both generality and ease of exposition, we abstract the details of the FF's collision resolution method for now. We show how to tailor Chucky to Cuckoo filter in Section 4.5.

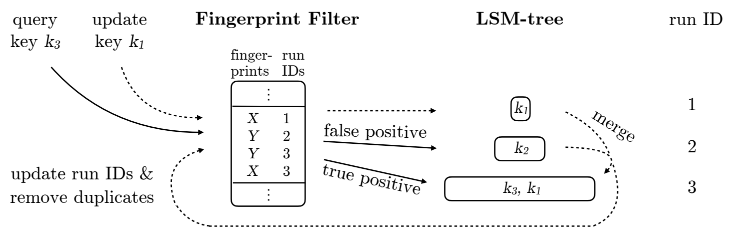
4.1 Integration with LSM-Tree

Figure 3 illustrates the architecture of Chucky, which uses an FF to map each physical entry in the LSM-tree to one FF entry consisting of a fingerprint and a run ID. The figure also illustrates the query and update workflows with solid and dashed lines, respectively.

Query Example. In Figure 3, keys k_1, k_2 and k_3 reside across various runs but happen to be mapped by the FF's hash function to the same FF bucket. Keys k_2 and k_3 have a colliding fingerprint Y while key k_1 has a different fingerprint X. The application queries key k_3, and so we reach the bucket shown in the figure and traverse its fingerprints from those belonging to younger runs first (i.e., to find the most recent entry's version). For Run 1, we have a negative as the fingerprint is different. For Run 2, we have a false positive leading to a wasted storage I/O. For Run 3, we have a true positive, and so the target entry is returned to the application.

Update Workflow. Whenever the LSM-tree's buffer flushes a new batch of application updates to storage, Chucky adds an FF entry for each key in the batch (including for tombstones). For example, consider entry k_1 in Figure 3, for which there is originally one version at Run 3. A new version of this entry is then flushed to storage as a part of Run 1. As a result, Chucky adds a new FF entry to account for this updated version. This leads to temporary space-amplification (SA), which is later resolved through merging while entries are brought to memory to be sort-merged[4]. For every obsolete entry identified and discarded while merging runs, Chucky removes the corresponding entry from the FF. For every other entry, Chucky updates its run ID to the ID of the new run being created.

[4]This SA is modest since the LSM-tree's exponential structure restricts the average number of versions per entry (e.g., $T/(T-1) < 2$ with Leveling or Lazy-Leveling). In fact, BFs exhibit exactly the same memory SA since each version of an entry across different runs' BFs takes up M bits per entry.



Term	Definition
B	FF bucket size in bits
F	FF fingerprint size in bits
D	run ID size in bits
S	entries per FF bucket
p_i	fraction of data at Level i
f_j	probability of Run ID j
H	entropy of the run ID probability distribution
H_{comb}	entropy the combination probability distribution
C	the set of all run ID combinations
C_{freq}	the set of most probable run ID combinations

Figure 3: Chucky's high-level architecture and terms used to describe it.

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

Table 3: Chucky's invocation complexities.

	Leveling	Lazy-Leveling	Tiering
binary encoded	$O(2^{-M} \cdot L)$	$O(2^{-M} \cdot L \cdot T)$	$O(2^{-M} \cdot L \cdot T)$
entropy encoded	$O(2^{-M})$	$O(2^{-M})$	$O(2^{-M} \cdot T)$

Table 4: FPR bounds w/o run ID compression.

Hence, Chucky maintains the FF's run IDs without requiring any additional storage I/Os.

Furthermore, Chucky allows run IDs to be inherited across merge operations to obviate FF updates and save memory I/Os. It does this by setting the run ID of the j^{th} oldest run at Level i of the LSM-tree to $(i-1) \cdot K + j$. Thus, the run IDs range from 1 to A, where A the number of runs (from Equation 1). Effectively, this means that an entry's run ID only changes when the entry is merged into a new level, but not when a given entry stays at the same level after a merge. For example in Figure 3, when merging Runs 1, 2 and 3 into a new run at Level 3, the new run also gets assigned a run ID of 3. During the merge operations, we identify and remove entry k_1's older version from the FF and update the run IDs of entry k_2 and of the new version of entry k_1 to 3. However, we keep Entry k_3's run ID the same since the new run inherits the older Run 3's ID.

Fewer Memory I/Os. An application query probes the FF once while an update accesses it $\approx L$ amortized times (once for each time the updated entry moves into a new level). Table 3 summarizes these properties. Relative to the memory I/O complexities of BFs in Table 1, Chucky reduces querying cost to a constant. Furthermore, it cheapens update cost for greedier merge policies and thereby decouples the memory I/O costs of queries and updates. In this way, Chucky dominates Bloom filters in terms of memory bandwidth.

4.2 Compressing Run IDs

As we saw earlier, binary encoded run IDs within FF buckets grow with data size thus taking bits from fingerprints and increasing the false positive rate. To prevent this problem, we now explore in detail how to keep run IDs as small as possible using compression.

Run ID Probabilities. Run IDs are extremely compressible because they follow an approximately geometric probability distribution. We formalize this using Equation 5, which denotes p_i as the fraction of user data at Level i of the LSM-tree.

$$p_i = \frac{T-1}{T^{L-i}} \cdot \frac{T^{L-1}}{T^L - 1} \approx \frac{T-1}{T} \cdot \frac{1}{T^{L-i}}, \tag{5}$$

A run with ID j resides at Level $\lceil j/K \rceil$ of the LSM-tree. Its frequency is therefore that Level's probability $p_{\lceil j/K \rceil}$ (from Equation 5) divided by the number of runs at that level $A_{\lceil j/K \rceil}$ (from Equation 1). Thus we denote f_j as the frequency of the j^{th} run ID in Equation 6.

$$f_j = \frac{p_{\lceil j/K \rceil}}{A_{\lceil j/K \rceil}} \tag{6}$$

These probabilities decrease exponentially for runs at smaller levels. Hence, it is possible to represent larger runs' IDs with few bits and smaller runs' IDs with more bits. Since smaller runs' IDs are exponentially less frequent, the average number of bits used to represent a run ID would stay small.

Entropy. To establish a limit on how much run IDs can be compressed, we derive their Shannon entropy, which represents a lower bound on the average number of bits needed to represent items within a given probability distribution. We do so in Equation 7 by stating the definition of entropy over the different run IDs' probabilities, plugging in Equations 1 and 5 for A_i and p_i, respectively, and simplifying. Interestingly, the entropy converges to a constant that is independent of the number of levels and hence does not grow with data size. The intuition is that the exponential decrease in run ID probabilities for smaller levels trumps the fact that run IDs at smaller levels would require more bits to represent uniquely.

$$H = \sum_{j=1}^{A \to \infty} -f_j \cdot \log_2(f_j) = \log_2\left(Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \right) \tag{7}$$

By plugging Equation 7 as the run ID length D of Equation 4, we obtain FPR bounds in Table 4. These bounds hold for any FF for which the number of fingerprints checked per lookup is a small constant (i.e., all FFs to date in practice). The fact these bounds are lower than those in Table 2 for optimal BFs reaffirms our approach; an FF with compressed run IDs may be able to match or even improve on BFs in terms of FPR.

Huffman Coding. To compress the run IDs in practice, we use Huffman coding [34]. The Huffman encoder takes as input the run IDs along with their probabilities (from Equation 6). As output, it returns a binary code to represent each run ID and whereby more frequent run IDs are assigned shorter codes. It does so by creating a binary tree from the run IDs by connecting the least probable run IDs first as subtrees. A run ID's ultimate code length corresponds to its depth in the resulting tree.

Example. Figure 4 illustrates a Lazy-Leveled LSM-tree[5] with labeled run IDs, each with a corresponding frequency from Equation 6. We feed these run IDs and their frequencies into a Huffman encoder to obtain the Huffman tree shown alongside. The code for a run is given by concatenating the tree's edge labels on the path from the root node to the given run ID's leaf node. For instance, the codes for run IDs 4, 8 and 9 are 011011, 010 and 1, respectively.

[5]This tree's parameters are $T = 5$, $K = 4$, $Z = 1$.

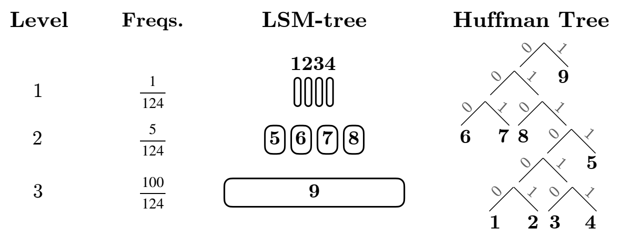


Figure 4: A Huffman tree represents each run ID with a unique code such that the average code length is minimized.

Decodability. With Huffman coding, no code is a prefix of another code [34]. 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 run ID at the given leaf, and then restarting at the root. For example, the input bit stream 11001 gets uniquely decoded into run IDs 9, 9 and 7 based on the Huffman tree in Figure 4. This property allows us to uniquely decode all run IDs within a bucket without the need for delimiting symbols.

Average Code Length. We measure the encoded run IDs' size using their average code length (ACL) as defined in Equation 8, where l_j is the code length assigned to the j^{th} run. For example, this equation computes 1.52 bits for the Huffman tree in Figure 4. This is a saving of 62% relative to binary encoding, which would require four bits to represent each of the nine run IDs uniquely.

$$ACL = \sum_{j=1}^{A} l_j \cdot f_j \tag{8}$$

Improved Memory Footprint Scalability. It is well-known in information theory that an upper bound on a Huffman encoding's ACL is the entropy plus one[6] [34]. We express this as $ACL \le H + 1$, where H is the entropy from Equation 7. We therefore expect the ACL in our case to converge and become independent of the data size, the same as Equation 7. We verify this in Figure 5 by increasing the number of levels for the example in Figure 4 and illustrating the Huffman ACL, which indeed converges. The intuition is that while runs at smaller levels get assigned longer codes, these codes are exponentially less frequent. In contrast, a binary encoding requires more bits to represent all run IDs uniquely. Thus, Huffman encoding allows to better scale memory footprint.

Tight ACL Upper Bound. Among compression methods that encode one symbol at a time, Huffman coding is known to be optimal in that it minimizes the ACL [34]. 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 Equation 8 than before by assuming a less generic coding method and observing that the Huffman ACL will be at least as short. For example, we can represent each run ID 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 runs at Level i uniquely[7]. We derive this encoding's average length in Equation 9 as ACL_{UB} and illustrate it in Figure 5 as a reasonably tight upper bound of the Huffman ACL.

$$ACL_{UB} = \sum_{i=1}^{L} p_i \cdot \left(L - i + 1 + \log_2(A_i)\right) = \frac{T}{T-1} + \log_2(K^{\frac{1}{T}} \cdot Z^{\frac{T-1}{T}}) \tag{9}$$

[6]The intuition for adding one is that each code length is rounded up to an integer.

[7]This is effectively a Golomb encoding [31], which is also applicable to our problem and easier to analyze. However, we focus on Huffman encoding as it allows to encode multiple symbols at a time. We harness this capability momentarily.

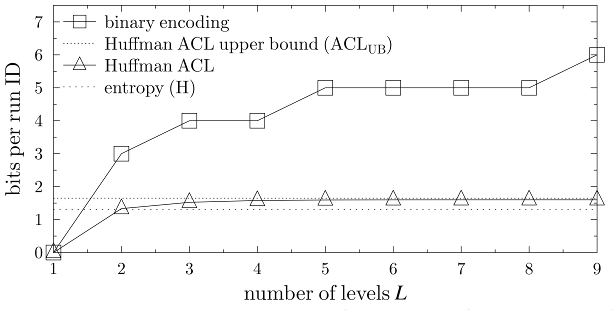


Figure 5: Compression causes the run IDs' average code length to converge with respect to data size.

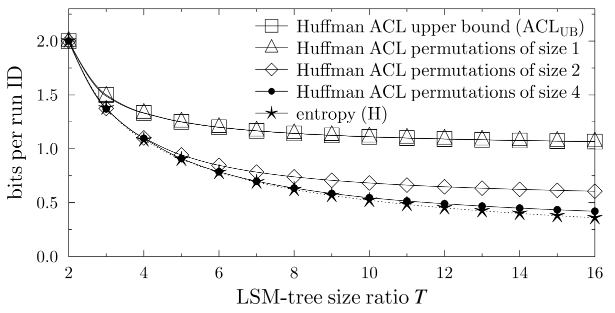


Figure 6: The run IDs' average code length approaches the entropy as larger permutations of run IDs are encoded.

Proximity to Entropy. Figure 5 further plots the entropy of the run IDs' frequency distribution from Equation 7. As shown, there is a gap between the Huffman ACL and the entropy. In fact, in Figure 6 we show that as we increase the LSM-tree's size ratio T, the gap between the ACL and the entropy grows[8]. The reason is that so far we have been encoding one run ID at a time, meaning that each run ID requires at least one bit to represent with a code. Hence, the ACL cannot drop below one bit per run ID. On the other hand, the entropy continues to drop towards zero as the probability distribution becomes more skewed since the information content (i.e., the amount of surprise) in the distribution decreases. A general approach in information theory to overcome this limitation is to encode multiple symbols at a time, as we now continue to explore.

Encoding Run IDs as Permutations. A common technique for an FF to achieve a high load factor at a modest FPR sacrifice is to store multiple fingerprints per bucket [11, 28, 69]. We now show how to leverage this FF design decision to collectively encode all run IDs within a bucket to further push compression. Figure 7 gives an example of how to encode permutations of two run IDs at a time for a Leveled LSM-tree (with two levels and size ratio T of 10). The probability of a permutation is the product of its constituent run IDs' probabilities from Equation 6. For example, the probability of permutations 21 and 22 are $10/11 \cdot 1/11$ and $(10/11)^2$, respectively. By feeding all possible run IDs permutations of size two along with their probabilities into a Huffman encoder, we obtain the Huffman

[8]The figure is drawn for a Leveled LSM-tree (i.e., $K = 1$ and $Z = 1$).

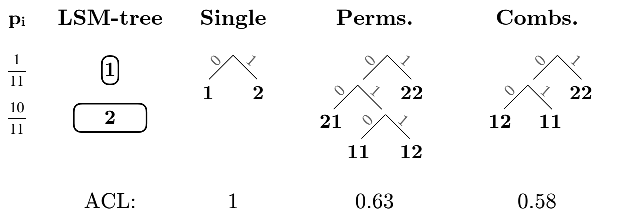


Figure 7: Encoding run IDs collectively as permutations or combinations allows reducing the average code length.

tree labeled Perms with an ACL of 0.63 in Figure 7. This is an improvement over encoding one run ID at a time. The intuition for the improvement is that we can represent the most common permutations with fewer bits than the number of symbols in the permutation. Figure 6 shows that as we increase the permutation size, the ACL of the resulting Huffman tree approaches the entropy.

Encoding Run IDs as Combinations. In the example in Figure 7, there are two permutations of the same run IDs: 21 and 12. For a query that encounters either permutation, the same lookup process ensues: we check Run 1 for the key (i.e., first the fingerprint and in case of a positive also in storage) and if we did not find it we proceed to check Run 2. The fact that both permutations trigger the same process implies that permutations encode redundant information about order. Instead, we can encode combinations of Run IDs, as shown in Figure 7, where the combination 12 replaces the two prior permutations. As there are fewer combinations than permutations ($\binom{S+A-1}{S}$ as opposed to A^S), we need fewer bits to represent them, and so the ACL can drop even lower than before.

To lower bound the ACL with encoded combinations, we derive a new entropy expression H_{comb} in Equation 10 by subtracting all information about order from our original entropy expression H (from Equation 7). This order information amounts to $\log_2(S!)$ bits to permute S run IDs while binomially discounting $\log_2(j!)$ bits for any run ID that repeats j times[9]. We divide by S to normalize the expression to be per entry rather than per bucket.

$$H_{comb} = H - \frac{1}{S} \cdot \left(\log_2(S!) - \sum_{i=1}^{A} \sum_{j=0}^{S} \binom{S}{j} \cdot f_i^j \cdot (1-f_i)^{S-j} \cdot \log_2(j!) \right) \tag{10}$$

Figure 8 compares H_{comb} to H as we increase the number of collectively encoded run IDs[10]. We observe that the more collectively encoded run IDs, the more H_{comb} drops as it eliminates more redundant information about order relative to H.

To use encoded combinations in practice, we must sort the fingerprints within each bucket by their run IDs to be able to identify which fingerprint corresponds to which run ID. To do the actual encoding, we feed all possible combinations along with their probabilities into a Huffman encoder. We express the probability c_{prob} of a combination c in Equation 11 using the multinomial distribution, where $c(j)$ denotes the number of occurrences of the jth run ID within the combination. For example, for the combination 12 in Figure 7, we have $S = 2$, $c(1) = 1$ and $c(2) = 1$. Hence, the probability

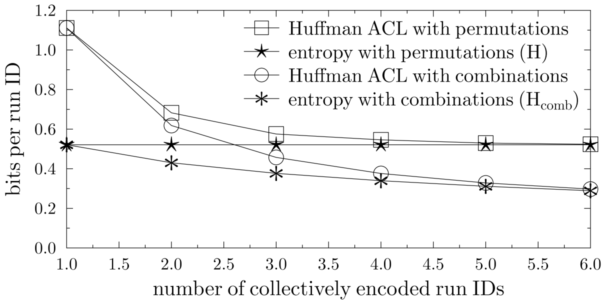


Figure 8: Encoding run IDs as large combinations minimizes their average code length.

is $2! \cdot (1/11) \cdot (10/11) = 20/121$.

$$c_{prob} = S! \cdot \prod_{j=1}^{A} \frac{f_j^{c(j)}}{c(j)!} \tag{11}$$

With combinations, the ACL 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 run ID rather than per bucket). We observe that the combinations ACL dominates the permutations ACL in Figure 8, and that it converges with the combinations entropy as we increase the number of collectively encoded run IDs. In the rest of the paper, we continue with encoded combinations as they achieve the best compression.

4.3 Aligning Codes with Fingerprints

With run ID codes being variable-length due to compression, aligning them along with fingerprints within FF buckets becomes a challenge. We illustrate this in Figure 9 Part (A) by aligning one run ID combination code for two entries along with two five-bit fingerprints (FPs) within sixteen-bit FF buckets. This example is based on the LSM-tree instance in Figure 4 except we now encode run ID combinations instead of encoding every run ID individually. The term $l_{x,y}$ in the figure is the code length assigned to a bucket with coinciding run IDs x and y. We observe that while some codes and fingerprints perfectly align within a bucket (Row I), others exhibit underflows (Row II) and overflows (Rows III and IV). Underflows occur within buckets with frequent run IDs as a result of having shorter codes. They are undesirable as they waste bits that could have otherwise been used for increasing fingerprint sizes. On the other hand, overflows occur in buckets with less frequent run IDs as a result of having longer codes. They are undesirable as they require storing the rest of the bucket content elsewhere, thereby increasing memory overheads.

We illustrate the contention between overflows and underflows in Figure 10 with the curve labeled *uniform fingerprints*[11]. The figure varies the maximum allowed fraction of overflowing FF buckets and measures the maximum possible corresponding fingerprint size. As shown, with a uniformly sized fingerprints, the fingerprint size has to rapidly decrease to guarantee fewer overflows.

To address this, our insight that the run ID combination distribution (in Equation 11) is heavy-tailed since the underlying run ID distribution is approximately geometric. Our approach is to therefore to guarantee that codes and fingerprints perfectly align

[9]Since combinations are multinomially distributed, an alternative approach for deriving the same expression is through the entropy function of the multinomial distribution.
[10]This example uses a Leveled LSM-tree with $T = 10$, $K = 1$, $Z = 1$ and $L = 6$.

[11]The figure is drawn for a Lazy-Leveled LSM-tree with configuration $T = 5$, $K = 4$, $Z = 1$, $L = 6$ and an FF with 32 bit buckets containing 4 entries.

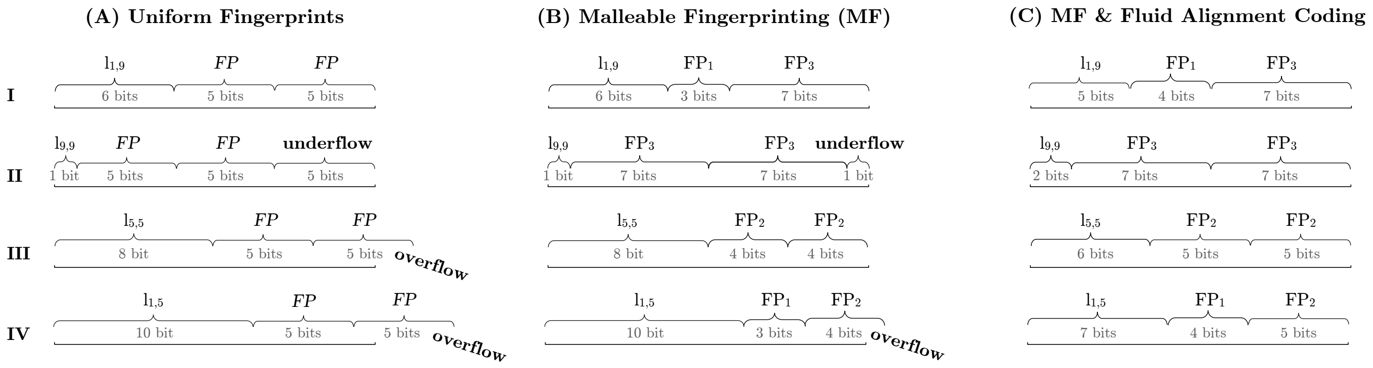


Figure 9: Storing compressed run ID codes with uniformly sized fingerprints leads to poor bucket alignment (Part A). We solve this problem using Malleable Fingerprinting (Part B) and Fluid Alignment Coding (Part C).

within the most probable combinations by adjusting their sizes, while allowing all the other combinations along the distribution's heavy tail to overflow. We achieve this in two steps using two complementary techniques: Malleable Fingerprinting (MF) and Fluid Alignment Coding (FAC).

Malleable Fingerprinting (MF). To facilitate alignment, MF allows entries from different LSM-tree levels to have different fingerprint sizes. However, an individual entry's fingerprint length stays the same even if it gets swapped across buckets by the FF's collision resolution method. This means that no fingerprint bits ever need to be dynamically chopped or added. Once an entry is moved into a new level, MF assigns it a new fingerprint size if needed while it is brought to memory to be sort-merged.

The question that emerges with MF is how to choose a fingerprint length for each level to strike the best possible balances between fingerprint sizes and overflows. We frame this as an integer programming problem[12] whereby FP_i denotes the (positive integer) length of fingerprints of entries at Level i. The objective is to maximize the average fingerprint size as expressed in Equation 12[13].

$$\text{maximize} \quad \sum_{i=1}^{L} FP_i \cdot p_i \tag{12}$$

We constrain the problem using an additional parameter NOV for the fraction of **n**on-**ov**erflowing buckets we want 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 run ID combinations in C whose cumulative probabilities fall just above NOV[14]. We add it to the problem in Equation 13 as a constraint requiring that for all $c \in C_{freq}$, the code length (denoted as l_c) plus the cumulative fingerprint length (denoted as c_{FP}) do not exceed the number of bits B in the bucket[15].

$$\forall c \in C_{freq}: \quad c_{FP} + l_c \leq B \tag{13}$$

[12]Note that the problem is only defined for $2^B > |C| = \binom{S+A-1}{S}$, meaning the bucket size B has to be at least large enough to identify all combinations uniquely.

[13]While more advanced objective functions are possible (e.g., minimizing the average FPR per fingerprint $\sum_{i=1}^{L} 2^{-FP_i} \cdot p_i$), such functions lead to a more complex optimization workflows without bringing significant additional benefit.

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

[15]More concretely, for a combination c let $c(j)$ denote the number of occurrences of the j^{th} run ID. 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} run ID, which is at Level $\lceil j/K \rceil$.

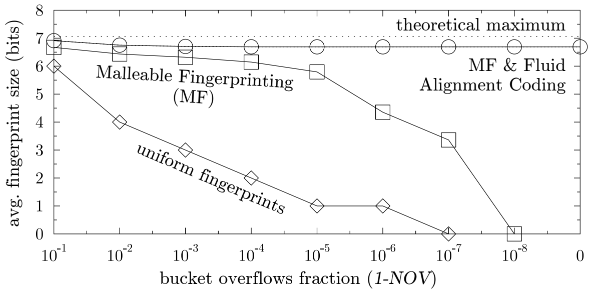


Figure 10: Malleable Fingerprinting and Fluid Alignment Coding solve the bucket alignment problem.

While integer programs are NP-complete and thus difficult to globally optimize, we exploit the particular structure of our problem with an effective hill-climbing approach shown in Algorithm 1. The algorithm initializes all fingerprint sizes to zero. It then increases larger levels' fingerprint size as much as possible, moving to a next smaller level when the overflow constraint in Equation 13 is violated. The rationale for lengthening larger levels' fingerprints first is that their entries are more frequent. In this way, the algorithm follows the steepest ascent. Figure 9 shows how MF reduces the severity of underflows (Row II) while at the same time eliminating some overflows (Row III). As a result, it enables better balances between overflows and average fingerprint size as shown in Figure 10.

.

Fluid Alignment Coding (FAC). Figure 9 Part (B) illustrates that even with MF, underflows and overflows can still occur (Rows II and IV, respectively). To further mitigate them, we introduce FAC. FAC exploits a well-known trade-off in information theory that the smaller some codes are set within a prefix code, the longer other codes must be for all codes to remain uniquely decodable. This trade-off is embodied in the Kraft-McMillan inequality [38, 49], 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 that occupy the underflowing bits for very frequent bucket combinations. As a result, the codes for all other bucket combinations can be made shorter. This creates more space in less frequent bucket combinations, which can be exploited to reduce overflows and to increase fingerprint sizes for smaller levels. We illustrate this idea in Figure 9

```
1 for $i \leftarrow 1$ to $L$ by 1 do  $FP_i \leftarrow 0$ end
2 for $i \leftarrow L$ to 1 by $-1$ do
3     for $b \leftarrow 1$ to $M$ by 1 do
4         curr $\leftarrow FP_i$
5         $FP_i \leftarrow$ b
6         if overflow constraint is violated then  $FP_i \leftarrow$ curr
```
Algorithm 1: Maximizing average fingerprint size with hill-climbing.

Part (C). The combination in Row II, which is the most frequent in the system, is now assigned a longer code than before. 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 overflows constraint (Equation 13) by a new constraint, shown in Equation 14. 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}. Furthermore, it ensures that all other buckets combinations not in C_{freq} can be uniquely identified using unique codes that are at most the size of a bucket B.

$$1 \geq \sum_{c \in C} \begin{cases} 2^{-(B-c_{FP})}, & \text{for } c \in C_{freq} \\ 2^{-B}, & \text{else} \end{cases} \tag{14}$$

Note that Equation 14 does not rely on knowing Huffman codes in advance (i.e., as Equation 13 does). Thus, we can run the Huffman encoder after rather than before finding the fingerprint lengths using Algorithm 1. Third, we run the Huffman encoder only on combinations in C_{freq} while setting the frequency input for a combination c as $2^{-(B-c_{FP})}$ as opposed to using its multinomial probability (in Equation 11) as before. This causes the Huffman encoder to generate codes that exactly fill up the leftover bits $B-c_{FP}$. Fourth, 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. In this way, we can identify and decode all codes across both sets uniquely.

Figure 10 shows that MF and FAC eliminate the contention between overflows and fingerprint size when applied together. In fact, they keep the average fingerprint size close (within half a bit in the figure) of the theoretical maximum, obtained by subtracting the combinations entropy (in Eq. 10) from the number of bits per entry M. We use MF and FAC by default for the rest of the paper.

Construction Time. Algorithm 1's run-time is $O(L \cdot M \cdot |C|)$, where $L \cdot M$ is the number of iterations and $|C|$ is the cost of evaluating the constraint in Equation 14. In addition, the time complexity of the Huffman encoder is $O(|C| \cdot \log_2(|C|))$[16]. This workflow is seldom invoked (i.e., 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 10 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 from the onset. Instead, we give a conservative approximation to still allow reasoning about system behavior. First, we observe that with FAC, the average code length is always at least one bit per entry, and so we use our upper bound ACL_{UB} from Equation 9 to slightly overestimate it. Hence, we approximate the average fingerprint size as $M - ACL_{UB}$ and thus the FPR over a single fingerprint as $2^{-(M-ACL_{UB})}$. We multiply this expression by a factor of Q, which denotes average number of fingerprints searched by the underlying FF per probe (e.g., for a Cuckoo filter with four entries per bucket $Q \approx 8$). Thus, we obtain Equation 15, for which the interpretation is the expected number of false positives for a query to a non-existing key. Practically, the actual FPR tends to be off from this expression by a factor of at most two.

$$FPR_{approx} = Q \cdot 2^{-M+ACL_{UB}} \tag{15}$$

4.4 Optimizing Decoding & Recoding

We now discuss the data structures needed to decode run IDs on application reads and to recode them on writes. Specifically, we show how to prevent these structures from becoming bottlenecks.

Cached Huffman Tree. Since Huffman codes are variable-length, we cannot generally decode them in constant time (e.g., using a lookup table) as we do not know from the onset how long a given code in question is. Hence, decoding a Huffman code is typically done one bit at a time by traversing the Huffman tree from the root to a given leaf based on the code in question. A possible problem is that if the Huffman tree is large, traversing it can require up to one memory I/O per node visited. To restrict this cost, we again use the insight that the bucket combination distribution in Equation 11 is heavy-tailed. Hence, it is feasible to store a small Huffman Tree partition in the CPU caches to allow to quickly decode only the most common combination codes.

To control the cached Huffman tree's size, we set the parameter NOV from the last subsection to 0.9999 so that the set of combinations C_{freq} for which we construct the Huffman tree includes 99.99% of all combinations we expect to encounter. Figure 11 measures the corresponding tree's size[17]. Since it occupies a few tens of kilobytes, it is small enough to fit in the CPU caches. In fact, the figure highlights an important property that as we increase the data size, the cached Huffman tree's size converges. The reason is that the probability of a given bucket combination (in Equation 11) is convergent with respect to the number of levels, and so any set whose size is defined in terms of its constituent combinations' 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, we use a Decoding Table in main memory for all other combination codes not in C_{freq}. To ensure fast decoding speed for DT, we exploit the property given in the last subsection that all bucket combinations not in C_{freq} are assigned uniformly sized codes of size B bits. As these codes all have the same size, we know from the onset how many bits to consider, and so we can map these codes to labels in a lookup array as opposed to a tree. This guarantees decoding speed

[16] To express these bounds more loosely but in closed-form, note that $|C| = \binom{A+S-1}{S} < (A+S-1)^S \cdot (\frac{e}{S})^S < A^S$.

[17] We continue here with the LSM-tree configuration from Figure 4. Each Huffman tree node is eight bytes.

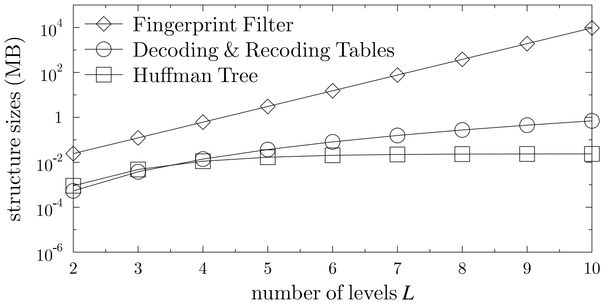


Figure 11: The Huffman tree size converges while the de/recoding table sizes grow slowly with respect to data size.

in at most one memory I/O. Figure 11 measures the DT size as we increase the number of levels on the x-axis (each DT entry is eight bytes). As the DT contains $\approx |C| = \binom{A+S-1}{S}$ entries, its size grows slowly as we increase the number of levels (and thus the number of runs A). We observe that it stays smaller than a megabyte even for a very large LSM-tree instance with ten levels.

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 FF 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 run IDs while handling application writes, we employ a Recoding Table. We use a fixed-width format to represent a run ID combination, and so the RT can also be structured as a lookup array. It costs at most one memory I/O to access and its size scales the same as the Decoding Table in Figure 11. Note that the most frequent RT entries are in the CPU caches during run-time and thus cost no memory I/Os to access.

Memory Cost Summary. Figure 11 also illustrates the FF size as we increase the number of LSM-tree levels. We observe that all auxiliary data structures are comparatively small, and we've seen that they entail few memory accesses. Thus, Chucky prevents de/recoding from becoming a performance or space bottleneck.

4.5 Integration with Cuckoo Filter

We now show how to integrate Chucky with Cuckoo Filter (CF) [28], which we employ due to its design simplicity and ease of implementation. CF consists of an array of buckets, each with four fingerprint slots. During insertion, an entry with key x is hashed to two buckets b_1 and b_2 using Equations 16 and 17. A fingerprint of key x is then inserted into whichever bucket has space.

$$b_1 = hash(x) \tag{16}$$

$$b_2 = b_1 \oplus \ hash(x's \ fingerprint) \tag{17}$$

If both buckets are full, however, some fingerprint y from one of these buckets is evicted to clear space. The fingerprint y is swapped into its alternative bucket using Equation 18, which does not rely on the original key (by virtue of using the xor operator) but only on the fingerprint and the bucket i that currently contains y.

$$j = i \oplus \ hash(y) \tag{18}$$

The swapping process continues recursively either until a free bucket slot is found for all fingerprints or until a swapping threshold is reached, at which point the original insertion fails. Querying requires at most two memory I/Os as each entry is mapped to two possible buckets. Henceforth in the paper, we employ a Cuckoo filter with four slots per bucket. Such a tuning is known to be able to reach 95% capacity with high probability without incurring insertion failures and with only 1-2 amortized swaps per insertion.

To implement Chucky on top of CF, we place a combination code at the start of each CF bucket followed by variable-sized fingerprints as shown in Section 4.3. We represent empty fingerprint slots using a reserved all-zero fingerprint coupled with the most frequent run ID to minimize the corresponding combination code length. In addition, we make the following adjustments.

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, specifically whenever LSM-tree's capacity just crosses a power of two. To fix this, we borrow from Vacuum filter [69] the idea of partitioning a CF into multiple independent CFs, each of which is a power of two, but where the overall number of CFs is flexible. In this way, capacity becomes adjustable by varying the number of CFs, and we map each key to one of the constituent CFs using a hash modulo operation. We set each CF to be 8MB.

Sizing & Resizing. When Chucky reaches capacity, it needs to be resized to accommodate new data. However, a CF cannot be resized efficiently. The simplest approach is to rebuild Chucky from scratch when it reaches capacity. However, this approach forces an expensive scan over the dataset to reinsert all entries into the new instance of Chucky. Instead, 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 instance of Chucky and thereby obviate the need for an additional scan. We set the size of the new instance of Chucky to be larger by a factor of $\frac{T}{T-1} \cdot 1.05$ than the current data size to accommodate data growth until the next full merge and to always maintain $\approx$ 5% spare capacity across all the CFs to prevent insertion failures[18].

Minimum Fingerprint Size. Since Chucky assigns variable fingerprint sizes to entries at different levels, a problem arises whereby the CF can map different versions of an entry from across different levels to more than two CF buckets. We resolve this by ensuring that all fingerprints comprise at least X bits, and we adapt the CF to determine an entry's alternative bucket based on its first X bits[19]. This forces all versions of the same entry to reside in the same pair of CF buckets. In accordance with the Cuckoo filter paper [28], we set the minimum fingerprint size to 5 bits to ensure that an entry's two buckets are independent enough to achieve a 95% load factor.

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 eight versions of a given entry. Some FFs can address this problem out-of-the-box using embedded fingerprint counters (e.g., Counting Quotient Filter [55]). For our CF design, however, we address this issue using an additional hash table (AHT), which maps from bucket

[18]We assume here that entry sizes are generated using a non-changing distribution. However, if incoming updated or inserted entries are significantly smaller than earlier data, Chucky may fill up prematurely. In this case, we preempt the next full merge.
[19]While this constraint slightly reduces the average fingerprint size given by Algorithm 1, it provides lower FPR variance as no entries are assigned very small fingerprints.

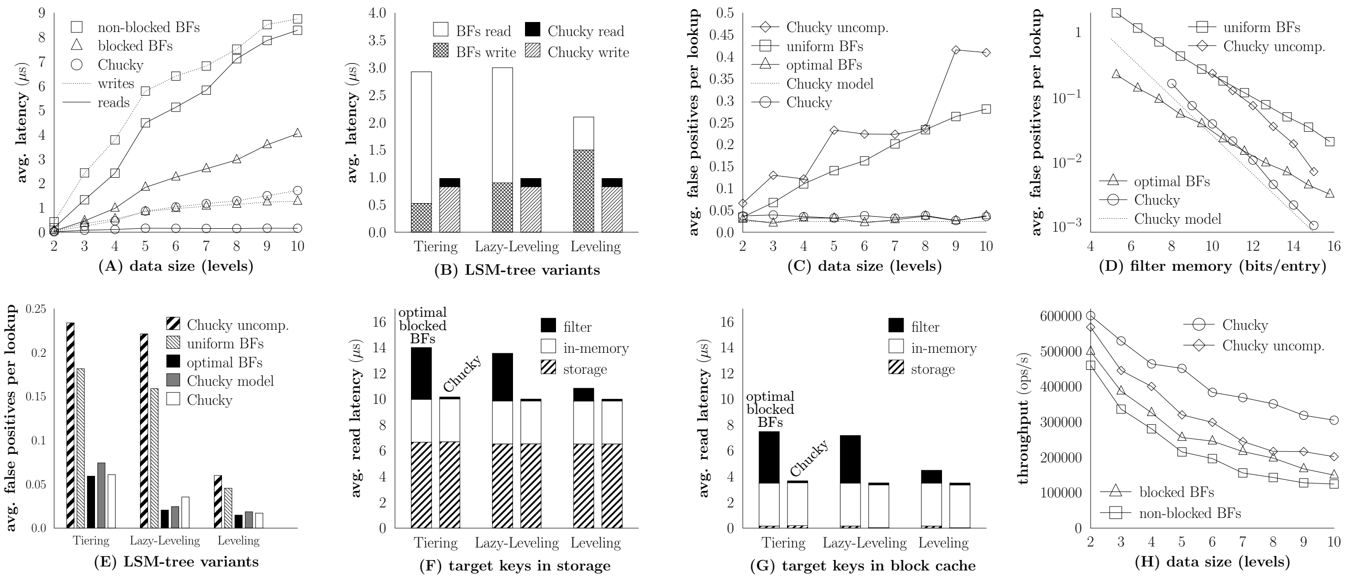


Figure 12: Chucky scales memory I/Os with data size (A) and for any LSM-tree variant (B). At the same time, Chucky's false positive rate scales with data sizes (C) with memory footprint (D), and for any LSM-tree variant (E). Hence, it improves read latency when target data is in fast storage (F) or cached (G), resulting in more scalable throughput (H).

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 FF bucket that's encountered during a query or update thus adding to them at most $O(1)$ additional memory access.

Persistence. For each run, we persist its entries' fingerprints in storage. During recovery, we read only the fingerprints from storage and thus avoid a full scan over the data. We insert each fingerprint along with its run ID into a brand new CF series 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.

5 EVALUATION

We now show experimentally that unlike existing designs, Chucky scales in terms of memory I/O and storage I/O at the same time.

Infrastructure. We use a machine with 32GB DDR memory and four 2.7 GHz cores with 8MB L3 caches running Ubuntu 18.04 LTS and connected to a 512GB SSD through PCIe.

Baselines. We use our own LSM-tree implementation, designed based on Dostoevsky [20], and which we are gearing towards commercial use. We added as baselines blocked [58] and non-blocked BFs with uniform false positive rates (FPRs) to represent design decisions in RocksDB [27] and Cassandra [3], respectively. We also support optimal FPRs [19]. We implemented Chucky as discussed in Section 4, and we support a version of Chucky with uncompressed run IDs as a loose representative of SlimDB [60].

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 ≈ 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. In Figure 12 Parts (A) to (E), we evaluate filter performance in isolation from other parts of the system (e.g., memtable, storage I/Os, block cache, block index). We then focus on end-to-end performance in Parts (F) to (H).

Memory I/O Scalability. Figure 12 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. 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 mulitple memory I/Os per filter across a growing number of filters. We drop non-blocked BFs henceforth in the evaluation as they are noncompetitive. 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 run IDs need to be updated. Crucially, we observe that Chucky is the only baseline that's able to keep read latency stable with data size as each read requires a constant number of memory I/Os.

Figure 12 Parts (B) stacks read and write latency with Chucky against blocked BFs with different LSM-tree variants. Chucky offers better cost balances across the board, mostly for its lower read latency. Nevertheless, Chucky also improves write cost for Leveled LSM-tree designs. The reason is that with Leveling, merging is greedy and so BFs are rapidly reconstructed leading to multiple BF insertions per entry per level. In contrast, Chucky always requires just one update per entry per level. Overall, Chucky not only

improves the filter read/write cost balances but also makes them independent of the underlying LSM-tree variant. This makes the system easier to reason about and tune.

FPR Scalability. Figure 12 Part (C) compares the FPR for Chucky with both compressed and uncompressed run IDs to blocked BFs with both uniform and optimal space allocation. As we increase the data size, the FPR of Chucky with uncompressed run IDs increases since the run IDs 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 run ID code length converges, thus allowing most fingerprints to stay large. The figure also includes the FPR model of Chucky from Equation 15 to show that it gives a reasonable approximation of the FPR in practice.

Figure 12 Part (D) 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[20]. Part (E) show that these results hold for any LSM-tree variant. Overall, Parts (C) to (E) show that Chucky is on par with optimal BFs with respect to scaling the FPR vs. memory budget trade-off.

Data in Storage vs. Memory. Figure 12 Parts (F) and (G) measure end-to-end read latency with uniform and Zipfian (with parameter $s = 1$) workloads, respectively. Read latency is broken in three components: (1) storage I/Os, (2) in-memory search across the fence pointers, buffer, and block cache, and (3) 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 (G), 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.

Throughput Scalability. Figure 12 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 [18]). The BF baselines do not scale well as they issue memory I/Os across a growing number of BFs. Chucky with uncompressed run IDs also exhibits deteriorating performance as its FPR grows and leads to more storage I/Os. Chucky with compressed run IDs 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.

[20]To allow Chucky to operate with fewer than eight bits per entry while also keeping the FPR low, it is possible to use a BF at the largest level of the LSM-tree and an FF for all smaller levels. We keep such a design out of scope for now due to space constraints.

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 [25, 27, 48]), there is significant interest in optimizing LSM-tree performance [43]. Most designs to date focus on managing compaction overheads, for example by separating values from keys [14, 41], partitioning runs into files and merging based on maximal file intersections [5, 65, 67], keeping hot entries in the buffer [5], packing the buffer more densely [10], opportunistically merging while scanning [56], scheduling carefully to prevent tail latencies [6, 42, 64], using customized or dedicated hardware [1, 33, 68, 70, 73], and to control delete persistence [63].

Another strain of work uses lazier compaction policies [50, 59, 60, 71, 72], 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 [19–21]. SlimDB [60] shows how to reduce memory I/Os using a Cuckoo filter, but its FPR does not scale well as discussed in Section 3. In contrast, we show how to scale the FPR and memory I/Os at the same time (for any merge policy including lazy ones) by replacing the BFs by an FF with compressed run IDs.

Fingerprint Filters. While we build Chucky on top of Cuckoo filter [28] for its design simplicity, many other FF designs with nuanced properties have been proposed. Many of them strive for better cache locality by using linear probing [7, 55], biasing cuckoo hashing towards using one bucket [11], or ensuring both candidate buckets are physically close [69]. Vacuum filter offers better memory utilization by allowing the filter size to not be a power of two [69]. Some designs allow to delay resizing the filter by chaining overflow filters [17, 69] or by sacrificing fingerprint bits [55]. Xor filter supports a better FPR in exchange for higher construction time [32, 54]. Adaptive Cuckoo filter modifies fingerprints to prevent repeated false positives due to access to the same entries [52]. Other designs prevent overflows due to duplicate insertions using internal counters [55]. 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 [46, 74], which can save storage I/Os for range reads but also require more memory I/Os to probe and construct. Making such filters more memory I/O efficient is an open challenge.

Bloom Filters (BF). Numerous BF variants have been proposed [12, 45, 66], which enable counting [9, 29, 62], compressibility [51], vectoriziation [57], deletes for some but not all entries [61], efficient hashing [23, 37] and cache locality [13, 22, 39, 40, 58]. Bloomier filter allows to associate values with keys but is unable to compress values and is more complicated than fingerprint filters [15, 16].

7 CONCLUSION

This paper proposes Chucky as a more scalable alternative to Bloom filters for LSM-tree point read filtering. Chucky uses a compact hash table in memory to map all entries to their locations within the LSM-tree, and it compresses this location information to scale the false positive rate. In this way, Chucky is the first LSM-tree filter that scales memory I/Os and storage I/Os at the same time.

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] 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).

[6] 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).

[7] 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.

[8] Bloom, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM 13*, 7 (1970), 422–426.

[9] Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., and Varghese, G. An improved construction for counting bloom filters. In *European Symposium on Algorithms* (2006).

[10] 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.

[11] Breslow, A. D., and Jayasena, N. S. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. In *VLDB* (2018).

[12] Broder, A. Z., and Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. *Internet Mathematics 1* (2002), 636–646.

[13] 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.

[14] Chan, H. H. W., Li, Y., Lee, P. P. C., and Xu, Y. HashKV: Enabling Efficient Updates in KV Storage via Hashing. *ATC* (2018).

[15] Charles, D., and Chellapilla, K. Bloomier filters: A second look. In *European Symposium on Algorithms* (2008).

[16] 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).

[17] Chen, H., Liao, L., Jin, H., and Wu, J. The dynamic cuckoo filter. In *IEEE ICNP* (2017).

[18] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. *SoCC* (2010).

[19] Dayan, N., Athanassoulis, M., and Idreos, S. Monkey: Optimal Navigable Key-Value Store. *SIGMOD* (2017).

[20] 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).

[21] Dayan, N., and Idreos, S. The log-structured merge-bush & the wacky continuum. In *SIGMOD* (2019).

[22] Debnath, B., Sengupta, S., Li, J., Lilja, D. J., and Du, D. H. Bloomflash: Bloom filter on flash-based storage. In *ICDCS* (2011).

[23] Dillinger, P. C., and Manolios, P. Bloom Filters in Probabilistic Verification. *Formal Methods in Computer-Aided Design* (2004).

[24] 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).

[25] Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., and Strum, M. Optimizing Space Amplification in RocksDB. *CIDR* (2017).

[26] 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).

[27] Facebook. RocksDB. *https://github.com/facebook/rocksdb*.

[28] Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. Cuckoo Filter: Practically Better Than Bloom. *CoNEXT* (2014).

[29] 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.

[30] 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).

[31] Golomb, S. Run-length encodings. *IEEE transactions on information theory* (1966).

[32] Graf, T. M., and Lemire, D. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics* (2020).

[33] 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).

[34] Huffman, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[35] 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).

[36] Jagadish, H. V., Narayan, P. P. S., Seshadri, S., Sudarshan, S., and Kanneganti, R. Incremental Organization for Data Recording and Warehousing. *VLDB* (1997).

[37] Kirsch, A., and Mitzenmacher, M. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms 33*, 2 (2008), 187–218.

[38] Kraft, L. G. *A device for quantizing, grouping, and coding amplitude-modulated pulses.* PhD thesis, MIT, 1949.

[39] Lang, H., Neumann, T., Kemper, A., and Boncz, P. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. In *VLDB* (2019).

[40] Lu, G., Debnath, B., and Du, D. H. C. A Forest-structured Bloom Filter with flash memory. *MSST* (2011).

[41] 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).

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

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

[44] 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).

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

[46] 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).

[47] Manegold, S., Boncz, P. A., and Kersten, M. L. Optimizing Database Architecture for the new Bottleneck: Memory Access. *VLDBJ 9*, 3 (2000), 231–246.

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

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

[50] 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).

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

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

[53] 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.

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

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

[56] Pilman, M., Bocksrocker, K., Braun, L., Marroquin, R., and Kossmann, D. Fast Scans on Key-Value Stores. *PVLDB 10*, 11 (2017), 1526–1537.

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

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

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

[60] 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.

[61] 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.

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

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

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

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

[66] 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.

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

[68] 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).

[69] 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).

[70] 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).

[71] 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).

[72] 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.

[73] 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).

[74] Zhang, Y., Li, Y., Guo, F., Li, C., and Xu, Y. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. *HotStorage* (2018).