Zeno Filter: To Infinity in Tiny Steps

HYUHNG MIN KIM, University of Toronto, Canada
NAVID ESLAMI, University of Toronto, Canada
NIV DAYAN, University of Toronto, Canada

Filters are compact probabilistic data structures that answer "have I seen this before?" without storing everything they have seen so far. Databases rely on them to skip unnecessary storage lookups and save time. In many applications, the total data size is unknown in advance, creating the need for filters that can expand as the dataset grows. However, existing expandable filters suffer from two forms of space inefficiency. First, they double in size to prevent hash collisions, leaving 50% of space wasted immediately after expansion. Second, they expand out-of-place, temporarily keeping both the old and new filters in memory, further inflating space usage.

We introduce Zeno Filter to resolve this wastage. It addresses the first problem by enabling expansions of less than 2×, achieved by stretching the distance between entries until hash collisions can be resolved. It addresses the second by expanding in-place, using a succinct and efficient indirection layer to relocate entries without duplication. The result is an expandable filter that cuts space by up to 60% while matching the speed and accuracy of existing designs.

CCS Concepts: • **Information systems → Data structures**; • **Theory of computation → Bloom filters and hashing**.

Additional Key Words and Phrases: Approximate Set Membership, Bloom Filter, Quotient Filter, Data Growth, Scalability.

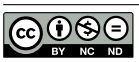
ACM Reference Format:
Hyuhng Min Kim, Navid Eslami, and Niv Dayan. 2026. Zeno Filter: To Infinity in Tiny Steps. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 251 (June 2026), 26 pages. https://doi.org/10.1145/3802128

1 Introduction

Filters. A filter is a compact probabilistic data structure that represents a set of keys and supports approximate set-membership queries. It guarantees no false negatives while allowing a small probability of false positives in exchange for significant savings in space. Owing to their compactness, filters are often stored in fast memory (e.g., DRAM or SRAM) even when the underlying data resides on disk or across the network. This enables systems to quickly rule out missing keys without accessing slower storage, thereby reducing latency and I/O overhead [15, 18, 19, 40, 44]. Since the introduction of the Bloom filter [3], filters have become fundamental components in data management systems [6, 13, 41], network [4, 28], security [24], and analytics [11, 12, 34].

The Need for Expandable Filters. Many modern applications handle continuously growing datasets whose final size is unknown. Traditional filters, such as Bloom filters, are static and require a fixed capacity to guarantee a target false-positive rate. The simplest way to handle growth — rebuilding a new filter from scratch [20, 51] — is costly, as it involves scanning the entire dataset. Over the past two decades, research has focused on making filters dynamically expandable without full reconstruction. A common technique is to allocate new filters in a chain as data grows [1, 9, 49];

Authors' Contact Information: Hyuhng Min Kim, hmkim@cs.toronto.edu, University of Toronto, Toronto, Canada; Navid Eslami, navideslami@cs.toronto.edu, University of Toronto, Toronto, Canada; Niv Dayan, nivdayan@cs.toronto.edu, University of Toronto, Toronto, Canada.



ACM 2836-6573/2026/6-ART251
https://doi.org/10.1145/3802128

insertions occur in the newest filter, but queries must check all filters, raising both query costs and false-positive rates. Other methods employ consistent hashing [32, 52] to link small filters in a ring, enabling elastic expansion, yet still incurring high query overhead due to the need to search across multiple filters.

State-of-the-art Expandable Filters. Modern expandable filters store fingerprints for each entry in a compact hash table (e.g., quotient filters [2] or cuckoo filters [27]). To accommodate growth, they migrate entries into a hash table twice the original size [10, 16, 17, 26, 39, 53, 54], repurposing one fingerprint bit to uniformly map entries to the larger table without rehashing the original keys. The latest design in the family, Aleph Filter [17], supports constant-time queries, insertions, and deletes, while maintaining a scalable false positive rate as data grows.

The space efficiency of expandable filters is critical in high-performance data systems. In append-only key-value stores [8, 21–23, 33], filters often serve as the primary in-memory index for on-disk data. Recent LSM-tree and B-tree designs rely on global filters to accelerate point and range lookups [20, 25, 26, 43, 48]. In these settings, the filter can become a memory bottleneck and compete with other critical structures, such as the buffer pool, over the available system memory.

Problem 1. Persistent Space Amplification. With traditional hash tables, expansion by an arbitrary small factor is possible by rehashing the keys to a new address space, which limits space amplification immediately after resizing. For filters, however, the original keys are unavailable and cannot be rehashed. The standard workaround is to reassign one bit from each fingerprint to the slot address, effectively doubling the filter's size. Consequently, this results in at least 50% space wastage immediately after expansion. The problem is difficult to mitigate because a bit is the smallest unit of storage — there is no natural way to reallocate less than one bit to achieve an expansion factor smaller than two.

Problem 2. Transient Space Amplification. During expansion, a larger filter is allocated while the original filter remains in memory, as entries are gradually migrated from the smaller to the larger filter. Although this excess space usage is temporary and disappears after migration, it still requires the system to reserve additional memory, which can force other resources to be swapped out and degrade performance.

Contributions. We introduce Zeno Filter[1], an expandable filter that addresses the above two problems. Zeno Filter is also a compact hash table that stores fingerprints of keys to support dynamicity. It allows to expand by a factor smaller than two using a novel technique called Stretching. The technique gradually increases the distance between entries until reaching the next power of 2, at which point it sacrifices a bit to map entries into the larger filter. Furthermore, it expands in-place by using an efficient layer of indirection that scales sub-linearly with respect to data size. This eliminates excess space usage during expansion while keeping the indirection layer cache-resident and thus fast to access.

Additional Contributions:

- We study how the expansion threshold dictates the trade-off between persistent space amplification and write cost.
- We show an efficient bit manipulation workflow that speeds up access through the indirection layer.
- We introduce a variant of Zeno Filter that leverages kernel primitives, improving performance at the cost of portability.
- We show how to support deletes without using auxiliary metadata as with previous approaches.

[1]We name this after Zeno of Elea, well known for his paradoxes concerning infinity.

Table 1. Terms used throughout the paper and the corresponding definitions.

Term	Definition
$h(\cdot)$	hash function
q	size of quotient in bits
f	initial fingerprint size in bits
ϵ	false positive rate
n	initial filter size in number of slots ($n = 2^q$)
N	current filter size in number of slots
α	expansion threshold ($0 < \alpha < 1$)

2 Background

Here we describe the Rank-and-Select Quotient Filter [39], InfiniFilter [16], and Aleph Filter [17], on top of which we build Zeno Filter.

2.1 Rank-and-Select Quotient Filter (RSQF)

An RSQF compactly stores the hashed representation of a key in a hash table. The key is first hashed into $f + q$ bits. The most significant f bits are the *fingerprint* and the least significant q bits are the *quotient*. The quotient represents the key's canonical slot, into which the key's fingerprint is inserted. As we'll show, however, hash collisions may push the fingerprint to the right of its canonical slot. Overall, the filter contains $n = 2^q$ slots.

Figure 1 illustrates an RSQF with $n = 8$ slots. In Figure 1-(a), three keys are inserted into different canonical slots based on the least significant $q = 3$ bits of their hash values. Throughout the paper, slot addresses are expressed in binary and are placed above their respective slots in all figures. The remaining $f = 4$ bits, marked in red, are stored in the slot as fingerprints.

Collision Resolution. The hash function may map multiple keys to the same canonical slot. Such hash collisions are resolved using Robin Hood hashing [7], where fingerprints with the same quotient are stored consecutively, pushing any other fingerprints to the right to make space. The set of fingerprints mapped to the same canonical slot is referred to as a *run*. The accumulated fingerprints of a run may also shift subsequent runs from their canonical slots.

Inserting three keys in Figure 1-(a) forms three runs: Run `000`, Run `011`, and Run `100`. Figure 1-(b) shows the state of the filter after an insertion of three additional keys. The three keys coincidentally map to the same three canonical slots, increasing the length of each run to two slots. Run `011` shifts Run `100` by one slot to the right.

Metadata. Since a fingerprint may be pushed to the right due to collisions, RSQF uses two bitmaps to track which canonical slot a fingerprint belongs to: the `occupieds` bitmap and `runends` bitmap. In each bitmap, there are as many bits as there are slots in the filter, and the i-th bit is associated with the i-th slot. An `occupieds` bit, when set, indicates that there exists at least one entry associated with this canonical slot. In contrast, a set `runends` bit indicates that this slot is the last slot of a particular run. In Figure 1-(b), the `occupieds` bit at Slot `000` is set because there is at least one entry belonging to this canonical slot, and the `runends` bit at Slot `001` is set because it holds the last entry of a run.

To optimize query performance, RSQF groups every 64 slots into a block. It also maintains an 8-bit `offset` field for each block to indicate how many of the block's first slots contain entries that overflowed from the block to its left. In Figure 1-(b), the offset field of the second block is set to 1 as there is one overflowing entry. Any query into such a block can now skip irrelevant data pushed from the left. With two metadata bits per slot and an 8-bit field for each block, RSQF uses 2.125 metadata bits per slot.

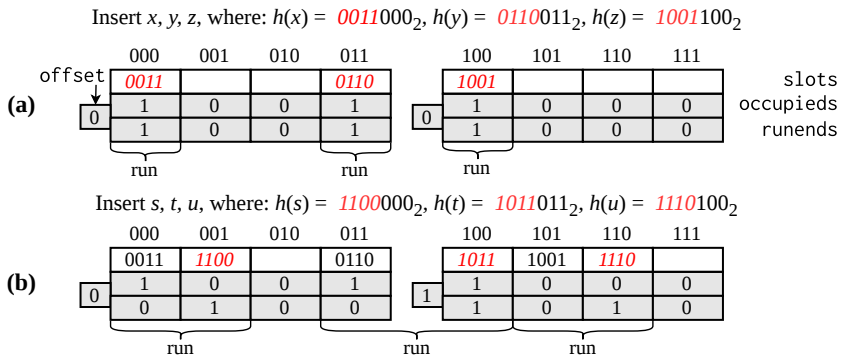


Fig. 1. RSQF uses the least significant bits of a key's hash as the canonical slot address and inserts the most significant bits as fingerprint.

Run Identification. Since a run may have been pushed to the right, any operation (i.e., query, insertion, or deletion) must first locate the target run if it exists. RSQF first hashes the key and finds the block where the key's canonical slot resides. It then skips entries pushed from preceding blocks using the offset field. The final step is done using rank and select primitives. A rank(i) returns the number of set bits up to position i in a bitmap, while select(i) returns the position of the i-th set bit. By applying a rank operation on the `occupieds` bitmap followed by a select operation on the `runends` bitmap, RSQF locates the end of the run belonging to the target canonical slot. These rank and select commands are optimized using advanced x86 instructions (TZCNT, PDEP, and POPCOUNT [14, 29, 39]) to obviate looping over the bitmaps.

Insertions and Deletions. An insertion identifies and inserts a fingerprint into the target run, potentially shifting other adjacent runs one slot further to the right. A deletion removes a matching fingerprint from the target run, potentially shifting pushed adjacent runs one slot back to the left. For example, in Figure 1-(b), inserting Key t grows the length of Run `011` to two and shifts Run `100` to the right by one slot. As the end of Run `011` is changed from Slot `011` to `100`, it sets the `runends` bit in Slot `011` to 0 and the `runends` bit in Slot `100` to 1. Also, the `offset` field of the second block is set to 1 since the entry for Key t in Slot `100` is shifted from the left block.

Query. A query also first locates the run of the target key. If the `occupieds` bit is not set, it returns negative. If it is set, it locates the associated run, scans through the run, and returns positive if a matching fingerprint is found and negative otherwise.

False Positive Rate. A query to a non-existing key may result in a *false positive* if the queried key maps to a run with a matching fingerprint. The false positive rate (FPR) is $\epsilon \leq 2^{-f}$ [39] since each run consists of ≈ 1 entry on average, and a fingerprint match occurs with a probability of 2^{-f}.

Expansion Threshold. As RSQF fills up, the cost of insertions increases as more entries on average need to be pushed to make space. To mitigate this cost and make space for more entries, it must expand. Expansion is triggered when the fraction of non-empty slots exceeds a configurable threshold α ($0 < \alpha < 1$). That is, the filter expands when the number of non-empty slots exceeds $n \cdot \alpha$. A higher threshold makes the filter more densely populated, but harms performance as there are more entries on average in a run. A user can tune the threshold to control this trade-off.

Fingerprint Sacrifice. To expand without rereading the original keys and rehashing them, RSQF repurposes the least significant bit from each fingerprint and assigns it as the most significant bit of the slot address. This process maps the fingerprints uniformly across a 2x larger RSQF while narrowing each slot by one bit. This method of filter expansion is referred to as the Fingerprint Sacrifice scheme, as a single bit of each fingerprint is sacrificed on each expansion.

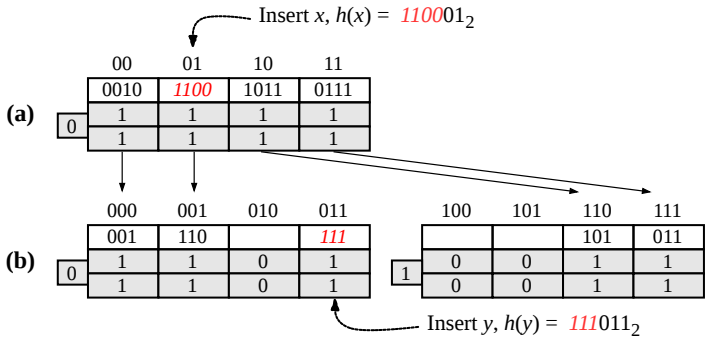


Fig. 2. RSQF expands by taking a single bit from the fingerprint and mapping the entry to a larger filter. Keys assigned after expansion are assigned one less bit.

Figure 2 provides an example. Key x is inserted at Canonical Slot `01` before expansion. During expansion, the key gets mapped to Slot `001`, as the least significant bit of its fingerprint is appended to its slot address. Keys inserted after expansion are assigned smaller fingerprints to fit the narrow slots (e.g., Key y in the figure).

Since the fingerprint becomes shorter by a bit on each expansion, the FPR doubles accordingly. In other words, the FPR of the RSQF is $\epsilon \leq 2^{-f+p}$, where p is the number of expansions. If the current filter size is N and the initial filter size was n, p is $\log_2 \frac{N}{n}$ since the filter size doubles in every expansion. Hence, the FPR of the RSQF deteriorates linearly with the data size. Moreover, the filter supports up to f expansions, at which point we run out of fingerprint bits.

2.2 InfiniFilter

InfiniFilter addresses the limitations of the Fingerprint Sacrifice method by providing a more scalable false positive rate and allowing an unbounded number of expansions to take place[2].

Variable-Length Fingerprints. InfiniFilter's core innovation is supporting a variable-length fingerprint in each slot. This allows sacrificing a bit from existing fingerprints to expand, while still setting longer fingerprints to newer entries. It does so by fixing the slot width across expansions, padding each fingerprint with a unary code to fill the remaining space in the slot.

The unary code consists of a sequence of ones terminated by a zero (e.g., `0`, `10`, `110`), where the number of consecutive ones encodes an integer value. The number of ones in this encoding represents the number of expansions the fingerprint has gone through. Importantly, this encoding is self-delimiting, allowing the filter to parse the slot to determine exactly where the unary code ends and the fingerprint begins.

InfiniFilter also expands by employing Fingerprint Sacrifice, taking one bit from each fingerprint and appending it to the canonical slot address. The lost bit shortens the fingerprint length, but the slot width remains the same by extending the unary code by one bit. When a new entry is inserted after expansion, it is assigned the original slot width and its unary code is set to one bit. Figure 3-(a) illustrates a simplified but valid filter configuration, initialized with a single slot containing one entry with hash 11 and has undergone two expansions. Key x is inserted with a 2-bit fingerprint and loses a bit on expansion. Key y, which is inserted after the expansion, is assigned the original length of two bits per fingerprint. Unary codes are marked in green in the figures.

False Positive Rate. Since each expansion enlarges the filter by a factor of two, the fingerprints' length follows a geometric distribution. That is, fingerprints inserted p expansions ago populate

[2]Though InfiniFilter is built on top of an older variant of the quotient filter [2], its core ideas are immediately applicable to RSQF [39]. We describe it on top of RSQF in this section for consistency and ease of exposition.

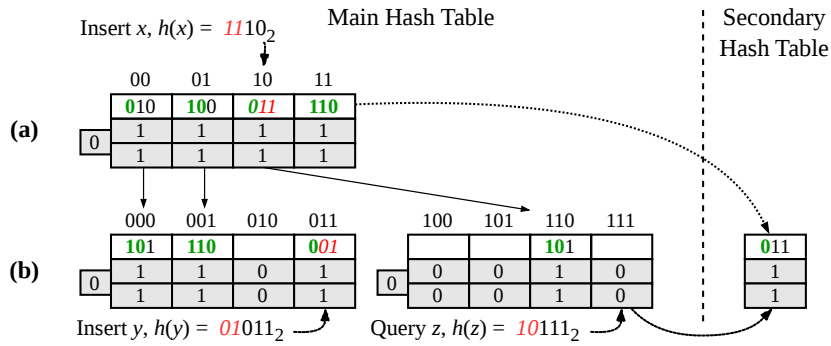


Fig. 3. InfiniFilter adds a unary padding to the fingerprint to support variable-length fingerprints. Void entries are moved to the secondary hash table.

approximately $\approx 2^{-p-1}$ of the occupied slots. As these fingerprints are left with $f - p$ bits, the collision probability of each fingerprint is 2^{-f+p}. The overall FPR of InfiniFilter at the p-th expansion is a weighted sum: $\epsilon = \sum_{i=0}^{p} 2^{-i-1} \cdot 2^{-f+i} \leq (p+2) \cdot 2^{-f-1}$. Since $p = \log_2 \frac{N}{n}$, the FPR increases logarithmically with respect to the filter size, limiting the scalability of InfiniFilter in terms of FPR.

Widening Regime. To achieve a constant FPR, InfiniFilter employs a so-called *Widening Regime*, whereby slots widen across expansions, supporting longer fingerprints for newer entries. Specifically, the length of fingerprints assigned to entries inserted after the p-th expansion is $f + 2 \cdot \log_2(p+1)$ bits. This causes the FPR to converge to a constant smaller than 2^{-f} [16]. In contrast, the variant of InfiniFilter whereby slots remain fixed-length across expansions is referred to as the Fixed-Width Regime.

Void Entries and Chaining. After the first f expansions, the oldest entries lose all remaining fingerprint bits, with the unary code filling up the entire slot width. Any such entry with no valid fingerprint bits is called a *void entry*. The entry at Slot 11 in Figure 3-(a) is an example of a void entry. Any query targeting a run with at least one void entry returns a positive.

Void entries beg the question of how to make the filter continue to expand, as they have no more fingerprint bits to sacrifice. InfiniFilter solves this problem by removing void entries from the main hash table at expansion and inserting their original hash to a smaller *secondary hash table*. For example, as the filter expands in Figure 3-(b), the void entry at Slot 11 in Figure 3-(a) is removed and its original hash (11) is inserted into the secondary hash table.

The secondary hash table has the same structure as the main filter, meaning that it also expands when it reaches capacity. If the oldest entry in the secondary hash table becomes void, it is sealed and is added to the *chain* of secondary hash tables. A new secondary hash table is allocated to take its place. This slows down queries and deletes of InfiniFilter as they may traverse a longer path through the chain. For example, a query for Key z in Figure 3-(b) does not find a matching fingerprint in the main hash table and has to also search the secondary hash table, which is where it finds a matching fingerprint and returns a positive. Generally, the worst-case query and deletion cost with InfiniFilter is $O(\log N/f)$ in the Fixed-Width Regime and $O(\log N/(f+\log\log N))$ in the Widening Regime. In the latter, fingerprints are longer upfront and thus slower to become void, leading to a shorter chain.

2.3 Aleph Filter

Aleph Filter improves the scalability of InfiniFilter by providing worst case $O(1)$ time queries and deletes. Instead of storing and accessing void entries through a chain of additional hash tables, it duplicates each void entry to both candidate slots that it would have mapped to if it had more valid

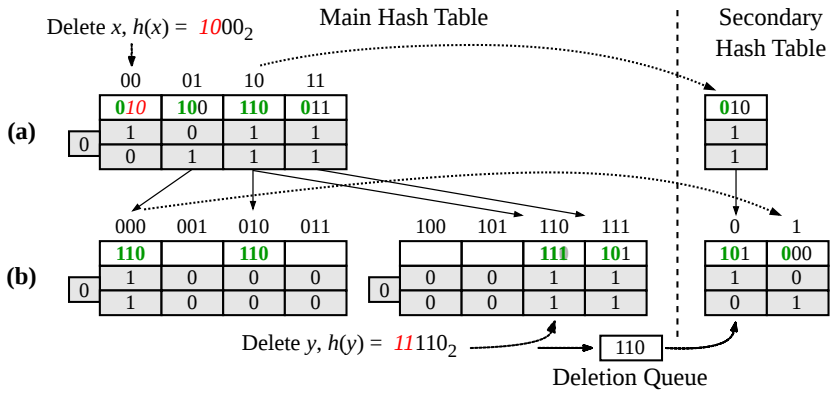


Fig. 4. Aleph Filter duplicates void entries for faster queries. Deleting a void entry replaces it with a tombstone and inserts its canonical slot to the deletion queue. Actual deletion of the void entry duplicates is deferred until the next expansion.

fingerprint bits. For example, the void entry at Slot `10` in Figure 4 is duplicated to Slots `010` and `110` after expansion, because the original hash would have had either `0` or `1` as its least significant fingerprint bit. As a void entry will have been duplicated into all possible slots that the key could have been mapped to, a query on the original key will always encounter a copy of the void entry. This allows only querying the main hash table in $O(1)$ time while guaranteeing no false negatives. The proportion of void entries stays small despite duplicating them since only a small fraction of all entries are old enough to become void [17].

Deletion. In both Aleph Filter and InfiniFilter, a deletion removes the longest matching fingerprint from the target run. For example, consider Run `00` in Figure 4-(a). A deletion for Key x should be deleting the entry on Slot `00` with the longest matching fingerprint of `10`. If it instead deletes the entry on Slot `01`, and the original hash for that entry happened to be `0000`, a future query on that key will return a false negative.

The longest matching fingerprint for deletion could be a void entry, which may have been duplicated multiple times. Because the number of duplicated void entries increases exponentially with the number of expansions since the entry became void, eagerly identifying and deleting all duplicates is expensive in the worst case. Aleph Filter bounds the deletion time to a constant by replacing a deleted void entry with a *tombstone* and deferring the actual deletion of its duplicates until the next expansion. A tombstone is represented as a special bit string of all 1s ($1 \cdots 11$), and if a query that did not match any other fingerprints in its canonical slot encounters a tombstone, it returns a negative.

Removing Duplicate Void Entries. During expansion, tombstones are removed along with their duplicates. To correctly infer the number of duplicates, Aleph Filter maintains two auxiliary data structures — the *secondary hash table* and the *deletion queue*. When an entry becomes void, the secondary hash table stores its original hash, as shown in Figure 4-(a). As the entry at Slot `10` becomes void, its original hash `10` is inserted into the secondary hash table. The secondary hash table of Aleph Filter, similarly to that of InfiniFilter, expands at capacity and gets added to the chain of secondary hash tables when any of its entries become void. The main difference is that it is not referred to during queries or deletions.

If a deletion encounters a void entry, its canonical slot address is inserted into the deletion queue. Before the next expansion, queued addresses are queried against the secondary hash tables for the longest matching original hash. The difference in length between the queued address and the matched hash tells how many expansions occurred since the entry became void. For example, in

Table 2. Terms and definitions to describe Zeno Filter.

Term	Definition
p	number of elapsed *periods* from the start
r	the *growth coefficient* for each expansion
S	space amplification of Zeno Filter
W	write amplification of Zeno Filter
ϵ_r	false positive rate of Zeno Filter under r

Figure 4-(b), deleting Key y targets Canonical Slot `110`. As there is no matching fingerprint, a void entry is replaced with a tombstone (`111`) and its slot address is inserted into the deletion queue. Since this slot address is three bits and the matching hash in the secondary hash table is `10` with two bits, Aleph Filter infers that there are $2^{3-2} = 2$ void duplicates of Key y. Thus, before the next expansion, it deletes a void entry from Slots `010` and `110`, and removes the original hash from the secondary hash table.

3 Problem Analysis

The purpose of a filter is to be space-efficient; if it becomes too large, it loses practicality, as we may as well store the full data keys. However, the state-of-the-art expandable filters like InfiniFilter [16] and Aleph Filter [17] exhibit three forms of inefficient space use. We now analyze these overheads in terms of space amplification, defined as the ratio between the total number of slots and non-empty slots in the filter.

Persistent Space Amplification. Because existing expandable filters expand by a factor of two, the filter right after expansion is at least half empty. The empty slots right after expansion contribute a factor of $\frac{2}{\alpha}$ to the space amplification, where α is the expansion threshold. For example, if the filter expands at $\alpha = 0.8$, the expanded filter will be only 40% full, leaving the space amplification to be 2.5. Since this source of space amplification persists after expansion, we refer to it as *persistent space amplification*.

To reduce persistent space amplification, we would like to expand by less than a factor of two. The challenge is that the Fingerprint Sacrifice method we saw in Section 2 does not generalize to expansion factors that are not powers of 2. The reason is that a bit is an atomic unit of data, and there is no natural way to sacrifice less than one fingerprint bit to map an entry to a larger filter.

Transient Space Amplification. Existing expandable filters expand out-of-place by migrating fingerprints from the original filter to the newly allocated filter. The co-existence of both the original and new filter during expansion adds a factor of $\frac{1}{\alpha}$ to the space amplification until the expansion is complete and the original filter is deallocated. Despite being temporary, this overhead is necessary as further expansion is not possible otherwise. We refer to this overhead as *transient space amplification*.

Ideally, we would like the filter to expand in-place to resolve this issue. The challenge with in-place expansion is memory fragmentation. Memory allocators typically cannot extend a data structure in-place since the physically adjacent space to the filter may have already been allocated for a different purpose. Hence, expanding in-place requires a layer of indirection. The simplest approach is having a small directory that maps to fixed-size blocks, each consisting of B slots. During expansion, new blocks are allocated to accommodate the new filter size, and pointers to them are added to the directory, which doubles in size when it reaches capacity. For a filter with N slots, the directory contains $\lceil N/B \rceil$ pointers.

Nevertheless, the directory also contributes to space amplification. If left unchecked, it may eventually outgrow the CPU caches and lead to performance degradation. Generally, the directory

consists of at most $O(N/B)$ empty slots as it is half-empty when it expands. At the same time, the most recently allocated block contains at most $O(B)$ empty slots. The number of wasted slots is thus $O(\texttt{max}(N/B, B))$. As this expression indicates, reducing the block size B lowers the space wasted when allocating a new block, but it also increases the wasted space in the directory[3]. These two sources of space wastage are therefore in contention with one another. Expanding in-place using cache-resident scalable layer of indirection is therefore a non-trivial challenge.

Auxiliary Space Amplification. Both InfiniFilter and Aleph Filter maintain a chain of secondary hash tables and deletion queues to support deletes, adding a factor of approximately $\frac{2^{-f}}{\alpha}$ to the space amplification. Although their contribution to the overall space amplification is smaller than the other two sources, they still increase the filter size by as much as 1 bit per entry. This effect is especially magnified when the filter contracts, making the auxiliary data structures larger in proportion relative to the main hash table.

Total Space Amplification. The three sources of space amplification sum up to be $\frac{3+2^{-f}}{\alpha} > 3$. Across many applications, filters are typically assigned a budget of 10 bits per entry to deliver an FPR of 1% [3, 27, 36, 42]. However, if space is amplified by at least a factor of 3 as indicated by the equation above, users may as well store full integer identifiers to represent all keys rather than using filters. This raises the primary research question of this paper: to what extent can the space amplification of expandable filters be mitigated, and at what cost?

4 Zeno Filter

We introduce Zeno Filter, an expandable filter that tackles all three sources of space amplification. Section 4.1 addresses persistent space amplification by allowing the filter to expand by less than a factor of two using a novel technique called Stretching. Section 4.2 eliminates transient space amplification by expanding the filter in-place using a sub-linear layer of indirection. Section 4.3 obviates the need for auxiliary metadata structures using a new encoding scheme for void entries. Together, these techniques bound the space amplification to $\frac{2^{1/r}}{\alpha}$ in exchange for a write cost of $O(r)$, where r is a tuning parameter.

Unlike InfiniFilter or Aleph Filter, Zeno Filter takes the most significant q bits of an entry's hash as the quotient and the least significant f bits as the fingerprint. This inversion is crucial for the void entry encoding scheme, as we will explain in detail in Section 4.3. All descriptions and figures in this section that describe Zeno Filter reflect this design. In Figure 5, for example, Key x is mapped to Canonical Slot 11 using the most significant $q = 2$ bits.

4.1 Stretching

We first address the problem of persistent space amplification by combining the Fingerprint Sacrifice method from Section 2 with a new expansion mechanism called *Stretching*. Stretching maps an entry to a fractionally larger filter unambiguously, such that we can still always find each entry in constant time. The trade-off is that each entry is copied more times on average.

With Stretching, we expand the filter by some factor $x < 2$ and 'stretch' the distance (i.e., number of empty slots) between runs by a factor of x. Stretching does not split runs like Fingerprint Sacrifice, but it allows the filter to maintain good insertion and deletion performance by increasing the average distance between adjacent runs, thus preventing clustering. However, we cannot use Stretching alone, as it will make the length of runs grow indefinitely, harming the filter's FPR and query cost. We avoid this problem by combining Stretching with Fingerprint Sacrifice.

[3]It is possible to alleviate this trade-off by recursively adding more indirection layers [54]. However, it logarithmically increases access costs as the filter size grows.

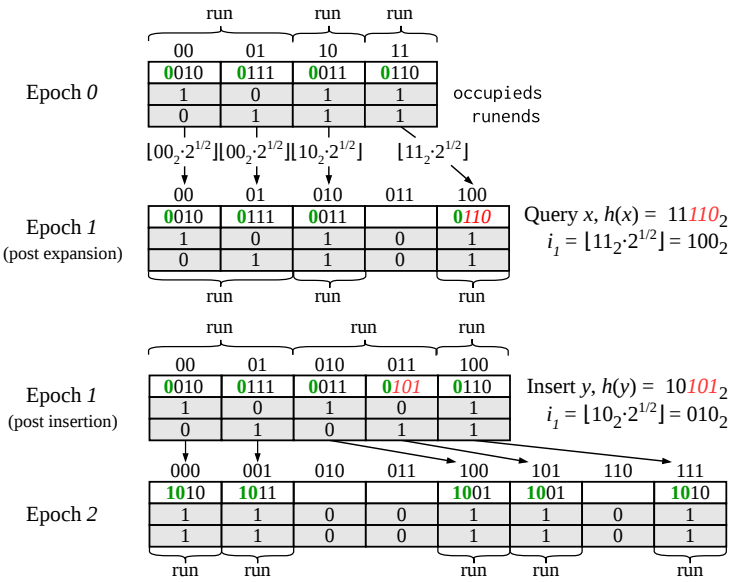


Fig. 5. Breakdown of a period of Zeno Filter. Zeno Filter uses Stretching to expand by a factor of $2^{\frac{1}{r}}$ at every epoch. At the r-th epoch, Zeno Filter uses Fingerprint Sacrifice to map entries to their respective slots in the expanded filter.

Specifically, we set Zeno Filter to always expand by a factor of $2^{1/r}$ for some positive integer r, forcing the filter size to return to a power of 2 after r expansions. Thus, a key with slot address i before expansion is mapped to slot $\lfloor i \cdot 2^{1/r} \rfloor$ after expansion. We call r the *growth coefficient*, and it allows trading between write cost and persistent space amplification. During the $r-1$ intermediate expansions, Zeno Filter applies Stretching to space out the runs, and at the r-th expansion, it applies Fingerprint Sacrifice to split the runs. We refer to the interval between a filter expansion by a factor of two as a *period* and each intermediate expansion within a period as an *epoch*.

Example. Figure 5 illustrates a toy example for a filter with initially four slots. The growth coefficient r is 2, and the expansion threshold α is 0.8. At Epoch 0, Key x maps to Canonical Slot $i_0 = 11$ in Run 11, leading to an expansion as the fraction of non-empty slots exceeds the threshold. As indicated by the arrows, this entry is moved to Slot $i_1 = 100$ after expansion due to Stretching. Querying for Key x at Epoch 1 follows the same Stretching logic. First, the key is hashed to derive its canonical slot, $i_0 = 11$, in the original address space. Next, the target slot in the stretched address space is calculated using the growth factor: $i_1 = \lfloor i_0 \cdot 2^{1/r} \rfloor$. Finally, the filter scans the run at Slot $i_1 = 100$ for the fingerprint. Key y is then inserted to the filter at Epoch 1, leading to another expansion. The filter now proceeds to the next period and its size returns to a power of 2, which is when it applies Fingerprint Sacrifice to each entry. For instance, Key x is mapped to Canonical Slot 111, and Key y, with its canonical slot originally being 10, is mapped to Canonical Slot 100 in the larger filter.

Space and Write Amplification. Since the number of slots in the new filter increases by a factor of $2^{1/r}$ in every expansion and at most α of these slots are non-empty, the persistent space amplification is:

$$S = \frac{2^{1/r}}{\alpha} \tag{1}$$

We also analyze Zeno Filter's write amplification, which is the average number of times an inserted entry gets rewritten across expansions after it is inserted. It is widely known that the write amplification for a dynamic array with growth factor g is $\frac{g}{g-1}$ [45]. This expression holds for an expandable hash table such as Zeno Filter as well. By applying a first-order Taylor expansion to Zeno Filter's growth factor $2^{1/r}$, its write amplification in terms of r is:

$$W = \frac{2^{1/r}}{2^{1/r} - 1} \lesssim \frac{r}{\ln 2} + 1 \tag{2}$$

Thus, Stretching reduces space amplification from $\frac{2}{\alpha}$ to $\frac{2^{1/r}}{\alpha}$ for a write cost of $O(r)$, thereby opening up a new trade-off frontier.

Fingerprint Length Distribution. To analyze the FPR of Zeno filter, we examine the distribution of fingerprint lengths in the filter. Since entries lose a fingerprint bit on every period, an entry that has gone through i periods would have $f - i$ valid fingerprint bits. After p periods, the approximate fraction of such entries is

$$F_r(i) \lesssim 2^{-i-1/r}, 0 \leq i \leq p. \tag{3}$$

When a period ends and the filter size reaches the next power of 2, a fraction of $2^{\frac{r-1}{r}} - 1$ entries in the filter that would have had a fingerprint length of f bits without fractional expansion lose a bit to have $f - 1$ fingerprint bits.

False Positive Rate. Because runs do not split until the filter reaches the next period, nearly twice as many fingerprints are checked per query on average during each epoch. The FPR of Zeno Filter in the fixed-width regime can thus be expressed as follows:

$$\epsilon_r = \sum_{i=0}^{p} \alpha \cdot F_r(i) \cdot 2^{-f+i} \lesssim \alpha \cdot (p+2) \cdot 2^{-f-1/r} \tag{4}$$

In the Widening Regime, the FPR of Zeno Filter converges to a constant as follows[4]:

$$\begin{aligned} \epsilon_r &= \sum_{i=0}^{p} \alpha \cdot F_r(i) \cdot 2^{-\ell(p-i)+i} \\ &= \alpha \cdot 2^{-f-1/r} \cdot \sum_{i=0}^{p} \frac{1}{(p-i+1)^2} \lesssim \alpha \cdot 2^{-f-1/r} \cdot \frac{\pi^2}{6} \end{aligned} \tag{5}$$

When $r = 1$, the FPR of Zeno Filter is identical to that of Aleph Filter with the same fingerprint size, both in the Fixed-Width and Widening Regime. As r increases, Zeno Filter's FPR converges to the FPR of Aleph Filter using one fewer bit for the fingerprint. However, initializing the filter with one extra bit per entry effectively compensates for this loss, since the space saved by Zeno Filter offsets the cost of adding this bit.

4.2 In-Place Expansion

We now address the problem of transient space amplification. Zeno Filter expands in-place using a layer of indirection, i.e., a directory, to allow allocating one new block of slots at a time, as discussed in Section 3. To keep the directory small and cache-resident, we draw inspiration from a data structure called *singly resizable array* [5]. The core idea is to gradually increase the block size as the data grows. This keeps both sources of space wastage moderate and approximately equal. At the same time, it slows down the growth of the directory size, making it more likely to fit in the caches. This structure was originally proposed to implement dynamically expandable arrays. We

[4]This approximation uses the closed-form expression for the infinite sum of squared reciprocal natural numbers (i.e., $\sum_{i=0}^{\infty} \frac{1}{(i+1)^2} = \frac{\pi^2}{6}$).

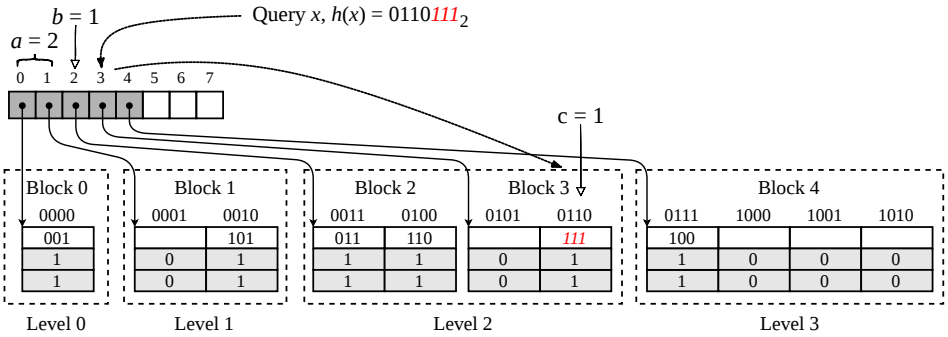


Fig. 6. Zeno Filter structure and query path. Blocks are clustered into levels, and the directory has pointers to each block.

are the first to leverage it in the context of a filter and describe a full integration. Furthermore, we show how to speed up access to this structure using bit manipulation techniques.

Indirection Structure. A singly resizable array consists of multiple *levels* of exponentially increasing capacities. When it is first allocated, there is only one level consisting of one block with Q slots. In our design, each slot of a block is an RSQF slot as described in Section 2.1. We set Q to 64 by default, making the first allocated block correspond to one RSQF block. During expansion, we allocate as many new blocks and levels as necessary to accommodate the new filter size. The pointers for each new block are inserted into the directory. Crucially, both the block size and the number of blocks across larger levels grow at a rate of $O(\sqrt{N})$. This is done by alternately doubling the number of blocks and doubling the size of each block across larger levels. Specifically, the l-th level contains $2^{\lfloor l/2 \rfloor}$ blocks of $Q \cdot 2^{\lceil l/2 \rceil}$ slots each, making it hold $Q \cdot 2^l$ slots in total. Hence, the total number of wasted slots is at most $O(\sqrt{N})$ in both the last allocated block and in the directory.

Since the directory only maps $O(\sqrt{N})$ blocks, it grows at a slow rate of $O(\sqrt{N})$ with respect to data size. This makes the directory likely to stay in the CPU caches even as N grows very large. At the same time, the number of wasted slots in the directory and in the last allocated block is also $O(\sqrt{N})$. The resulting space amplification is a fraction of $O(1/\sqrt{N})$. It becomes vanishingly small as data grows.

Figure 6 illustrates this structure with $N = 8$ slots and $Q = 1$. The array shown at the top is the directory, with pointers to each block. Notice how, due to the alternation in the growth of the number of blocks and block sizes across larger levels, Level 0 has one block with one slot, Level 1 has one block with two slots, and Level 2 has two blocks with two slots each. Level 3 in this example contains an unfilled block with four slots, though this level has capacity for two such blocks. The address space for slots spans the blocks in the order they are added. For instance, Slot 6 is at the end of Block 3 in Level 2, while Slot 7 is within Block 4 in Level 3. While some space is wasted at the end of the directory and at the end of the last occupied block, it is asymptotically lower than with out-of-place expansion or with methods utilizing fixed-length blocks.

Fast Access. The increasing size of blocks complicates filter access, as the correct block for an entry must first be inferred. To access an entry at Slot i, we first identify the level l that the entry belongs to. As the capacity of each level grows exponentially, we know that l is correlated with the most significant bit of the binary representation of the target slot. Accordingly, let j be the binary representation of $i + 1$, with all leading zeros removed. We find l by subtracting 1 from the length of j, i.e., $l = |j| - 1$. In Figure 6, for example, for the query on Key x at Slot 6 ($j = 111_2$), we identify its level as $l = |j| - 1 = 2$.

Table 3. The values of j, l, a, b, and c for the first 11 slots. Values of j and c are in binary encoding.

slot(i)	binary(i_2)	j	l	a	b	c	block index($a+b$)
0	0000	1	0	0	0	0	0
1	0001	10	1	1	0	0	1
2	0010	11				1	
3	0011	100	2	2	0	0	2
4	0100	101				1	
5	0101	110			1	0	3
6	0110	111				1	
7	0111	1000	3	4	0	00	4
8	1000	1001				01	
9	1001	1010				10	
10	1010	1011				11	

Listing 1. Pseudocode of indexing Zeno Filter

```
element query(Directory* d, unsigned i) {
    unsigned j = i + 1;
    // Count the size of j in bits
    unsigned l = sizeof(j) - cntlz(j) - 1;
    unsigned floor_l = l >> 1;        // floor(l/2)
    unsigned ceil_l = (l + 1) >> 1;   // ceil(l/2)
    unsigned a = (1 << floor_l) * (2 + (l & 1)) - 2;
    // Mask the most significant `floor_l` bits
    unsigned b = (j >> ceil_l) & (1 << floor_l) - 1;
    // Mask the least significant `ceil_l` bits
    unsigned c = j & (1 << ceil_l) - 1;
    Block* block = d[a + b];
    return block[c];
}
```

We then compute the total number of blocks in all preceding levels before Level l, which we refer to as a. When calculating a, we use the fact that the number of blocks per level doubles once every two levels, and for odd levels, the same number of blocks must be added as in the previous level. Thus, we calculate this value using the formula[5] $a = 2^{\lfloor l/2 \rfloor} \cdot (2 + l \bmod 2) - 2$. Since Key x is in Level 2, the sum of the number of blocks in Levels 0 and 1 is $a = 2$.

Next, we compute the index of the target block within Level l. We denote this value as b. We take this value from the most significant $\lfloor l/2 \rfloor$ bits of j right after the leading 1-bit, since Level l consists of $2^{\lfloor l/2 \rfloor}$ blocks. For Key x, we compute this as $b = 1_2$.

The values a and b together determine which pointer to follow in the directory. That is, the sum $a + b$ corresponds to the number of blocks that precede the block being accessed. In Figure 6, the access path for Key x follows the pointer in Slot 3 of the directory, since the respective values for a and b are 2 and 1.

Lastly, we infer the number of slots to skip within the target block. We refer to this value as c, and it is obtained from the least significant $\lceil l/2 \rceil$ bits of j, since each block in Level l consists of $2^{\lceil l/2 \rceil}$ slots. The value of c for Key x is computed as $c = 1_2$. In sum, we locate Key x at Slot 1 of Block 3, contained within Level 2. Table 3 lists the calculations of the above values for each slot in Figure 6.

[5]The original paper [5] has an error in calculating a. We provide a corrected equation.

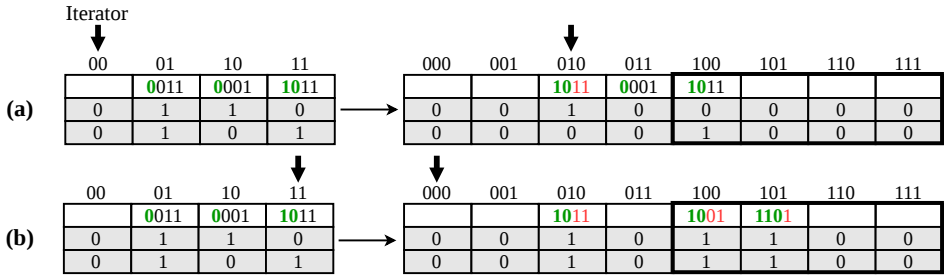


Fig. 7. Filter expansion under (a) forwards and (b) backwards iteration. Newly allocated filter slots are indicated with a thicker outline. Migrated entries are tinted in red, but the iterator cannot make this distinction.

Listing 2. Pseudocode of Zeno Filter-VM allocation

```
int prot = PROT_READ | PROT_WRITE;
int flags = MAP_ANONYMOUS | MAP_PRIVATE | MAP_NORESERVE;
size_t filter_size = num_slots * (fingerprint_length + 2.125) / 8;
char* filter = mmap(NULL, filter_size, prot, flags, -1, 0);
```

Listing 3. Pseudocode of Zeno Filter-VM expansion

```
size_t new_size = filter_size * pow(2, 1 / r);
filter = mremap(filter, filter_size, new_size, MREMAP_MAYMOVE);
filter_size = new_size;
```

Note that all steps in computing the values k, a, b, and c only involve constant time operations. We further optimize each step of the calculation using only bitwise operators and CPU instructions. For example, $\lfloor j \rfloor$ is obtained by the `cntlz` instruction to count leading zeros, avoiding floating point calculations in $\lfloor \log_2 j \rfloor$. The values for $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$, along with their most and least significant bits, are derived through bitwise shifting and masking. Similarly, the modulo operation is simplified as a bitwise masking on the least significant bit. Listing 1 presents the pseudocode.

The directory is small compared to the filter. For example, for a filter with $f = 8$ and N being 2^{20}, 2^{30}, and 2^{40}, the directory has 2^{12}, 2^{17}, and 2^{21} slots each. For each value of N, the directory size is only 32KB, 1MB, and 16MB, making the directory small enough to fit in the CPU caches for any realistic data sizes.

Migrating Entries. After increasing the filter size during expansion, entries are migrated to their respective slots in the enlarged filter as described in Section 4.1, either through Stretching or Fingerprint Sacrifice. However, with in-place expansion, iterating from the start of the filter is infeasible because migrated entries may be re-encountered by the iterator and migrated again. For example, in Figure 7-(a), the entry at Slot `01` is migrated to Slot `010`, but since the iterator cannot tell apart migrated and nonmigrated entries, the entry is migrated again. This results in a false negative, as the entry is indeed stored in the filter but at the incorrect location.

We prevent this problem by traversing the filter backwards. By so doing, we are guaranteed to only encounter entries that were not already migrated during the current expansion. The reason is that when an entry is migrated, it is always mapped to a larger slot address by appending a `0` or `1` as the least significant bit of its slot address. Figure 7-(b) illustrates this approach. Once entries are migrated to larger slot addresses by their most significant fingerprint bit, the iterator will never re-encounter them again, as it will have advanced to smaller slot addresses.

Widening Regime. In the Widening Regime, we increase the slot width across expansion to stabilize the FPR. To support this, each block is reallocated during expansion with wider slots

while keeping the same number of slots. Since we reallocate one block at a time, transient space amplification stays low.

Virtual Memory Alternative. While the software-based directory described above is cache efficient, the bitwise address translation (Listing 1) inevitably incurs a slight latency overhead. To eliminate this software overhead entirely, we propose a variant of Zeno Filter that leverages virtual memory as opposed to a singly resizable array. In this case, virtual memory provides the abstraction of a contiguous address space. This variant is faster as the indirection layer is implemented in hardware through page tables. Further, as memory pages are fixed-length, this scheme forgoes the $O(\sqrt{N})$ upper bound on wasted slots and instead uses at most B wasted slots from a block. However, it sacrifices portability as it relies on platform-specific system calls. We refer to this variant as **Zeno Filter-VM**.

We allocate virtual memory using the `mmap` POSIX system call as shown in the code in Listing 2. When Zeno Filter-VM reaches capacity and expansion is triggered, it uses `mremap` to remap the existing filter to a larger address space. The code in Listing 3 recalculates the filter size and expands the filter size according to the growth coefficient r. Although this remapping may change the virtual address, it does not move objects in physical memory.

These variants of Zeno Filter cater to applications with different performance and portability requirements. We evaluate both in Section 5.

4.3 Void Entry Encoding

Recall from Section 4.1 that at the end of a period (i.e., after r epochs as the filter's size returns to a power of 2), Zeno Filter performs Fingerprint Sacrifice by repurposing one bit from each fingerprint to the slot address, splitting runs to evenly distribute their entries across the larger filter. In this process, a new void entry is created whenever an entry runs out of fingerprint bits. Similarly to Aleph Filter [17], Zeno Filter duplicates a void entry into two distinct slots in the subsequent period as described in Section 2.3 to maintain a worst-case $O(1)$ query cost. However, these void duplicates create a challenge for deletes.

In general, a deletion removes the entry with the longest matching fingerprint in the target run. However, if the run consists only of void entries, it is nontrivial to determine how many other void duplicates must be removed from other locations in the filter to eliminate the entry entirely. Aleph Filter addressed this problem through auxiliary structures as discussed in Sections 2.3 and 3. In contrast, Zeno Filter tackles this challenge without using any additional metadata. Instead, it ensures void duplicates of the same entry are adjacent and self-delimiting. We show that this provides enough information to infer how many duplicates to remove.

Adjacency of Void Duplicates. We refer to the set of void duplicates originating from the same key as a *void sequence*. During Fingerprint Sacrifice, Zeno Filter repurposes the most significant bit of each fingerprint to become the least significant bit of the slot address. Thus, entries are migrated after a period to one of two adjacent slots; that is, an entry at Slot i is migrated to either Slot $2 \cdot i$ or $2 \cdot i + 1$. Consequently, a void entry is duplicated into two adjacent slots, placing entries of the same void sequence in adjacent runs. This is in contrast to the state-of-the-art expandable filters [16, 17] where void duplicates from the same entry are scattered across the filter, making it difficult to relate them back to the original key.

Self-Delimiting Void Sequence Encoding. In addition to making a void sequence adjacent, we must also make it self-delimiting to be able to tell apart void sequences from different entries that happen to be right next to each other. To this end, we draw inspiration from the classic symmetric unary code [50]. We reserve three logical markers `A`, `B`, and `C` for this purpose. A void sequence with

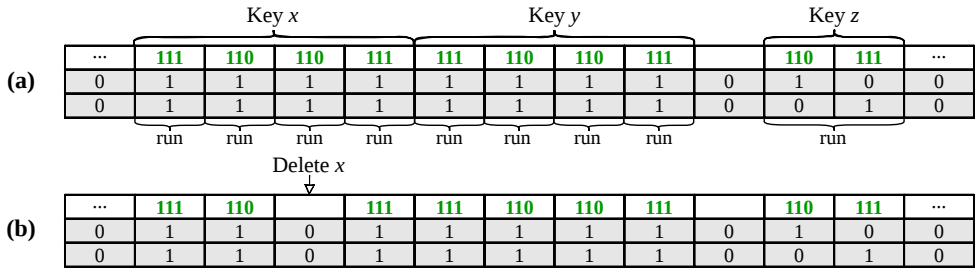


Fig. 8. Self-delimiting void sequences allow Zeno Filter to correctly identify and remove the void sequence when its entry is lazily deleted.

length one is encoded with a single A, while a longer void sequence is enclosed by delimiting Bs in both ends with Cs in between, thereby allowing its length to be inferred.

Physical Encoding. Although we want to encode the three markers unambiguously, the variable-length fingerprint scheme physically has only two unused unary symbols. We solve this by assigning markers B and C one physical symbol each, and using a combination of both for marker A. Specifically, we use the symbol with all 1s ($1\cdots11$) as B and the max-length unary symbol ($1\cdots10$) as C, while A is encoded as a concatenation of the two symbols. Figure 8-(a) illustrates an example of three void sequences. Keys x and y have length four and are encoded as BCCB, whereas Key z has length one and is encoded as A, physically represented as two unary symbols connected. As shown, the sequences for Keys x and y can be told apart due to their self-delimiting encoding. Also, when void entries with different markers fall into the same canonical slot, we distinguish them by sorting the markers in a predefined order. The ordering places As first, Bs next, and Cs last, thereby grouping identical markers together.

Lazy Deletion. To support fast deletions, Zeno Filter exploits the adjacent and self-delimiting encoding of a void sequence. In particular, a deletion only erases the target void entry, allowing it to be done in constant time. For example, lazily deleting Key x in Figure 8-(b) erases a single C marker, but keeps the remaining void duplicates untouched. Since the deleted void entry no longer exists, queries for the deleted key return a negative in most cases (unless there is a hash collision with some other key, i.e., a false positive). The mechanism for reclaiming the remaining void duplicates is described later in this section.

Deleting the Shortest Void Sequence. When void sequences with different lengths overlap, we must also decide which sequence to remove. In such a case, deleting the correct sequence is crucial since otherwise it could lead to false negatives. To be specific, overlaps happen because the original hashes of the sequences share a prefix, with the shorter sequence having a longer hash. Among the overlapping sequences in Figure 9-(a), Key x maps to the shortest void sequence and has the longest hash. Deletes must target the shortest of overlapping sequences; removing longer ones would discard the broader hash coverage of shorter hashes, leading to potential false negatives. For example, in Figure 9-(a), if deleting Key y at Canonical Slot 010 removed the longest sequence, a query for Key z would return a false negative if its hash were 000 or 001.

By separating and sorting overlapping void entries, we can always lazily delete the entry belonging to the shortest void sequence without referring to any surrounding slots. The reason is that the A marker always belongs to the shortest sequence, and a B marker always belongs to a shorter sequence than a C marker. An A marker is always followed by a trailing escape token and is identifiable as a sequence of length one. For longer void sequences, the shorter sequence is always fully contained within the longer one. Since shorter sequences start at length one and grow at the same rate as the longer sequence, their ranges never intersect. As such, any B markers that appear

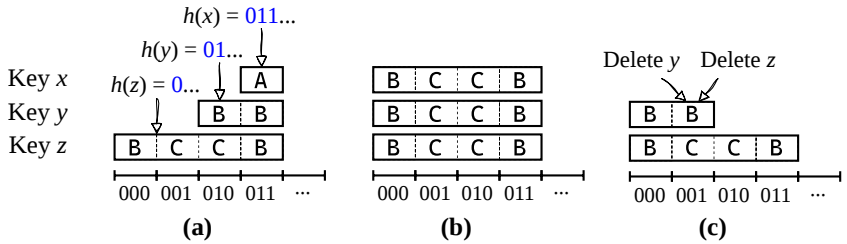


Fig. 9. (a) When void sequences of different lengths overlap, Zeno Filter always deletes the shortest sequence. (b) An ambiguous configuration caused by lazy deletes removing an entire sequence. (c) When deletions target all B and C markers in the same run, all markers are erased completely.

together in the same run with C markers must belong to a shorter void sequence. In sum, when different types of void entries overlap, we always delete the shortest void sequence by deleting A markers first, B markers second, and C markers last.

B Markers as Placeholders. Lazy deletes on different canonical slots could erase an entire void sequence. In such a case, we lose all information about the deletions and cannot remove the relevant sequences. For example, the setting in Figure 9-(b) can be interpreted in two ways: there were either three void sequences all along, or four void sequences were deleted in each of the four slots. To solve this, we keep both B markers as placeholders and do not remove them immediately, and instead append a new B marker at the end of the run. If there are subsequent C markers in the run, the new B marker is appended after the C markers; otherwise, we insert a C marker as a delimiter. The only case we remove B markers is when the last C marker is deleted and there remains no delimiters, as shown in Figure 9-(c). This implies that all overlapping void sequences had been deleted, which is inferrable from the number of surrounding void entries.

Cleanup. Since lazy deletion removes only the target void entry, residual void duplicates persist in the filter after a deletion. Zeno Filter therefore performs a cleanup step during the scan of the next expansion, where these leftover entries are removed. The cleanup mechanism leverages the fact that a void sequence of length greater than two deterministically spans from an even index to an odd index. To support cleanup, Zeno Filter maintains a void sequence stack, a void sequence counter, and a deletion counter.

When the backwards iterator visits an odd index containing s B markers, it pushes the index onto the stack and increments the sequence counter by s. If fewer void entries are observed at a slot than indicated by the sequence counter, the difference corresponds to lazily deleted entries and is added to the deletion counter. When the iterator reaches an even index with s B markers, it identifies the termination of s void sequences. Assuming the deletion counter is d, the iterator first pops and discards the top d indices from the stack, removing the shortest overlapping void sequences. It then pops $s - d$ indices, doubles the corresponding sequence lengths and reinserts them into the enlarged filter, and decreases the sequence counter by $s - d$.

4.4 Concurrency

The concurrency scheme of Zeno Filter allows thread-safe access in both steady-state operations and during expansion. The scheme is orthogonal to the designs of Zeno Filter and is compatible with prior works, including RSQF, InfiniFilter, and Aleph Filter.

Mechanics. Zeno Filter adapts the concurrency scheme of the underlying RSQF [39] but extends it to support finer-grained locking. The filter is divided into regions of R slots, where R is set to 4096 by default to keep regions page-aligned. Unlike RSQF, which locks at most two regions and therefore employs a larger region size of $R = 65536$ to ensure correctness, Zeno Filter allows

threads to incrementally acquire locks on an unbounded sequence of regions. It is this support for incremental locking that enables Zeno Filter to safely employ a finer locking granularity than RSQF. The finer locking granularity mitigates access skew — while concurrent accesses to the exact same key must serialize, Zeno Filter's smaller regions lower the probability of distinct keys contending for the same lock compared to RSQF, thereby reducing conflicts on 'hot' regions. We call the collective regions locked incrementally a *group* of regions. A thread performing operations on the filter during an epoch needs to lock only a single group.

Concurrency During Expansion. As described in Section 4.2, Zeno Filter expands in-place to reduce transient space amplification. As such, expansion takes the steps of the iterator deleting an entry, recalculating its index, and reinserting the entry into an appropriate slot in the larger filter. Zeno Filter allows queries and updates to proceed concurrently by incorporating the expansion iterator with the locking mechanism. Specifically, the iterator locks two groups at a time: the source group where the iterator performs deletions, and the destination group where it inserts the deleted entries. In contrast, RSQF locks the entire filter during expansion.

A thread that accesses regions that were not yet scanned by the iterator operates normally, calculating the slot index as before the expansion. If a thread's operation involves a region that the iterator is scanning or has already scanned, the thread gives up its lock on that region and computes the index as it would after the expansion, as described in Section 4.1.

5 Evaluation

We now turn to empirically evaluate Zeno Filter. Experiments 1 to 4 evaluate its core features from Section 4. Experiment 5 compares Zeno Filter to other expandable filters from the literature.

Platform. We run all experiments on a Supermicro AS -4125GS-TNRT server running Ubuntu 24.04.3. The system is equipped with two AMD EPYC 9654 processors (192 cores, 384 threads, 2.4GHz), with 12MB L1, 192MB L2, and 768MB L3 cache, along with 755GB of RAM. We utilize a 447GB Micron 7450 NVMe SSD storage.

Implementation & Baselines. We build Zeno Filter on top of RSQF [39] using C++. Since the closest competitors, Aleph Filter and InfiniFilter, are built using Java on top of a less efficient quotient filter variant [2], we implemented C++ versions of these two filters on top of RSQF. This enables an apples to apples comparison against Zeno Filter, whereby only core features pertaining to expansion are varied across experiments. By default, these baselines all operate in the Fixed-Width Regime. We describe several other baselines from the literature in Experiment 5.

Setup. Across all experiments except Experiment 7, we use uniformly randomly generated 64-bit integer keys for both insertions and queries. For Experiment 7, we employ two non-synthetic workloads to simulate real-world access patterns. By default, a filter is initialized with 2^8 slots and 16-bit fingerprints (of which at least one bit serves as unary padding). In every experiment, we issue more inserts than the filter's initial capacity, causing it to expand at least once. The expansion threshold is set to $\alpha = 0.9$ by default. For insertions, we report the average latency across periods of multiple expansions to take expansion cost into account. For queries, we focus on the worst-case query performance just before expansion when the filter is full. Every point measuring query latency is an average across 100K queries. When measuring the filter size, we use an independent thread to monitor the memory usage every 10 ms.

Experiment 1: Stretching. We first evaluate the Stretching technique from Section 4.1. We experiment with four different values of the growth coefficient r (1, 2, 3, and 4). These correspond to expansions by factors of 2, 1.41, 1.26, and 1.19, respectively. Figure 10-(a) shows the impact on persistent space amplification, measured as the total number of slots in the newest filter divided by the number of occupied slots. As shown, space amplification fluctuates across expansions. It is highest right after expansion and decreases as the filter fills up. The amplitude of the fluctuations

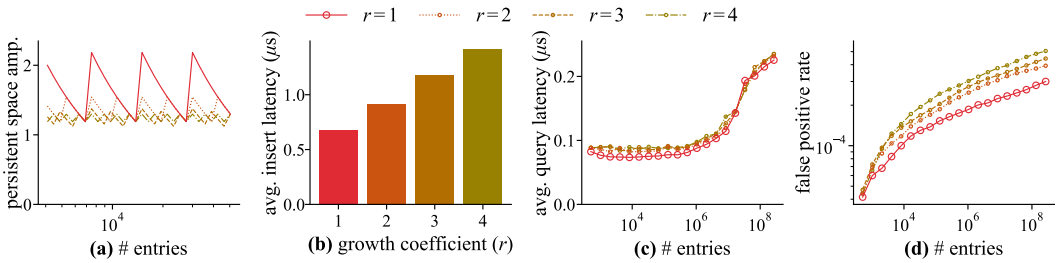


Fig. 10. Using Stretching, Zeno Filter can expand by less than a factor of two to curb persistent space amplification in exchange for a higher average write cost. The impact on query latency is negligible, while the FPR doubles at most.

is smaller for higher growth coefficients as the filter expands by smaller factors. When $r = 1$, the space amplification peaks at 2.22, as the filter doubles in size when it is $\approx 90\%$ full. As r increases, the maximum space amplification drops to 1.57 for $r = 2$, 1.40 for $r = 3$, and 1.32 for $r = 4$, matching our cost model in Equation 1.

Figure 10-(b) analyzes average insertion latency as we insert 2^{28} keys, letting the filters expand ≈ 20 times. The average insertion latency increases in small linear steps with respect to r due to more frequent expansions, as captured by our cost model in Equation 2. This is the price we pay in exchange for curbing space amplification using Stretching.

Figure 10-(c) shows the average query latency as the filter expands. All configurations exhibit an increase in latency going leftwards as they outgrow the CPU caches. For higher values of r, query latency slightly rises due to runs becoming slightly longer on average before splitting in the next period. Consequently, queries take slightly longer to traverse these runs. Nevertheless, the increase in query cost with respect to r is nearly negligible.

Figure 10-(d) illustrates the FPR measured using the same non-existing keys as in Part (c). All variants of Zeno Filter exhibit a logarithmic FPR as characteristic of the Fixed-Width Regime. Nevertheless, the FPR is slightly higher for higher values of r since runs are slightly longer, meaning there are more opportunities for false positives to occur. Yet, the FPR is higher by at most a factor of 2 for any value of r as runs at most double in length during a period and get split in half at the end of it.

Overall, we observe that Stretching significantly curbs persistent space amplification in exchange for a moderate increase in write cost and a slight increase in query cost and FPR. Thus, Stretching enables new attractive trade-offs for space-conscious applications.

Experiment 2: In-place Expansion. We now evaluate Zeno Filter's in-place expansion mechanisms, which employs a software-based indirection layer as described in Section 4.2. We also evaluate Zeno Filter-VM, which leverages virtual memory as an alternative to the software-based indirection layer. We compare these variants to Aleph Filter, which expands out-of-place. We evaluate Aleph Filter under two settings — one using equivalent 16-bit fingerprints, and another using 10-bit fingerprints to match the maximum memory budget against Zeno Filter. The results are shown in Figure 11. We set the Zeno Filter variants to expand by a factor of two in this experiment to remove the impact of Stretching and enable an apples to apples comparison with Aleph Filter.

Figure 11-(a) illustrates these baselines' memory footprint over time as we insert 2^{28} entries. Both configurations of Aleph Filter exhibit spikes in memory as the newer and older filters exist simultaneously during expansion. In contrast, neither Zeno Filter variant incurs such spikes since they expand in-place. Their maximum space amplification is smaller by a third compared to the 16-bit Aleph Filter and identical to the 10-bit Aleph Filter.

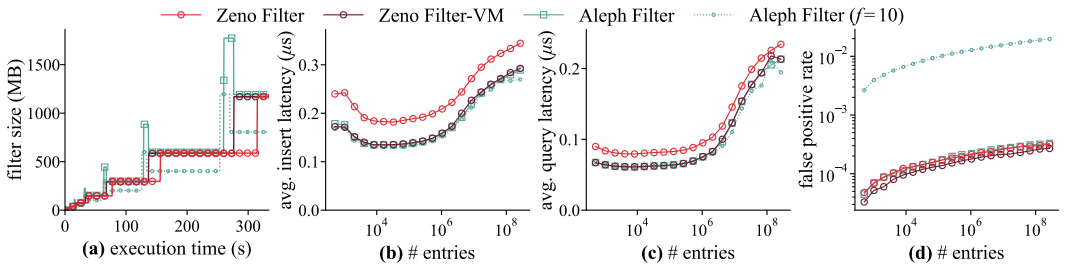


Fig. 11. Zeno Filter eliminates transient space amplification by expanding in-place. The price is a modest performance overhead due to the software-based layer of indirection. Zeno Filter-VM shrinks these overheads at the expense of portability.

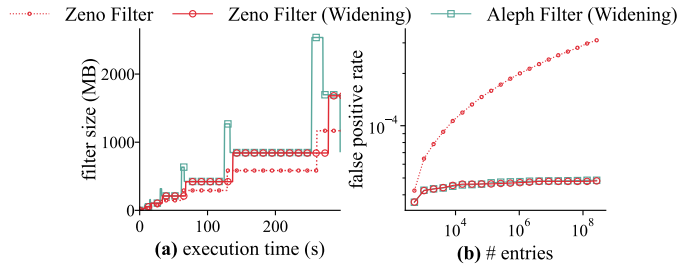


Fig. 12. By employing the Widening Regime, both Zeno Filter and Aleph Filter stabilizes the FPR to a constant at the cost of additional space overhead. Zeno Filter further eliminates transient space amplification.

Parts (b) and (c) of Figure 11 report the average insertion and query latencies as the filters grow. Zeno Filter is slightly slower than Aleph Filter due to the additional software-based indirection layer. Nevertheless, Zeno Filter-VM matches the performance of Aleph Filter by using virtual memory as a hardware-based indirection layer while sacrificing portability. Reducing the fingerprint size for Aleph Filter has almost no impact on insertion or query latency.

Figure 11-(d) reveals the cost of this memory-constrained configuration. While the standard Aleph Filter and both Zeno Filter variants maintain a low FPR, the 10-bit Aleph Filter suffers from significant FPR degradation to satisfy the stricter memory budget.

Overall, this experiment demonstrates that Zeno Filter's in-place expansion mechanism eliminates memory spikes during expansion for a modest price in either performance or portability. It further shows that matching Zeno Filter's memory efficiency with an out-of-place expanding filter requires a prohibitive sacrifice in FPR.

Experiment 3: Widening Regime. Figure 12 evaluates the memory footprint and FPR of Zeno Filter under the Widening Regime. We compare against Zeno Filter with the Fixed-Width Regime and Aleph Filter using the same widening strategy. The results show that widening allows both filters to stabilize the FPR at a constant level, unlike the Fixed-Width Regime where the FPR continues to grow. This stabilization comes at the cost of additional space overhead. However, the overhead is more pronounced for Aleph Filter due to transient space amplification caused by out-of-place expansion.

Experiment 4: Directory Size. We evaluate the impact of Zeno Filter's software-based in-place expansion scheme, which gradually increases the block size as data size grows. Recall that Zeno Filter configures its largest block size to be $O(\sqrt{N})$ for N entries, which in turn makes the directory grow at a slow rate of $O(\sqrt{N})$. This design keeps the directory cache-resident and quickly accessible. To isolate the effect of directory growth, we compare against a modified variant that allocates fixed-size blocks throughout, causing the directory to grow linearly with the data size.

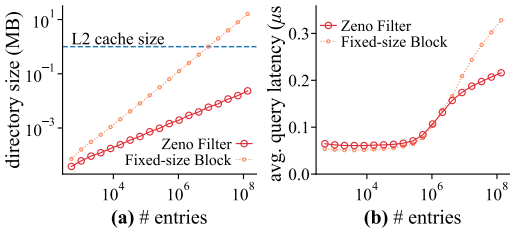


Fig. 13. Zeno Filter's directory size grows sublinearly with the data size, allowing it to remain cache-resident and enable fast filter access as data size increases.

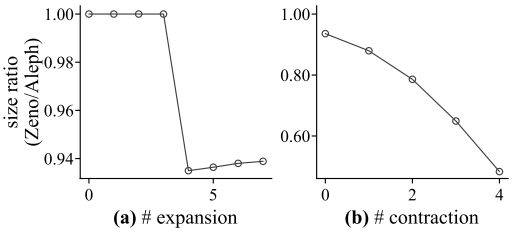


Fig. 14. While Aleph Filter's secondary hash table is initially small, its relative footprint increases under deletions and contractions. Zeno Filter avoids this overhead through its void entry encoding scheme.

Figure 13-(a) shows the directory size as a function of the number of inserted entries. Zeno Filter's directory grows sublinearly and remains within the L2 cache across the entire range of the experiment. In contrast, the fixed-size block scheme exhibits a linear increase in directory size, outgrowing the L2 cache capacity. This difference highlights the effectiveness of Zeno Filter's in-place expansion scheme in controlling directory growth as data size scales.

The cache behavior observed in Part (a) directly translates to query performance in Figure 13-(b). While the fixed-size block scheme achieves slightly lower latency at small scales due to simpler indexing, its query latency increases sharply once the directory outgrows the L2 cache. Zeno Filter avoids this by maintaining a cache-resident directory, resulting in consistently lower query latency at larger scales. These results demonstrate that Zeno Filter's expansion scheme trades a small amount of indexing complexity for substantially improved scalability.

Experiment 5: Void Entry Encoding and Deletion. We now measure the space savings gained using the void entry encoding scheme from Section 4.3. The experiment reflects a workload consisting of two phases whereby the dataset initially expands and later contracts. Each filter slot spans one byte in this experiment.

Figure 14-(a) reports the size ratio between Zeno Filter to Aleph Filter during the first phase of the experiment across eight expansions. As shown, Zeno Filter is smaller than Aleph Filter by $\approx 6\%$ after the fifth expansion. The reason is that at this point, Aleph Filter allocates a secondary hash table to store the hashes of its void entries. Zeno Filter obviates the need for this secondary hash table by making void sequences identifiable and unambiguous.

In the second phase of the experiment, we delete keys in the reverse order of their insertions. As an outcome, Zeno Filter and Aleph Filter both contract four times. With Aleph Filter, the secondary hash table's proportion of the memory footprint increases as the oldest keys are not affected by the deletions. Therefore, the relative size of Zeno Filter significantly decreases relative to Aleph Filter, as shown in Figure 14-(b). By the end of the experiment, Zeno Filter is nearly half the size of Aleph Filter. Overall, this experiment highlights the space reduction enabled by Zeno Filter's void encoding scheme especially for datasets with fluctuating sizes.

Experiment 6: Concurrency. Parts (a) and (b) of Figure 15 compare the insertion throughput of Zeno Filter and RSQF across varying thread counts. In Part (a), we initialize the filters with 2^{24} slots and perform insertions until the filters become full. Since Zeno Filter takes incremental locks on its regions, it is able to employ a finer locking granularity of $R = 4096$ compared to RSQF's coarser $R = 65536$. As a result, Zeno Filter outperforms RSQF as concurrency increases. Part (b) reports similar results, where insertions are performed until the filters undergo two expansions. However, overall scalability is limited because expansion remains single-threaded. Enabling multi-threaded expansion to migrate independent regions in parallel remains an interesting direction for future work.

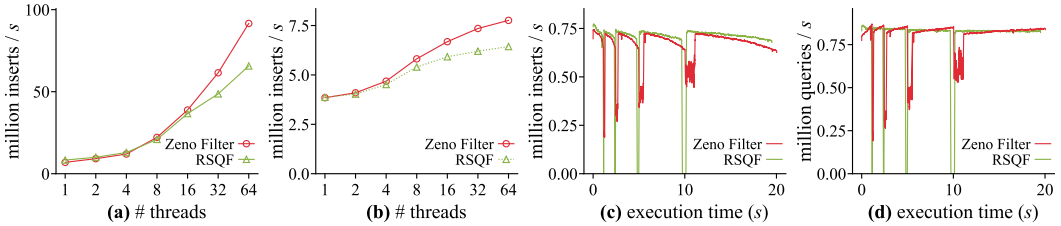


Fig. 15. Zeno Filter benefits from incrementally locking regions in a finer granularity. Zeno Filter also allows for concurrent filter access during expansion.

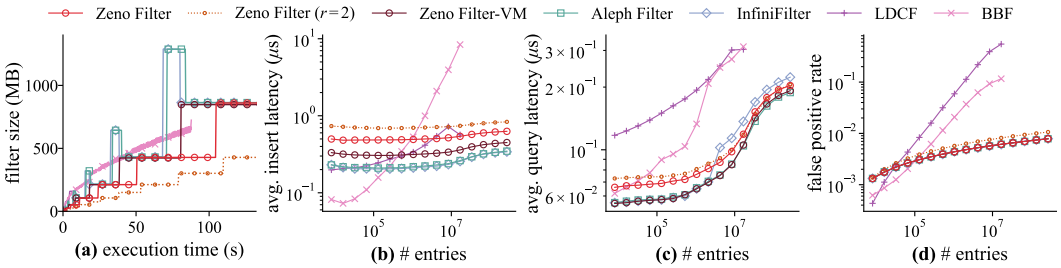


Fig. 16. Zeno Filter significantly reduces space amplification relative to state-of-the-art expandable filters while exhibiting similar performance and matching their FPR.

Parts (c) and (d) of Figure 15 illustrate Zeno Filter's ability to handle concurrent insertions and queries during expansion. We initialize a filter with 2^{20} slots and begin execution with two threads, where one performs random insertions and the other executes random queries. Keys are inserted to the point where the filter undergoes four expansions. Once expansion is triggered, Zeno Filter automatically spawns a new iterator thread to migrate entries, as described in Section 4.4. The four dips in throughput correspond to expansion phases. While in-place expansion introduces contention on the shared structure, our fine-grained locking scheme isolates this to the specific regions being migrated. Unlike RSQF, which halts operations during filter expansion, Zeno Filter sustains throughput, robustly processing concurrent insertions and queries throughout expansion.

Experiment 7: Holistic Evaluation. We now compare Zeno Filter holistically against four recent expandable filters from the literature. Aside to InfiniFilter, and Aleph Filter, which we discussed in Section 2, we also compare to Bamboo filter (BBF) [47] and to the Logarithmic Dynamic Cuckoo Filter (LDCF) [54]. In their core, BBF and LDCF are cuckoo filters that employ the fingerprint sacrifice method to expand without rereading the original data, and they expand at a fine granularity using variants of linear and extendible hashing, respectively. They also employ overflow chains for fingerprints they cannot find space for. We use the public implementations for BBF [46] and LDCF [30]. All baselines are initialized with 2^{12} slots and we then insert 2^{29} keys. The experiment features three variants of Zeno Filter. We include Zeno Filter using growth coefficients 1 and 2 to showcase results with and without Stretching. We also include Zeno Filter-VM with growth coefficient 1. The average insertion cost as measured in this experiment accounts for the amortized cost of expansions.

In figures Figure 16-(a) and Figure 16-(b), we observe that the memory footprint and insertion speed of both BBF and LDCF rapidly deteriorate. The reason is that as they expand, they lose fingerprint bits. As a result, the underlying cuckoo filter is not able to effectively resolve collisions, as its collision resolution strategy rehashes a fingerprint to an alternative bucket. Consequently, more swapping takes place for each insertion and overflow chains are assigned when no valid location can be set for a given entry. These baselines also exhibit a deteriorating query performance

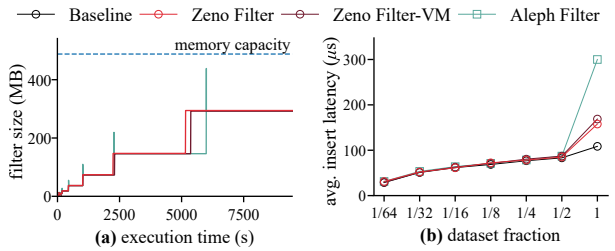


Fig. 17. Zeno Filter does not exhibit spikes in memory, allowing a larger buffer pool to be allocated. This maintains insertion performance, even when a portion of the system memory is traded for the filter.

on account of having to traverse longer overflow chains, and the FPR increases since the fingerprints shrink as the filter expands.

InfiniFilter and Aleph Filter perform similarly throughout the experiments. Their curves in Figure 16-(a) match each other, both showing memory spikes due to out-of-place expansions. For InfiniFilter, query performance is higher than Aleph Filter towards the end of the experiment, as a secondary hash table is allocated and has to be traversed by queries. Aleph Filter eliminates this overhead by duplicating void entries within the main hash table.

As shown in Figure 16-(a), none of the Zeno Filter variants exhibit memory spikes due to expansions as they expand in-place. We further observe that the Zeno Filter variant with Stretching has a memory footprint that increases in more granular steps, more closely matching the data size in the filter. For the Zeno Filter variants with software-based indirection, the price is a slight performance degradation as shown in Figure 16-(b) and (c), yet this is a good trade-off for its space savings. That said, Zeno Filter-VM matches the performance of Aleph Filter and InfiniFilter using virtual memory rather than the software-based indirection approach. The FPR for the Zeno Filter variants is the same as for Aleph Filter and InfiniFilter as they all employ the Fixed-Width Regime in this experiment.

Overall, the experiment against other baselines demonstrates that Zeno Filter is able to significantly reduce space amplification relative to the state of the art while exhibiting similar performance and matching their FPR.

Experiment 8: End-to-End Evaluation. To verify the practical benefits of Zeno Filter in a production-grade storage engine, we integrate it into WiredTiger [38], the storage engine backend for MongoDB [37]. This setup allows us to evaluate the filter's impact on a complex system where memory management is critical. We use the following real-world datasets [31, 35]:

- **Books**: 200M 64-bit Amazon booksale popularity scores.
- **OSM**: 200M 64-bit Open Street Map geocoordinates.

Both datasets exhibit nearly identical memory and performance trends; for clarity, we present results for the Books dataset and omit the other. Following previous work [25, 26], we evaluate with subsets of 100M keys from each dataset, and we pair each key with a random 504-byte value to generate 512-byte records. We measure insertion throughput at logarithmic intervals (from 1/64th to the full dataset) to capture performance characteristics across varying data scales.

We compare Zeno Filter and Zeno Filter-VM against Aleph Filter and a vanilla WiredTiger configuration. We restrict the total system memory (WiredTiger buffer pool + filter) to 1% of the maximum dataset size. For filter-enabled baselines, the memory required for the filter is deducted from the total system memory.

Crucially, we configure the buffer pool to strictly enforce the memory budget during expansion events. Since Aleph Filter expands out-of-place, we reduce its steady-state buffer pool allocation to

reserve headroom for its transient space amplification. In contrast, Zeno Filter's in-place expansion eliminates this need for reservation, allowing it to dedicate a larger portion of the memory budget to the buffer pool.

Figure 17-(a) plots the memory footprint of the filters as the dataset is fully ingested. As shown, Aleph Filter exhibits transient spikes during expansion, nearly hitting the memory capacity limit. In contrast, Zeno Filter expands in-place, resulting in a monotonic memory profile that stays within safe bounds without requiring temporary headroom.

Figure 17-(b) illustrates the impact of this memory behavior on insertion latency. As the dataset fraction increases, the system becomes increasingly memory-constrained. Aleph Filter suffers severe performance degradation at these later stages. This is a direct consequence of its transient space amplification. To safely accommodate the expansion spikes shown in Part (a), we were forced to configure WiredTiger with a significantly smaller buffer pool. This reduced cache size leads to frequent disk I/O, stalling insertions. Conversely, Zeno Filter's lack of expansion spikes allowed us to allocate a larger portion of the budget to the buffer pool, maintaining higher cache hit rates and lower latency.

Finally, notice that the performance difference between Zeno Filter and Zeno Filter-VM is minimal in this experiment. Unlike previous in-memory microbenchmarks, the WiredTiger workload is I/O-bound, masking the overhead of the software-based indirection layer.

6 Conclusion

We introduced Zeno Filter, a dynamic filter that tackles both persistent and transient space amplification. It does so by expanding in-place by a smaller factor. We showed that Zeno Filter achieves scalable performance and FPR, while maintaining the smallest memory footprint among all baselines.

References

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261.

[2] Michael A Bender, Martin Farach-Colton, Rob Johnson, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. 2011. Don't thrash: How to cache your hash on flash. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*.

[3] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[4] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

[5] Andrej Brodnik, Svante Carlsson, Erik D Demaine, J Ian Ian Munro, and Robert Sedgewick. 1999. Resizable arrays in optimal time and space. In *Algorithms and Data Structures: 6th International Workshop, WADS'99 Vancouver, Canada, August 11–14, 1999 Proceedings 6*. Springer, 37–48.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[7] Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin hood hashing. In *26th annual symposium on foundations of computer science (sfcs 1985)*. IEEE, 281–288.

[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*. 275–290.

[9] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. 2017. The dynamic cuckoo filter. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 1–10.

[10] Yuvaraj Chesetti, Navid Eslami, Huanchen Zhang, Niv Dayan, and Prashant Pandey. 2026. Aeris Filter: A Strongly and Monotonically Adaptive Range Filter. *Proceedings of the ACM on Management of Data* 4, 1 (2026), 1–26.

[11] Rayan Chikhi, Jan Holub, and Paul Medvedev. 2021. Data structures to represent a set of k-long DNA sequences. *ACM Computing Surveys (CSUR)* 54, 1 (2021), 1–22.

[12] Rayan Chikhi and Guillaume Rizk. 2013. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology* 8, 1 (2013), 22.
[13] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. {SplinterDB}: closing the bandwidth gap for {NVMe}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 49–63.
[14] Intel Corporation. 2025. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://cdrdv2.intel.com/v1/dl/getContent/671200
[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.
[16] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. Infinifilter: Expanding filters to infinity and beyond. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
[17] Niv Dayan, Ioana-Oriana Bercea, and Rasmus Pagh. 2024. Aleph Filter: To Infinity in Constant Time. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3644–3656.
[18] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.
[19] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.
[20] Niv Dayan and Moshe Twitto. 2021. Chucky: A succinct cuckoo filter for LSM-tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.
[21] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. {ChunkStash}: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.
[22] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1414–1425.
[23] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 25–36.
[24] Ozgun Erdogan and Pei Cao. 2007. Hash-AV: fast virus signature scanning by cache-resident filters. *International Journal of Security and Networks* 2, 1-2 (2007), 50–59.
[25] Navid Eslami, Ioana O Bercea, and Niv Dayan. 2025. Diva: Dynamic range filter for var-length keys and queries. *Proceedings of the VLDB Endowment* 18, 11 (2025), 3923–3936.
[26] Navid Eslami and Niv Dayan. 2024. Memento Filter: A Fast, Dynamic, and Robust Range Filter. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–27.
[27] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.
[28] Deke Guo, Jie Wu, Honghui Chen, and Xueshan Luo. 2006. Theory and network applications of dynamic bloom filters. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 1–12.
[29] AMD Inc. 2024. AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf
[30] David Ivekovic. [n. d.]. LDCF. https://github.com/DavIvek/BioInf1
[31] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).
[32] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard TB Ma, Xueshan Luo, and Bangbang Ren. 2019. The consistent cuckoo filter. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 712–720.
[33] Sajad Faghfoor Maghrebi and Niv Dayan. 2025. Sphinx: A Succinct Perfect Hash Index for x86. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4424–4437.
[34] Camille Marchet, Christina Boucher, Simon J Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. 2021. Data structures based on k-mers for querying large collections of sequencing data sets. *Genome research* 31, 1 (2021), 1–12.
[35] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.
[36] Meta. 2021. RocksDB Bloom Filter. https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter
[37] MongoDB. 2025. The Developer Data Platform. https://www.mongodb.com/
[38] MongoDB. 2025. WiredTiger Storage Engine. https://www.mongodb.com/docs/manual/core/wiredtiger/
[39] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM international conference on Management of Data*. 775–787.
[40] Prashant Pandey, Martín Farach-Colton, Niv Dayan, and Huanchen Zhang. 2024. Beyond Bloom: A Tutorial on Future Feature-Rich Filters. In *Companion of the 2024 International Conference on Management of Data*. 636–644.

[41] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.
[42] Redis. 2025. Bloom filter. https://redis.io/docs/latest/develop/data-types/probabilistic/bloom-filter/
[43] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.
[44] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM design space and its read optimizations. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3578–3584.
[45] Robert E Tarjan and Uri Zwick. 2024. Optimal resizable arrays. *SIAM J. Comput.* 53, 5 (2024), 1354–1380.
[46] Hancheng Wang, Haipeng Dai, Shusen Chen, Meng Li, Rong Gu, Huayi Chai, Jiaqi Zheng, Zhiyuan Chen, Shuaituan Li, Xianjun Deng, et al. [n. d.]. Bamboo Filters. https://github.com/wanghanchengchn/bamboofilters
[47] Hancheng Wang, Haipeng Dai, Shusen Chen, Meng Li, Rong Gu, Huayi Chai, Jiaqi Zheng, Zhiyuan Chen, Shuaituan Li, Xianjun Deng, et al. 2024. Bamboo filters: make resizing smooth and adaptive. *IEEE/ACM Transactions on Networking* 32, 5 (2024), 3776–3791.
[48] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. Grf: A global range filter for lsm-trees with shape encoding. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
[49] Jigang Wen, Shuyu Pei, Chuhan Yan, Kun Xie, and Wei Liang. 2024. Extensible bloom filters: Adaptive strategies for scalability and efficiency in network and distributed systems to handle increased data. *Tsinghua Science and Technology* (2024).
[50] Wikipedia contributors. 2025. Unary coding. https://en.wikipedia.org/wiki/Unary_coding. Accessed: 2025-10-05.
[51] Yuhan Wu, Jintao He, Shen Yan, Jianyu Wu, Tong Yang, Olivier Ruas, Gong Zhang, and Bin Cui. 2021. Elastic bloom filter: deletable and expandable filter using elastic fingerprints. *IEEE Trans. Comput.* 71, 4 (2021), 984–991.
[52] Minghao Xie, Quan Chen, Tao Wang, Feng Wang, Yongchao Tao, and Lianglun Cheng. 2022. Towards capacity-adjustable and scalable quotient filter design for packet classification in software-defined networks. *IEEE Open Journal of the Computer Society* 3 (2022), 246–259.
[53] Shuiying Yu, Sijie Wu, Hanhua Chen, and Hai Jin. 2020. The entry-extensible cuckoo filter. In *IFIP International Conference on Network and Parallel Computing*. Springer, 373–385.
[54] Fan Zhang, Hanhua Chen, Hai Jin, and Pedro Reviriego. 2021. The logarithmic dynamic cuckoo filter. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 948–959.

Received October 2025; revised January 2026; accepted February 2026