Aeris Filter: A Strongly and Monotonically Adaptive Range Filter

YUVARAJ CHESETTI, Northeastern University, USA
NAVID ESLAMI, University of Toronto, Canada
HUANCHEN ZHANG, Tsinghua University, China
NIV DAYAN, University of Toronto, Canada
PRASHANT PANDEY, Northeastern University, USA

Range filters are probabilistic data structures used to efficiently perform range emptiness queries, with applications in databases, big data analytics, and key-value stores. Modern range filters are compact and can guarantee a bounded false positive rate irrespective of the spatial skew in queries. However, existing range filters are still susceptible to temporal skew: in skewed workloads where a few queries are repeated disproportionately more often, the false positive rate of a range filter may be unbounded.

We introduce the Aeris filter, an adaptive expandable range filter that guarantees a robust false positive rate irrespective of spatial or temporal skew. The Aeris filter achieves this by dynamically resolving and adapting to false positives. More specifically, the Aeris filter is monotonic adaptive, i.e., it never forgets a previously encountered false positive. The Aeris filter introduces a novel encoding scheme to implement adaptivity in a range filter with no additional space or operational overhead. Furthermore, the Aeris filter deamortizes the I/O cost to expand monotonic adaptive filters by utilizing on-disk adaptivity structures, resulting in fewer system disruptions.

Experimental results demonstrate that the Aeris filter achieves up to a 10× reduction in false positive rates on skewed query distributions compared to other non-adaptive range filters. When integrated into a database, the Aeris filter delivers 1.5 − 8× higher throughput for adversarial workloads, and is able to deliver this high throughput using a cache of smaller size. The Aeris filter also reduces expansion overhead by up to 3× compared to the Memento filter, a spatially-robust expandable range filter. These improvements ensure scalable, efficient, and adaptive range query handling in dynamic environments.

CCS Concepts: • **Theory of computation → Sketching and sampling**; **Data structures design and analysis**; **Bloom filters and hashing**; • **Information systems → Unidimensional range search**.

Additional Key Words and Phrases: Range Filter; Adaptive; Dictionary data structure; Databases

ACM Reference Format:
Yuvaraj Chesetti, Navid Eslami, Huanchen Zhang, Niv Dayan, and Prashant Pandey. 2026. Aeris Filter: A Strongly and Monotonically Adaptive Range Filter . *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 7 (February 2026), 26 pages. https://doi.org/10.1145/3786621

1 Introduction

A filter is a compact data structure that approximately represents a set of keys and supports querying whether a given key exists or not. Traditional filters like Bloom [5], Quotient [41], and Cuckoo filters [23] only support point queries (i.e., for the existence of one key at a time). Recent ***range filters*** [9, 14, 22, 28, 34, 47, 51] extend this capability to efficiently identify whether a given range of

Authors' Contact Information: Yuvaraj Chesetti, Northeastern University, Boston, MA, USA, chesetti.y@northeastern.edu; Navid Eslami, University of Toronto, Toronto, ON, Canada, navideslami@cs.toronto.edu; Huanchen Zhang, Tsinghua University, Beijing, China, huanchen@tsinghua.edu.cn; Niv Dayan, University of Toronto, Toronto, ON, Canada, nivdayan@cs.toronto.edu; Prashant Pandey, Northeastern University, Boston, MA, USA, p.pandey@northeastern.edu.

ACM 2836-6573/2026/2-ART7
https://doi.org/10.1145/3786621

keys is empty or nonempty. Similarly to traditional filters, a range filter cannot return a false negative (i.e., reporting that a nonempty range is empty). However, it may return a false positive (i.e., that an empty range is nonempty). Formally, a range filter represents a key set S from a universe U and answers *range emptiness queries* of the form $[a,b] \cap S = \emptyset$ while striving to maintain a bounded false positive rate (FPR) ϵ [26].

As a range filter is much smaller than the full set of keys that it represents, it can be stored at a higher layer of the memory hierarchy than the full data set, which typically resides on a storage device or over a network. As such, a range filter obviates redundant computations, storage I/Os and/or network hops over parts of the data that do not contain the keys being searched for. Range filters have been an active area of research over the past decade and are finding applications in social networks [13], replication in distributed key-value stores [46], search engines [25], databases [32, 42], time series [27, 29], scientific spatial models [50], bioinformatics [10], etc.

Existing range filters offer weak guarantees. With most existing range filters [9, 24, 28, 38, 47, 51], the FPR varies significantly depending on the degree of spatial skew in the data set and queries. The reason is that such filters effectively store the keys' prefixes while truncating their least significant bits to save space. Hence, queries to empty ranges with end points picked from the same key distribution tend to collide with the prefixes of existing keys. In contrast, a *robust* range filter guarantees a bounded FPR ϵ independently of spatial skew. A well-known lower bound states that a spatially robust range filter over intervals of length at most R keys must use at least $\Omega(\log(R/\epsilon))$ bits per key [26]. To date, three spatially robust range filters have been proposed [14, 22, 34].

However, the existing spatially robust range filters are still not robust to temporal skew. For example, consider monitoring applications that employ filters to identify a given subset of events for further processing. Such applications include packet tracking in routers [36], vehicle or people tracking in smart cities [8, 31], and event tracking in industrial distributed monitoring [6]. In such applications, an event with an attribute within a given range may occur repeatedly within a short time span, potentially leading to repeated false positives and thus multiple redundant system checks. Worse, an adversary can identify which queries lead to false positives as such queries tend to incur higher latency. Such an adversary can then issue these queries at will to trigger I/Os and/or pollute the cache [44]. The underlying cause for this vulnerability is that existing range filters are constructed only with respect to the dataset but not with respect to the query workload. Hence, the observed FPR can exceed ϵ and even approach 1.

Adaptive filters. Adaptive filters are a new family of filters that modify their internal structure in response to queries to prevent false positives from repeating [3, 30, 33, 36, 43, 48]. For example, the recent AdaptiveQF [48] stores a fingerprint (i.e., a hash digest) for each key in a compact hash table [41]. In response to a false positive, it extends the fingerprint of the key that the query collided with to prevent the same query from causing a false positive in the future. While adaptive filters offer robustness to temporally skewed queries, they exhibit two shortcomings. (1) Existing adaptive filters only support point queries. This leaves applications relying on range filters vulnerable to temporal skew. (2) Adaptations add information into the filter, eventually filling it up and causing it to have to expand. Existing adaptive filters do this by scanning the raw dataset and rebuilding the entire filter from scratch. Expansion is therefore currently an expensive process. An adaptive filter must be able to access the original keys over which a false positive occurred to allow adding information to the filter to prevent the false positive from recurring. This is typically done using a disk-resident *reverse map* that maps each fingerprint in the filter to the full key.

Research goal. Is it possible to design a range filter that simultaneously guarantees (1) a robust FPR with respect to any spatial and/or temporal skew in queries, (2) efficient expandability without scanning the original data, and (3) fast operations?

Most existing range filters are unable to achieve these goals at the same time. For example, SuRF [51] constructs a succinct truncated trie over the key set. However, it cannot adapt to correct false positives since the trie is static. Many other range filters hash each key to one or more bits in a bitmap [9, 14, 24, 28, 28, 34, 38], setting them from 0s to 1s. As multiple keys may map to each bit, there is no way to disambiguate which bits correspond to which keys. Hence, there is no natural way to adapt the filter without introducing the possibility of false negatives [35].

Insights and challenges. The recent Memento filter [22] holds promise with respect to our research goal as it is the only range filter to date that supports spatial robustness, fast queries, and fast deletes and updates. It is also able to expand without rereading the original set of keys using recent techniques on filter expansion [16, 17]. If we could also make the Memento filter temporally robust, we would achieve our research goal. However, achieving temporal robustness, i.e., guaranteeing bounded false positive rate for any arbitrary sequence of queries is non-trivial.

The Memento filter partitions the universe from which keys are drawn and stores a fingerprint within a compact hash table corresponding to every nonempty partition. Alongside each fingerprint, it stores a sorted list of suffixes for all keys within that partition. Similarly to the AdaptiveQF, this design is appropriate for the elimination of recurring false positives by extending fingerprints. However, extending fingerprints in the Memento filter is challenging. Due to hash collisions, a fingerprint may correspond to multiple keys across multiple partitions. This introduces the challenge of efficiently splitting and merging entries within the filter without introducing false negatives.

This paper. We present the ***Aeris filter***, an adaptive range filter that provides fast operations and a spatially and temporally robust FPR for any possible dataset and query workload. Built on top of the Memento filter [22], the Aeris filter introduces a novel internal encoding scheme that allows fingerprints to be extended when false positives occur. We show how to split and merge suffixes within a partition without introducing false negatives while maintaining high performance.

The Aeris filter exhibits the property of *monotonic adaptivity*, meaning that a false positive that occurs once can never repeat [48]. As a result, more information has to be added to the filter with response to queries, eventually forcing the filter to expand. The Aeris filter expands efficiently without rereading the original set of keys by reassigning bits from existing fingerprints to remap entries during expansion while assigning longer fingerprints to newly inserted entries. However, efficiently supporting expandability can compromise the *strong adaptivity*—i.e., the number of false positives remains tightly concentrated around $\epsilon \cdot n$ [48] , where n is the number of queries the filter has seen so far, even if the queries are adversarially generated. In the Aeris filter, we demonstrate that by assigning a small number of additional fingerprint bits—specifically, $\log_2 \log_2 \frac{1}{\epsilon}$—we can still maintain strong adaptivity.

Our results:

- We design and implement the Aeris filter, an adaptive range filter that supports monotonic adaptivity.
- The Aeris filter employs a new encoding scheme that efficiently encodes variable length fingerprints for range queries using 1 extra bit per slot compared to the Memento filter.
- The Aeris filter enables databases to maintain consistent throughput even for adversarial workloads and prevent throughput drops. When using the Aeris filter, WiredTiger [37] achieves 1.5×–2.5× higher throughput compared to when using the Memento filter.
- We show how to adapt a fingerprint by retrieving the original key from a storage-based reverse map and rehashing it. Unlike past work on adaptive filters, we structure this reverse map as a write-optimized index to alleviate bottlenecks on the write path, and we exploit it to support efficient expansions.

Symbol	Definition
ϵ	The target false positive rate
R	Maximum query range length
n	Number of queries
q	Size of quotient in bits
r	Size of remainder in bits
s	Size of slot in bits
α	Load factor of the filter
$h(x)$	General hash function
P	Partition size
$p(k)$	Partition of key k
$m(k)$	Memento of key k (Offset of key within partition)
k_p	Keepsake box of partition p in filter (identified by variable-length fingerprint)
l	Initial length of fingerprint
μ	Average number of items per partition
$h(p(k))$	Partition hash of key k
$h_0(p(k))$	Canonical slot of fingerprint (First q bits of $h(p(k))$)
$h_1(p(k))$	Remainder of fingerprint (Next r bits of $h(p(k))$)

Table 1. A table of notations split into three sections: for the general range filtering problem, for the quotient filter, and for the Aeris filter respectively.

- The Aeris filter achieves an order of magnitude lower false positive rate compared to the Memento filter and the Grafite filter, two state-of-the-art range filters with robust guarantees, on skewed workloads (Zipfian distribution).
- The Aeris filter has no overheads on query performance due to adaptivity and offers similar performance as the Memento filter and Grafite for query workloads.

2 Preliminaries

We now describe the structures on top of which we build Aeris filter. Table 1 summarizes the symbols used throughout the paper.

2.1 Quotient filters

The quotient filter (QF) [4, 18, 19, 40, 41][1] is a point-query filter that represents a multiset of keys while supporting dynamic insertions and deletes. It provides the common infrastructure for all other filters described in this paper. At its core, it uses a hash function to map each key from the universe to a l-bit fingerprint. A false positive occurs when the fingerprint of a queried key matches the fingerprint of some other key stored in the filter.

The QF divides a Key x's fingerprint $h(x)$ into a q-bit ***quotient*** $h_0(x)$ and an r-bit ***remainder*** $h_1(x)$. It maintains a hash table of 2^q slots, each of which can hold one r-bit remainder. When a key x is inserted, the quotient filter attempts to store the remainder $h_1(x)$ at index $h_0(x)$, which we call x's ***canonical slot***.

[1]For simplicity, we refer to Pandey et al.'s [41] counting quotient filter as the quotient filter (QF) throughout the paper. The counting quotient filter is the more space-efficient and performant version of the original quotient filter (QF) introduced by Bender et al.'s [4].

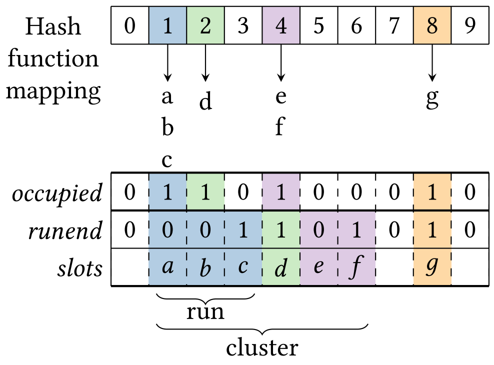


Fig. 1. The Quotient Filter [41] resolves hash collisions by storing colliding entries along contiguous runs that push each other to the right to form clusters.

To resolve hash collisions caused by multiple fingerprints being mapped to the same canonical slot, the QF uses Robin Hood hashing, a variant of linear probing that pushes items to the right to make space. Robin Hood hashing effectively sorts keys based on their hashes by maintaining the invariant that if $h(a) < h(a')$, then the fingerprint for Key a will be stored to the left of the fingerprint for Key a'. All keys that share the same canonical slot are stored contiguously in a ***run***. A sequence of runs whereby all but the first have been pushed to the right away from their canonical slot is called a ***cluster***.

Figure 1 illustrates an example. As shown, three keys are mapped to Slot 1, forming a run consisting of three slots. At the same time, one entry is mapped to Slot 2 and two entries are mapped to Slot 4. The entries Slots 2 and 4 are pushed to the right by the Run from Slot 1 to form a cluster.

As runs may shift to the right, the QF must allow operations to identify the correct run associated with a given canonical slot. It does this using 2 bits of additional metadata (*occupied* and *runend*) per slot. The *occupieds* bit for a slot is set if there is at least one entry for which this slot is the canonical slot. In Figure 1, for example, the *occupieds* bit is set to 1 only for Slots 1, 2, 4, and 8. The *runends* bit is set for a given slot if it contains the last fingerprint in a given run. In Figure 1, the *runends* bit is set to 1 for Slots 3, 4, 6 and 8.

To speed up queries, the QF divides the filter into 64 slot chunks and stores an 8-bit integer offset for each chunk to indicate across how many slots the contents of this chunk have been pushed to the right. Within each chunk, we observe that the i^{th} *runend* bit set to 1 corresponds to the end of the run belonging to the i^{th} slot for which the *occupied* bit is set to 1. This one-to-one correspondence is used to find the start and end of a run using constant-time *RANK* and *SELECT* primitives [41].

The QF supports efficient insertions for a load factor α of up to 95%. Beyond this point, long clusters make insertions prohibitively expensive. Thus, the QF uses $(\log_2 \frac{1}{\epsilon} + 2.125)/\alpha$ bits per key to provide an FPR of ϵ and expected constant-time operations.

For all traditional point filters such as the QF, the FPR guarantee ϵ holds if queries are sampled randomly. However, in real-world applications and/or in the presence of an adversary, the same key may be queried multiple times within a short time span, thus potentially leading to repeated false positives. We refer to this phenomenon as *temporal skew*, and it can cause the FPR to exceed ϵ and approach 1.

2.2 Adaptive Filters and the AdaptiveQF

Adaptive filters were designed to maintain a more robust FPR in the presence of temporal skew by correcting false positives. In particular, if querying for a non-existing Key x leads to a false positive due to a collision with the fingerprint of some Key y, then Key y's fingerprint is modified to resolve the hash collision. Doing so prevents the false positive from repeating when Key x is queried for again. To modify Key y's fingerprint, Key y is retrieved from storage using a reverse map from each key's fingerprint to the original key. Key y is then rehashed to resolve the hash collision.

The first adaptive filter [36] uses multiple hash functions to generate remainders, and it records which of these hash functions was used to generate each remainder. Adaptation to a false positive takes place by regenerating the remainder over which the false positive occurred using one of the different hash functions. However, since adaptation cycles across several known hash functions, the filter is vulnerable to temporal skew if we query for multiple keys that collide with many of the possible remainders for some Key y. Hence, the FPR may still exceed ϵ and approach 1.

Monotonic adaptivity. Monotonically adaptive filters ensure that the filter never "forgets" a false positive. In other words, any query to Key x that results in a false positive the first time will result in a negative for all subsequent times that Key x is queried for. Such filters ensure that every query has a probability of at most ϵ of returning a false positive independently of the query workload [2]. They are therefore robust to temporal skew. In addition, they are useful for applications such as URL blocklists, where users are impacted negatively by the experience of an innocuous website being identified repeatedly as malicious and needing to be repeatedly reverified by the user as being non-malicious. [1].

The adaptive quotient filter. The AdaptiveQF [48] is a monotonically adaptive filter based on the QF design. It adapts to false positives by extending a fingerprint over which a false positive occurred to span multiple slots. It supports this by using one additional metadata bit with each slot (*extension bit*) that marks whether the given remainder in the slot is a new remainder or an extension of the remainder in the slot to the left. Since fingerprints monotonically grow over time in response to false postives, the AdaptiveQF never forgets information and thus repeats a false positive.

Although every fingerprint extension takes up an additional slot in the filter, the rate at which extensions are created is slow. The reason is that the set of non-existing queries that users issue is typically small relative to the universe size (e.g., common misspellings of a website). The number of extensions needed to eliminate false positives in such a case is moderate and manageable. Even in an adversarial setting where non-existing queries are issued uniformly randomly, subsequent adaptations after the first one occur exponentially less frequently. A given fingerprint is extended the first time after an average of n/ϵ queries have taken place globally, The second adaptation occurs after at least another n/ϵ^2 queries since the fingerprint is now twice as long. In general, the i^{th} adaptation occurs after at least n/ϵ^i queries. Hence, the rate at which adaptations fill up the filter in the worst case slows down exponentially over time.

Despite its advantages, the AdaptiveQF still exhibits several core issues. First, adaptations still add content to the filter, potentially causing it to have to expand. Expansions occur by traversing and rehashing the entire dataset, and so it is an expensive process. Moreover, the AdaptiveQF does not support range queries. It is therefore inapplicable to numerous applications that perform both point and range queries [10, 29, 50].

2.3 Range Filters and the Memento filter

Range filters have been an active area of research over the past decade due to their use-cases in KV-stores and beyond [25, 27, 32, 42, 51]. While all range filters to date are vulnerable to temporal skew, earlier designs were also vulnerable to the equally important issue of *spatial skew*. The source of this latter vulnerability is that many earlier range filters forgo the keys' lower-order bits to save space (e.g., by truncating keys [51] or mapping them based on their more significant bits to a smaller domain [9, 47]). Hence, if the end points of many empty range queries are close to the keys in the dataset, they are likely to match the prefixes of some existing keys and return false positives. The FPR may therefore exceed ϵ and approach 1.

A well-known lower bound states that a range filter that is robust to spatial skew while supporting range queries of up to length R must use $\log_2(R/\epsilon)$ bits per entry [26]. A few recent range filters meet this lower bound while providing spatial robustness [14, 22, 34]. However, no range filter to date provides spatial and temporal robustness simultaneously. Is it possible for a range filter to achieve both properties so that the FPR is guaranteed to be at most ϵ for any possible dataset and workload? If so, can it also maintain fast operations?

Memento filters. The Memento filter [22] is the range filter that provides spatially robust queries while meeting the aforementioned lower bound and providing fast queries, insertions, deletions. In addition, it can expand and contract along with the dataset size. If we could also make the Memento filter temporally robust, we would achieve our research goal. We therefore use the Memento filter as the starting point for our design.

Partitions and mementos. The Memento filter divides a finite universe of keys (e.g., 64-bit integers) into equally-sized partitions of P keys each, where P is the largest power of 2 that is smaller than or equal to the maximum range query size R (i.e., $P = 2^{\lceil \log_2(R) \rceil}$). The binary representation of each key is then divided into a prefix and a suffix. The prefix identifies the partition that the key belongs to, while the suffix identifies the offset of the key within the partition. The suffix, referred to as the key's ***memento***, comprises the least significant $\log_2(p)$ bits of the key. Figure 2 illustrates this partitioning.

Keepsake box. For each insertion, the Memento filter hashes the prefix of the key to generate a fingerprint. It inserts this fingerprint into a quotient filter as described in Section 2.1 by mapping it to a canonical slot and storing the remainder of the fingerprint in that slot (and using Robin Hood Hashing to clear space). Next to the fingerprint's remainder, it stores the key's memento. If more keys within the same partition are inserted, they get mapped to the same fingerprint and stored as a contiguous list of sorted mementos alongside the fingerprint's remainder. This collection of a partition fingerprint and associated fingerprints is referred to as a ***keepsake box***. Hence, the Memento filter preserves the locality of nearby keys without sacrificing their lower-order bits. This is the key to achieving fast operations and spatial robustness simultaneously. Partitions and keepsake boxes form a many-to-one-relationship, and Figure 2 illustrates the mementos from different partitions being stored in a single keepsake box.

Range queries. A range query is handled by checking if there is an existing keepsake box in the filter for each overlapping partition. Since the partitions are sized to approximately match the maximum query length, at most two keepsake boxes need to be checked in the filter. For each keepsake box, we check for mementos that lie within the query range. If there is at least one, the query returns a positive.

FPR & space. Since the Memento filter generates fingerprints based on the prefixes of keys, it is possible for keys from different partitions to map to the same fingerprint and thus to the same keepsake box. This can lead to false positives. The remainder is set to $\log_2(1/\epsilon)$ bits to guarantee an FPR of ϵ. Overall, the Memento filter uses $(\log_2(\frac{R}{\epsilon}) + 3.125)/\alpha$ bits per key.

Example. Consider a set $S = \{12, 22, 24, 35, 38, 55, 57, 66\}$. We let the partition size be $P = 10$ in this example as decimals are easier for illustrative purposes, though in reality the partition size is always a power of two. Given a Key k, its partition $p(k)$ is computed as $\lfloor k/10 \rfloor$ and its memento $m(k)$ as k mod 10. Thus, Keys 55 and 57 both belong to Partition 5, and their respective mementos are 5 and 7. Suppose that the hash function generates the same fingerprint for Partitions 1 and 5. This is a hash collision, and so the mementos from both of these partitions get stored within the same keepsake box (i.e., 2, 5 and 7 corresponding to Keys 12, 55 and 57). In the figure, a query arrives targeting a part of Partition 1. It visits the corresponding keepsake box, finds that both Mementos 1 and 5 overlap with the specified range, and returns a positive. Note that this is a true positive since Key 12 actually falls within the specified key range, though Memento 5 comes from an entirely different partition and would have led to a false positive if Key 12 did not also exist.

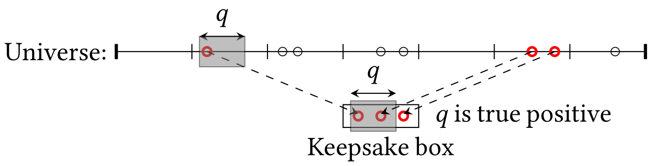


Fig. 2. The Memento filter partitions the universe and maintains a keepsake box for each non-empty partition. A range query is answered by checking at most two keepsake boxes. The figure shows one keepsake box with mementos highlighted and an example of a true positive query q.

2.4 Challenges

While the Memento filter provides a good foundation for supporting both spatially and temporally robust queries, several technical challenges remain to be addressed.

Adaptivity for keepsake boxes. To build an adaptive range filter, we can extend the fingerprints of the keepsake boxes in the Memento Filter to resolve false positives, similar to the AdaptiveQF. The challenge is that each keepsake box also stores a list of mementos from potentially different partitions due to hash collisions. Extending the fingerprint of a keepsake box naively could thus result in future false negatives. How can we correctly fix false positives while ensuring that it is impossible for false negatives to ever occur?

Space overhead. Since filters reside in memory, space efficiency is critical. As we have seen, the QF uses $(\log_2(1/\epsilon)+2.125)/\alpha$ bits per key while the AdaptiveQF uses one additional bit per key to support adaptivity. Similarly, the Memento filter uses $(\log_2(R/\epsilon)+3.125)/\alpha$ bits per key. Can we match these space overheads for an adaptive range filter that provides both spatial and temporal robustness?

Stalls due to expansions. Monotonically adaptive filters accumulate information by extending fingerprints. Eventually, the filter runs out of space and must expand to accommodate new adaptations or insertions. The AdaptiveQF expands by rereading keys from storage and rehashing them to construct a 2× larger filter. Can we expand the filter without incurring prolonged I/O overheads and stalls?

Reverse map. Existing adaptive filters structure the reverse map as a flat on-disk hash map. For every application insertion, the reverse map incurs a read I/O and a write I/O to read and add a new entry to a hash bucket. The challenge is that keeping the reverse map up-to-date can become an I/O bottleneck for insertion-heavy workloads. Can we alleviate the reverse map bottleneck on the write path while still supporting adaptivity to cope with temporal query skew?

3 Aeris filter

This section describes Aeris filter and how it overcomes the technical challenges described in Section 2.4. Similarly to the Memento filter, the Aeris filter is built on top of a quotient filter and consists of multiple keepsake boxes, each of which stores mementos from a set of partitions. The core difference between the Aeris filter and the Memento filter is that the fingerprints in keepsake boxes of the Aeris filter can be extended to support adaptivity.

Extendable fingerprints. The keepsake boxes in Aeris filter use extendable fingerprints to support adaptivity. Each prefix $p(k)$ is mapped to an l-bit fingerprint using hash function $h(p(k))$. The hash function $h(p(k))$ generates a long hash and higher-order l bits are extracted for the l-bit fingerprint. Initially, a keepsake box in the Aeris filter has a fingerprint of length $l=(q+r)$ bits. The higher order $q=\lceil\log_2(n)\rceil$ bits form the quotient (canonical slot) and the remaining r bits form the remainder.

When adapting in response to a false positive, the fingerprint of a keepsake box is extended to a longer fingerprint of length $l' > l$. Since both the fingerprints are generated from the same long

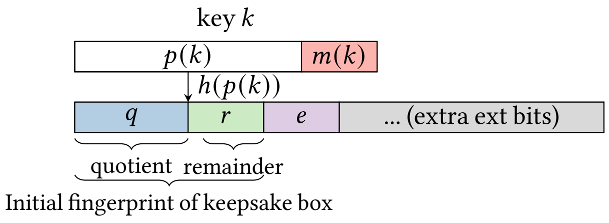
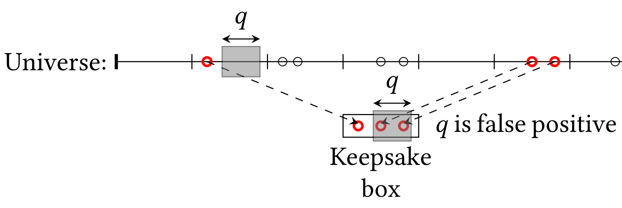
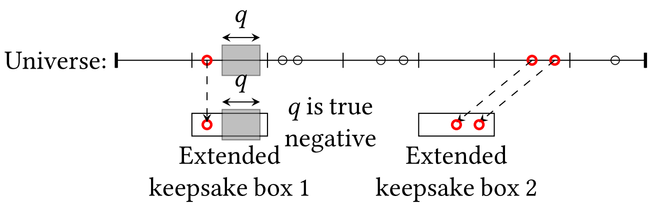


Fig. 3. Variable length fingerprints in the Aeris filter. The key is divided into a partition $p(k)$ and memento $m(k)$ bits. The fingerprint is generated by extracting the higher-order bits of a long hash $h(p(k))$. The fingerprint is initially $(q+r)$ bits, where q is the length of the quotient and r the remainder. As the fingerprint is extended for adaptivity, additional extension bits (denoted by e) are added.



(a) Example of a false positive query in the Aeris filter. The query is a false positive due to mementos added by partitions not in the query range lying in the keepsake box.



(b) After adapting, the keepsake boxes are split by extending their fingerprints. The mementos of the corresponding partitions are moved to their new keepsake box.

Fig. 4. Range query adaptivity in the Aeris filter

hash $h(p(k))$, the shorter fingerprint is a prefix of the longer one. Thus, extending a fingerprint does not change its canonical slot and remainder. We refer to the additional bits added as *extension* bits. Figure 3 illustrates how fingerprints that can be extended are generated in the Aeris filter.

Queries and false positives. To answer a range query $[a,b]$ (i.e., is there any key k in the dataset such that $a \leq k \leq b$), the Aeris filter splits the range query into subqueries, generating one for each intersecting partition. Since the partition size is chosen to be $2^{\lceil \log_2 R \rceil}$, there can be at most 2 such subqueries.

For ease of discussion, we will assume that $[a,b]$ completely lie in a single partition , both the endpoints of the query range a and b belong to partition p. To answer the query $[a,b]$, the Aeris filter will check if the keepsake box corresponding to p contains a memento m that lies between $[m(a),m(b)]$. The query is a false positive if the dataset does not actually contain any key between $[a,b]$ but the keepsake box contains a memento in the query range. This can happen if there is another partition $p' \neq p$ that also maps to the keepsake box of p and contributes a memento within the query range. Figure 4a illustrates an example of a false positive range query.

Adapting to false positives. To adapt to a false positive range query $[a,b]$ (lying within a single partition p for ease of discussion), the Aeris filter splits the keepsake box corresponding to p into multiple keepsake boxes with extended fingerprints.

Let k_p be the keepsake box corresponding to p and that it currently has a l-bit length fingerprint. The false positive arises from there being one or more non-empty partitions that also map to k_p and contribute a memento that lies within the query range. Each such partition p' maps to k_p due to a hash collision — both p and p' have the same l bit fingerprint.

To adapt to the false positive, the Aeris filter extends fingerprint length to l' such that the colliding partitions no longer map to the same keepsake box. The Aeris filter creates new keepsake boxes with l'-bit length fingerprints for all partitions (including those that do not contribute mementos in the query range) that currently map to k_p. All the mementos of k_p are then redistributed across these new keepsake boxes. This adaptation prevents query $[a,b]$ from being a false positive again — any colliding partition p' that contributed a memento in the query range is now part of a different keepsake box. Figure 4 illustrates the process of adaptation in the Aeris filter.

3.1 Reverse map

Similarly to other adaptive filters, the Aeris filter maintains a reverse map on disk to map from each fingerprint to the original key. The Aeris filter structures this reverse map as a write-optimized B^ϵ-tree [7] to avoid bottlenecks on the insertion path. We now describe the access patterns targeting the reverse map, and how we structure the reverse map to optimize for these access patterns.

Access patterns. Every time a key-value pair is inserted into or deleted from the filter, a corresponding entry must be added to or removed from the reverse map to maintain the association between the key's fingerprint and the full key. Consequently, workloads with frequent insertions or deletions generate many writes to the reverse map. Updates that modify only the value associated with an existing key do not affect the reverse map.

In contrast, the reverse map is not accessed during most queries (i.e., true positives or true negatives). It is only consulted on a false positive to retrieve the key corresponding to the colliding fingerprint so that the filter can extend that fingerprint. Initially, a false positive is expected to occur at most once every $1/\epsilon$ queries. In fact, as discussed in Section 2.2, false positives become less frequent over time as fingerprints grow longer. That is, after n/ϵ^i queries, a false positive occurs at most once every $1/\epsilon^i$ queries. Hence, accesses to the reverse map along the query path are infrequent to begin with and become rarer over time.

Reverse map structure. We exploit the fact that queries to the reverse map are infrequent, whereas insertions are comparatively frequent and costly. Accordingly, we implement the reverse map using a write-optimized data structure, trading a modest increase in read latency for substantially higher insertion throughput.

In particular, we structure the reverse map as a B^ϵ-tree [7]. A B^ϵ-tree is a B-tree [49] wherein each internal node contains a write buffer for each of its children. Updates to a given child accumulate within its parent's buffer and only get flushed into the child when this buffer is full. As this approach pushes multiple entries to the child at a time with one I/O, it amortizes write cost. Generally, a B^ϵ-tree supports insertions in amortized $O\left(\frac{1}{B^{1-\epsilon}} \cdot \log_{B^\epsilon} \frac{N}{M}\right)$ I/Os and queries in $O\left(\log_{B^\epsilon} \frac{N}{M}\right)$ I/Os. In these expressions, N is the number of entries in the dataset, B is the fanout of the tree, M is the number of entries that fit in memory, and ϵ is a parameter that trades between the tree's fanout and the amount of buffering per node. Our B^ϵ-tree implementation uses SplinterDB [11], though other write-optimized structures such as RocksDB [45] would have also been suitable.

In contrast, insertions in a B-tree cost $O\left(\log_B \frac{N}{M}\right)$ I/Os, which is asymptotically higher compared to the B^ϵ−tree. Therefore, the B^ϵ−tree is the more appropriate choice for the reverse map compared to the B-Tree, as the workload typically consists of a higher proportion of updates compared to lookups.

Each entry in the B^ϵ-tree maps a fingerprint to a list containing a subset of the keys within its corresponding keepsake box. Since insertions are performed out of place, multiple entries corresponding to the same keepsake box may exist at multiple levels of the tree, but always along a single root-to-leaf path. Together, these entries contain the full set of keys within the keepsake box. When an entry is flushed into a node that already contains an entry with the same fingerprint, the entries are merged by concatenating their key lists.

Handling a false positive involves a single root-to-leaf traversal of the B^ϵ-tree to retrieve all keys in the affected keepsake box. We then insert a tombstone with the original fingerprint to mark the mapping entries for eventual removal. Finally, we rehash the retrieved keys, potentially splitting the keepsake box, and reinsert them along with their updated fingerprints.

Parameterization. Under typical parameter settings for a B^ϵ-tree, insertions are highly efficient. Since a key is typically no larger than 16 bytes while a fingerprint is capped at 1-2 bytes, a 4 KB disk page can hold roughly 256 fingerprints and keys (i.e., $B \approx 256$). We set M such that $N \gg M \gg B$ (e.g., M corresponds to several megabytes of entries), which further reduces the cost of both insertions and queries. Substituting these parameters into the cost expression above, we find that insertions into the reverse map consume as little as 1–2 % of the system's total write bandwidth. In contrast, handling a false positive may require a few additional I/Os to traverse the B^ϵ-tree—an acceptable trade-off given the rarity of such events.

Space. Aside from its small in-memory buffer, the reverse map is stored entirely on disk. Each entry contains only a fingerprint and its corresponding key, without storing any values or rows, which are typically much larger. In many practical settings, keys are 8 bytes, while values range from 64 bytes to several kilobytes [13, 15, 39]. Consequently, the reverse map is typically one to two orders of magnitude smaller than the primary dataset.

3.2 Implementation

We now describe how the quotient filter is extended to support range queries and adaptivity. We describe the data layout and operations needed to support queries, updates and adaptivity.

High-level overview. The Aeris filter is an array of $n = 2^q$ slots of size $(r + \log_2(R))$ bits, where r is the length of the fingerprint remainder and R is the maximum range query length that the Aeris filter supports[2]. The Aeris filter maintains 3 metadata bits per slot: *occupied*, *runend* and *extension* bits. Note that this is similar to structure of the AdaptiveQF, except that the slots in the AdaptiveQF are of size r bits. A keepsake box is stored as a single unit in the Aeris filter and can span multiple slots. A group of keepsake boxes having the same canonical slot are referred to as a *quotient run.*

Keepsake box encoding. In the Aeris filter, a keepsake box stores the remainder from the fingerprint, metadata bits, and mementos sequentially. The *extension* bit is set to 1 in slots that store extension bits, and the *runend* bit marks the end of a keepsake box. This differs from the standard quotient filter encoding that uses the *runend* bit to mark the end of a quotient run. Instead, the Aeris filter reuses the *extension* bit to mark the end of a quotient run: the last slot of a quotient run will have both its *extension* and *runend* bit set. Since an extension slot can never be the last slot of a keepsake box, it is always possible to determine whether an *extension* bit is marking an extension slot or the end of quotient run by checking its corresponding *runend* bit. This encoding avoids introducing additional metadata bits while supporting both keepsake boxes and extension slot information.

[2]Similar to the Memento filter, the Aeris filter also supports queries of length greater than R but cannot guarantee false positive rate of ϵ.

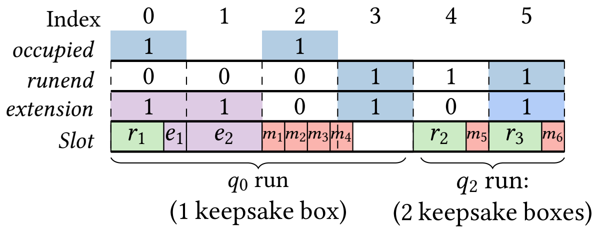

Index
0
1
2
3
4
5
occupied
runend
extension
Slot
r_1 e_1 e_2 m_1 m_2 m_3 m_4 r_2 m_5 r_3 m_6
q_0 run (1 keepsake box)
q_2 run: (2 keepsake boxes)

Fig. 5. Keepsake box encoding in Aeris filter. The keepsake box stores the remainder, extension bits, and mementos in contiguous slots. The above figure stores two quotient runs, q_0 with one keepske box and q_2 with two keepsake boxes. Runend bits mark the end of a keepsake box (Slot 3, 4 and 5) and are used in conjunction with the extension bit to mark the end of quotient runs (Slot 3 and 5).

Extension bits. To keep the extension bits consistent with the keepsake box encoding, fingerprints are extended in discrete increments. The first extension is the size of a memento ($\lceil\log_2(R)\rceil$) bits. Subsequent extensions bits are added in increments of ($\lceil\log_2(R)\rceil + r$) bits, which is the size of one full slot. Figure 5 illustrates the encoding of a keepsake box that has been extended twice.

Storing keepsake boxes. Similar to the quotient filter, the Aeris filter uses Robin Hood hashing to store keepsake boxes. Robin Hood hashing maintains the invariant that items are stored in order of their canonical slots — items having the same canonical slot are clustered together. The Aeris filter will first attempt to store the keepsake box in contiguous slots starting at its canonical slot. If any of the slots between the canonical slot and the last slot required to store the keepsake box are occupied, the Aeris filter will shift move keepsake boxes according to the Robin Hood hashing scheme to make space for the new incoming keepsake box. A group of keepsake boxes having the same canonical slot are referred to as a *run*, while a group of runs with no empty slots in between them is referred to as a *cluster*. Within a quotient run, keepsake boxes having the same canonical slot and remainder are ordered lexicographically by their extension bits.

Locating keepsake boxes. The Aeris filter uses the quotient filter's rank and select-based metadata operations to locate the start of a keepsake box. The Aeris filter finds the start and end of a run using constant-time *RANK* and *SELECT* primitives [41]. It then scans the remainders and extension bits of the keepsake box within the run to find the one with a matching fingerprint.

Queries. The Aeris filter divides a range query $[a,b]$ (Is there any key k in the dataset such that $a \le k \le b$?) into smaller subqueries that lie completely in a partition. For each subquery, the Aeris filter locates their corresponding keepsake boxes and checks if it contains any mementos that lie within the subquery mementos. Range queries of length up to R can be divided into at most two subqueries as the Aeris filter uses a partition size of $2^{\lceil\log_2(R)\rceil}$. Queries of length greater than R will be subdivided into multiple subqueries and answered in a similar fashion [2].

Adaptations. The Aeris filter adapts a query when it receives feedback from the database that the query was a false positive. The query is adapted by extending the keepsake boxes of all the subqueries the original query was divided into. For each keepsake box, the Aeris filter consults the reverse map to retrieve all the keys the map to the keepsake boxes. Each keepsake box is split into multiple keepsake boxes with extended fingerprints and the mementos are redistributed.

The fingerprints of the keys in the keepsake box are extended until they no longer collide with the query partition. A subtle case arises when the query is a false positive, but the keepsake box contains mementos from the query range. This occurs when the dataset includes keys from the query partition but are not in query range. Extending these fingerprints to not collide with the query partition is not possible as they represent the same partition. Instead, the fingerprints from the query partition must be extended after all the other fingerprints have been extended, once the required extension length is known. This case also highlights the complexity of supporting adaptivity for range queries compared to point queries. The AdaptiveQF does not encounter this situation as it only supports point queries.

Inserts and deletes. The Aeris filter inserts a key by adding its memento to the keepsake box it maps to, creating a new keepsake box if it does not exist. The case when the keepsake box already exists but has extension bits is handled specially. If the extension bits match, the memento is added to that keepsake box. Otherwise, a new keepsake box is created with distinguishing extension bits. This is to prevent ambiguity in mapping keys to keepsake boxes as otherwise it is possible a key might match both keepsake boxes. The Aeris filter deletes keys by removing its memento from the keepsake box, removing the keepsake box if it becomes empty.

3.3 Expandability

In this section, we discuss how the Aeris filter expands while avoiding costly I/O stalls. As outlined in Section 2.4, expandability is necessary to guarantee monotonicity in adaptive filters as the filter cannot discard accumulated adaptivity information.

Expanding quotient filters. The easiest way to expand a filter is to reread the keys from storage and reinsert them from scratch into a larger filter. However, this entails a full pass over the data and is therefore expensive. A cheaper alternative is to sacrifice a bit from each remainder to become a part of the slot address and remap all entries into a 2x larger filter. The issue with this approach is that the remainders shrink across expansions, thus causing the false positive rate to linearly increase.

InfiniFilter. To address these issues, InfiniFilter [16] maintains fingerprints of variable length in the filter. When a filter expands, existing fingerprints are shortened in the expanded filter. However, fingerprints from new keys inserted after expanding have the full length remainder. The Memento filter utilizes the InfiniFilter technique to support expandability.

Eventually, the oldest entries in the filter run out of bits, thus begging the question how to continue expanding when some entries have no more bits to allow remapping an entry to a larger filter. This problem can be addressed by storing such entries within auxiliary structures at the expense of query cost [16] (InfiniFilter), to duplicate them within the filter at the expense of space and complicating deletions [17] (Aleph filter), or by reconstructing the filter from scratch by rehashing the original keys [22] (Memento filter). We now show that Aeris filter offers a fourth way out by virtue of having a reverse map.

Expansions in the Aeris filter. The Aeris filter expands using the InfiniFilter technique. As in the InfiniFilter, each keepsake fingerprint maintains a unary age counter of the form 00..1 to determine how many expansions ago the fingerprint was inserted. For example, a unary counter of 001 indicates that the fingerprint was inserted 2 expansion cycles ago. New fingerprints into the filter are always inserted with an age counter of 1, indicating the fingerprint was inserted after the most recent expansion. Keepsake boxes are transferred to the expanded filter by transferring one bit from the remainder to the canonical slot and incrementing the unary counter. Figure 6 illustrates how the Aeris filter implements the InfiniFilter expansion technique.

The Aeris filter overcomes the drawback of Memento Filter needing a full scan of the dataset despite only a fraction of the fingerprints running out of remainder bits. It does so by utilizing the reverse map to selectively rejuvenate only those fingerprints that have run out of bits. When the Aeris filter expands, it iterates over the filter to find all fingerprints that were inserted r expansion cycles ago by inspecting the unary counter of the fingerprint. For each of these fingerprints, the Aeris filter queries the reverse map to retrieve the keys mapping to this fingerprint. The keepsake

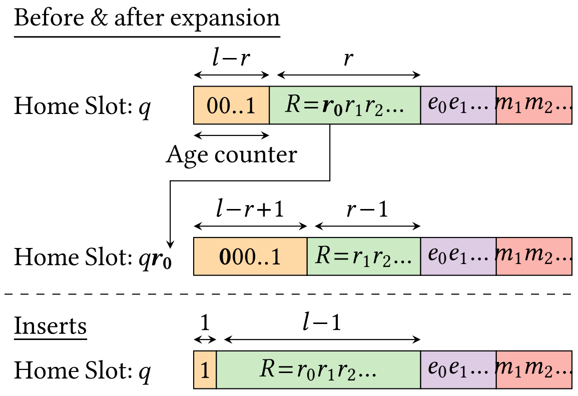


Fig. 6. The Aeris filter expands using variable-length fingerprints that encode how many expansion cycles ago a partition run was inserted. Partition runs are transferred to the expanded filter by moving one bit from the remainder to the quotient and incrementing the unary counter by appending a 0 bit to the start. Insertions always insert a full length fingerprint with an age counter of 1.

boxes for these fingerprints are not transferred to the new filter. Instead, new keepsake boxes with rejuvenated fingerprints are created and inserted into the new filter.

The Aeris filter deamortizes the I/O cost across expansions and avoids read amplification. Instead of incurring significant I/O every r expansions to perform a full scan of the dataset, the Aeris filter utilizes the reverse map to spread the I/O cost into smaller parts across r expansions, eliminating periodic long stalls in Memento filter's expansions.

Queries and adaptations. In the expandable Aeris filter, a partition can match keepsake boxes from different expansion cycles. More specifically, a partition can match at most one keepsake box from each expansion cycle. Queries in the expandable Aeris filter therefore check for matching keepsake boxes from each expansion cycle within the relevant partitions.

To adapt to a false positive query, all keepsake boxes that match the query's target partition are merged and adapted into a new partition run with age counter set to 1. This approach opportunistically rejuvenates fingerprints and maximizes the utility of the I/O performed in querying the reverse map to adapt.

4 Theoretical analysis

In this section, we analyze the false positive rate and space usage of the Aeris filter. We show that the Aeris filter is strongly and monotonically adaptive, while requiring only one extra bit compared to the Memento filter, which was already near space-optimal.

Static false positive rate. For a fingerprint of size f, the Aeris filter and the Memento filter have the same *static* false positive rate, i.e., the probability that the filter returns a positive result for an empty query. This holds because given the same hash function, both filters store the same set of fingerprints, and adaptations only decrease the overall false positive rate.

A query $[a,b]$ lying entirely in partition p is a false positive if there exists another partition p' having the same fingerprint, and p' also has keys with at least one memento in the query range $[m(a),m(b)]$. Since these two events — partition hash collision and partition memento overlap — occur independently, the probability of a false positive follows from the product of their independent probabilities. Given that the partitions are mapped to l-bits fingerprints, the probability of two partitions colliding is 2^{-l}. Given n items and an average of μ items per partition, the overall probability can be bounded by $(n/\mu)(2^{-l}) \le n \cdot 2^{-l}$. Since partition mementos overlapping is dependent on the dataset, the overall false positive rate of this query is bounded by $n \cdot 2^{-l}$. By setting the partition size (P) to be equal to $2^{\lceil log_2(R) \rceil}$, any query Q not contained in a single partition can be subdivided into at most 2 sub-queries that do lie within a partition, bounding the overall maximum false positive rate as $\epsilon = n \cdot 2^{1-l}$.

Strong adaptivity. The Aeris filter improves the Memento filter by also guaranteeing *strong adaptivity* [3]: the probability of an empty query being a false positive is ϵ, including adversarial ones that has access to results of prior queries. We show that the false positive probability of for an empty query $[a,b]$ in partition p is ϵ. Since the Aeris filter store mementos exactly, the false positive rate can be bounded by the probability of partition hash collisions. If query $[a,b]$ is a false positive, then it must be due to a partition p' that never collided with p in previous queries. This is because if p' had collided with p before, then the filter would have adapted and resolved this collision. The probability that a user (including adversarial users) chose a partition p' that collides with a partition p containing items in the dataset is $n \cdot 2^{-l}$. Given that any general query Q can be divided into at most 2 sub-queries, the overall sustained false positive rate is bounded by $\epsilon \le n \cdot 2^{1-l}$.

Space usage. The Aeris filter allocates $N = 2^q$ slots to store n items with load factor $\alpha = n/N$. Each slot size is $r+m$ bits, where r is the size of remainder of the fingerprint and $m = \log_2 P$ is the space required to store a memento. To guarantee a false positive rate of $\epsilon \le n \cdot 2^{1-f}$, the Aeris filter needs to use fingerprints of size of at least $l = \log_2(n/\epsilon)+1$ bits. The first q bits of the fingerprint are stored implicitly as the home slot index in the Aeris filter, thus needing r to be at least $\log_2(\alpha/\epsilon)+1$ bits. Thus, each slot in the Aeris filter occupies $r+m = \log_2(R\alpha/\epsilon)+1$ bits. Each slot also an overhead of 3.125 bits for the metadata bits. Putting everything together, for a false positive rate of ϵ and maximum query length R, the Aeris filter requires $\alpha n \left(\log_2 \frac{R\alpha}{\epsilon} + 4.125\right)$ bits.

Without any adaptions, at least $1-\alpha$ fraction of the N slots will be free after n insertions. During adaptions, we re-purpose the empty slots to store extension bits to resolve collisions. This approach is similar to the variable-length counter encoding technique in the counting quotient filter [41]. In expectation, only 2 bits are required to resolve collisions [3] to support strong adaptivity. However, the Aeris filter over-adapts by growing the extension first by $m = \log_2 R$ bits and subsequently in increments of $r+m$ bits.

Monotonic adaptivity. Here we consider an Aeris filter instance that has expanded r times. Observe that, by rejuvenating its fingerprints, at most a fraction of 2^{-1} of the Aeris filter's non-empty slots have remainders of length r, at most a fraction of 2^{-2} of the non-empty slots have remainders of length $(r-1)$, and so on. Note that the oldest remainders are of length 0, and constitute a fraction of 2^{-r} of all non-empty slots. As such, the probability that a negative query matches one of these remainders is at most $\alpha \cdot 2^{-r} + \sum_{i=1}^{r} \alpha \cdot 2^{-r+i-1} \cdot 2^{-i} = \alpha \cdot (r+2) \cdot 2^{-r-1}$. Here, α is Aeris filter's load factor, i.e., the fraction of non-empty slots. Since range queries check at most two keepsake boxes, their false positive rate can be bounded by $2 \cdot \alpha \cdot (r+2) \cdot 2^{-r-1}$.

The above expression implies that using $r = \log_2 \frac{1}{\epsilon}$ bit remainders similarly to a standard QF yields an FPR of $(\log_2 \frac{1}{\epsilon} + 2) \cdot \epsilon$. This resulting FPR is higher than the desired FPR of ϵ by a factor of $(\log_2 \frac{1}{\epsilon} + 2)$. To counteract this extra multiplicative factor and achieve monotonic adaptivity, we slightly enlarge the remainders by an additional $\log_2 \log_2 \frac{1}{\epsilon} + 1$ bits, resulting in $r = \log_2 \frac{1}{\epsilon} + \log_2 \log_2 \frac{1}{\epsilon} + 1$ bit remainders. The added $\log_2 \log_2 \frac{1}{\epsilon}$ bits shrinks the exponential term at the end of the FPR expression and turns the extra multiplicative factor into a constant. The final additional bit ensures that this constant is smaller than 1, resulting in a total FPR of $\frac{\log_2 \frac{1}{\epsilon} + \log_2 \log_2 \frac{1}{\epsilon} + 3}{2 \log_2 \frac{1}{\epsilon}} \cdot \epsilon$ which is always at most ϵ for reasonably small ϵ, i.e., $\epsilon \le 2\%$.

The Aeris filter guarantees both strong and monotonic adaptivity. Strong adaptivity is inherently guaranteed by extending fingerprints to completely eliminate collisions that lead to false positives similar to the AdaptiveQF. Monotonic adaptivity is achieved through infinite expansion and rejuvenation of old fingerprints, ensuring that a false positive never repeats again. As demonstrated in the previous paragraph, these expansions maintain the maximum false positive rate by utilizing an asymptotically small number of additional fingerprints bits.

Supporting variable-length queries and keys. As Aeris filter is spatially robust and meets Goswami's lower bound [26], it uses at least $\log(R/\epsilon)$ bits per entry to handle range queries of size up to R while guaranteeing an FPR of ϵ. The implication is that the FPR guarantee does not hold with a larger than expected range size. For Aeris and Memento filter, for instance, the the FPR deteriorates linearly with respect to a range query length larger than the preconfigured value of R. In addition, the lower bound implies that a spatially robust range filter cannot support variable-length keys, as this would imply an infinite number of keys within any possible query range. By plugging infinity for R in the lower bound expression, we find that supporting var-length keys and/or queries would require infinite memory. For these reasons, spatially robust range filters [14, 22, 34] cannot support variable-length queries and variable-length keys with a bounded false positive rate.

The recent Diva range filter [21] overcomes these limitations by offering a relaxed *semi-robust* FPR guarantee that holds for smooth data distributions (e.g., Normal, Zipfian, Uniform, etc.). It approximates the data distribution through sampling and stores infixes of keys in-between two samples within an order preserving quotient filter. We observe that the techniques proposed in this paper, to elongate fingerprints of colliding keys on the query path, are applicable to Diva as well by elongating its infixes in response to false positives. In fact, since Diva is order-preserving, it can allow to identify colliding keys without the use of a reverse map and thereby thus simplifying the system and reduce I/O overheads. It would make for intriguing and impactful future work to make Diva adaptive and thus temporally robust as well.

5 Evaluation

In this section, we evaluate the accuracy and performance of the Aeris filter and compare it with state-of-the-art range filters, all of which are non-adaptive. We perform full system benchmarks by integrating the filters with WiredTiger [37], a production key-value store (Section 5.1). We also compare the performance of filters on microbenchmarks in a standalone setting (Section 5.2).

We compare the Aeris filter against four range filters: SuRF [51], SNARF [47], Grafite [14], and Memento filter [22]. The **Succinct Range Filter (SuRF)** is a trie-based range filter that stores the shortest unique prefix and suffix bits for each key in a succinct trie. The **Sparse Numerical Array-Based Range Filter (SNARF)** is a learning-based filter that learns a linear spline model of the key distribution, which is then used to map the keys to a bitmap. **Grafite** is a bitmap-based filter that provides robust guarantees against spatial skew. It uses a locality-preserving hash function to map each key to a bit in a bitmap. The resulting bitmap is then compressed using using Elias-Fano coding [20]. Finally, the **Memento filter** is a fingerprint-based range filter that also has robust guarantees against spatial skew. We use the open-source `C/C++` implementations of the above filters. All the filters are compiled with `gcc-9.4`.

For inserts, we only compare against the **Memento filter** and **SNARF**, as these are the sole baselines supporting inserts. For expansions, we only compare against the **Memento filter** as it is the only other expandable range filter apart from the Aeris filter.

Our code and experimental setup is available as an open source repository. [3]

System specification. All experiments were run on a server with a 64-core 2-way hyperthreaded Intel Xeon Gold 6338 CPU @ 2.00GHz with 1008 GB of memory and a 96 MiB L3 cache. The machine has a 4TB KIOXI KXG80ZN84T09 NVME drive and runs Linux kernel version `5.4.0-155`.

[3]https://github.com/saltsystemslab/AerisFilter

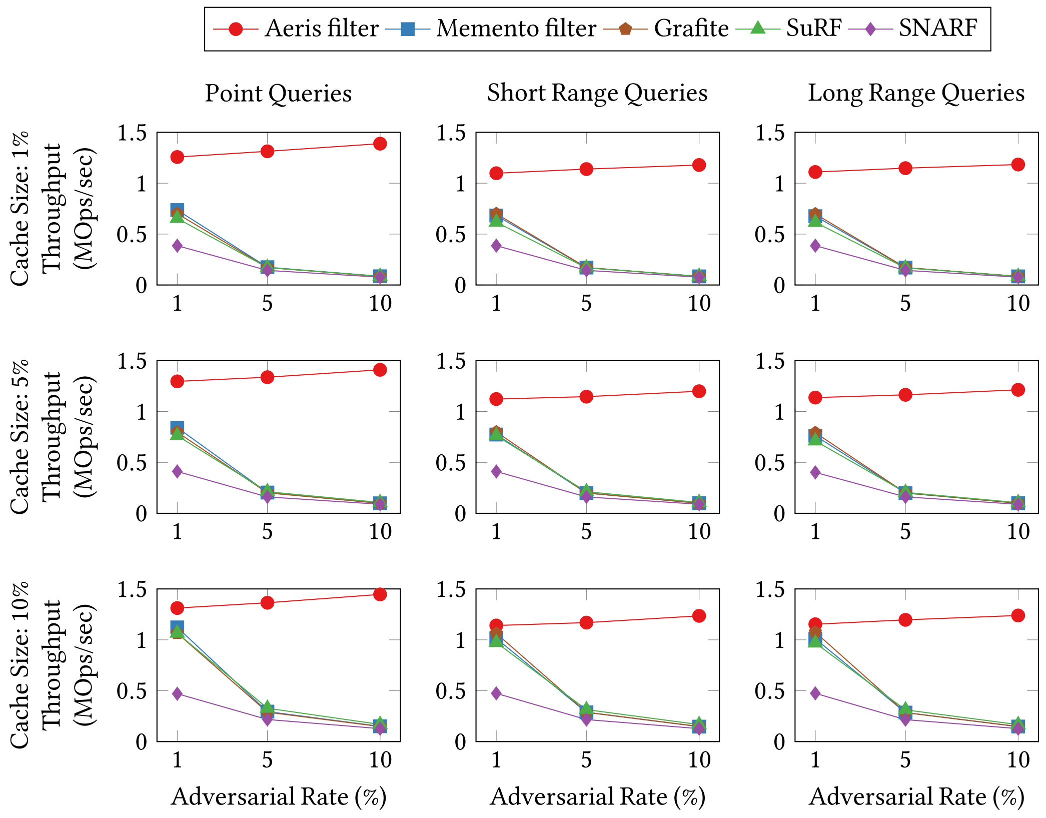


Fig. 7. Throughput on adversarial-query workload. All the filters are configured with a false positive rate of 2^{-9}.

5.1 Application benchmarks

In our application benchmarks, we evaluate the impact of the filters on the performance of WiredTiger [37], a B-Tree-based key-value store. The range filters, being small in size, are kept in memory and help avoid unnecessary disk accesses. Similar to the benchmarks from Wen et al. [48], we evaluate the effectiveness of the filters against adversarial workloads. To evaluate the effectiveness of the deamortized I/O approach for expandability, we compare the Aeris filter with the Memento filter. Finally, we study the overhead on inserts incurred due to supporting adaptivity.

Setup. Our system consists of a WiredTiger instance on disk and an in-memory range filter. Range queries first consult the filter and then proceed to the on-disk WiredTiger instance if the filter returns a positive result. The Aeris filter employs SplinterDB [12], a disk-resident and write-optimized key value store, with a 64 MB cache as its reverse map. To offset the additional memory used by the reverse map cache, we reduce the memory allocated to the WiredTiger cache in the Aeris filter setup by the same amount.

Adversarial workloads. We measure the impact of an adversarial query workload on the database. This workload simulates an attacker aiming to degrade database performance by repeatedly issuing false positive queries that will induce redundant false disk accesses. The attacker collects false positives by randomly issuing queries and then measuring the difference in response latencies to determine which ones accessed the disk. A slow response to a negative query indicates that the query was a false positive. The attacker then attempts to degrade the system performance by repeating the collected false positive queries to force the database to access the disk. To prevent false positive from being served from the cache, the attacker continues issuing random queries while periodically injecting false positives.

We setup the test by loading the system (database and filter) with 100 million 64-bit keys drawn uniformly at random. For each key, we also generate a random 504 byte value, resulting in the database storing 512-byte key-value pairs. The database is stored on disk, while the filter is kept in memory. For the query workload, we generate 200 million queries of the form $[x,x+R-1]$ for the query workload, where x is chosen uniformly at random and R is the length of the range query.

We run the test across different cache sizes. The cache size is measured as the ratio of the available memory to the dataset size, and we test for cache sizes of 1%, 5% and 10%. For the Aeris filter, the cache size includes the cache allocated to the database (WiredTiger) and the reverse map (SplinterDB), while for the other filters, the space is entirely allocated to WiredTiger's cache. We run our tests on three separate query lengths: point queries($R=1$), short range queries ($R=32$) and long range queries ($R=1024$). We also vary the rate at which adversarial queries are injected from 1% to 10%. We configure all the range filters to have the same false positive rate of $\epsilon=2^{-9}$ (roughly 0.1%).

The workload proceeds in two phases: a warm-up phase and an adversarial phase. In the warm-up phase, we perform the first half (100 million queries) of the query workload and record all the false positive results. In the adversarial phase, we perform the second half of the query workload while periodically replacing queries with the previously recorded false positives, according to the adversarial frequency.

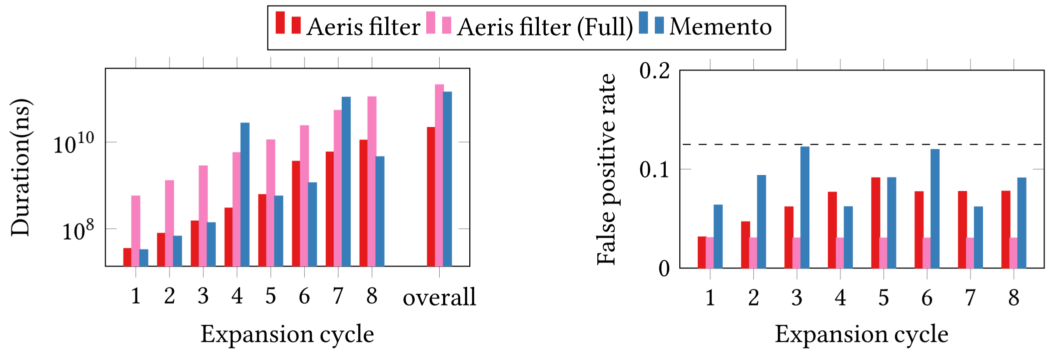
Figure 7 shows the overall application throughput for varying range query lengths, cache sizes, and adversarial query rates. As shown, an adversarial query rate of 1% is enough to cause performance degradation in databases using a non-adaptive range filter. On the other hand, the Aeris filter adapts to false positive queries in the warm-up phase, allowing the Aeris filter-based database to maintain high performance regardless of the percentage of adversarial queries in the workload.

With a cache size of 1%, an adversarial rate of 1% results in a 1.5× loss in performance for the non-adaptive filters. Increasing the adversarial rate results in further loss of performance — 4× at 5% and 8× at 10% adversarial frequency. Although increasing the cache size mitigates performance degradation in the database using a non-adaptive filter, a larger adversarial-query rate still overwhelms the system. In contrast, using the Aeris filter helps maintains a stable throughput that is $1.5-8\times$ higher, even with the smaller cache.

Table 2 reports the overall I/O (measured from `/proc/self/io`), for the adversarial test (5% cache size, 10% adversarial rate, $R=32$). The Aeris filter transfers almost $8-14\times$ fewer bytes compared to the other non-adaptive range filters, including the cost of accessing the reverse map to adapt to false positives.

Table 2 also reports the memory and disk usage of the filters. The reverse map of the Aeris filter occupies 5.1 GB on disk, a 10% overhead on the WiredTiger database consisting of 100 million pairs of 8-byte keys and 504-byte values occupying 50 GB of disk space. The Aeris filter uses slightly more main memory space compared to robust range filters: 6% over the Memento filter and 14% over the Grafite filter. The load factor of the Aeris filter increases from 90.0% to 90.2% over the course of the adversarial workload. The 0.2% increase in the load factor is from the slots used to store adaptivity bits. The Aeris filter adapts 390K false positives (200 million queries at a FPR of $\epsilon=2^{-9}$). Since the Aeris filter was initialized with 110 million slots (100 million items at 90% load factor), this is roughly 0.2% of the overall slots in the Aeris filter.

Expansions. We evaluate the time taken to expand the filter and the resulting false positive rate for the Aeris filter and the Memento filter across several expansion cycles. As more keys are inserted into



(a) Time to expand for filters. (Log scale, lower is better)

(b) False positive rate across expansions. (Lower is better). The theoretical maximum false positive rate is plotted as a dashed line.

Fig. 8. Filter expansion duration and false positive rate.

a filter with fixed initial capacity, it eventually runs out of space and must expand. These expansions are required to maintain a bounded false positive rate as the dataset grows.

For this workload, we initialize the filters with an initial capacity of $\frac{100}{2^8}$ million slots, and insert a total of 100 million keys. Each expansion doubles the filter capacity, resulting in a total of 8 expansions. We configure the filters to have a maximum false positive rate of $\epsilon = 2^{-3}$. To guarantee this false positive rate, the Memento filter uses fingerprint remainders of size $\log_2(\frac{1}{\epsilon})+1=4$ bits, while Aeris filter uses $\log_2(\frac{1}{\epsilon})+1+\log_2\log_2(\frac{1}{\epsilon})=5$ bits. The extra one bit in the remainder enables Aeris filter to guarantee strong adaptivity across expansions as explained in Section 3.3.

We compare the Aeris filter against the Memento filter as a baseline. The Memento filter uses InfiniFilter [16], a state-of-the-art filter expansion technique that uses variable-length fingerprints to achieve a good balance between expansion speed and false positive rate. In this method, the fingerprints from older expansions will eventually run out of bits, at which point the Memento filter will rebuild the filter by retrieving all the keys from the database.

To measure the impact of the deamortized expansion technique in the Aeris filter, we evaluate the Aeris filter in two variants. The first variant we refer to as Aeris filter (Full) expands the filter by rejuvenating all fingerprints using the reverse map. This variant is a baseline to demonstrate how the reverse map can also be employed to speed up the naïve expansion technique. It offers a lower false positive rate at the cost of slightly increased expansion time compared to the Memento filter. The other variant, which we refer to simply as the Aeris filter, deamortizes the I/O cost of expanding by rejuvenating only those fingerprints (approximately 2^{-r} fraction of keys, where r is remainder size) that have run out of bits as described in Section 3.3. The Aeris filter variants use the reverse map to rejuvenate the fingerprints, while the Memento filter uses WiredTiger to rejuvenate fingerprints.

Figure 8a plots the expansion time for the filters. For the first three expansion cycles, the Aeris filter and Memento filter take similar time to expand. Both these filters expand completely in memory using the InfiniFilter's [16] technique. None of the fingerprints in the filters have run out of bits. As a result, no I/O is needed to expand the filter. In contrast, the Aeris filter (Full) reads all the keys from the reverse map, resulting in a longer time to expand.

From cycle 4 onward, the Aeris filter uses the reverse map to selectively rejuvenate only those fingerprints (approximately 2^{-r} fraction of the fingerprints) that have run out of bits. In comparison, the Memento filter rejuvenates all fingerprints by performing a full database scan periodically (cycle

	I/O		Space Usage	
Filter	Read (MB)	Write (MB)	Memory (MB)	Disk (GB)
Aeris filter	15989.49	1058.33	227.06	5.1
Memento filter	129490.26	0	213.80	0
Grafite	129834.26	0	198.18	0
SuRF	118160.27	0	135.53	0
SNARF	238315.10	0	174.03	0

Table 2. Total I/O and space used by the filters in the adversarial test (5% cache size, 10% adversarial frequency, short range length queries ($R=32$)).

4 and 7 in Figure 8a) to retrieve all keys. A drawback of this approach is that it also requires reading the values associated with the keys, which amplifies the I/O cost to read keys. The Aeris filter (Full) rejuvenates all fingerprints using the reverse map, but avoids the read-amplifiaction problem faced by the Memento filter.

The Aeris filter completes all expansions in 3× less time compared to the Memento filter. Additionally, the Aeris filter deamortizes the I/O cost across expansions while the Memento filter periodically pays the full I/O cost of reading all keys every few expansion cycles. It is also worth pointing out that while the Aeris filter (Full) reads all keys from the reverse map on every expansion, it is is still only overall 1.47× slower compared to the Memento filter. The key difference is that the Aeris filter reads keys from the reverse map to rejuvenate fingerprints resulting in lower read-amplification while the Memento filter uses the underlying database.

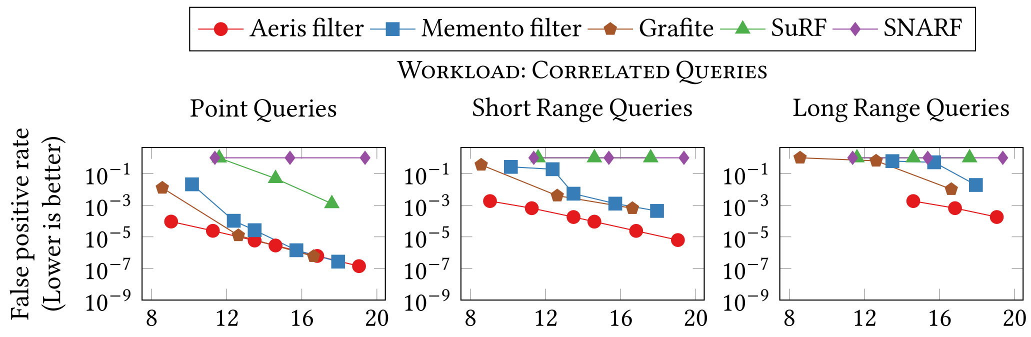
Figure 8b plots the instantaneous false positive rate for the filters. The false positive rate is measured using queries drawn from a uniform distribution. The horizontal line in the plots the theoretical maximum false positive rate ($\epsilon=2^{-3}$ in our experiment) guaranteed by the filters. The Aeris filter (Full) is able to achieve a $(2-4\times)$ lower and stable false positive rate with the trade-off of requiring more time for expansions. The Aeris filter, which is the deamortized I/O variant, has a lower false positive rate and lower expansion time compared to the Memento filter. While the Aeris filter has a higher false positive rate compared to the Aeris filter (Full) variant, it offers a balanced trade-off between false positive rate and expansion time. Note that, the Aeris filter still guarantees strong adaptivity with the overall false positive rate bounded by $\epsilon=2^{-3}$ across expansions.

Insertion. We measure the overhead on inserts of using a reverse map for adaptivity by measuring the throughput of insertions from a uniform random dataset. On a single thread, we measure a throughput of 3483.85 ops/sec for the Memento filter and 3456.85 ops/sec for the Aeris filter. In addition to inserting keys in the in-memory filter and the database, the Aeris filter also updates the reverse map. However, the Aeris filter employs the SplinterDB as the reverse which is write-optimized and results in negligible overhead (1% drop) in the insertion throughput compared to the database employing a non-adaptive filter.

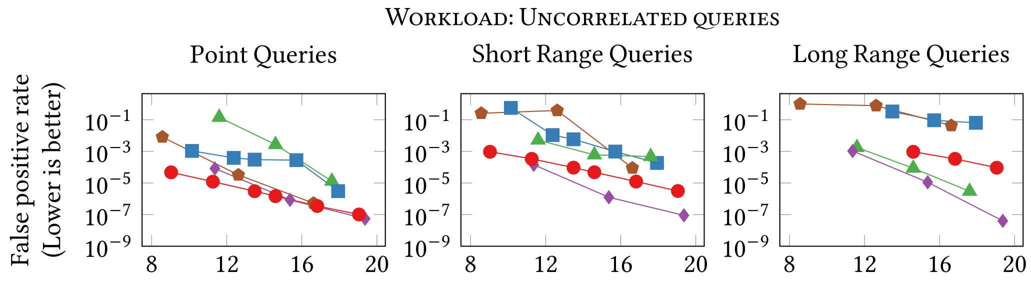
5.2 Microbenchmarks

We now evaluate the Aeris filter against other range filters as a standalone in-memory data structure to determine the CPU overheads of supporting adaptivity.

Query workloads. We generate workloads consisting of 200 million empty queries of the form $[x,x+R-1]$ using the following methods:



(a) False positive rate for a Zipfian distribution with queries drawn from a correlated workload (query endpoints are close to points in the dataset). The Aeris filter has the lowest false positive rate as it is able to adapt to repeated false positives. SuRF and SNARF have a high false positive rate (close to 1) as they are not robust to spatial skew.



(b) False positive rate for a Zipfian distribution with queries drawn from a uncorrelated workload (query endpoints are uniform randomly chosen; no spatial skew). The Aeris filter has a the lowest false positive rate among filters that guarantee a robust false positive rate against spatial skew.

Fig. 9. False positive rate versus space usage of filters on Zipfian query workloads. (Log scale, lower is better)

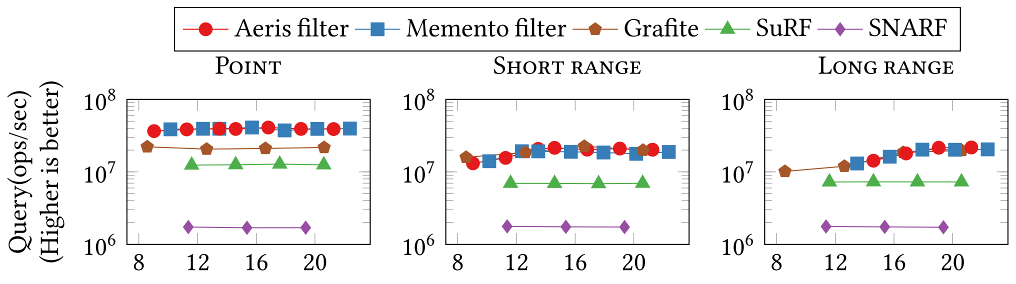


Fig. 10. Microbenchmark for query throughput on the filters on Zipfian distribution query workload (Log scale, higher is better).

- Correlated: We first choose x' by sampling with replacement from a Zipfian distribution using $\alpha = 1.5$, and then choosing x by uniform randomly sampling from $[x', x' + 2^{30(1-D)}]$, where, $D = 0.8$ is the correlation degree.
- Uncorrelated: x is chosen by sampling with replacement from a Zipfian distribution using $\alpha = 1.5$.

All benchmarks are repeated with three types of queries: point queries ($R = 1$), short-range queries ($R = 32$) and long-range queries ($R = 1024$). The Memento filter and Aeris filter use a memento size of

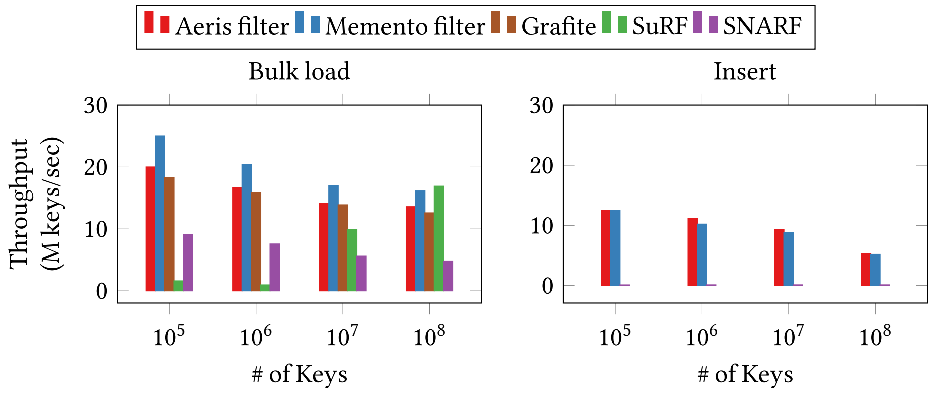


Fig. 11. Bulk load and insert microbenchmark performance. (Higher is better)

2 bits for point queries, short-range queries use a memento size of 5 bits and long-range queries use a memento size of 10 bits. All the workloads are run on a dataset of 100 million 64-bit keys chosen uniformly at random. The filters are constructed with a load factor of 90%.

False positive rate vs. memory for skewed workloads. In this benchmark, we evaluate the false positive rate and space usage trade-off for the filters. The false positive rate of the filters is evaluated by varying the memory budget per key from 8 to 20 bits/key.

Figure 9a plots the false positive rate and space usage trade-off for the filters on a Zipfian workload consisting of CORRELATED queries (queries with spatial skew). Among the filters evaluated, the Aeris filter, Memento filter and Grafite provide robust false positive guarantees while SuRF and SNARF have no such guarantees. SuRF and SNARF have a false positive rate close to 1 on this workload for range queries. In most practical filter configurations, where 8-16 bits are allocated per key, the Aeris filter has an order of magnitude lower false positive rate for similar space usage compared to the non-adaptive filters. This is due to the Aeris filter's ability to adapt to repeated false positives in a workload with a skewed distribution such as a Zipfian distribution.

Figure 9b plots the same false positive rate and space usage trade-off on a Zipfian workload consisting of UNCORRELATED queries (no spatial skew). SuRF and SNARF have a lower false positive rate in the absence of spatial skew, which is consistent with prior evaluation [14, 22]. Among the filters that provide robust guarantees the Aeris filter has the lowest false positive rate that is an order of magnitude lower than Grafite and Memento filter. **Thus, the Aeris filter is the only range filter that is robust against both spatial skew and temporal skew.**

Query performance. We also measure the query throughput of the filters and show the results in Figure 10. We only plot the throughput for the CORRELATED workload as they are similar to the results from the UNCORRELATED workload. The Aeris filter matches the best performing range filters on queries such as the Memento filter and Grafite, showing that the Aeris filter supports adaptivity without any loss in filter query performance. SNARF and SuRF have lower query performance throughput compared to the other filters.

Bulk construction and insert performance. Figure 11 shows the average bulk-load construction and insertion throughput of the filters. The throughput is measured as the median of multiple runs made with different memory budgets and datasets. The bulk-load construction throughput measures the rate at which the filter can be constructed when all keys are known in advance and sorted. Insertion throughput measures the rate at which keys can be inserted into the filter when they arrive dynamically and are not known in advance. Among the filters being evaluated, the Aeris filter, Memento filter and SNARF are dynamic filters, while Grafite and SuRF are static filters. The Aeris filter matches the performance of the state of the art range filters on both bulk-load and dynamic insertion performance.

6 Conclusion

The Aeris filter introduces a novel approach to adaptive range filtering, addressing key limitations of existing methods by ensuring strong and monotonic adaptivity while maintaining high performance. By leveraging variable-length fingerprints and integrating a reverse map, the Aeris filter significantly reduces false positives, improves query efficiency, and optimizes filter expansions. Our evaluation demonstrates that Aeris filter achieves up to a 10× reduction in false positive rates on skewed workloads while maintaining minimal memory overhead. Moreover, its integration with a production database results in 1.5×–8× higher throughput under adversarial conditions, highlighting its robustness in real-world applications.

A key insight from our work is that most traditional range filters struggle with adaptability due to their static structures and fixed-length fingerprints. The Aeris filter overcomes this by dynamically extending fingerprints only when needed while ensuring correctness. Additionally, our novel expansion strategy deamortizes I/O costs using the reverse map, reducing the overhead of filter growth by up to 3×. These contributions position the Aeris filter as an efficient and scalable solution for modern databases, search engines, and large-scale analytics systems that require adaptive range filtering.

With its strong and robust theoretical guarantees and practical efficiency, we expect Aeris filter to become the go-to filter data structure in modern databases. As we show in our evaluation, databases can offer robust performance guarantees even against skewed and adversarial workloads by employing the Aeris filter.

7 Acknowledgments

This research was funded in part by NSF grant OAC 2517201, 2513656 and NSERC grant RGPIN-2023-03580.

References

[1] Mangadevi Atti and Manas Kumar Yogi. 2024. Review of Variants of Bloom Filters for Detection of Malicious URL. *Journal of Intelligent Decision Technologies and Applications* 1, 1 (2024), 6–12.

[2] Michael A. Bender, Rathish Das, Martin Farach-Colton, Tianchi Mo, David Tench, and Yung Ping Wang. 2021. Mitigating False Positives in Filters: to Adapt or to Cache?. In *2nd Symposium on Algorithmic Principles of Computer Systems, APOCS 2020, Virtual Conference, January 13, 2021*, Michael Schapira (Ed.). SIAM, 16–24. doi:10.1137/1.9781611976489.2

[3] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. 2018. Bloom Filters, Adaptivity, and the Dictionary Problem. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, Mikkel Thorup (Ed.). IEEE Computer Society, 182–193. doi:10.1109/FOCS.2018.00026

[4] 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. *Proc. VLDB Endow.* 5, 11 (2012), 1627–1637. doi:10.14778/2350229.2350275

[5] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. doi:10.1145/362686.362692

[6] Gedare Bloom, Gianluca Cena, Ivan Cibrario Bertolotti, Tingting Hu, and Adriano Valenzano. 2017. Optimized event notification in CAN through in-frame replies and Bloom filters. In *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*. IEEE, 1–10.

[7] Gerth Stolting Brodal and Rolf Fagerberg. 2003. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Baltimore, Maryland) *(SODA '03)*. Society for Industrial and Applied Mathematics, USA, 546–554.

[8] Alex Carluccio. 2023. *Anonymous People Flow Monitoring System Leveraging Bloom Filters*. Ph. D. Dissertation. Politecnico di Torino.

[9] Guanduo Chen, Zhenying He, Meng Li, and Siqiang Luo. 2024. Oasis: An Optimal Disjoint Segmented Learned Range Filter. *Proc. VLDB Endow.* 17, 8 (may 2024), 1911–1924. doi:10.14778/3659437.3659447

[10] Rayan Chikhi, Jan Holub, and Paul Medvedev. 2021. Data Structures to Represent a Set of k-long DNA Sequences. *ACM Comput. Surv.* 54, 1, Article 17 (March 2021), 22 pages. doi:10.1145/3445967

[11] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[12] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P. Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[14] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2024. Grafite: Taming Adversarial Queries with Optimal Range Filters. *Proc. ACM Manag. Data* 2, 1 (2024), 3:1–3:23. doi:10.1145/3639258

[15] Transaction Processing Performance Council. 2011. TPC Benchmarks. http://www.tpc.org/information/benchmarks.asp.

[16] 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 (jun 2023), 27 pages. doi:10.1145/3589285

[17] Niv Dayan, Ioana-Oriana Bercea, and Rasmus Pagh. 2024. Aleph Filter: To Infinity in Constant Time. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 3644–3656. doi:10.14778/3681954.3682027

[18] Peter C. Dillinger and Panagiotis (Pete) Manolios. 2009. Fast, All-Purpose State Storage. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software* (Grenoble, France). Springer-Verlag, Berlin, Heidelberg, 12–31. doi:10.1007/978-3-642-02652-2_6

[19] Gil Einziger and Roy Friedman. 2016. Counting with TinyTable: Every Bit Counts!. In *Proceedings of the 17th International Conference on Distributed Computing and Networking* (Singapore, Singapore) *(ICDCN '16)*. Association for Computing Machinery, New York, NY, USA, Article 27, 10 pages. doi:10.1145/2833312.2833449

[20] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM* 21, 2 (1974), 246–260. doi:10.1145/321812.321820

[21] Navid Eslami, Ioana O. Bercea, and Niv Dayan. 2025. Diva: Dynamic Range Filter for Var-Length Keys and Queries. *Proc. VLDB Endow.* 18, 11 (Sept. 2025), 3923–3936. doi:10.14778/3749646.3749664

[22] Navid Eslami and Niv Dayan. 2024. Memento Filter: A Fast, Dynamic, and Robust Range Filter. *Proc. ACM Manag. Data* 2, 6 (2024), 244:1–244:27. doi:10.1145/3698820

[23] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and*

Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014, Aruna Seneviratne, Christophe Diot, Jim Kurose, Augustin Chaintreau, and Luigi Rizzo (Eds.). ACM, 75–88. doi:10.1145/2674005.2674994

[24] Zhuochen Fan, Bowen Ye, Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Zirui Liu, and Bin Cui. 2024. Enabling space-time efficient range queries with REncoder. *The VLDB Journal* (07 Aug 2024). doi:10.1007/s00778-024-00873-w

[25] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. 2017. BitFunnel: Revisiting Signatures for Search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Shinjuku, Tokyo, Japan) *(SIGIR '17)*. Association for Computing Machinery, New York, NY, USA, 605–614. doi:10.1145/3077136.3080789

[26] Mayank Goswami, Allan Grønlund Jørgensen, Kasper Green Larsen, and Rasmus Pagh. 2015. Approximate Range Emptiness in Constant Time and Optimal Space. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, Piotr Indyk (Ed.). SIAM, 769–775. doi:10.1137/1.9781611973730.52

[27] T. Kahveci and A. Singh. 2001. Variable Length Queries for Time Series Data. In *Proceedings 17th International Conference on Data Engineering* (2001-04). 273–282. doi:10.1109/ICDE.2001.914838

[28] Eric R. Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1670–1684. doi:10.1145/3514221.3526167

[29] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut Palm: Static and Streaming Data Series Exploration Now in Your Palm. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1941–1944. doi:10.1145/3299869.3320233

[30] Tsvi Kopelowitz, Samuel McCauley, and Ely Porat. 2021. Support Optimality and Adaptive Cuckoo Filters. In *Algorithms and Data Structures - 17th International Symposium, WADS 2021, Virtual Event, August 9-11, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12808)*, Anna Lubiw and Mohammad R. Salavatipour (Eds.). Springer, 556–570. doi:10.1007/978-3-030-83508-8_40

[31] Mandeep Kumar and Amritpal Singh. 2024. Anomalous vehicle recognition in smart cities using persistent bloom filter: Memory efficient and intelligent monitoring. *Transactions on Emerging Telecommunications Technologies* 35, 1 (2024), e4896.

[32] Cockroach Labs. 2015. https://github.com/cockroachdb/cockroach

[33] David J. Lee, Samuel McCauley, Shikha Singh, and Max Stein. 2021. Telescoping Filter: A Practical Adaptive Filter. In *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference) (LIPIcs, Vol. 204)*, Petra Mutzel, Rasmus Pagh, and Grzegorz Herman (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 60:1–60:18. doi:10.4230/LIPICS.ESA.2021.60

[34] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2071–2086. doi:10.1145/3318464.3389731

[35] Yuxin Meng and Lam-for Kwok. 2014. Adaptive blacklist-based packet filter with a statistic-based approach in network intrusion detection. *J. Netw. Comput. Appl.* 39 (2014), 83–92. doi:10.1016/J.JNCA.2013.05.009

[36] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2020. Adaptive Cuckoo Filters. *ACM J. Exp. Algorithmics* 25, Article 1.1 (March 2020), 20 pages. doi:10.1145/3339504

[37] MongoDB. [n. d.]. WiredTiger. https://github.com/wiredtiger/wiredtiger.

[38] Bernhard Mößner, Christian Riegger, Arthur Bernhardt, and Ilia Petrov. 2023. bloomRF: On Performing Range-Queries in Bloom-Filters with Piecewise-Monotone Hash Functions and Prefix Hashing. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 131–143. doi:10.48786/EDBT.2023.11

[39] Pat O'Neil, Betty O'Neil, and Xuedong Chen. 2007. The Star Schema Benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[40] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. 2005. An optimal Bloom filter replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 823–829. doi:10.5555/1070432.1070548

[41] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 775–787. doi:10.1145/3035918.3035963

[42] Prashant Pandey, Shikha Singh, Michael A. Bender, Jonathan W. Berry, Martin Farach-Colton, Rob Johnson, Thomas M. Kroeger, and Cynthia A. Phillips. 2020. Timely Reporting of Heavy Hitters using External Memory. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1431–1446. doi:10.1145/3318464.3380598

[43] Pedro Reviriego, Jim Apple, Álvaro Alonso, Otmar Ertl, and Niv Dayan. 2024. Cardinality Estimation Adaptive Cuckoo Filters (CE-ACF): Approximate Membership Check and Distinct Query Count for High-Speed Network Monitoring. *IEEE/ACM Trans. Netw.* 32, 2 (2024), 959–970. doi:10.1109/TNET.2023.3302306

[44] Pedro Reviriego, Miguel González, Niv Dayan, Gabriel Huecas, Shanshan Liu, and Fabrizio Lombardi. 2024. On the Security of Quotient Filters: Attacks and Potential Countermeasures. *IEEE Trans. Computers* 73, 9 (2024), 2165–2177. doi:10.1109/TC.2024.3371793

[45] RocksDB 2013. https://rocksdb.org/, Last Accessed Sept. 7, 2025.

[46] Russell Sears, Mark Callaghan, and Eric Brewer. 2008. Rose: compressed, log-structured replication. *Proc. VLDB Endow.* 1, 1 (Aug 2008), 526–537. doi:10.14778/1453856.1453914

[47] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: A Learning-Enhanced Range Filter. *Proc. VLDB Endow.* 15, 8 (apr 2022), 1632–1644. doi:10.14778/3529337.3529347

[48] Richard Wen, Hunter McCoy, David Tench, Guido Tagliavini, Michael A. Bender, Alex Conway, Martin Farach-Colton, Rob Johnson, and Prashant Pandey. 2024. Adaptive Quotient Filters. *Proc. ACM Manag. Data* 2, 4, Article 192 (Sept. 2024), 28 pages. doi:10.1145/3677128

[49] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees. *Proc. VLDB Endow.* 16, 11 (2023), 2976–2989. doi:10.14778/3611479.3611502

[50] Eleni Tzirita Zacharatou, Darius Šidlauskas, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2019. Efficient Bundled Spatial Range Queries. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19)*. Association for Computing Machinery, New York, NY, USA, 139–148. doi:10.1145/3347146.3359077

[51] 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. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 323–336. doi:10.1145/3183713.3196931

Received July 2025; revised October 2025; accepted November 2025