InfiniFilter: Expanding Filters to Infinity and Beyond

NIV DAYAN, University of Toronto, Canada
IOANA BERCEA, BARC, IT University of Copenhagen, Denmark
PEDRO REVIRIEGO, Universidad Politécnica de Madrid, Spain
RASMUS PAGH, BARC, University of Copenhagen, Denmark

Filter data structures have been used ubiquitously since the 1970s to answer approximate set-membership queries in various areas of computer science including architecture, networks, operating systems, and databases. Such filters need to be allocated with a given capacity in advance to provide a guarantee over the false positive rate. In many applications, however, the data size is not known in advance, requiring filters to dynamically expand. This paper shows that existing methods for expanding filters exhibit at least one of the following flaws: (1) they entail an expensive scan over the whole data set, (2) they require a lavish memory footprint, (3) their query, delete and/or insertion performance plummets, (4) their false positive rate skyrockets, and/or (5) they cannot expand indefinitely.

We introduce InfiniFilter, a new method for expanding filters that addresses these shortcomings. InfiniFilter is a hash table that stores a fingerprint for each entry. It doubles in size when it reaches capacity, and it sacrifices one bit from each fingerprint to map it to the expanded hash table. The core novelty is a new and flexible hash slot format that sets longer fingerprints to newer entries. This keeps the average fingerprint length long and thus the false positive rate stable. At the same time, InfiniFilter provides stable insertion/query/delete performance as it is comprised of a unified hash table. We implement InfiniFilter on top of Quotient Filter, and we demonstrate theoretically and empirically that it offers superior cost properties compared to existing methods: it better scales performance, the false positive rate, and the memory footprint, all at the same time.

CCS Concepts: • **Information systems → Data structures**; • **Theory of computation → Data structures design and analysis**.

Additional Key Words and Phrases: Probabalistic data structures, approximate set membership, Bloom filter, quotient filter, expandability, data growth, scalability.

ACM Reference Format:
Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond . *Proc. ACM Manag. Data* 1, 2, Article 140 (June 2023), 27 pages. https://doi.org/10.1145/3589285

1 INTRODUCTION

What is a Filter? A filter is a compact probabilistic data structure that represents keys in a set. As it is smaller than the keys that it represents, it can be stored at a higher level of the memory hierarchy (e.g., DRAM or SRAM), even if the keys themselves reside over a network or on a disk drive due to their larger size. Filters answer queries for whether a given key exists in a set, and some can also store and retrieve a payload for each key. A filter cannot return false negatives, but does return false positives with a probability that depends on the amount of memory assigned to it.

Authors' addresses: Niv Dayan, University of Toronto, Canada; Ioana Bercea, BARC, IT University of Copenhagen, Denmark; Pedro Reviriego, Universidad Politécnica de Madrid, Spain; Rasmus Pagh, BARC, University of Copenhagen, Denmark.

2836-6573/2023/6-ART140 $15.00
https://doi.org/10.1145/3589285

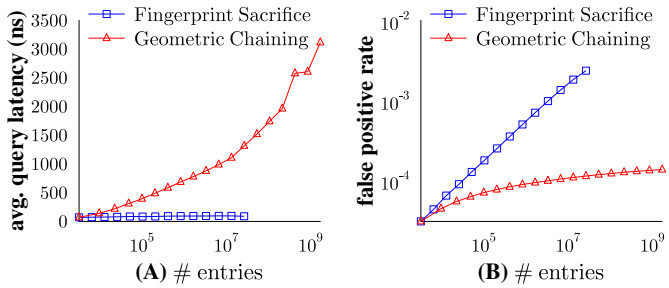

Fig. 1. Existing methods for expanding filters do not support a stable query cost, a stable false positive rate, and indefinite expansion, all at the same time.

Thus, a filter can quickly rule out the existence of a key without searching the full data set. This boosts query performance by eliminating hops across a network and/or expensive storage accesses.

Filters are Ubiquitous. Since the invention of the original Bloom filter [9] in the early 1970s, filters have been ubiquitously used in various areas of computer science. Database management systems (e.g., Cassandra [57], HBase [46]) and key-value stores (e.g., Speedb [83], RocksDB [67], LevelDB [40]) use filters to avoid searching for a particular data item in storage [20–26, 80]. Hash join algorithms employ filters to rule out non-matching entries for a given key [6]. Filters are used in network applications to prevent redundant communication [12], in security to eliminate denial of service attacks or detect malicious URLs [38], and in search engines [39] to rule out documents that do not match a given search term.

The Need for Filter Expandability. Many modern applications manage dynamic data that grows over time. Such applications need to expand their filter/s as the size of the data that they represent grows. In the architecture community, the 2020 ASPLOS best paper replaces the traditional multi-level translation lookaside buffer (TLB) with a Cuckoo filter for virtual memory translation and identifies expansion as a pivotal challenge [82]. In the networks community, efficient filter expansion was recently identified as a central problem for supporting black lists, MAC address lookups, multi-cast routing, and longest prefix matching [89]. In the operating systems and databases communities, modern storage engines update data out-of-place and employ a filter to map the location of every data entry in storage [2, 16, 24, 25, 27, 76, 84]. As the data grows, they must expand this filter as more data is inserted. Such storage engines include web caches, key-value stores, relational databases, and file systems, and they are used across numerous applications including gaming and deduplication [27], real-time monitoring and analytics [16], data-series discovery [53–55], social networks [66], internet advertising [84], etc. While the problem of filter expansion is prevalent across myriad applications and communities, existing solutions, which we describe next, are lacking.

Existing Expansion Methods are Insufficient. The difficulty common to all filters with respect to expansion is that they store a hashed representation of each key rather than the original key itself. Hence, the original keys cannot be rehashed when expanding, as done with a regular hash table. The obvious workaround is to scan the original data and construct a filter with greater capacity from scratch. However, the cost of traversing the whole data set can be prohibitive. Another possibility is to pre-allocate a very large filter in advance, but this wastes a lot of memory from the get-go and restricts the ultimate set size the filter can represent. Yet another option is to create a chain (i.e., a linked list) of filters with geometrically increasing capacities, and to add new filters to this chain as the data grows. We refer to this as Geometric Chaining. However, this method increases query costs as all filters along the chain potentially need to be searched.

Over the past decade, a new family of filters emerged, including Quotient filter [7, 32], Cuckoo filter [36], and others [35, 87]. These filters store a fingerprint (i.e., a hash digest) for each key

within a compact hash table. We refer to such filters collectively as tabular filters. Tabular filters provide limited support for expansion: it is possible to double their capacity and sacrifice one bit from each fingerprint to map it to the expanded hash table. We coin this the Fingerprint Sacrifice method. The problem is that the fingerprints shrink as the data grows, and this increases the false positive rate. Furthermore, the fingerprint bits eventually run out, at which point the filter becomes useless: it returns a positive for every query, and it cannot continue expanding. While it is possible to delay this problem by initializing longer fingerprints from the onset, this does not fundamentally solve the problem and costs additional memory.

Figure 1 illustrates how query latency and the false positive rate scale as we increase the data size with Geometric Chaining and Fingerprint Sacrifice, both implemented on top of a Quotient filter [7]. The full experiment is described in Section 5. Geometric Chaining exhibits rapidly increasing query costs as there are more filters to search. In contrast, Fingerprint Sacrifice exhibits a catapulting false positive rate and cannot expand indefinitely as eventually, all fingerprints run out of bits.

Research Challenge. We identify the following research problem: is it possible to expand a filter indefinitely without rereading the original data and while maintaining (1) fast queries, insertions, and deletes, (2) a stable false positive rate, and (3) a stable memory footprint? What are the best possible cost trade-offs among these metrics that we can achieve?

Insight: Variable-Length Fingerprints. Our insight is that tabular filters can be adapted to store fingerprints of different lengths for different entries. This insight can be utilized to efficiently expand tabular filters. During expansion, even though we must sacrifice a bit from each fingerprint to map it to the expanded hash table, new entries inserted after the expansion can still be assigned longer fingerprints. This approach promises to keep the average fingerprint length longer and thus the weighted false positive rate more stable. Moreover, since this approach keeps all entries within a unified hash table, it promises to maintain high performance.

Our Solution. We introduce *InfiniFilter*, a novel method for expanding filters. InfiniFilter is a tabular filter that stores a fingerprint along with a unary encoded age counter for each entry within a compact hash table. The age counter counts how many expansions ago a given entry was inserted. During expansion, one bit from each fingerprint is sacrificed to map the fingerprint to an expanded hash table with double the capacity. At the same time, the unary counter is incremented. Hence, all entries remain uniformly sized and perfectly aligned within the hash table's slots.

Since InfiniFilter doubles in capacity during each expansion, half the entries in the filter are new, another quarter are slightly less recent, and so on. In particular, the fingerprint lengths follow a geometric distribution, meaning that most fingerprints are long and exponentially fewer are shorter. This keeps the average fingerprint length long and thus the weighted false positive rate stable. Moreover, queries, deletes, and insertions are fast as entries from across different expansions co-exist within a unified hash table. Due to this novel design, InfiniFilter provides a stable false positive rate, modest memory footprint, and high performance, all at the same time.

Contributions. In summary, our contributions are to:

(1) Show how to assign longer fingerprints for newer entries within the same tabular filter using unary encoded age counters, which serve as parsable self-delimiting padding.
(2) Show that variable-length fingerprints complicate deletes, yet deletes can still be performed correctly and efficiently by targeting the longest matching fingerprint within a hash slot.
(3) Show that the basic version of InfiniFilter can only be expanded a finite number of times as eventually the oldest fingerprints run out of bits and cannot be remapped to a larger filter. However, InfiniFilter can be combined with chaining to expand indefinitely while only slightly degrading query and delete cost, both of which become $O(\log(N)/F)$, where F is the initial fingerprint length.

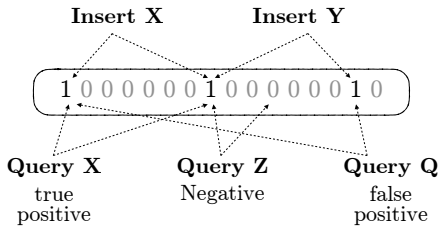

Fig. 2. A Bloom filter hashes each key to k bits, setting them from 0s to 1s or keeping them set to 1s. It returns a positive for a query if all bits for the key in question are 1s or a negative otherwise.

(4) Show that with a fixed number of bits per entry, InfiniFilter supports a worst-case $O(\log N)$ false positive rate. However, by slightly increasing the fingerprint length assigned to newer entries at a rate of $O(\log \log N)$, the false positive rate stays $O(1)$.
(5) Show that queries to existing keys can rejuvenate (i.e., lengthen) the fingerprints of older entries. This eliminates the cost contention between the false positive rate and the number of bits per entry, allowing to keep both of these cost metrics constant as the data grows.
(6) Implement InfiniFilter and other State of the Art expansion methods on top of Quotient Filter and open-source the code[1].
(7) Empirically and theoretically evaluate InfiniFilter against other expansion methods to show that it achieves superior cost properties.

2 BACKGROUND

We now provide background on Bloom and Quotient filters.

Bloom Filters. Bloom filter [9] is the most traditional, common, and easy-to-implement kind of filter. It consists of a bitmap initially set to all zeros. A key is inserted by hashing it using k hash functions to k random positions in the bitmap and setting all bits in these positions to ones. A lookup involves hashing the key in question to its k positions. If at least one of the bits in these positions is set to zero, the key could not have been inserted and so the filter returns a negative answer. Otherwise, the filter returns a positive answer. Since the bits in question could have been coincidentally set to 1s by other keys, however, there is a chance of a false positive. The probability θ of a false positive is approximated by Equation 1, where b is the number of bits in the filter, i is the number of inserted keys, and k is the number of hash functions. Figure 2 illustrates a Bloom filter with sixteen bits and two hash functions.

$$\theta \approx \left(1 - e^{-k \cdot \frac{i}{b}}\right)^k \tag{1}$$

As more keys are inserted into a Bloom filter, the false positive rate in Equation 1 increases as more of the bits in the filter get set to 1s. If too many keys are inserted, the Bloom filter becomes useless: it reports a positive for any query, regardless of whether a key has been inserted or not. Hence, a Bloom filter requires knowing the data set size in advance when it is allocated to guarantee a given false positive rate to the user.

Bloom filters do not support deletes. To see why, observe that resetting bits from 1s to 0s to reflect a deletion of some key X could affect a bit that also belongs to some other existing key Y. This would lead to future false negatives when querying for key Y. False negatives violate the semantic guarantees of a filter and cannot be tolerated by most applications that filters are used for.

[1]The code is available at https://github.com/nivdayan/FilterLibrary.

term	definition
n	number of slots currently in the filter
S	initial number of slots in the filter
N	current capacity divided by initial capacity (i.e., $N = n/S$)
X	number of expansions since the start (i.e., $X = \lceil \log_2(N) \rceil$)
M	total memory used for the filter (bits / entry)
F	initial fingerprint size when the filter is first allocated
h(...)	hash function
c	number of Basic InfiniFilters in the Chained InfiniFilter
θ	false positive rate
α	space utilization expansion threshold ($0 < \alpha < 1$)

Table 1. Terms used throughout the paper to describe Quotient Filter and InfiniFilter.

For similar reasons, Bloom filters cannot be efficiently expanded. A Bloom filter does not retain information about which keys had set off which bits. Hence, there is no way of remapping the bits belonging to each key to a Bloom filter with greater capacity without rereading the original data.

Tabular Filters. Since the early 2010s, a new family of filters emerged as an alternative to Bloom filters. These structures store a fingerprint (i.e., a hash digest) for every key within a compact hash table [14], and they typically differ in their collision resolution strategy. Examples include Quotient Filter [7, 32], Cuckoo Filter [36], and others [11, 35, 73, 87]. These new tabular filters offer more promise concerning expandability. We focus on the Quotient filter because, as we will show, its collision resolution strategy is particularly well-suited for InfiniFilter. We also discuss and evaluate other types of filters in Sections 5 and 6.

Quotient Filter. Table 1 lists terms used to describe Quotient filter (QF) and InfiniFilter henceforth in the paper. A QF [7, 32] is a hash table with 2^x slots. Each slot can store one fingerprint, and collisions are handled via Robin Hood hashing [15]. The QF version employed in this paper maps a given key y to a *canonical slot* using the least significant x bits of its hash $h(y)$, and it derives a fingerprint for the key based on the next F bits of the key's hash. If the canonical slot is empty, the key's fingerprint is stored there. If it is non-empty, however, the fingerprint will be stored in some slot to the right. Fingerprints belonging to the same canonical slot are stored along contiguous slots in a so-called *run*. A *cluster* is a group of contiguous runs of which the first run begins at its canonical slot and the subsequent runs have been shifted to the right.

Figure 3 Part (A) illustrates a QF with eight slots and three keys mapped to different canonical slots based on the least significant three bits of their hashes (slot numbers are expressed in binary throughout the paper). The rest of their hashes, marked in red, are stored as fingerprints. Part (B) of Figure 3 shows how the filter's state changes after two more insertions into canonical Slot 011, leading to hash collisions. The result is a cluster comprising two runs between Slots 011 and 110. Note that in this paper, all binary notations assume that more significant bits are to the left.

QF Metadata Flags. To mark the start and end of runs and clusters, each slot includes three metadata flags. The is_occupied flag indicates whether a given slot is a canonical slot for at least one existing key. The is_shifted flag is set for a slot that contains a fingerprint that had been shifted to the right from its canonical slot. The is_continuation flag indicates whether the slot contains a continuation of a run that started to the left.

In Figure 3 Part (B), for example, Slot 011 only has the is_occupied flag set to true because it stores a fingerprint for which Slot 011 is the canonical slot. Slots 100 and 101 both have the is_continuation and is_shifted flags set to true because they are part of a run that starts to the left

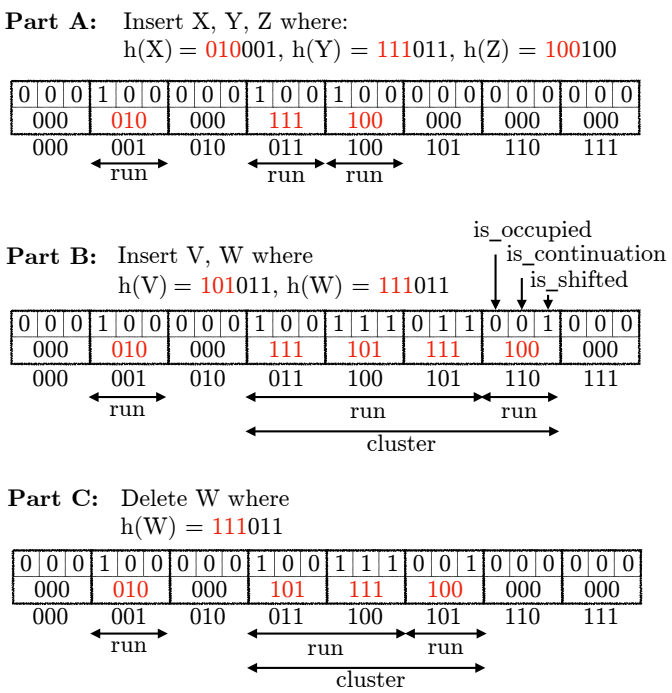

Fig. 3. A Quotient filter stores a fingerprint for each key in a hash table and resolves collisions by organizing fingerprints into runs and clusters, which are demarcated using three metadata bits per slot. Fingerprints in the figure are illustrated in red.

(at Slot 011) but the fingerprints they contain are shifted to the right due to collisions. Slot 100 also has the is_occupied flag set to true because it is the canonical slot for a key that is shifted to the right (to Slot 110). Slot 110 has the is_shifted flag set to true because it belongs to a cluster starting to the left, yet its is_continuation flag is false to mark the start of a new run within this cluster.

QF Queries. A query begins at a given key's canonical slot and moves leftwards until finding the start of a cluster (i.e., a slot with only the is_occupied flag set to true). It then scans the cluster rightwards, keeping a running counter of the number of subsequent runs we must skip. Each slot to the left of the canonical slot with the is_occupied flags set to true indicates one additional run to be skipped. This increments our running counter. On the other hand, each slot with the is_continuation flag set to false indicates the start of a new run. This decrements the running counter. When the running counter's value is zero, we have reached the target run. The query then scans the run's fingerprints and returns a positive if there is at least one exact match.

QF Inserts. An insertion commences similarly to a query by first finding the run to which the fingerprint should be inserted. The fingerprint is added to this run by shifting all subsequent keys in the cluster one slot to the right and potentially adding new runs to the cluster by pushing them to the right from their canonical slots.

QF Deletes. Unlike a Bloom filter, a QF supports deletes to keys we know had previously been inserted. It executes a delete by identifying a key's run and removing from it a matching fingerprint. It then shifts any subsequent key in the cluster one slot to the left, potentially also splitting clusters by shifting some runs back to their canonical slots. Figure 3 Part (C) illustrates a delete operation to fingerprint 111 at canonical slot 011. There are two matching fingerprints, so the first one that is encountered is removed, and all subsequent entries in the cluster are pushed leftwards.

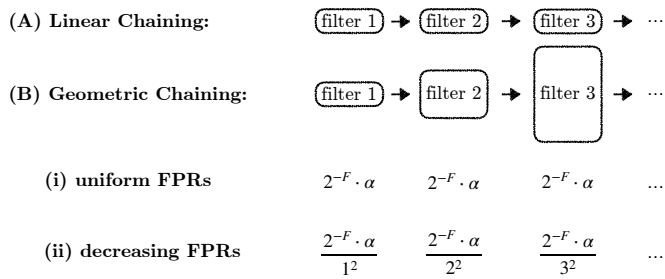

Fig. 4. Linear Chaining allocates linearly more filters as the data size grows. Geometric Chaining allocates exponentially larger ones. Geometric Chaining further supports setting slightly smaller false positive rates to larger filters to cause the overall false positive rate to converge to a constant.

QF Iteration. A QF supports iteration over its fingerprints using a linear left-to-right scan. For each fingerprint encountered along the way, it is possible to infer and report its canonical slot using the metadata flags.

QF Allocation. If too many keys are inserted into a quotient filter and utilization increases beyond $\approx$ 90%, the average cluster length starts growing rapidly until eventually, the performance of queries becomes impractical. Hence, similarly to Bloom filters, a quotient filter has to be allocated with a maximum capacity in mind. Throughout the paper, we let the parameter α denote the fraction at which point a Quotient filter becomes full.

Analysis. For a query to a non-existing key, the false positive rate for a Quotient filter is known to be $\lesssim 2^{-F} \cdot \alpha$. The reason is that each canonical slot is associated with a run containing on average α fingerprints, each of which matches the search key with probability 2^{-F}. The overall memory footprint is $M = \frac{F+3}{\alpha}$ bits per entry. This accounts for one fingerprint and three metadata bits per slot, and the fact that a fraction α of the slots are non-empty.

3 PROBLEM ANALYSIS

This section describes existing techniques for filter expansion and analyzes them as they would apply to a Quotient Filter.

1. Full Reconstruction. The simplest method of expanding a filter is by scanning the full data set, rebuilding a filter with greater capacity from scratch, and disposing of the original filter [24, 89]. The problem is that the original data often resides on a slower storage medium (e.g., disk or SSD) and/or over a network. It can therefore be expensive to fully scan. In some applications, data is regularly scanned in the background (e.g., compaction operations in LSM-trees [3, 67, 70]). This provides an opportunity for full filter reconstruction without additional I/O overheads [20]. However, for applications where data is not regularly scanned, full reconstruction is disruptive. Hence, this paper focuses on expansion methods that do not require re-scanning the data at all.

2. Pre-Allocation. A common approach for circumventing the challenge of filter expansion is to pre-allocate a large static filter in advance. For example, the Pliops data processor allocates a large static filter occupying $\approx$100GB when the system is deployed to map from data entries to their location in storage [25]. However, this method requires a lot of memory from the system's get-go, even while the data set is still small. Moreover, this method restricts the maximum number of entries that can ultimately be inserted into the data set. In this paper, we rather focus on methods that enable gradual expansion and do not restrict the maximum data size.

3. Linear Chaining. Several papers propose to accommodate data growth by creating a chain of similarly-sized filters, each with the same false positive rate [17, 44, 45]. New insertions are made to the last filter in the chain, and a new filter is allocated when the last filter reaches capacity. In

	query/ delete	insert	false positive rate	fingerprint bits / key	max. expansions
Linear Chaining [17, 44, 45]	$O(N)$	$O(1)$	$O(2^{-F} \cdot N)$	F	∞
Geom. Chaining [1, 88, 90]	$O(\lg N)$	$O(1)$	$O(2^{-F} \cdot \lg N)$	F	∞
Geom. Chaining & Growing Mem. [71]	$O(\lg N)$	$O(1)$	$O(2^{-F})$	F+$O(\lg \lg N)$	∞
Fingerprint Sacrifice [7, 92]	$O(1)$	$O(1)$	$O(2^{-F} \cdot N)$	$F - O(\lg N)$	F

Table 2. A comparison of different techniques for expanding a Filter. No existing method is scalable in terms of the costs of queries/deletes, the false positive rate, the memory footprint, and the maximum number of expansions, all at the same time.

practice, this is the method used in the FIFO compaction policy within RocksDB [67]. We illustrate this method in Figure 4 Part (A). The downsides are that the costs of queries and deletes increase linearly with data size as possibly all filters must be searched for a matching fingerprint. Moreover, as there are N filters along the chain, each with a false positive rate of $2^{-F} \cdot \alpha$, the overall false positive rate is $\theta \lesssim 2^{-F} \cdot N \cdot \alpha$. The memory footprint is $M = \frac{F+3}{\alpha}$ bits per entry as with a regular quotient filter. We summarize these properties in Row 1 of Table 2.

4. Geometric Chaining. Other papers propose creating a chain of geometrically larger filters as the data expands [1, 71, 88, 90]. New insertions are made to the largest filter, and a new filter with doubled capacity is appended to the chain when the last filter is at capacity. We illustrate this approach in Figure 4 Part (B). This approach entails logarithmic query/delete overheads as the number of filters in the chain is logarithmic with data size. Moreover, the false positive rate is $\theta \lesssim 2^{-F} \cdot \alpha \cdot (\log_2(N) + 1)$, and the memory footprint is $M = \frac{F+3}{\alpha}$ bits per entry. We summarize these properties in Row 2 of Table 2. Note that Geometric Chaining dominates Linear Chaining across all the different cost metrics.

To fully stabilize the overall false positive rate, it is possible to assign lower false positive rates to larger filters along the chain [1, 71, 88]. The best-known method is using the reciprocals of squares: the false positive rate assigned to the i^{th} filter along the chain is smaller by a factor of i^{-2} than the false positive rate assigned to the $(i-1)^{\text{th}}$ filter [71]. This causes the sum of false positive rates across all filters to converge to $\theta \lesssim 2^{-F} \cdot \alpha \cdot \frac{\pi^2}{6}$ (as per the famous Basel Problem solved by Euler). The cost of this method is a higher memory footprint: $M \lesssim \frac{1}{\alpha} \cdot (F + 3 + 2 \cdot \log_2(\log_2(N) + 1))$ bits per fingerprint. Part (B) (ii) of Figure 4 illustrates the false positive rates assigned to filters along the chain using this method with respect to the initial fingerprint length F and the capacity threshold α. We summarize this method's properties in Row 3 of Table 2

5. Fingerprint Sacrifice. Unlike Bloom filters, tabular filters such as Quotient Filters lend themselves to yet another form of expansion that we coin Fingerprint Sacrifice. The idea is to derive the original hash of every entry by concatenating its slot offset with its fingerprint. We can then employ this full hash to reinsert the key into a new tabular filter with doubled capacity [7]. Specifically, an entry belonging to slot x in the older filter is placed in the new filter's slot x if its least significant fingerprint bit is zero. Otherwise, it is placed at slot $x + q/2$ of the new filter, where q is the number of slots in the new filter. Figure 5 shows an example of an expansion from four to eight slots. Note that in the older filter, there is a collision at Slot 1 leading to a run occupying two slots. This collision is resolved after expansion as the least significant bits of the fingerprints of these two entries are different. These entries are therefore mapped to different slots in the expanded filter.

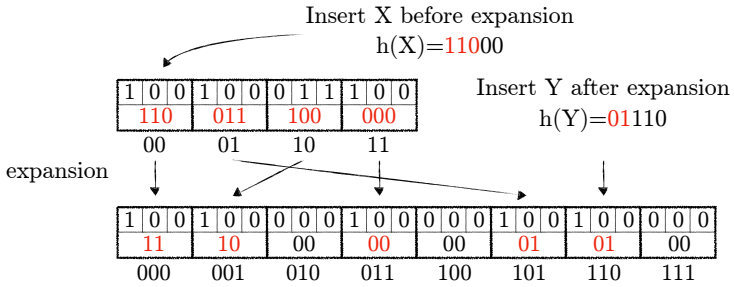

Fig. 5. Quotient Filter can be expanded by sacrificing one bit from each fingerprint to map it to an expanded hash table. We use black and red in the figure to illustrate slot addresses and fingerprints, respectively.

The problem with this approach is that it transfers one bit from each entry's fingerprint to become a part of its slot address during an expansion. Hence, this method does not support infinite expansions: the fingerprints run out of bits after F expansions, where F is the initial fingerprint size. Furthermore, sacrificing one bit from each fingerprint during each expansion causes the false positive rate to double every time the data size doubles. Hence, the false positive rate is $\lesssim 2^{-F} \cdot \alpha \cdot N$, while the memory requirement is $M = \frac{1}{\alpha} \cdot (F - \lceil \log_2(N) \rceil)$ bits per entry. As this method reinserts all existing entries to a new filter whenever the data size doubles, the amortized insertion cost is $\approx 2 \in O(1)$. Hence, it exhibits approximately half of the maximum throughput of the Pre-Allocation or Chaining methods. We summarize these properties in Row 4 of Table 2.

Summary. Geometric Chaining offers indefinite expansion and a superior false positive rate relative to the Fingerprint Sacrifice method. On the other hand, the fingerprint sacrifice method offers faster query/delete operations. The next section introduces InfiniFilter to combine the best of both worlds. Section 6 includes further details on lower bounds and theoretical algorithms for the filter expansion problem from the theory community.

4 INFINIFILTER

InfiniFilter is a new method for expanding set-membership filters. Similarly to the Bit Sacrifice method, InfiniFilter doubles in size when it reaches a configurable capacity threshold, and it sacrifices a bit from each fingerprint to map it to the expanded version. The core innovation is a new entry format that allows setting longer fingerprints to newer entries. This allows InfiniFilter to expand while maintaining stable operation costs, memory footprint, and false positive rate.

Section 4.1 describes the Basic InfiniFilter, which supports a finite number of efficient expansions. Section 4.2 shows how to support infinite expansions by chaining multiple basic InfiniFilters together. Section 4.3 shows how to trade slightly more memory to fully stabilize the false positive rate in the worst case.

We build InfiniFilter on top of Quotient Filter. We chose Quotient filter because it resolves collisions without relying on entries' fingerprints (i.e., in contrast to, say, Cuckoo or Morton filters [11, 36]). Hence, it is straightforward to extract the hash associated with an entry and use it to remap the entry to an expanded version of the filter.

4.1 The Basic InfiniFilter

The Basic InfiniFilter is initialized as a quotient filter with S slots (S is a power of 2) and a fingerprint size of F bits. Figure 6 illustrates a basic InfiniFilter with four slots. The first three bits in each slot are the usual is_occupied, is_shifted, and is_continuation flags of a quotient filter used to resolve hash collisions. The remaining bits are divided into two fields. The first is a unary age counter, which counts how many expansions ago the given entry was inserted. The second is the fingerprint.

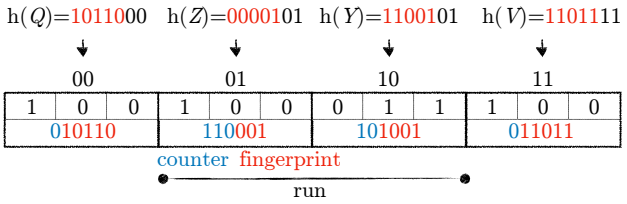

Fig. 6. An example of an InfiniFilter with four slots. For each data entry, there is a fixed-length entry that consists of a unary age counter and a fingerprint.

Together, the counter and fingerprint occupy $F + 1$ bits. The longer the unary counter of a given entry is, the shorter the fingerprint is.

Figure 6 shows an example of an InfiniFilter with 4 slots and an initial fingerprint length of 5 bits. There are four entries across four slots. The full hashes for the original keys, Q, Z, Y, and V are given above the cell that contains a fingerprint for that key. As shown, the four entries at Slots 00 to 11 were inserted zero, two, one, and zero expansions ago, respectively, as indicated by their age counters (i.e., 0, 110, 10, and 0, respectively).

Insertion. A new entry is inserted with a fingerprint comprising F bits and an age counter initialized to 0 to indicate that no expansions have taken place since the entry was inserted. The rest of the insertion procedure is identical to that of a regular quotient filter.

Queries. A query commences as with a regular quotient filter by first finding the run belonging to the target slot. For each entry in this run, the query parses the self-delimiting unary counter to infer how long the fingerprint is. It then checks whether this fingerprint matches the least significant bits of the fingerprint of the key in question. It returns a positive if so, and it continues scanning the run otherwise. If it finishes scanning the run without a match, it returns a negative.

In Figure 6, for example, consider a query to key Y, which maps to a run comprising Slots 01 and 10. While the fingerprints at these slots have different lengths, they both match the fingerprint of key Y causing the filter to return a positive.

Deletes. Similarly to a regular Quotient Filter, InfiniFilter supports deletes, though it requires being more careful to maintain correct semantics. The reason is that fingerprints within a run can have different lengths, and removing a shorter (lower-resolution) fingerprint associated with the wrong key could lead to false negatives later on.

For instance, suppose the user deletes key Y in Figure 6. The hash of key Y matches both fingerprints at this run, so there is a question of which to remove. If we delete the shorter one (i.e., 001), we would get a false negative later when querying for key Z.

To prevent false negatives, InfiniFilter always deletes the longest matching fingerprint (i.e., 1001 in the above example). This guarantees that queries to other keys will not result in false negatives as they will always still match the shorter remaining fingerprint.

Expansion. When the fraction of occupied slots in the filter reaches a threshold of α, an expansion begins. This process first allocates a new InfiniFilter with double the capacity of the existing one. It then iterates over the smaller InfiniFilter from left to right. For each entry, it concatenates its slot address with its fingerprint to derive its original hash, and it uses this hash to reinsert the entry into the new InfiniFilter. Specifically, an entry from Slot i in the former InfiniFilter is placed either at Slot i or at Slot $i + q/2$ of the newer InfiniFilter depending on whether its least significant fingerprint bit is 0 or 1, where q is the number of slots in the new InfiniFilter. The new fingerprint for each entry does not include the former least significant bit as the information contained in this bit is now implicit in the entry's new slot address. Finally, the former InfiniFilter is de-allocated.

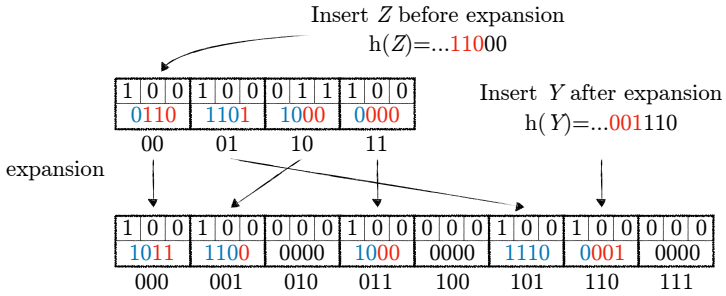

Fig. 7. InfiniFilter doubles in capacity by incrementing every entry's age counter and transferring the least significant bit of its fingerprint to become the most significant bit of its slot address. The figure uses black, blue, and red to illustrate slot addresses, unary age counters, and fingerprints, respectively.

Figure 7 depicts the expansion example from Figure 5 reapplied to an InfiniFilter. For each entry, the age counter is incremented while the least significant bit of its fingerprint becomes the most significant bit of its slot address. Hence, every entry still occupies the same number of bits after expansion. Crucially, even though entries from before the expansion now have shorter fingerprints, newer entries inserted after the expansion are still assigned F bits, the maximum fingerprint length.

Expansion Threshold. The expansion threshold α controls a trade-off. The higher it is, the better the filter's memory utilization is as more of the filter is full. On the other hand, queries and inserts become slower as clusters become longer on average. Our design employs a threshold of 80% by default to strike a reasonable balance.

Insertion Cost. Half of all the entries in InfiniFilter are new and will not have participated in any expansion. A quarter will have participated in one expansion, an eighth in two expansions, and so on. Hence, the amortized insertion cost follows a geometric sum that converges to $\approx 2 \in O(1)$.

Expansion Limit. After F expansions, the oldest entries in the filter, which were inserted before the first expansion took place, run out of fingerprint bits. At this point, we can no longer employ parts of their fingerprint to map them to a larger filter with greater capacity, so the filter cannot continue expanding. Hence, the basic InfiniFilter accommodates at most F expansions, leading to a maximum data size of $S \cdot 2^F$ entries. Section 4.2 shows how to overcome this limitation to continue expanding indefinitely.

Age Distribution. Since InfiniFilter doubles in size when it expands, the distribution of age counters is geometric: there are generally half as many entries with age $i+1$ as there are of age i. Equation 2 approximates the fraction of entries in the filter with age i after X expansions (i.e., $0 \leq i \leq X$).

$$f(i) \lesssim 2^{-i-1} \qquad 0 \leq i \leq X \tag{2}$$

False Positive Rate. Entries of age i have a fingerprint size of $F - i$ bits and thus a false positive probability of $2^{-F+i} \cdot \alpha$. Equation 3 derives the weighted average false positive rate by multiplying this expression with the age distribution in Equation 2.

$$\begin{aligned} \theta = \sum_{i=0}^{X} f(i) \cdot 2^{-F+i} \cdot \alpha &\lesssim (X+2) \cdot 2^{-F-1} \cdot \alpha \\ &\leq (F+2) \cdot 2^{-F-1} \cdot \alpha \end{aligned} \tag{3}$$

As shown in Equation 3, the false positive rate increases linearly with the number of expansions X (and hence logarithmically with the data size). The intuition is that even though there are exponentially fewer older entries in the filter, these entries exhibit exponentially higher false positive rates due to their shorter fingerprints. Hence, each generation of entries contributes equally

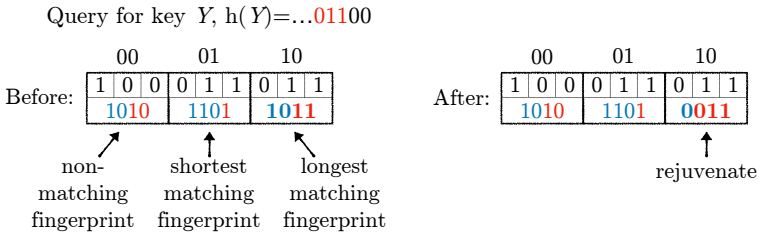

Fig. 8. After a query to an existing key, it is possible to rejuvenate the longest matching fingerprint in the target run to lower the false positive rate.

to the false positive rate. Since the basic InfiniFilter supports up to F expansions, the false positive rate reaches a maximum after F expansions.

Memory. Each slot comprises $F + 1$ bits for the unary counter and fingerprint plus the three usual metadata bits of a standard quotient filter. We divide this by the expansion threshold α in Equation 4 to derive M as the number of bits per entry at the moment before expansion takes place.

$$M = {}^{(F+4)}/_{\alpha} \tag{4}$$

Rejuvenation. The analysis above indicates that the false positive rate of InfiniFilter so far increases logarithmically as the data grows. To slow down and even halt the rate at which the false positive rate increases, InfiniFilter leverages queries to opportunistically rejuvenate (i.e., lengthen) the fingerprints of older entries. The insight is that a query to the filter that returns a positive is typically followed by a lookup to storage to retrieve the original key-value pair. If the key is found, we can rehash it to lengthen its fingerprint. Similarly to deletes, however, rejuvenation carries a risk with respect to correctness. If there are multiple matching fingerprints of different lengths in the run and we lengthen a shorter matching fingerprint, false negatives can occur later on if the fingerprint we lengthened corresponds to a different key. To prevent false negatives, we must lengthen the longest matching fingerprint within the run. Queries to the other keys will still match the shorter remaining fingerprints, thus eliminating the possibility of false negatives.

Figure 8 illustrates a query to key Y, for which the fingerprint resides in a run consisting of three slots. The fingerprints at slots 01 and 10 match key Y's fingerprint, so the filter returns a positive. We assume the original key Y is then retrieved from storage. This allows us to rehash it and lengthen the longest matching fingerprint (at Slot 10) to F bits. Future queries arriving at this run will now, on average, exhibit fewer false positives.

Batch Rejuvenation. In many storage applications, data is occasionally read and reorganized in large batches. This includes garbage-collection in log-structured file systems [77], compaction in log-structured merge-trees [70], and defragmentation in B-trees [42]. Such processes also serve as opportunities to rejuvenate the fingerprints of older entries. For each entry read from storage during such an operation, we can derive its original hash and rejuvenate the longest matching fingerprint within the corresponding run.

Contraction. If many deletes occur and utilization significantly decreases, InfiniFilter contracts. The contraction threshold is ${}^{\alpha}/_{4}$ to ensure that the first delete operation after expansion would not lead to an immediate subsequent contraction. When utilization drops to ${}^{\alpha}/_{4}$, we allocate a new InfiniFilter with half as many slots and iterate over the existing InfiniFilter's entries. An entry from Slot i or Slot $i + {}^{q}/_{2}$ of the larger InfiniFilter is placed at Slot i of the smaller one, and the most significant bit of its slot address is appended as the least significant bit of its fingerprint. To maintain the same slot width, every entry whose age counter is greater than zero is decremented. Otherwise, the most significant bit of its fingerprint is truncated. Figure 9 depicts an example where a delete operation triggers a contraction from eight to four slots.

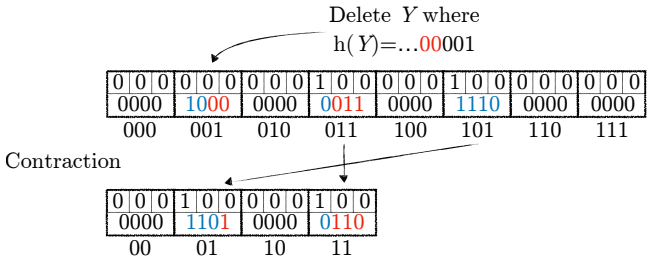

Fig. 9. InfiniFilter contracts by halving the number of slots and transferring the most significant bit of each entry's canonical slot address to become the least significant bit of the entry's fingerprint.

InfiniFilter (IF) type	query/ delete	insert	false positive rate	fingerprint bits / entry	max. expansions
Basic IF	$O(1)$	$O(1)$	$O(2^{-F} \cdot \lg N)$	F	F
Chained IF	$O(\frac{\lg N}{F})$	$O(1)$	$O(2^{-F} \cdot \lg N)$	F	∞
Chained IF & Growing Mem.	$O(\frac{\lg N}{F+\lg \lg N})$	$O(1)$	$O(2^{-F})$	$F + O(\lg \lg N)$	∞

Table 3. The different variants of InfiniFilter offer new and superior cost properties for filter expansion.

Summary. We summarize the properties of the basic InfiniFilter in Row 1 of Table 3. As shown, it matches the Fingerprint Sacrifice method in Table 2 in terms of the maximum number of supported expansions and in terms of the performance of queries/inserts/deletes. At the same time, it scales the false positive rate logarithmically rather than linearly, a tremendous improvement. In the next two sections, we introduce two more variants of InfiniFilter that support infinite expansions and better scale the false positive rate in the worst case.

4.2 Infinite Expansions via Chaining

The basic InfiniFilter from the previous section supports a finite number of expansions. The reason is that, eventually, the oldest entries in the filter run out of fingerprint bits. We refer to such entries as *void entries*. The problem with void entries is that they have no more spare fingerprint bits that can be sacrificed to map them to an expanded filter. We now show how to overcome this limitation by organizing void entries along a chain of InfiniFilters.

The Active InfiniFilter. Figure 10 illustrates the chaining architecture, which consists of multiple basic InfiniFilters as building blocks. While these InfiniFilters have different numbers of slots, they each have the same slot width. Insertions are made into the so-called *Active InfiniFilter*. Once the Active InfiniFilter reaches the expansion threshold α, it expands using the process as described in Section 4.1. As we iterate over the Active InfiniFilter during this expansion, we migrate every void entry that we encounter into a so-called *Secondary InfiniFilter*.

The Secondary InfiniFilter. The Secondary InfiniFilter is smaller than the Active InfiniFilter by a multiplicative factor of 2^{F+1} slots. For every void entry migrated from the Active InfiniFilter into the Secondary InfiniFilter, we employ the most significant F bits of its canonical slot address in the Active InfiniFilter as an F bit fingerprint to be inserted into the Secondary InfiniFilter. The remaining lesser significant bits of the entry's canonical slot address in the Active InfiniFilter are assigned as the entry's canonical slot address in the Secondary Infinifilter.

The Chain. Just before the Active InfiniFilter expands, we expand the Secondary InfiniFilter (also using the expansion algorithm from Section 4.1). Eventually, the oldest entries in the Secondary

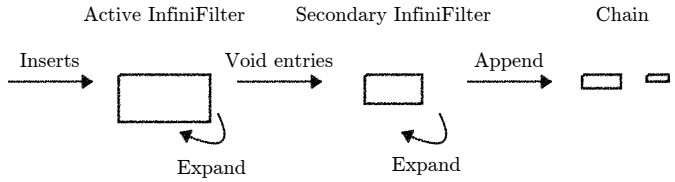

Fig. 10. The Chained InfiniFilter supports indefinite expansion using a short chain of basic InfiniFilters. New insertions go into the Active InfiniFilter. When entries become void, they are transferred into the Secondary InfiniFilter, which is, in turn, appended to the chain of static, older InfiniFilters when it fills up.

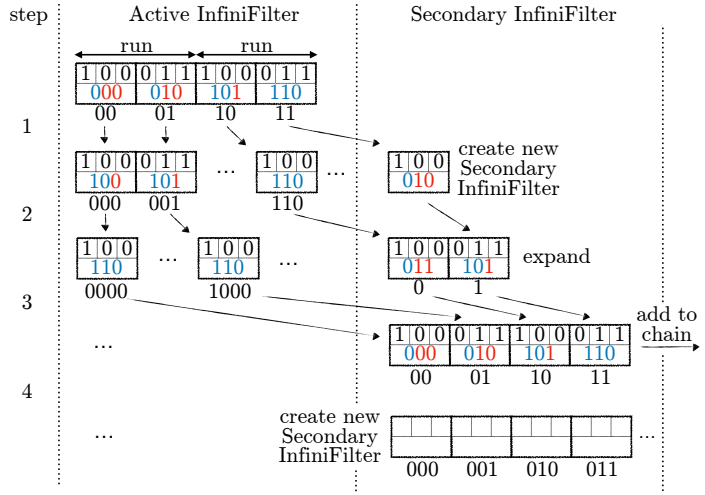

Fig. 11. An example illustrating the migration of entries from the Active InfiniFilter into the Secondary InfiniFilter and finally into the chain across several expansions.

InfiniFilter become void. At this point, the Secondary InfiniFilter becomes static and appended to a chain of static InfiniFilters. A new empty Secondary InfiniFilter is then allocated.

Example. Figure 11 illustrates an example of four expansions. The example commences with an Active InfiniFilter consisting of four slots and 2-bit fingerprints. There is a void entry at Slot 11 that belongs to a run starting at Canonical Slot 10. As the first expansion begins, a new Secondary InfiniFilter is allocated with one slot. We migrate the void entry to the Secondary InfiniFilter during the expansion and employ its former canonical slot address as its fingerprint.

In Step 2, just before the Active Infinifilter expands, we first expand the Secondary InfiniFilter and map its only entry to Canonical Slot 0 based on the most significant bit of its fingerprint. Then, as we expand the Active InfiniFilter, we encounter a void entry in Slot 110. Based on this slot address, we migrate it with fingerprint 11 to Slot 0 of the Secondary InfiniFilter. There are now two entries in the Secondary InfiniFilter mapped to canonical slot 0. The mechanics of the underlying quotient filter resolve this hash collision by storing these entries as a run starting at Slot 0 and comprising two slots.

In Step 3, as in Step 2, we first expand the Secondary InfiniFilter. We then expand the Active InfiniFilters while migrating all void entries into the Secondary InfiniFilter. At this point, the oldest entries in the Secondary InfiniFilter become void, so we seal the Secondary InfiniFilter and append it to the chain. We then allocate a new empty Secondary InfiniFilter as shown in Step 4.

Number of InfiniFilters. After the initial F expansions, the Active InfiniFilter and any InfiniFilter along the chain contains entries spanning $F + 1$ consecutive generations. The number of generations X since initialization is $\lceil \log_2(N) + 1 \rceil$. The total number of InfiniFilters is therefore $c = \lceil {(\log_2(N)+1)}/{(F+1)} \rceil$.

Queries. A query first searches the Active InfiniFilter, then the Secondary InfiniFilter, and then the chain from younger to older InfiniFilters. When it finds a matching fingerprint, it terminates and returns a positive to the user. If it finishes traversing all InfiniFilters with no match, it returns a negative. The worst-case query cost is $O(\lg(N)/F)$ memory accesses. However, as most entries are in the Active InfiniFilter, most positive queries finish after just one access to the Active InfiniFilter.

Note that most often, with an initial fingerprint size of, say, $F = 10$ bits per entry, the Active and Secondary InfiniFilter will support 20 expansions before the chain becomes non-empty. This implies increasing the initial data size by a vast factor of $\approx 2^{20}$. Hence, while the chain is a construction used to guarantee indefinite expansion, it will typically be empty and not influence performance. So in most cases, the chained InfiniFilter only requires one or two cache misses per query, one to the Active InfiniFilter and one to the Secondary InfiniFilter.

Deletes. In Section 4.1, we saw that a delete operation has to remove the entry with the longest matching fingerprint from a run to prevent future false negatives. This principle also holds for the chained InfiniFilter. If we delete a matching entry Y in an older InfiniFilter while there is an entry Z with a matching fingerprint in a newer InfiniFilter, this could result in future false negatives. This could happen if the entry Y corresponds to a different entry for which the hash is different from entry Z's hash only along more significant bits that are not stored as a part of entry Y's fingerprint.

To prevent false negatives, we traverse the different InfiniFilters from largest to smallest. For each InfiniFilters along this traversal, we attempt we apply the delete procedure described in Section 4.1. If we successfully find and remove a matching fingerprint, the procedure terminates. Otherwise, we continue to the next smaller InfiniFilter. This approach removes the entry with the longest matching hash across all InfiniFilters.

The worst-case delete cost is $O(\lg(N)/F)$ memory accesses as we must potentially traverse all InfiniFilters. Since most of the entries are in the Active InfiniFilter, however, a delete is likely to find and remove the target in constant time after just searching the Active InfiniFilter. If a filter in the Chain runs out of entries due to deletes, it is de-allocated.

False Positive Rate. When querying for a non-existing entry, a false positive can occur along any of the InfiniFilters in the chain. To derive the overall false positive rate, one can multiply the number of InfiniFilters c by the false positive rate of an individual InfiniFilter. We provide Equation 5 as a smooth function that approximates the false positive rate as the data size grows. As shown, the false positive rate increases logarithmically. Later in Section 5, we verify this model empirically and show that it is very accurate in practice.

$$\theta \lesssim (\log_2(N) + 2) \cdot 2^{-F-1} \cdot \alpha \tag{5}$$

Rejuvenation Operations. The Chained InfiniFilter also supports rejuvenation operations to lengthen the fingerprints of older entries after a query to an existing entry. If the true positive for this entry occurs in the Active InfiniFilter, the rejuvenation process is identical to the one described in Section 4.1. However, the true positive could also occur in the Secondary InfiniFilter or one of the InfiniFilters along the chain. In this case, after we retrieve the target key from storage, we rehash the key, delete the longest matching fingerprint from the filter where the true positive occurred, and reinsert the lengthened fingerprint into the Active infiniFilter. If a filter in the Chain empties due to rejuvenation operations, it is de-allocated.

Summary. The Chained InfiniFilter supports a logarithmic false positive rate and infinite expansions. Hence, it dominates the Fingerprint Sacrifice method, which supports only F expansions and has a linear false positive rate.

Moreover, the Chained InfiniFilter improves query/delete costs relative to Geometric Chaining by a significant factor of F (i.e., $O(\lg N/F)$ vs. $O(\lg N)$). The trade-off is that InfiniFilter's periodic expansions can slow down insertion throughput by a factor of up to ≈ 2 relative to Geometric

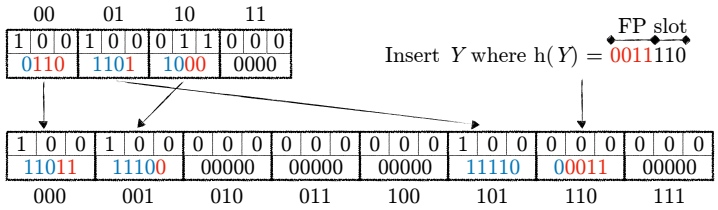

Fig. 12. InfiniFilter can stabilize the false positive rate by increasing fingerprint sizes for newer entries while providing the necessary padding for older entries by lengthening their unary counters.

Chaining. Nevertheless, assuming a typical assignment of, say, 10 to F, an x10 improvement in query/delete throughput is worth an x2 slowdown in insertion throughput for a wide spectrum of applications. In terms of overall system throughput, this trade-off is beneficial whenever queries constitute between 10% to 100% of the workload relative to insertions. Such workloads characterize all the default workloads in YCSB [19] and many applications in practice (e.g., HTAP [62, 81] and Social Graphs [5, 33]). Moreover, in many applications (e.g., storage engines such as HBase [46], Cassandra [3], RocksDB [67], etc.) filters are queried on the critical path of performance and directly impact the latency experienced by users. In contrast, new data is first buffered and inserted into a filter in the background, possibly during idle time, so insertion performance can be less critical to optimize for users' experience.

4.3 Stabilizing the False Positive Rate

The Chained InfiniFilter so far offers a logarithmic false positive rate, and rejuvenation operations can be employed to opportunistically decrease the false positive rate when users query for existing keys. However, the effectiveness of rejuvenation operations depends on the query workload. If only a small fraction of the existing keys are queried for, most fingerprints will not be touched by queries and thereby not get rejuvenated. With a vision towards navigable systems that can span and adapt across a wide spectrum of trade-offs to optimize diverse workloads [47–50], this section shows how to obtain better worst-case guarantees for the false positive rate in exchange for slightly more memory, regardless of the query workload.

The technique we employ is to increase the slot width of the Active InfiniFilter as the data grows to allow storing even longer fingerprints for newer entries. By gradually increasing the average fingerprint length as a function of the data size, the longer fingerprints of newer entries counterbalance the effect of the shortening fingerprints of older entries and guarantee a lower and more stable false positive rate overall.

Fingerprint Growth. Our goal is to assign longer fingerprints to newer entries such that the false positive rate across the filter as a whole converges to a constant with respect to the number of expansions that have taken place. At the same time, we would like to grow the fingerprints at a slow rate to prevent the memory footprint from significantly increasing over time. Inspired by [71], we strike this balance using the reciprocal of square numbers, which produce a convergent series ($\sum_{i=0}^{\infty} i^{-2} = \pi^2/6$). We use this series to decrease the target false positive rate by a factor of $(X+1)^{-2}$ for entries inserted after the X^{th} expansion (belonging to Generation X). To do so, Equation 6 gives the fingerprint length assigned to entries inserted after the X^{th} expansion.

$$\ell(X) = F + \lceil 2 \cdot \log_2(X+1) \rceil \tag{6}$$

As this approach increases the slot width, we increase the unary code of older entries to provide the necessary padding in each slot, as shown in Figure 12.

Memory Footprint. The requisite number of bits per entry is given in Equation 7. It is derived by considering that the number of expansions X is $\log_2(N)$. Furthermore, one bit is needed as a unary counter, three bits are needed for a quotient filter to resolve collisions, and only a fraction of up to α of the filter's slots are used when at full capacity.

$$M = \frac{4 + F + \lceil 2 \cdot \log_2(\log_2(N) + 1) \rceil}{\alpha} \tag{7}$$

Fingerprint Size Distribution. We now turn to derive the false positive rate. To do so, we first reason about the lengths of different fingerprints within an individual InfiniFilter along the chain. Let us consider a filter whose oldest fingerprints were created in generation t (after the t^{th} expansion). Such a filter stores entries inserted from across $\ell(t)$ consecutive generations of entries, as after $\ell(t)$ expansions the oldest entries become void. For such an InfiniFilter, entries inserted at generation $t+i$ will have had a fingerprint of length $\ell(t+i)$ when they were first inserted. However, by the time this InfiniFilter is appended to the chain, $\ell(t) - i$ additional expansions must have taken place, and so entries of this generation must have lost $\ell(t) - i$ bits of their original fingerprints. Hence, Equation 8 derives the fingerprint lengths of entries belonging to Generation $t + i$ for an InfiniFilter created at Generation t.

$$FP_t(i) = \ell(t+i) - (\ell(t) - i) = F + 2 \cdot \log_2(t + i + 1) - \ell(t) + i \tag{8}$$

For an InfiniFilter created at Generation t, Equation 9 denotes θ_t as the overall false positive rate. It is derived by weighting the false positive rates for entries with a given age using Equation 8 by the distribution of different ages in a filter from Equation 2.

$$\theta_t = \sum_{i=0}^{\ell(t)} f(\ell(t) - i) \cdot 2^{-FP_t(i)} \cdot \alpha \lesssim 2^{-F-1} \cdot \alpha \sum_{i=1}^{\ell(t)+1} \frac{1}{(t+i)^2} \tag{9}$$

Constant False Positive Rate. To obtain the overall false positive rate, the left-hand side of Equation 10 sums up the false positive rate for every existing InfiniFilters in the system. The subsequent derivation in Equation 10 shows that the overall false positive rate converges to a constant with respect to the number of expansions that have taken place. The reason is that newer InfiniFilters have longer fingerprints on average and thus a lower false positive rate.

$$\begin{aligned} \theta &\leq 2^{-F-1} \cdot \alpha \cdot \sum_{i=1}^{X+1} \frac{1}{i^2} \lesssim 2^{-F-1} \cdot \alpha \cdot \sum_{i=0}^{\infty} \frac{1}{i^2} \\ &= 2^{-F-1} \cdot \alpha \cdot (\pi^2/6) \lesssim 2^{-F} \cdot \alpha \end{aligned} \tag{10}$$

Faster Queries and Deletes. While we have already achieved our goal of stabilizing the false positive rate with respect to the data size, the approach introduced in this section also asymptotically improves the performance of queries and deletes. The reason is that newer InfiniFilters store entries from across a greater number of consecutive generations as they are assigned longer fingerprints to begin with. This restricts the number of InfiniFilters c to be at most $O(\log_2 N / (F + \log_2 \log_2 N))$. This is a lower expression for query and delete cost than what we were able to obtain for the Chained InfiniFilter in Section 4.2. To prove this formally, one can show by induction that a chain of c InfiniFilters stores entries from at least $c \cdot (F + \log_2 c)$ consecutive generations.

Summary. We summarize the properties of the Chained InfiniFilter with Growing Fingerprints in Row 3 of Table 3. In contrast to the Chained InfiniFilter in Row 2, it offers a constant rather than a logarithmic false positive rate and faster queries/deletes in exchange for a slightly higher memory footprint. Hence, it offers a new attractive design choice for applications that require even stabler performance guarantees as the data grows.

5 EVALUATION

We now turn to evaluate InfiniFilter against several baselines[2].

Platform. Our machine has two Intel Xeon E5-2690v4 (2.6 GHz, 14 cores) with a total of 28 cores and 56 hyper-threads. There are 512GB of RAM, 35MB of L3 cache, 256KB of L2 cache, and 32KB of L1 cache. There are two 960GB SSDs and four 1.8TB HDDs, though we do not use these drives in the experiments. An Ubuntu 18.04.5 LTS operating system is installed.

Baselines. We first compare the Chained InfiniFilter against the Bit Sacrifice and Geometric Chaining methods from Section 3. We do not compare against Linear Chaining as it is dominated by the Geometric Chaining and is therefore non-competitive.

We implemented InfiniFilter and all the other baselines in Java. We chose Java to make our filter implementations compatible with popular key-value stores that heavily use filters, including HBase [4] and Cassandra [3], both of which are written in Java.

Our implementation consists of a Quotient Filter class, which is inherited by each of the baselines as a separate class to provide a means of expansion. Since all baselines share the same Quotient Filter implementation, any differences in their performance arise purely from the expansion strategy rather than implementation idiosyncrasies. We use version 11.0.16 of the Java compiler. We employ xxhash [18] as the hash function for all the baselines.

All baselines are initialized with the same slot widths. Three bits of each slot are employed by the quotient filter as metadata bits to resolve hash collisions. With the Geometric Chaining method and Bit Sacrifice methods, this leaves the remaining bits to be used as fingerprints. For Infinifilter, one additional bit is used as a unary age counter, meaning its fingerprints are initialized with one bit less than the other baselines.

Setup. All experiments involve inserting or querying uniformly randomly distributed eight-byte integer keys generated using the java.util.Random class. We set the capacity threshold at which each of the baselines expands to 80%.

All of the evaluation figures show how different cost metrics evolve as we insert more data into the filter. Between every two adjacent points in any curve in each figure, one expansion occurs, meaning the data size doubles. We consider this as a *phase*. Each phase commences with an expansion and proceeds to fill up the filter to 80% capacity. We measure the average insertion latency for each phase by dividing the duration of the phase by the number of insertions that took place. Hence, our measurements for average insertion latency account for the cost of expansion.

To focus on the worst-case, we measure performance for queries to non-existing keys (i.e., negative queries). Such queries traverse the entire chain of InfiniFilters and thus highlight its worst-case behavior. Moreover, we measure query latency and the false positive rate at the end of each phase, right before the next expansion. At this point, queries and insertions need to traverse the longest possible clusters on average. Furthermore, there are more opportunities for false positives to occur as runs are longer.

At the end of each phase, we also measure memory footprint as the total filter size divided by the number of entries that have been inserted. Hence, our memory measurements account for all memory overheads including the 20% spare capacity in each filter at the end of each phase.

InfiniFilter Offers Superior Cost Balances. Figure 13 Parts (A) to (D) compare the different baselines with an initialization of 16-bit slots. The Bit Sacrifice method only supports 13 expansions until its fingerprints run out of bits. The other baselines can expand indefinitely. This experiment focuses on the Chained InfiniFilter from Section 4.2 with fixed slot widths.

Part (A) focuses on query cost. The Geometric Chaining method exhibits the fastest-growing query costs since a new filter is added to the chain in each expansion, and the entire chain is

[2]The code of InfiniFilter and all baselines is available at https://github.com/nivdayan/FilterLibrary.

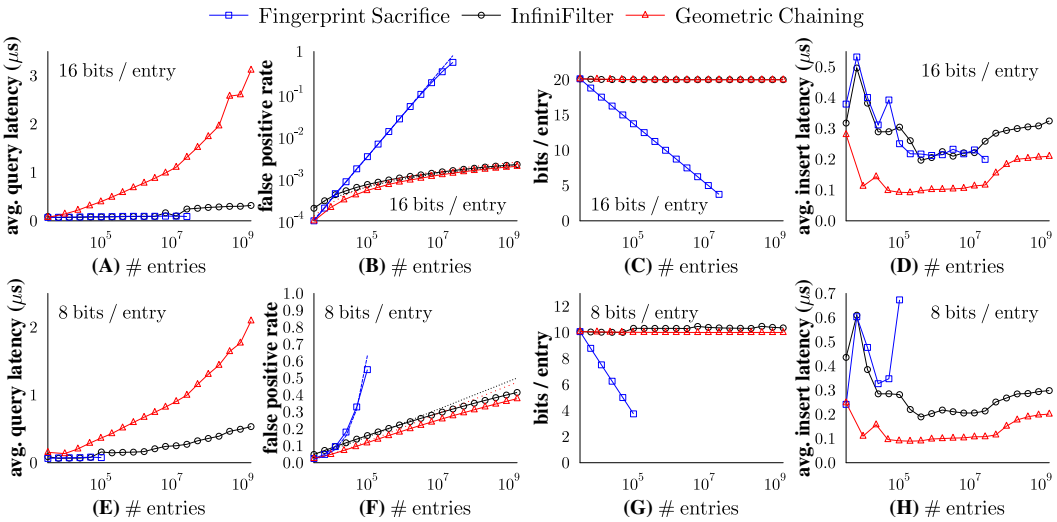

Fig. 13. As the data size grows, Geometric Chaining exhibits a rapidly deteriorating query cost, while the Fingerprint Sacrifice method exhibits a rapidly increasing false positive rate and cannot expand indefinitely. In contrast, InfiniFilter maintains a stabler query cost, false positive rate, memory footprint and insertion cost as the data size grows, all while being able to expand indefinitely.

traversed in each query. The Bit Sacrifice method exhibits stable performance but cannot expand indefinitely. The Chained InfiniFilter maintains modest query overheads that increase slightly towards the end of the experiment as a Secondary Infinifilter is allocated, meaning that two filters are accessed in each query.

Part (B) measures the false positive rate. The Bit Sacrifice method exhibits the fasting increasing false positive rate as one bit from each fingerprint is sacrificed in each expansion. Therefore, the false positive rate doubles in each phase of the experiment. InfiniFilter and Geometric Chaining exhibit logarithmic false positive rates as the data size grows. The dotted lines with matching colors for each of the baselines reflect the analytical cost models (from Equation 5 for the Chained InfiniFilter and from Section 3 for the other two baselines). The model error is proportional to the actual false positive rate, meaning that as the false positive rate grows, the error becomes more noticeable in the figure.

Part (C) measures the memory footprint. The Bit Sacrifice method exhibits decreasing memory overheads as all fingerprints shrink by one bit after each expansion. In contrast, InfiniFilter and the Geometric Chaining method exhibit a stable memory footprint since their slot widths stay fixed as the data grows.

Part (D) focuses on insertion overheads. The Bit Sacrifice method and InfiniFilter exhibit a higher insertion cost by a modest constant factor of ≈ 2 relative to Geometric Chaining. The reason is that unlike the Geometric Chaining method, which simply allocates a new empty filter to expand, the other two methods must also migrate all existing entries into a new filter during expansion. This is also what allows them to support faster queries.

Overall, InfiniFilter dominates the Bit Sacrifice method by better scaling the false positive rate and supporting infinite expansions. Compared to Geometric Chaining, InfiniFilter supports >10x faster queries in exchange for $\approx$ 2x slower insertions.

To show these results hold in general, Figure 13 Parts (E) to (H) repeat the experiment with initialization of 8-bit rather than 16-bit slots. The Bit Sacrifice method in this case runs out of fingerprint bits after five expansions, while the other two methods can continue expanding indefinitely.

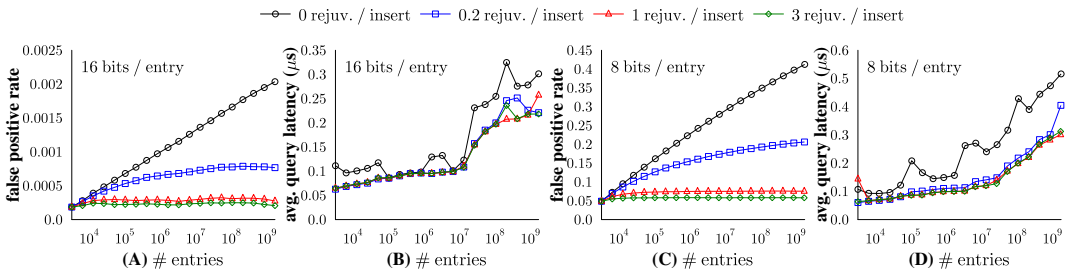

Fig. 14. The more positive queries take place relative to insertions, the more InfiniFilter is able to keep the false positive rate and query costs stable by rejuvenating older fingerprints.

InfiniFilter has a slightly higher false positive rate than Geometric Chaining because it employs some of its memory for unary counters, though it continues to significantly improve on Geometric Chaining in terms of query cost.

Rejuvenation Operations. Figure 14 showcases InfiniFilter's rejuvenation operations. The experiment interleaves uniformly random queries to existing entries (i.e., positive queries) along with insertions of new entries. Each query rejuvenates the longest matching fingerprint in the target run, as described in Sections 4.1 and 4.2. The experiment illustrates four curves, each with a different ratio between the number of queries to the number of insertions. A ratio of zero means there are no queries, 0.2 means we issue one query for every five writes, etc. At the end of each phase, we issue queries to non-existing entries to measure the false positive rate. Parts (A) and (B) of the figure focus on an initialization with 16-bit slots while Parts (C) and (D) employ 8-bit slots. As this experiment only affects query costs and the false positive rate, we do not illustrate memory or insertion overheads as they are the same as in Figure 13.

Parts (A) and (C) of Figure 14 show that with a higher proportion of positive queries in the workload, the false positive rate stays lower and more stable. The reason is that the positive queries rejuvenate fingerprints and keep them longer on average. Meanwhile, Parts (B) and (D) of Figure 14 show that a higher proportion of positive queries also keeps query costs lower and more stable. The reason is that rejuvenation operations migrate older entries from the secondary InfiniFilter or from the chain into the Active InfiniFilter. This reduces the cluster lengths that queries must traverse in InfiniFilters containing older entries. Overall, these experiments establish that even a small proportion of positive queries in an application workload can effectively prevent the false positive rate and query costs from deteriorating as the data grows.

Growing Fingerprints. Rejuvenation operations only help stabilize the false positive rate if queries to existing entries are evenly distributed in the data set. To stabilize the false positive rate as the data grows regardless of the query distribution, Section 4.3 proposes to increase the lengths of fingerprints of newer entries as a function of the data size. Figure 15 showcases this technique using the curves labeled *Growing Mem.* and compares it to the version of InfiniFilter from Section 4.2 with fixed slot widths. As a baseline, we also add Geometric Chaining with Growing Memory (GCGM) from [71], whose properties are summarized in Row 3 of Table 2. In this experiment, there are no positive queries and thus no rejuvenation operations. All baselines are initialized with 8-bit slots.

Part (A) of Figure 15 shows that InfiniFilter with growing fingerprints completely stabilizes the false positive rate. The price is a slowly increasing memory footprint, as shown in Part (B). The dotted lines with matching colors in Part (A) for each of the baselines reflect the analytical cost models from Equations 5 and 10 for InfiniFilter and from Section 3 for GCGM. While GCGM offers a similar false positive rate and memory footprint to Infinifilter with growing fingerprints, it is non-competitive in terms of its query cost, as shown in Part (C). Part (C) also demonstrates that

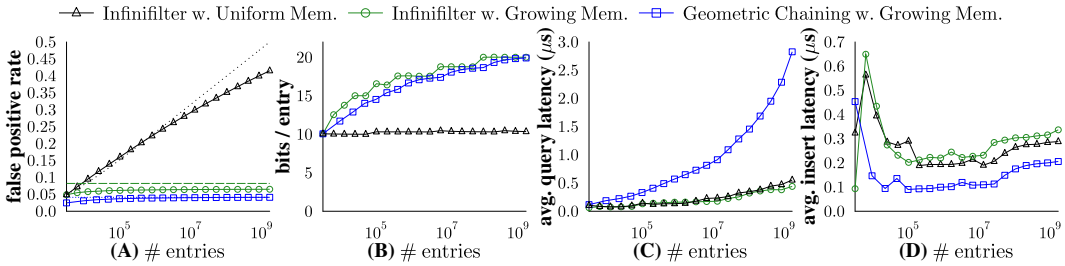

Fig. 15. Increasing the fingerprint lengths of newer entries inserted into the filter allows to stabilize the false positive rate even in the absence of many positive queries and rejuvenation operations.

query overheads stay more modest with InfiniFilter with growing fingerprints as this technique slows down the rate at which the chain of infiniFilter grows. This improves query costs toward the end of the experiment. Part (D) demonstrates that growing fingerprint lengths does not significantly impact insertion performance in the context of InfiniFilter.

Comparison to Static Baselines. We now compare InfiniFilter to three static filters with a fixed capacity pre-allocated in advance: a regular Quotient filter [7], a Bloom filter [9], and a Cuckoo filter [36]. This goal is to compare InfiniFilter to different instances employing the Pre-Allocation method from Section 3. Moreover, these experiments illuminate the effects of choosing a Quotient filter as the base method for InfiniFilter rather than other filter designs. Each of the baselines is initialized with 16 bits per entry, and we use the Chained InfiniFilter variant from Section 4.3 with growing fingerprints. The Cuckoo filter has four slots per bucket. The Bloom filter employs 11 hash functions, which is the optimal number given 16 bits per entry. This experiment measures performance in a finer resolution than before to show how InfiniFilter behaves across the board rather than only when it is just about to expand.

Figure 16 Part (A) compares the overall memory footprint across the baselines as we insert 10^9 data entries from scratch. The static filters exhibit a high memory footprint from the onset as they are pre-allocated to accommodate the maximum data size. Furthermore, they are unable to accommodate data growth beyond their pre-allocated capacity ($\approx 10^8$ entries). In contrast, InfiniFilter scales the memory footprint in proportion to the data size by expanding gradually. Hence, it requires less memory upfront and supports indefinite growth.

Figure 16 Part (B) measures latency for negative queries. InfiniFilter is initially the fastest baseline as it is smaller than the others and so it fully fits into the L3 cache. As the data grows, however, it outgrows the cache, and a secondary InfiniFilter is created. This results in a slowdown. The static quotient filter is initially the second fastest baseline as most slots are empty, so it only checks on average one "is_occupied" flag per query before encountering a zero and terminating. As it fills up, however, cluster lengths grow and query performance deteriorates. Bloom filter is initially the third slowest baseline. It also initially checks one bit on average before encountering a zero and terminating. As it fills up, however, the percentage of ones in the filter grows to $\approx$ 50%, and so on average, two bits are checked before encountering a zero and terminating. Hence, latency approximately doubles as the data size grows. The reason the Bloom filter is initially slower than the quotient filter is due to modulo operations to obtain the target bit from a hash value. We employ modulo operations since a Bloom filter's size is not generally a power of 2. The static Cuckoo filter exhibits stable performance across the experiment as it always searches eight fingerprint slots across two random bucket locations, resulting in two cache misses. Overall, InfiniFilter offers similar, if not better, query performance relative to its static counterparts, while also supporting gradual expansion and indefinite growth.

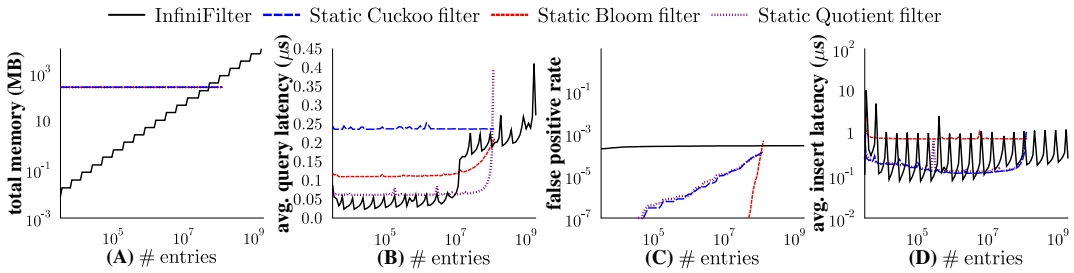

Fig. 16. Unlike static filters, InfiniFilter supports indefinite growth while (A) requiring less memory upfront, (B) having a competitive query cost, (C) preserving a stable false positive rate, and (D) paying a moderate toll in terms of insertion cost.

Figure 16 Part (C) shows that the false positive rate with a Bloom filter is initially lowest as the probability of all hash functions hitting ones is infinitesimal when the Bloom filter is nearly empty. As more data is inserted, however, its false positive rate exceeds all other baselines as predicted by Equation 1. The false positive rate for the static Cuckoo and Quotient filters grows in proportion to the number of entries until they fill up and can no longer expand. In contrast, InfiniFilter provides and preserves a stable and predictable false positive rate that is on par with the other baselines' ultimate false positive rate.

Figure 16 Part (D) shows that the Bloom filter exhibits the slowest insertion speed as each insertion entails one cache miss for each of the 11 hash functions. The cuckoo and Quotient filters exhibit similar performance across the board as each insertion usually only entails one cache miss. However, their performance deteriorates as the data grows and larger clusters need to be accessed in the Quotient filter and more swapping happens across the cuckoo filters' buckets. InfiniFilter, in contrast, exhibits fluctuating insertion performance due to its expansion operations. This is the core trade-off of InfiniFilter relative to the pre-allocation method. In summary, compared to the pre-allocated baselines, InfiniFilter exhibits slightly slower insertion throughout. In exchange, it requires less memory upfront and supports indefinite data growth while maintaining similar query performance and false positive rate.

6 RELATED WORK

Bloom Filters. Numerous Bloom filter variants have been proposed and surveyed [12, 63, 85]. They support counting [10, 37, 79], vectorization [61, 74], deletes [78], more efficient hashing [31, 51], and better cache locality [13, 28, 58, 60, 75]. While Bloom filters have not been shown to support efficient expansion, it is possible to accommodate dynamic data sets with them using Linear Chaining [44, 45], Geometric Chaining [1, 90], or by retrieving a fingerprint for each entry from storage [89]. Section 3 discusses these different options.

Another option is to compress a Bloom filter to allow its physical size to grow along with the data size. However, this would require compressing and decompressing parts of the filter for each query and insertion, leading to higher computational overheads. Existing work on compressed Bloom filters [68] rather aims to reduce network traffic when transmitting bloom filters across a network rather than to support expansions.

Filters as Hash Tables of Fingerprints. While we implement InfiniFilter on top of Quotient Filter [7], there exist various other filters that store a fingerprint for each entry within a compact hash table. The Vector and Counting Quotient Filters employ nested buckets and traverse them quickly using SIMD operations [72, 73]. Cuckoo filter resolves hash collisions using partial-key cuckoo hashing [36]. Morton Filter is a Cuckoo filter variant that biases insertions to one bucket to improve query cost [11]. Vacuum Filter is another Cuckoo filter variant that maps both candidate

buckets for each entry into the same chunk and therefore supports better cache locality and more flexible sizing (the number of buckets need not be a power of two) [87]. Other approaches [64, 91] form a conceptual hash ring of buckets and thus supports elastic expansion, though query, delete and insertion costs all become $O(\log_2 N)$ as a search tree has to be searched to find a given entry's bucket. The Crate Filter employs larger hash buckets and evicts overflowing entries to a smaller spare hash table [8]. Prefix Filter is a variant of the Crate Filter that evicts the largest fingerprint to help queries avoid searching the spare hash table [35]. TinySet adapts the bucket encoding based on the load targeting the bucket [34]. The Xor and Ribbon filters improve memory utilization in exchange for a higher construction cost [30, 43]. Across many of these filters, it is possible to employ the Fingerprint Sacrifice method, discussed in Section 3, to expand. However, this causes the false positive rate to rapidly increase. Integrating InfiniFilter with these filters to combine their nuanced properties with efficient expandability can make for intriguing future work.

Theoretical Algorithms and Bounds. Pagh, Segev, and Wieder prove the following lower bound: If we initialize a filter to constant capacity and expand it to contain N keys, the filter must at some point use at least $\log(\log(N))$ bits per key in addition to what is required for a filter with a fixed capacity of N keys [71].

Pagh, Segev, and Wieder also describe two expandable filters [71]: The first uses Geometric Chaining (see Section 3) with polynomially decreasing false positive rates assigned to larger filters along the chain as described in Section 3. This asymptotically improves the approach of [1] which uses geometrically decreasing false positive rates. This structure's space overhead matches the lower bound up to a constant factor, but queries must search $\log(N)$ filters.

The second data structure is a hash table of fingerprints that duplicates void entries across both candidate buckets in each expansion. This construction exhibits constant time queries and constant amortized insertions. However, it assumes an upper bound of U on the number of keys to achieve an overhead of roughly $O(\log\log U)$ bits per key. This is close to the lower bound when $\log\log U$ is close to $\log\log N$, which may be reasonable in some settings (e.g. when keys come from a finite set of size U and the set N is not too small). To support deletes, this structure encodes a binary age counter and a deletion flag for every fingerprint, and it employs an auxiliary dictionary to disambiguate entries with matching fingerprints. These components inflate space costs by a significant constant factor. Also, having to search both the filter and dictionary doubles query cost.

An asymptotically better filter was later presented by Liu, Yin, and Yu [59]. If N and U are polynomially related, this data structure achieves constant time per operation with high probability and a more modest space overhead of $\log\log N + O(\log\log\log N)$ bits per key, which matches the leading term of the lower bound. However, this structure has not been shown to support deletes.

The constructions of [59, 71] have to our knowledge never been implemented, but it is of course interesting to compare their theoretical properties to those of InfiniFilter. First, these structures either require a hard limit U on the number of keys or use $\Omega(\log N)$ time to answer queries. Analyzing these constant time filters for $N > U$, we find that either the space complexity or the false positive rate increases significantly, so the bound on N is necessary.

In contrast, the chained InfiniFilter from Section 4.3 meets the space lower bound of [71] even if N is unbounded while supporting a constant false positive rate, queries and deletes in $O(\frac{\log N}{F+\log\log N})$ time, and insertions in constant amortized time. Furthermore, InfiniFilter's novel deletion algorithm, which removes the longest matching fingerprint within a bucket, removes the need for additional machinery to support deletes.

Range Filters. Recently, several range filters have been proposed to allow filtering range queries [41, 52, 65, 86, 93]. Applying design elements from InfiniFilter to allow to efficiently expand range filters is an intriguing future direction of work.

Learning from Negative Queries. Recent approaches have been devised to learn from commonly issued negative queries (to non-existing keys) to reduce the false positive rate [29, 56, 69]. Integrating such techniques with InfiniFilter is an intriguing direction.

7 CONCLUSION

This paper introduces InfiniFilter, a novel method for expanding set-membership filters as the size of the data that they represent grows. InfiniFilter is a hash table of fingerprints that expands by doubling in size and sacrificing a fingerprint from each entry to map it to the expanded capacity. It employs a novel entry format that allows setting longer fingerprints to newer entries to stabilize the false positive rate, and it supports fast insertions/deletes/queries by virtue of using a unified hash table. It employs chaining as a technique to guarantee indefinite expansion without significantly increasing the cost of queries or deletes. It also employs a novel deletion algorithm that targets the longest matching fingerprint within a target slot to prevent false negatives. InfiniFilter can rejuvenate the fingerprints of older entries opportunistically during queries to existing entries, or it can assign increasing fingerprint sizes to newer entries to stabilize the false positive rate regardless of the query distribution. Overall, InfiniFilter scales the cost per operation, the false positive rate, and the memory footprint better than any existing method while supporting expansion up to an unbounded universe size.

ACKNOWLEDGEMENT

Ioana Bercea and Rasmus Pagh are part of BARC, supported by the VILLUM Foundation grant 16582. Pedro Reviriego is supported by the ACHILLES project PID2019-104207RB-I00 and the ENTRUDIT project TED2021-130118B-I00 funded by the Spanish Agencia Estatal de Investigación (AEI) 10.13039/501100011033 and by the Madrid Community research project TAPIR-CM grant no. P2018/TCS-4496. We thank the Reviewers for their invaluable feedback. We also thank Miguel González Sáiz for help with the implementation. Lastly, we thank Geffen Huberman for coining InfiniFilter.

REFERENCES

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom Filters. *Inform. Process. Lett.* (2007).
[2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. *SOSP* (2009).
[3] Apache. 2023. Cassandra. *http://cassandra.apache.org* (2023).
[4] Apache. 2023. HBase. *http://hbase.apache.org/* (2023).
[5] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a Database Benchmark Based on the Facebook Social Graph. *SIGMOD* (2013).
[6] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. 2014. Memory-Efficient Hash Joins. *VLDB* (2014).
[7] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *PVLDB* (2012).
[8] Ioana O Bercea and Guy Even. 2020. Fully-Dynamic Space-Efficient Dictionaries and Filters with Constant Number of Memory Accesses. *SWAT*.
[9] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM* 13, 7 (1970), 422–426.
[10] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An Improved Construction for Counting Bloom Filters. In *ESA*.
[11] Alex D Breslow and Nuwan S Jayasena. 2018. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, compression, and decoupled logical sparsity. In *VLDB*.
[12] Andrei Z. Broder and Michael Mitzenmacher. 2002. Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1 (2002), 636–646.

[13] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD Bufferpool Extensions for Database Systems. *PVLDB* (2010).

[14] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. 1978. Exact and Approximate Membership Testers. In *STOC*.

[15] Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin Hood Hashing. In *FOCS*.

[16] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. *SIGMOD* (2018).

[17] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. 2017. The Dynamic Cuckoo Filter. In *ICNP*.

[18] Yann Collet. 2023. XXHash. *https://github.com/Cyan4973/xxHash* (2023).

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. *SoCC* (2010).

[20] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. *SIGMOD* (2017).

[21] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *TODS* 43, 4 (2018), 16:1–16:48.

[22] 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. *SIGMOD* (2018).

[23] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *SIGMOD*.

[24] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD*.

[25] Niv Dayan, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Ofer Frishman, Evgeni Ginzburg, Igal Maly, et al. 2021. The End of Moore's Law and the Rise of the Data Processor. *VLDB* (2021).

[26] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *PVLDB* (2022).

[27] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-Value Store. *PVLDB* (2010).

[28] Biplob Debnath, Sudipta Sengupta, Jin Li, David J Lilja, and David HC Du. 2011. BloomFlash: Bloom filter on Flash-Based Storage. In *ICDCS*.

[29] Kyle Deeds, Brian Hentschel, and Stratos Idreos. 2020. Stacked filters: learning to filter by structure. *PVLDB* (2020).

[30] Peter C Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. 2022. Fast Succinct Retrieval and Approximate Membership Using Ribbon. *SEA* (2022).

[31] Peter C. Dillinger and Panagiotis Manolios. 2004. Bloom Filters in Probabilistic Verification. *FMCAD* (2004).

[32] Peter C. Dillinger and Panagiotis Pete Manolios. 2009. Fast, All-Purpose State Storage. In *SPIN*.

[33] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. *CIDR* (2017).

[34] Gil Einziger and Roy Friedman. 2017. TinySet—An Access Efficient Self Adjusting Bloom Filter Construction. *IEEE ACM Trans Netw* (2017).

[35] Tomer Even, Guy Even, and Adam Morrison. 2022. Prefix Filter: Practically and Theoretically Better Than Bloom. In *VLDB*.

[36] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. *CoNEXT* (2014).

[37] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE ACM Trans Netw* (2000).

[38] Shahabeddin Geravand and Mahmood Ahmadi. 2013. Bloom filter Applications in Network Security: A State-of-the-Art Survey. *Comput Netw* (2013).

[39] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. 2017. BitFunnel: Revisiting Signatures for Search. In *SIGIR*.

[40] Google. 2023. LevelDB. *https://github.com/google/leveldb/* (2023).

[41] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. 2014. Approximate Range Emptiness in Constant Time and Optimal Space. In *SODA*.

[42] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends Databases* 3, 4 (2011), 203–402.

[43] Thomas Mueller Graf and Daniel Lemire. 2020. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *JEA* (2020).

[44] Deke Guo, Jie Wu, Honghui Chen, and Xueshan Luo. 2006. Theory and Network Applications of Dynamic Bloom Filters. In *INFOCOM*.

[45] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. 2009. The Dynamic Bloom Filters. *IEEE Trans Knowl Data Eng* (2009).

[46] HBase. 2013. Online reference. *http://hbase.apache.org/* (2013).
[47] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. 2019. Learning Key-Value Store Design. *arXiv preprint arXiv:1907.05443* (2019).
[48] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *CIDR*.
[49] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE DEBULL* (2018).
[50] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike Kester, Niv Dayan, Demi Guo, Minseo Kang, and Yiyou Sun. 2019. Learning Data Structure Alchemy. *IEEE DEBULL* (2019).
[51] Adam Kirsch and Michael Mitzenmacher. 2008. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Struct Algorithms* (2008).
[52] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *SIGMOD*.
[53] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *PVLDB* (2018).
[54] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut Palm: Static and Streaming Data Series Exploration Now in your Palm. In *SIGMOD*.
[55] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut: Sortable Summarizations for Scalable Indexes over Static and Streaming Data Series. *VLDBJ* (2019).
[56] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. *SIGMOD* (2018).
[57] Avinash Lakshman and Prashant Malik. 2010. Cassandra - A Decentralized Structured Storage System. *SIGOPS Op. Sys. Rev.* (2010).
[58] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. 2019. Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput. In *VLDB*.
[59] Mingmou Liu, Yitong Yin, and Huacheng Yu. 2020. Succinct Filters for Sets of Unknown Sizes. In *ICALP*.
[60] Guanlin Lu, Biplob Debnath, and David H. C. Du. 2011. A Forest-Structured Bloom Filter with Flash Memory. *MSST* (2011).
[61] Jianyuan Lu, Ying Wan, Yang Li, Chuwen Zhang, Huichen Dai, Yi Wang, Gong Zhang, and Bin Liu. 2018. Ultra-Fast Bloom Filters Using SIMD Techniques. *TPDS* (2018).
[62] Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. 2019. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *EDBT*.
[63] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing Bloom filter: Challenges, solutions, and comparisons. *IEEE Commun. Surv. Tutor.* (2018).
[64] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard TB Ma, Xueshan Luo, and Bangbang Ren. 2019. The Consistent Cuckoo Filter. In *INFOCOM*.
[65] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *SIGMOD*.
[66] Meta. 2023. MyRocks. *http://myrocks.io/* (2023).
[67] Meta. 2023. RocksDB. *https://github.com/facebook/rocksdb* (2023).
[68] Michael Mitzenmacher. 2002. Compressed Bloom Filters. *IEEE ACM Trans Netw* (2002).
[69] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2018. Adaptive cuckoo filters. In *SIAM ALENEX*.
[70] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* (1996).
[71] Rasmus Pagh, Gil Segev, and Udi Wieder. 2013. How to Approximate a Set Without Knowing its Size in Advance. In *FOCS*.
[72] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *SIGMOD*.
[73] Prashant Pandey, Alex Conway, Joe Durie, Michael A Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. In *SIGMOD*.
[74] Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. *DAMON* (2014).
[75] Felix Putze, Peter Sanders, and Johannes Singler. 2010. Cache-, Hash-, and Space-Efficient Bloom Filters. *JEA* (2010).
[76] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *PVLDB* (2017).

[77] Mendel Rosenblum and John K Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *TOCS* (1992).
[78] Christian Esteve Rothenberg, Carlos Macapuna, Fabio Verdi, and Mauricio Magalhaes. 2010. The Deletable Bloom Filter: a New Member of the Bloom Family. *IEEE Commun. Lett* (2010).
[79] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. 2013. The Variable-Increment Counting Bloom Filter. *IEEE ACM Trans Netw* (2013).
[80] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM Design Space and its Read Optimizations. *ICDE* (2023).
[81] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. *SIGMOD* (2012).
[82] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *ASPLOS*.
[83] Speedb. 2023. Speedb. *https://github.com/speedb-io/speedb* (2023).
[84] V Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. 2016. Aerospike: Architecture of a Real-Time Operational DBMS. *VLDB* (2016).
[85] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2012. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Commun. Surv. Tutor* (2012).
[86] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a Learning-Enhanced Range Filter. *PVLDB* (2022).
[87] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. 2019. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. In *VLDB*.
[88] Robert Williger and Tobias Maier. 2019. *Concurrent Dynamic Quotient Filters: Packing Fingerprints into Atomics*. Ph.D. Dissertation. Karlsruher Institut für Technologie (KIT).
[89] 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* (2021).
[90] Kun Xie, Yinghua Min, Dafang Zhang, Jigang Wen, and Gaogang Xie. 2007. A Scalable Bloom Filter for Membership Queries. In *GLOBECOM*.
[91] 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* (2022).
[92] Fan Zhang, Hanhua Chen, Hai Jin, and Pedro Reviriego. 2021. The Logarithmic Dynamic Cuckoo Filter. In *ICDE*.
[93] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. *SIGMOD* (2018).

Received October 2022; revised January 2023; accepted February 2023