Dynamic Flat Filter: A Unified Framework for Scalable and Stable Fingerprint-Based Filters

YANG DU, Soochow University, China
SHANKUI JI, Soochow University, China
HE HUANG*, Soochow University, China
YU-E SUN, Soochow University, China
GE GAO, Soochow University, China
GUOJU GAO, Soochow University, China

Fingerprint-based filters, such as Cuckoo and Quotient Filters, are a key class of data structures for approximate membership query with deletion support. However, their fixed capacity is a major limitation in dynamic environments with unpredictable data volumes. While prior research has introduced dynamic filters, their resizing mechanisms are tightly coupled to specific filter structures, making their mechanisms difficult to generalize. Moreover, prior dynamic solutions introduce severe performance bottlenecks: chain-based or ring-based filters enable incremental growth yet degrade queries by probing multiple sub-filters, while global-resizing filters cause long, disruptive stalls during resizing as their single, large filter grows. In addition, prior incremental designs often leave scale-induced false positive rate growth unaddressed. In this paper, we present the Dynamic Flat Filter (DFF), a unified framework that decouples dynamic resizing from core filter logic, enabling fingerprint-based filters to achieve incremental scalability while preserving their original performance, memory efficiency, and false positive guarantees. DFF maintains a variable-sized set of independent segments, each reusing the base filter's native layout, enabling fine-grained resizing with only minor integration hooks. We achieve $O(1)$ insertion, query, and deletion by employing a flat structure for all segments and a lookup table that quickly maps any item to its corresponding segment. Additionally, DFF incorporates a fingerprint growth strategy that keeps the false positive rate essentially constant as the filter scales. We demonstrate the general applicability of DFF by integrating it with Cuckoo and Quotient Filters, each requiring fewer than 60 lines of code to be modified, and evaluate these implementations on real-world and synthetic datasets. Experimental results show that DFF outperforms SOTA baselines in insertion, query, and deletion, maintains high memory efficiency, and stabilizes the false positive rate. Notably, DFF attains at least 1.33× the overall throughput on query-intensive workloads and reduces worst-case insertion latency by up to 59.5% compared with the best prior dynamic filter.

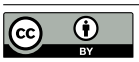
CCS Concepts: • **Theory of computation → Bloom filters and hashing**.

Additional Key Words and Phrases: Approximate Membership Query, Dynamic Filter

ACM Reference Format:
Yang Du, Shankui Ji, He Huang, Yu-E Sun, Ge Gao, and Guoju Gao. 2026. Dynamic Flat Filter: A Unified Framework for Scalable and Stable Fingerprint-Based Filters. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 166 (June 2026), 26 pages. https://doi.org/10.1145/3802043

*Corresponding Author.

Authors' Contact Information: Yang Du, Soochow University, Suzhou, Jiangsu, China, duyang@suda.edu.cn; Shankui Ji, Soochow University, Suzhou, Jiangsu, China, skjiskji@stu.suda.edu.cn; He Huang, Soochow University, Suzhou, Jiangsu, China, huangh@suda.edu.cn; Yu-E Sun, Soochow University, Suzhou, Jiangsu, China, sunye12@suda.edu.cn; Ge Gao, Soochow University, Suzhou, Jiangsu, China, ggao@stu.suda.edu.cn; Guoju Gao, Soochow University, Suzhou, Jiangsu, China, gjgao@suda.edu.cn.



ACM 2836-6573/2026/6-ART166
https://doi.org/10.1145/3802043

1 Introduction

Approximate Membership Query (AMQ) data structures, also known as *filters*, are fundamental building blocks for data-intensive systems [21, 25, 29, 36, 43, 48]. They provide a compact probabilistic representation of a set to efficiently test whether an element is present, with a controllable false positive rate. Because a filter is typically much smaller than the data it represents, it can be stored in fast memory (*e.g.*, SRAM), while the underlying dataset resides in slower storage. This allows systems to quickly answer membership queries without expensive data lookups, a property essential to high-performance applications such as key-value stores [6, 26, 27, 31, 34], web caches [22, 30, 37], and database engines [13, 14, 49].

The Bloom Filter (BF) [4] is the most widely known AMQ design. However, modern systems have largely migrated to *fingerprint-based filters*, such as the Cuckoo Filter (CF) [18] and the Quotient Filter (QF) [3]. By replacing bit arrays with compact per-item fingerprints, these filters enable deletions and maintain high memory efficiency. Crucially, fingerprint-based filter has diversified to address distinct system trade-offs [5, 45]. For instance, Cuckoo Filters are favored for fast random lookups, while Quotient Filters optimize for data locality in cache-sensitive or storage-bound workloads. No single filter design dominates across all metrics, making the choice of the underlying filter highly application-dependent.

However, most of the existing filters remain static in capacity, a mismatch with practical deployments that increasingly require feature-rich, dynamically scalable filters [40]. Real-world datasets, such as keys in storage engines [6, 22, 26, 31, 34] or entries in network telemetry systems [41, 42], evolve continuously, so filters must scale up or down efficiently. Existing fingerprint-based filters struggle with such dynamics: reconstructing a new, larger filter from the original dataset is often infeasible when data resides in slow or distributed storage. Dynamic resizing has been studied in three families, including global-resizing methods [11, 12, 45, 46], chain-based methods [2, 8, 20, 44], and ring-based methods [32, 33]. While these studies enable some degree of resizing, they often suffer from performance degradation: global-resizing methods trigger long blocking during resizes and reduce data locality as a single large filter grows, and chain- or ring-based methods achieve incremental growth but degrade queries by probing multiple sub-filters. Moreover, incremental designs often leave scale-induced false positive rate (FPR) growth unaddressed: FPR can increase either by accumulating checks across sub-filters or by consuming effective fingerprint bits for routing as the structure expands.

Beyond performance, a deeper limitation is **lack of generality**. In existing dynamic solutions, the resizing mechanism is often coupled with a specific filter variant. For example, a resizing strategy tailored for the random-probing Cuckoo Filter cannot be directly applied to the deterministic-probing Quotient Filter. This creates a dilemma for system designers: if they require the specific advantages of a specialized filter (*e.g.*, the data locality of QF), they cannot easily make it dynamic without re-engineering complex expansion logic. Therefore, a general expansion framework is not merely a convenience but a necessity. It allows applications to retain the distinct trade-offs of their chosen filter while gaining incremental scalability, thereby separating the selection of filter-specific performance characteristics from the logic of dynamic scalability.

This observation motivates a fundamental question: *Can we design a general dynamic framework that enables any fingerprint-based filter to scale incrementally, without compromising its algorithmic $O(1)$ performance or requiring structural redesign?* Such a framework would unify dynamic scalability across existing filters, simplify extensions, and eliminate the need for tightly coupled solutions.

In this paper, we present the Dynamic Flat Filter (DFF), a unified framework that decouples resizing from core filter logic, thereby enabling fingerprint-based filters to scale incrementally with dataset size while preserving the underlying filter's $O(1)$ operation performance and false

positive guarantees. DFF adopts a backend-agnostic design that treats the global structure as a variable-sized set of independent *segments*—each a sub-filter that *reuses the base filter's native layout* and collision-resolution logic—and resizes by adding or removing segments (§ 4.2). Therefore, DFF only requires minimal integration effort (§ 4.3) to adapt existing fingerprint-based filters, without redesigning their internal layout or collision-resolution logic. To make this segment-based abstraction effective, DFF introduces a directory-style lookup table (§ 4.4), inspired by extendible hashing (EH) [17], that provides single-hop, constant-time item-to-segment localization, thereby preserving $O(1)$ operations of the underlying filter. To control false positive growth as capacity scales, DFF employs a fingerprint-growth mechanism (§ 4.5) routed through a universal matcher interface (§ 4.3.2), allowing it to lengthen fingerprints in newer segments without redesigning internal layouts. Together, these mechanisms make DFF robust and easy to integrate: existing fingerprint-based filters require only minor adjustments yet gain incremental scalability.

Our contributions are summarized as follows:

- We present a novel AMQ framework, the Dynamic Flat Filter (DFF), enabling fingerprint-based filters to scale incrementally with dataset size while preserving their original $O(1)$ operation performance and false positive guarantees.
- We provide a minimal compatibility interface that lets DFF wrap existing fingerprint-based filters with negligible changes, along with a pluggable matcher interface to support fingerprint growth in a generalized manner. We demonstrate ease of adoption by integrating DFF with Cuckoo and Quotient Filters, each requiring fewer than 60 lines of code to be modified.
- We design a dynamic flat structure with a lookup table that provides single-hop, constant-time item-to-segment localization, ensuring $O(1)$ insertion, query, and deletion operations.
- We evaluate the performance of DFF on both synthetic and real-world datasets. Experimental results show that DFF outperforms competitive baselines in insertion, query, and deletion operations, maintains high memory efficiency, and stabilizes the false positive rate via its fingerprint-growth strategy. Notably, DFF attains at least 1.33× the overall throughput on query-intensive workloads and reduces worst-case insertion latency by up to 59.5% compared with the best prior dynamic filter.

2 Background and Related Work

2.1 Basic Structures

The Bloom filter (BF) [4] is a fundamental data structure used for approximate membership queries. It employs a bit array of size m, initially set to zero. During insertion, k independent hash functions map each item to k bits in the array, setting them to one. A drawback of BF is that it does not support item deletion.

The Cuckoo filter (CF) [18] improves upon the Bloom filter (BF) by supporting item deletion. It stores an f-bit fingerprint for each item x, denoted η_x. CF maintains a table of m buckets with b slots each. For x, it computes two candidate buckets $B_1 = \text{hash}(x)$ and $B_2 = B_1 \oplus \text{hash}(\eta_x)$. Insertion first tries to place η_x into an empty slot of either bucket. If both are full, CF runs bounded evictions: pick a victim in the current bucket, swap it with η_x, then chase the evicted fingerprint to its alternative bucket, repeating until an empty slot is found or a preset eviction bound is reached. Queries and deletions check the two buckets and remove one matching fingerprint. CF supports expected $O(1)$ operations and is more memory-efficient than BF at low false positive rates.

The Quotient Filter (QF) [3] is another fingerprint-based design that also supports deletions. It splits each item's hash into a quotient q (bucket index) and remainder r, storing r in a contiguous run for q with small metadata to delimit runs. Collisions are resolved by linear probing within that run, which enhances spatial locality. This spatial locality usually yields higher query throughput than

CF—especially at moderate load and on storage devices—but performance degrades as occupancy rises and run lengths grow.

2.2 Dynamic Filters

2.2.1 Dynamic Filters by Global Resizing. The Vacuum Filter (VF) [45] employs the same underlying data structure as CF. Periodically, VF reconstructs the entire filter by scanning the original data to accommodate dynamic datasets. However, this procedure is often bottlenecked by slow access to the original dataset, which typically resides in SSDs or remote storage.

Built on Bloom Filter (BF), the Elastic Bloom filter (EBF) [46] supports expansion and compression without accessing the original keys. EBF maintains an elastic fingerprint per item that dynamically releases or absorbs bits during resizing. During expansion, EBF builds a larger filter by reallocating all fingerprints and consuming one extra bit from each. Deletion removes the fingerprint and clears the corresponding bits in the BF. However, EBF incurs space overhead for fingerprints, and full-filter reallocation during expansion can cause significant blocking, resulting in extreme tail latency.

The InfiniFilter (IFF) [11] and Aleph Filter (AF) [12] are QF-based dynamic filters, with AF being an improved successor of IFF. When reaching capacity, they build a new QF with double the capacity and transfer one bit from each fingerprint to the slot index, allowing the new QF to hold twice as many items without accessing the original keys. They employ fingerprint growth by assigning longer fingerprints to new entries to control FPR as the dataset scales.

To handle rare cases where an initial fingerprint exhausts its indexable bits during expansion, IFF and AF add a short chain of QFs. This chain is typically very short and the extra filters are small, so most data remains in a single dominant QF. Thus, we still categorize IFF/AF as global-resizing approaches. As a result, they can incur substantial operation blocking during large expansions (similar to EBF). Moreover, as the dominant QF grows, clusters may span multiple cache lines or pages, reducing locality and degrading performance.

From the perspective of accuracy, global-resizing methods that resize without re-accessing original keys (*e.g.*, EBF, IFF, and AF) face a common challenge: as the structure grows, more hash-derived bits are effectively used for addressing, which would cause FPR to increase if fingerprints remained fixed-length. To stabilize FPR under growth, EBF relies on elastic fingerprints, while IFF/AF adopt fingerprint growth. In both cases, improved FPR stability comes with extra per-item space (fingerprint/metadata bits).

2.2.2 Dynamic Filters by Chained Structures. The Dynamic Bloom Filter (DBF) [20] and Dynamic Cuckoo Filter (DCF) [8] maintain a linked list of homogeneous sub-filters: DBF uses Counting Bloom Filters (CBFs) [19] and DCF uses Cuckoo Filters (CFs) [18]. They start with one sub-filter and append a new one when the active sub-filter reaches a preset threshold (*e.g.*, item count or load factor). DCF may also expand upon insertion failure due to CF's eviction bound. Queries and deletions probe sub-filters sequentially, so latency grows linearly with the number of sub-filters.

To reduce linear probing, the Logarithmic Dynamic Cuckoo Filter (LDCF) [47] organizes CFs as a tree, using fingerprint prefix bits to select the candidate CF at each level. Its compacted variant keeps only the top level to save space. However, both LDCF and its compacted version must traverse the tree to locate the target segment, yielding $O(\log N)$ time complexity for query and deletion.

Inspired by linear hashing [28], the Bamboo Filter (BBF) [44] organizes items into segments and introduces "partial-key linear hashing" to split one segment at a time, achieving amortized $O(1)$ operations. However, like linear hashing, BBF splits segments in order and relies on overflow chains to handle overflows that occur before a segment's turn to split. Hence, queries may still probe a few segments, and overflow segments often run at lower load factors, slightly increasing memory cost.

In terms of accuracy and space, chained designs tend to see FPR grow with scale: the overall FPR accumulates when queries probe multiple sub-filters, and some variants also consume fingerprint/prefix bits for localization, reducing effective discrimination. To the best of our knowledge, this scale-induced FPR growth has not been explicitly addressed in prior chain-based designs.

2.2.3 Dynamic Filters by Consistent Hashing Rings. Beyond chain-based methods, ring-based methods place sub-filters or buckets on a consistent hashing ring [23, 24], a classic load-balancing technique in distributed systems. The Consistent Cuckoo Filter [32, 33] and Bucket-level Elastic Cuckoo Filter [39] use this ring to support bucket-level expansion as item counts change. However, locating the responsible node on the ring requires an $O(\log N)$ search over virtual nodes and multiple hops along the ring.

2.2.4 Analysis. Recent advances have improved dynamic support for fingerprint-based filters, but fundamental trade-offs remain. Global-resizing methods reallocate all items during resizing, causing substantial blocking. They also collect items in a single growing filter, worsening data locality and degrading performance. Chain- and ring-based methods scale by appending sub-filters, so queries and deletions must traverse multiple substructures and cannot maintain constant-time localization. In terms of accuracy, scale-induced FPR growth under continued expansion remains underexplored beyond global-resizing designs. In addition, existing approaches are tightly coupled to specific filter internals, hindering portability and reuse across designs. Consequently, the field still lacks a general, efficient, reusable resizing mechanism.

3 Problem Statement and Design Goals

3.1 Problem Statement

An approximate membership query (AMQ) structure answers whether an item belongs to a set with no false negatives and a bounded false positive rate. We study the dynamic setting: starting from an initial capacity c chosen independently of the eventual data size, the structure maintains a changing set S under insertions and deletions, and ideally keeps its false positive rate stable under unbounded growth (potentially by using additional bits as needed). When insertions push occupancy beyond c, it must grow to larger capacities (c_2, c_3, *etc.*). When insertions and deletions are roughly balanced, capacity should stabilize near the working-set size. When deletions dominate, it should shrink and reclaim space. Resizing should be incremental and rely only on information stored in the filter, without re-scanning the original dataset.

3.2 Our Goal

Our goal is to design a unified framework that enables fingerprint-based filters to scale incrementally with dataset size while preserving the underlying filter's $O(1)$ operation performance, memory efficiency, and false positive guarantees of their static counterparts. Concretely, we target the following properties:

- **Broad compatibility with minimal integration effort.** We aim to design a framework that can seamlessly integrate with a wide range of existing and future fingerprint-based filters, with minimal modifications to their internal logic.
- **Incremental and memory-efficient resizing.** We aim to enable fine-grained resizing operations that enhance memory efficiency, avoiding long blocking events and reducing tail latency during growth and shrinkage.
- **Constant-time complexity for core operations.** We aim to keep the time complexity of insertion, query, and deletion at $O(1)$ whenever the static design offers $O(1)$ per-operation cost.

- **Stable false positive rate under growth.** We aim to provide an optional mechanism to keep the false positive rate stable under growth, trading additional space (and potential overhead) for improved accuracy.
- **High throughput.** We aim to achieve high throughput across all operations (*i.e.*, insertion, query, and deletion), in addition to low tail latency during resizing.

4 The Dynamic Flat Filter Framework

Here, we provide a comprehensive overview of the Dynamic Flat Filter (DFF), detailing its core concept, data structure, and the lookup-table-based localization. We then present DFF's operations (insertion, query, deletion) and scaling mechanism, and discuss an optional matcher-based fingerprint-growth mechanism that stabilizes false positive rate under growth.

4.1 Main Idea

Our main idea is to develop a unified framework that decouples dynamic resizing from core filter logic, thereby enabling fingerprint-based filters to evolve capacity incrementally as the dataset scales, while retaining the same $O(1)$ operation performance, memory efficiency, and false positive guarantees as their original static form. DFF adopts a backend-agnostic design that *relies solely on a minimal compatibility contract that most existing designs already satisfy* (§ 4.3). It treats the global structure as a variable-sized set of independent "segments"—each a sub-filter that reuses the underlying filter's native layout—and adjusts capacity via fine-grained segment splits and reclamation, avoiding global rebuilds and reducing tail latency (§ 4.2). Since each segment runs its native logic unchanged, resizing is limited to high-level coordination rather than structural changes, enabling seamless integration of diverse fingerprint-based filters (*e.g.*, CF, QF) without redesigning their internal layouts.

However, this segment-based abstraction alone is not sufficient, and two key challenges remain. First, DFF must preserve constant-time item-to-segment localization as segments are continuously added and removed. Without such $O(1)$ localization, fingerprint-based filters that rely on constant-time operations (*e.g.*, CF and QF) would degrade. Existing approaches that organize segments into chains or rings and search across them cannot preserve constant-time behavior as the structure grows. Second, DFF must control scale-induced false positive rate (FPR) growth. While incremental resizing avoids global rebuilds, it can introduce accuracy side effects: FPR may rise not only from accumulating checks across segments, but also from consuming effective fingerprint bits for routing as the structure expands. Therefore, DFF needs mechanisms to bound and stabilize FPR under growth without major changes to filter logic.

To achieve constant-time localization, we introduce a novel directory-style lookup table (LUT) tailored to the partial-key regime of fingerprint-based filters. Inspired by extendible hashing (EH) [17], the LUT uses hash-prefix routing and directory growth but does not store or relocate full keys: routing and resizing are driven solely by hash prefixes and the fingerprints stored inside segments (§ 4.4). Each operation thus proceeds in two steps: (i) locate the segment via a single-hop LUT access, and (ii) perform insertion, query, or deletion within that segment (§ 4.6). Consequently, the end-to-end cost remains $O(1)$, matching the underlying filter.

However, this EH-style hash-prefix routing also introduces an accuracy challenge: as the structure expands, items assigned to the same segment share longer routing prefixes, which reduces the distinguishing fingerprint entropy and can increase the false positive rate (FPR). To address this, DFF supports an optional *fingerprint growth* mechanism inspired by InfiniFilter [11], which allocates longer fingerprints to newer segments to stabilize FPR under growth. Our key innovation is a universal matcher interface (§ 4.3.2) that decouples fingerprint derivation and matching from

segment storage, allowing DFF to apply fingerprint-growth policies across segments without redesigning their in-segment layouts.

These mechanisms make DFF robust and easy to integrate: existing fingerprint-based filters require only minor adjustments yet gain incremental scalability without sacrificing memory efficiency or false positive guarantees. The following sections detail the architecture and mechanisms.

4.2 Architecture and Workflow

The Dynamic Flat Filter (DFF) organizes a dynamic filter as a flat collection of fixed-capacity segments plus a directory-style lookup table (LUT). As depicted in Figure 1, the first layer is a dynamic set of N segments $S = \{s_1, s_2, \ldots, s_N\}$, where each segment is an instance of a backend fingerprint-based filters (*e.g.*, Cuckoo and Quotient filters [3, 18]) with capacity c. DFF scales by adding new segments (and reclaiming empty ones), so its total capacity evolves with N, while each segment preserves the backend's native in-segment layout and collision-resolution logic.

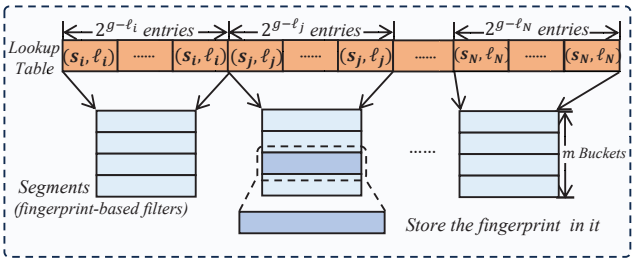


Fig. 1. The framework of the Dynamic Flat Filter.

The lookup table (LUT) provides one-hop item-to-segment localization. We denote the LUT as an EH-style directory $E[\cdot]$ (*i.e.*, an array) with 2^g entries, where g controls the initial entry assignment and each entry stores a pointer (*i.e.*, a segment ID) and multiple entries may point to the same segment. Each segment s_i has a local depth $\ell_i \leq g$ (*i.e.*, it has been split ℓ_i times) and therefore corresponds to a contiguous directory range of size $2^{g-\ell_i}$ consecutive entries in $E[\cdot]$ point to s_i. For an item x, DFF computes a fixed-width hash $h(x)$ and uses the g most significant bits (MSBs) of $h(x)$ as the directory index, *i.e.*, $idx(x) = h(x) \gg (|h(x)| - g)$; it then reads $E[idx(x)]$ to locate the responsible segment and invokes that segment's native operation. As the structure grows, DFF updates the directory following standard Extendible Hashing (EH): when a segment s_i splits, its local depth increases to $\ell_i + 1$ and its directory range is halved and rewired to the two post-split segments; if $\ell_i = g$, DFF first doubles the directory (increasing g by one) and then applies the same halving update. Our implementation also includes an optional optimization that reserves r least significant bits (LSBs) of $h(x)$ for initial partitioning, reducing MSB routing-bit consumption in exchange for more initial segments (§ 4.4).

However, this prefix-based routing also reduces in-segment fingerprint discrimination: for a segment s_i with local depth ℓ_i, all items routed to s_i share the same ℓ_i MSBs of $h(x)$, so using these MSBs for directory indexing effectively leaves fewer bits in an f-bit fingerprint to distinguish items within s_i. Consequently, the segment-level false positive rate (FPR) can increase as ℓ_i grows. DFF addresses this via an optional matcher interface (§ 4.3.2) that decouples fingerprint generation/matching from segment storage: it can share the routing MSB prefix as segment metadata and apply a fingerprint-growth policy (§ 4.5) that assigns longer fingerprints to newly created segments during expansion to stabilize the global FPR. For throughput-sensitive settings, DFF still supports fixed-length fingerprints and redundantly stores the routing MSB bits in each fingerprint to enable fast equality-based matching.

Because each operation touches exactly one segment via a single LUT lookup, DFF preserves the expected $O(1)$ insertion, query, and deletion complexity of the underlying static filter, while optionally stabilizing FPR under growth via fingerprint growth.

4.3 Framework Compatibility Interface

DFF is designed as a general-purpose framework that can accommodate a wide range of existing and future fingerprint-based filters. To achieve this broad compatibility, DFF treats each segment as an instantiation of a backend filter and interacts with it through a small set of standardized primitives, preserving the backend's in-segment layout and collision-resolution logic. This section specifies the minimal integration interface required by DFF.

4.3.1 Filter Primitives. As summarized in Table 1, DFF relies on only a few fundamental operations that are commonly supported by most fingerprint-based filters. Most of these primitives are already standard in existing designs: Insert, Query, and Delete are ubiquitous across AMQ structures, with Init and IsFull likewise present in any fingerprint-based filter.

Table 1. Minimal filter primitives used by DFF.

Op	Semantics
Init(cap, fp_bits)	Create empty filter with capacity and fingerprint length.
Insert(element)	Generate fingerprint and place in a candidate slot.
Query(element)	Match the element over fingerprints in candidate slots.
Delete(element)	Match and remove one fingerprint.
IsFull()	Returns true if load factor threshold reached.
Split(dst, Decide(fp))	Split current filter into itself and a new one, redistributing fingerprints based on a decision function.

The only novel primitive is Split, which is a simple composition of three operations already present in most fingerprint-based filters: enumerating stored fingerprints in a segment, deleting a chosen fingerprint, and inserting a fingerprint into a new segment. Guided by a decision function $\text{Decide}(fp) \rightarrow \{\text{MOVE}, \text{COPY}, \text{NONE}\}$, Split scans the source once and, for each fp, performs $\text{Delete}(fp)$ then $\text{Insert}(fp)$ into the destination on MOVE, inserts into the destination while keeping the original on COPY, or skips on NONE, thus reusing existing code paths without altering in-segment layouts.

4.3.2 Adapting to Fingerprint Growth. To support fingerprint growth (§ 4.5) while keeping segments backend-agnostic, DFF decouples fingerprinting logic (how fingerprints are generated and matched) from storage logic (how fingerprints are stored) via an optional *matcher* interface.

The matcher provides three functions used by Insert, Query, and Delete: (i) $\text{GenFp}(h, fp\_bits)$ derives a fingerprint from the item's hash; (ii) $\text{MatchFp}(h, fp)$ tests whether a stored fingerprint matches the item's hash; and (iii) $\text{PickVictim}(fp\_list)$ selects a fingerprint to remove when multiple candidates match during deletion. Accordingly, Init accepts an additional matcher parameter, and the three operations invoke the matcher for all fingerprinting tasks.

Some backends also derive auxiliary addresses from fingerprints (*e.g.*, the alternate bucket in Cuckoo Filters). For these filters, the matcher optionally provides $\text{GetOriginalFp}(fp)$ to expose a stable fingerprint view for addressing. In addition, to support fingerprint growth during segment splits, Split also accepts an additional $\text{Transform}(fp)$ callback, which can modify each fingerprint before inserting it into the new segment (*e.g.*, to lengthen or adjust it).

Integrating fingerprint growth requires only minimal edits and does not affect core data structures or control flow: replace inlined fingerprint generation and comparison with calls to these functions. In practice, this amounts to just a few lines of changes, with no modifications to the segment layout.

4.3.3 Ease of Adoption. Building on the primitives and the pluggable matcher defined above, the integration surface is intentionally small and non-intrusive. To demonstrate its generality, we integrated our DFF framework with two popular and distinct filters: the Cuckoo Filter (CF) and the Quotient Filter (QF). As shown in Table 2, achieving full compliance required modifying *fewer than 60 lines of code* in each case, without altering their core data structures or complex internal logic. This confirms that DFF can be adopted by existing filters with minimal engineering effort.

Table 2. # of code changed to integrate DFF with CF and QF.

Filter	Lines Changed	Language	Total Lines
Cuckoo Filter (CF)	58	C++	500+
Quotient Filter (QF)	54	C++	500+

4.4 Design Features of Lookup Table

In this section, we will discuss the three design features of the lookup table, which are *constant-time addressing*, *halving update*, and *directory doubling expansion*, jointly supporting DFF to resize while ensuring constant-time operations dynamically.

4.4.1 Constant-time Addressing. DFF must compute, for each item x, the directory index $idx(x)$ in $O(1)$ time even as segments split and the directory grows. Let $h_x = h(x)$ be a fixed-width hash word (*e.g.*, 32 bits). In a plain extendible-hashing (EH) directory with global depth g (*i.e.*, the number of entries is 2^g), the directory index is obtained by extracting the g most significant bits (MSBs) of h_x: $idx_{\text{EH}}(x) = h_x \gg (|h_x| - g)$. Intuitively, the expression $h_x \gg (|h_x| - g)$ discards the lower $|h_x| - g$ bits and keeps exactly the top g routing bits, yielding an integer in $[0, 2^g)$.

Our implementation also includes an optional optimization that reserves r least significant bits (LSBs) of $h(x)$ for initial partitioning, reducing MSB routing-bit consumption and mitigating FPR growth in exchange for more initial segments. Specifically, we reserve r least significant bits (LSBs) of h_x to select an initial group, where r is determined by the initial number of segments. The remaining directory routing is then performed by EH on the MSBs. Concretely, we first compute the group ID $grp(x) = h_x \& (2^r - 1)$, and then compute the EH index within the group using the current global depth g: $msb(x) = h_x \gg (|h_x| - g)$. Finally, the full LUT index is formed by concatenating the group bits and the EH bits: $idx(x) = (grp(x) \ll g) \mid msb(x)$, *i.e.*,

$$idx(x) = \Big(\big(h_x \& (2^r - 1)\big) \ll g\Big) \mid \Big(h_x \gg (|h_x| - g)\Big) \tag{1}$$

DFF then locates the target segment by reading $E[idx(x)]$. This computation uses only bit operations and remains constant-time.

4.4.2 Halving Update. When a segment s_i becomes full, DFF splits it into two segments: the original segment s_i and a newly allocated segment s_j. Let s_i have local depth ℓ_i and occupy a contiguous directory range of size $2^{g-\ell_i}$, we denote by e_i the starting index of this range, so entries $E[e_i], E[e_i + 1], \ldots, E[e_i + 2^{g-\ell_i} - 1]$ all point to s_i. A split increases the local depth to $\ell_i + 1$ and rewires exactly half of this range to the new segment: the lower half continues to point to s_i, while the upper half is updated to point to s_j. This exposes one additional routing bit for distinguishing s_i and s_j. To remain consistent with the directory update, DFF also redistributes the fingerprints stored in s_i: for each stored item x, we evaluate the next routing bit of its hash (the $(\ell_i + 1)$-th MSB

of $h(x)$); items with this bit equal to 0 remain in s_i, and items with this bit equal to 1 are moved to s_j (or copied when required to avoid false negatives) via the segment's Split primitive.

Notably, DFF derives fingerprints from the MSBs of $h(x)$, so the next routing bit used by the directory is consistent with the next bit available from the stored fingerprint. Therefore, although addressing uses $h(x)$ while redistribution during Split only inspects stored fingerprints, both follow the same routing rule. Using MSB-derived fingerprints also fits naturally with fingerprint growth, since newly extended fingerprints preserve the same MSB prefix.

4.4.3 Directory Doubling Expansion. A segment split requires one more routing bit to distinguish the two post-split segments. Therefore, when a full segment s_i already has local depth $\ell_i = g$, the current directory does not expose enough MSB routing bits to perform the next split. In this case, DFF expands the directory by doubling its size, increasing the global depth from g to $g + 1$. Concretely, each old directory entry is duplicated into two consecutive entries that both point to the same segment, *i.e.*, $E'[2j] = E[j]$ and $E'[2j + 1] = E[j]$ for all $j \in [0, 2^g)$. For example, $\{E[0], E[1], E[2], E[3]\}$ maps to $\{E[0], E[0], E[1], E[1], E[2], E[2], E[3], E[3]\}$. This preserves all existing mappings while making one additional routing bit available, so that the subsequent split can apply the halving update described above. With the optional LSB grouping optimization enabled, the same duplication is applied independently within each LSB group.

4.5 Fingerprint Growth Strategy

4.5.1 Motivation and Context. As discussed in § 4.4, DFF's lookup table consumes progressively more hash-prefix bits for routing as segments split. As a result, items mapped to the same segment share a longer prefix, leaving fewer effective bits in each segment's fingerprint for distinguishing items and causing false positive rate (FPR) to increase with scale. This phenomenon is inherent to dynamic approximate membership when the final set size is not known in advance: Pagh *et al.* [38] show that maintaining a constant target FPR during growth requires an additional $\Omega(\log \log n)$ bits per element beyond the $\Theta(\log(1/\epsilon))$ static bound. Practical dynamic filters explore different mechanisms to realize this growing-bit budget, including Elastic Bloom Filter [46] and fingerprint-growth-based designs such as InfiniFilter and Aleph Filter [11, 12]. Fingerprint growth has also recently appeared in dynamic range filters [15, 16] and expandable perfect hash table [35].

Motivated by the broad adoption of fingerprint growth for controlling FPR under dynamic expansion [11, 12, 15, 16, 35], DFF adopts fingerprint growth as an optional FPR-stabilization strategy. Fingerprint growth allocates longer fingerprints (*i.e.*, retains more hash-derived bits) to newer segments as the structure scales. DFF realizes this by increasing the target fingerprint length on expansion: when a segment splits, the post-split segments are initialized with (possibly) longer fingerprints under a configurable policy. We discuss the growth rule and its implications in § 4.5.4.

4.5.2 Fingerprint Representation. When a segment splits during expansion, approximately half of the resident fingerprints must be reassigned to the newly created segment. Under fingerprint growth, the new segment is configured with a longer target fingerprint length, so these migrated entries cannot be transferred verbatim. Meanwhile, DFF does not retain the original keys (nor full hashes), making it infeasible to regenerate longer fingerprints from scratch. Therefore, DFF converts migrated fingerprints into a format that remains compatible with the new segment while explicitly recording which bits are valid for future matching.

Specifically, DFF stores each fingerprint in a form consisting of a *valid payload* and an *age* marker. The payload (stored in the most significant bits) captures the hash-derived bits that were originally available when the item was inserted, while the age marker (stored in the least significant bits) encodes how many growth steps the fingerprint has undergone. We encode age in unary: the age suffix contains a single 1 bit followed by 0s. For example, for a stored fingerprint 10110100,

the suffix 100 is the unary age marker, whose age value is obtained by counting trailing zeros (CTZ), which is efficient on modern CPUs. The valid payload is then extracted by removing the age marker, *i.e.*, right-shifting the stored fingerprint by $age + 1$ bits. During a split, DFF applies a lightweight Transform to migrated fingerprints to update their age marker (and, depending on the configured growth policy, allocate additional payload bits in the new segment), enabling segments with different target fingerprint lengths to coexist without re-accessing original keys.

In addition, because LUT routing uses a shared MSB prefix per segment, DFF does not store these routing-prefix bits in every fingerprint. Instead, each segment keeps the prefix as metadata, and each stored fingerprint contains only the remaining payload bits and the age marker. During matching, the matcher combines the segment prefix with the extracted payload to form the comparable fingerprint, keeping matching consistent with LUT routing while reducing per-item redundancy.

4.5.3 Operations with Fingerprint Growth. For queries, the valid payload is extracted by removing the unary age marker, *i.e.*, right-shifting by $age + 1$ bits, where $age = \mathrm{CTZ}(fp)$. DFF also stores a per-segment routing prefix P (the MSB prefix implied by the local depth). A query matches by comparing the reconstructed prefix $P \| payload$ with the corresponding MSB prefix bits of $h(x)$.

For example, suppose the queried item's hash has MSB prefix 1011001 . . . , and a candidate stored fingerprint is $fp = 10110100$ (after including the segment's routing prefix). Here $\mathrm{CTZ}(fp) = 2$, so removing the age marker yields $payload = fp \gg (2 + 1) = 10110$. The query then checks whether $h(x)$ has the same prefix 10110 in the corresponding bits, which it does, indicating a match.

Deletion is more complex because we cannot delete the first prefix match. With fingerprint growth, multiple fingerprints of different ages may match under variable-length matching. Thus, DFF checks all backend-defined candidate slots (*e.g.*, all slots in the two CF buckets), collects the matches, and deletes the one with the smallest age (*i.e.*, the longest payload) via matcher.PickVictim.

For example, suppose we find two matching fingerprints, $fp_1 = 10110100$ and $fp_2 = 10110101$ (after including the segment's routing prefix), for an item hashed to 101101011 . . . . If we delete the first match fp_1 (age 2, payload 10110), we may introduce a false negative: fp_1 could belong to a different item (*e.g.*, 101100110 . . .) that shares the same prefix, while fp_2 is the correct representative for 101101011 . . . under fingerprint growth. Therefore, DFF must check all matches and delete the one with the smallest age, incurring modest overhead but avoiding false negatives.

4.5.4 Choosing Growth Policies. DFF's growth policy specifies the target fingerprint length for a segment as a function of its local depth ℓ_i (the number of splits applied to segment s_i). In particular, to stabilize FPR under continued growth, we adopt a logarithmic growth policy inspired by InfiniFilter [11]:

$$f(\ell_i) = f_0 + \left\lceil 2 \log_2(\ell_i + 1) \right\rceil \tag{2}$$

where f_0 is the initial fingerprint length. The underlying rationale is to distribute the overall false positive budget across "depth generations" so that older (shorter) and newer (longer) fingerprints can coexist without the total FPR drifting upward: we allocate the per-depth budget to decay polynomially, approximately as $\epsilon(\ell_i) \propto 1/(\ell_i + 1)^2$, and translate this budget into bits using the exponential dependence of FPR on effective bits (roughly $\epsilon \propto 2^{-f}$), which yields an additive growth of about $2 \log_2(\ell_i + 1)$ bits.

This policy grows sublinearly with the split count: $f(\ell_i) = f_0 + O(\log \ell_i)$. Since ℓ_i is at most logarithmic in the total cardinality under continued doubling-style growth, the additional fingerprint length is $O(\log \log n)$, aligning with the $\Omega(\log \log n)$ per-element overhead that is unavoidable for maintaining a stable target FPR when the final set size is unknown [38]. We evaluate the resulting accuracy/space trade-offs empirically in § 6.

4.6 Filter Operations

4.6.1 Insertion. Algorithm 1 outlines the process for inserting an item x into DFF. DFF first computes a fixed-width hash $h_x = h(x)$ and derives the directory index $idx(x)$ using the directory's global depth g by Equation 1. It then performs a single LUT lookup to locate the target segment s and invokes the backend filter's native Insert on s. If insertion fails (*e.g.*, due to bounded kickouts in CF) or s becomes full, DFF triggers expansion by splitting s (§ 4.7). This supports both QF-style backends (fail mainly when full) and CF-style backends (may fail before the load threshold).

Algorithm 1: Insert(x)

Input: The item to insert x

$h_x \leftarrow h(x)$;
$idx \leftarrow idx(x)$ by Equation 1;
$s \leftarrow E[idx]$;
ok $\leftarrow$ s.Insert(h_x);
if *not ok* ***or*** *s.IsFull()* **then**
  Expand(s) ; // § 4.7
end

Figure 2 illustrates insertion at a high level. DFF computes $h_x = h(x)$, locates the target segment via $idx(x)$ in Equation 1, and then invokes the backend's native Insert inside that segment. Query and Delete follow the same localization rule as depicted in Figure 2.

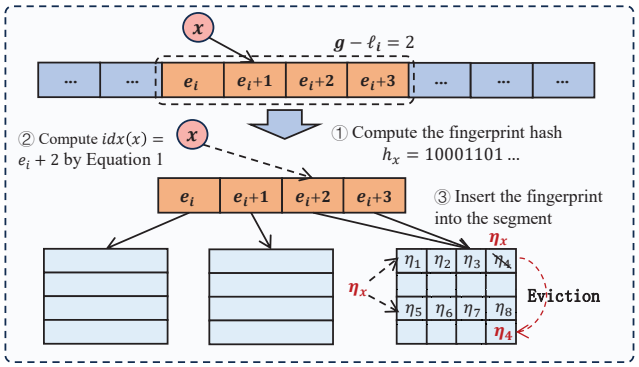


Fig. 2. The process of insertion (illustrated with CF).

4.6.2 Query. Algorithm 2 performs a membership query for an item x. DFF first computes a fixed-width hash $h_x = h(x)$ and derives the directory index $idx(x)$ by Equation 1. It then performs a single LUT lookup to locate the target segment $s \leftarrow E[idx(x)]$ and invokes the backend filter's native Query operation within that segment. When the optional matcher interface is enabled, the segment's Query uses matcher.MatchFp(h_x, fp) to test candidate fingerprints; otherwise, it falls back to the backend's default fingerprint comparison. Therefore, DFF preserves the same expected $O(1)$ query complexity as the underlying static filter.

For ease of exposition, we inline-expand s.Query(h_x) in Algorithm 2, where s.CandidateSlots(h_x) denotes the segment-determined candidate bucket(s)/slot(s). This expansion is only for illustration and does not mean DFF is limited to filters exposing such an API. § 4.6.3 similarly expands s.Delete(h_x) for clarity.

Algorithm 2: Query(x)

Input: The item to query x

$h_x \leftarrow h(x)$;
$idx \leftarrow idx(x)$ by Equation 1;
$s \leftarrow E[idx]$;
// Abstract demonstration of s.Query(h_x)
$C \leftarrow s$.CandidateSlots(h_x);
foreach $fp \in C$ **do**
    if *matcher.MatchFp(h_x, fp)* **then**
        return True;
    end
end
return False;

4.6.3 Deletion. Algorithm 3 outlines the deletion process. Similar to the query operation, DFF first computes a fixed-width hash $h_x = h(x)$ and locates the target segment via $idx(x)$ in Equation 1. It then invokes the backend filter's native Delete within that segment. When the optional matcher interface is enabled, deletion uses matcher.MatchFp(h_x, fp) to identify matching fingerprints and matcher.PickVictim($\cdot$) to select one to remove when multiple candidates match (*e.g.*, under fingerprint growth (§ 4.5)); otherwise, it typically reduces to a fast equality-based deletion.

Algorithm 3: Delete(x)

Input: The item to delete x

$h_x \leftarrow h(x)$;
$idx \leftarrow idx(x)$ by Equation 1;
$s \leftarrow E[idx]$;
// Abstract demonstration of s.Delete(h_x)
$C \leftarrow s$.CandidateSlots(h_x);
$Matches \leftarrow \emptyset$;
foreach $fp \in C$ **do**
    if *matcher.MatchFp(h_x, fp)* **then**
        append fp to $Matches$
    end
end
if $Matches = \emptyset$ **then**
    return False
end
idx $\leftarrow$ matcher.PickVictim($Matches$);
s.remove($Matches$[idx]);
return True

Additionally, the deletion operation may trigger a compression process, which reclaims the space of empty segments, serving as the inverse of space expansion. The deletion operation retains the same time complexity as the query operation.

4.7 Scaling Mechanism

4.7.1 Expansion. Expansion grows DFF incrementally by splitting a single full segment rather than resizing the entire filter. An expansion is triggered when an insertion targets a segment that is full (or the backend reports insertion failure, *e.g.*, due to bounded evictions in Cuckoo Filters). Let the full segment be s_i with local depth ℓ_i. DFF allocates a new empty segment s_j with the same backend type as s_i. If fingerprint growth is enabled, s_j may be initialized with a larger target fingerprint length according to the configured growth policy (§ 4.5.4). Algorithm 4 summarizes the expansion procedure, and Figure 3 provides an example.

Algorithm 4: Expand(s_i)

Input: The segment to expand s
Allocate a new empty segment s_j of the same backend type as s_i;
Let ℓ_i denote the local depth of s_i;
if *fingerprint growth enabled* **then**
  initialize s_j with target fingerprint length $f(\ell_i + 1)$ (§ 4.5.4);
end
if $\ell_i = g$ **then**
  DoubleDirectory() ; `// increase` $g \leftarrow g + 1$, § 4.4
end
Let e_i denote the starting index of s_i's directory range;
Rewire $\{E[e_i + 2^{g-\ell_i-1}], \ldots, E[e_i + 2^{g-\ell_i} - 1]\}$ to point to s_j;
Set local depth of both s_i and s_j to $\ell_i + 1$;
Split(s_i, s_j, Decide($\cdot$), Transform($\cdot$));
return True;

Before redistributing items, DFF ensures that the directory exposes enough routing bits to distinguish the post-split segments. If $\ell_i = g$, DFF first doubles the directory as described in § 4.4, increasing g by one while preserving existing mappings. DFF then performs the halving update: it rewires half of s_i's directory range to point to s_j and keeps the other half pointing to s_i (§ 4.4.2).

The core of expansion is fingerprint redistribution, implemented by the primitive Split(dst, Decide(fp), Transform(fp)) and driven by a decision function Decide(fp) $\rightarrow$ MOVE, COPY, NONE. For each stored fingerprint fp in s_i, DFF evaluates the next routing bit corresponding to the split, *i.e.*, the $(\ell_i + 1)$-th MSB of $h(x)$, and applies a rule: if the bit is 0, Decide returns NONE and the fingerprint remains in s_i; if the bit is 1, Decide returns MOVE and the fingerprint is migrated into s_j. When fingerprint growth is enabled, the migrated fingerprint is also passed through Transform(fp) before being inserted into s_j, updating its age marker and conforming it to s_j's target fingerprint length (§ 4.5). Notably, although addressing uses $h(x)$ while Split operates on stored fingerprints, the two remain consistent because fingerprints are derived from MSBs and DFF stores the segment's shared routing prefix as metadata (§ 4.5), allowing the next routing bit to be recovered for Decide.

Example of Expansion: To illustrate the expansion process (visualized in Figure 3), consider a directory with global depth $g = 3$ (containing $2^3 = 8$ entries). Suppose a segment s_i has local depth $\ell_i = 2$; this means it corresponds to a range of $2^{3-2} = 2$ directory pointers, specifically indices `000` and `001`. When s_i splits:

(1) **Update Directory:** The local depth of s_i increases to 3. The directory entry `000` continues to point to s_i, but the entry `001` is updated to point to the newly allocated segment s_j.
(2) **Redistribute Items:** We scan the items currently stored in s_i. For each item x, we inspect the $(\ell_i + 1)$-th MSB of its hash (in this case, the 3rd bit).

- If the bit is `0`, x belongs to index `000` and remains in s_i.
- If the bit is `1`, x belongs to index `001` and is moved to s_j.

This ensures that future lookups for hash prefix `001...` are routed directly to s_j without probing s_i.

Finally, DFF must handle the rare *fingerprint-exhaustion* case. During expansion, a stored fingerprint may no longer contain enough valid bits to determine the next routing bit (*e.g.*, when its valid payload is fully consumed by growth steps). In this case, Decide returns COPY, causing Split to insert a copy of the fingerprint into s_j while keeping it in s_i. This conservative duplication is necessary to avoid false negatives: since the item could belong to either post-split segment but the stored representation cannot disambiguate it, placing it in both ensures that future queries remain sound. Such COPY events affect only a small fraction of fingerprints and therefore have a negligible impact on the overall expansion cost. Additionally, since such fingerprints are duplicated across multiple segments, a deletion routed to a single segment cannot reliably remove all duplicates, potentially causing residual copies and FPR growth. However, given that fingerprint exhaustion affects only a very small fraction of items, this impact is typically negligible in practice.

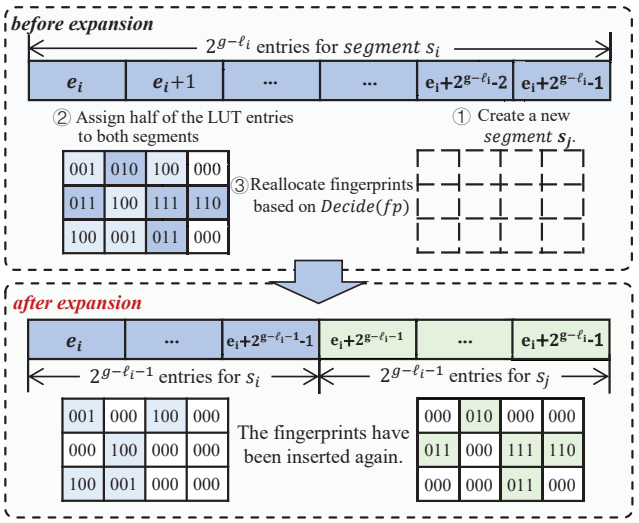


Fig. 3. The process of expansion.

Quantitative Analysis of Expansion Overhead: Unlike global resizing filters that incur memory overhead by rebuilding the entire structure during resizing, DFF decouples expansion costs from dataset size. Expansion in DFF is strictly bounded by the constant size of a single segment and the directory. For instance, in a workload of 10 million keys occupying ≈ 20 MB, splitting a standard DFF segment (2^{14} slots, 12-bit fingerprints) requires allocating only ≈ 24 KB—a negligible 0.12% overhead. Similarly, doubling a 1024-entry directory adds only ≈ 16 KB. Thus, DFF maintains a peak expansion overhead of mere kilobytes, whereas global-resizing-based baselines (*e.g.*, InfiniFilter or EBF) would require a transient memory spike of tens of megabytes to resize the same dataset.

4.7.2 Compression. Compression is the reverse of expansion and is checked upon deletions. After a deletion in segment s_j, DFF tests whether s_j and its buddy segment s_i (the sibling created by the most recent split) are jointly underutilized. If their combined load satisfies $\alpha_i + \alpha_j \leq \tau$ for a configurable threshold $\tau < 1$, DFF attempts to merge them by moving all fingerprints from one segment into the other using the backend's native insertion logic, rewiring the directory range to the surviving segment, and decreasing its local depth by one; if the merge fails (*e.g.*, due to insertion failure in CF backends), DFF aborts and leaves the structure unchanged.

5 Performance Analysis

In this section, we analyze the performance of DFF. Unless otherwise noted, we consider the CF-based instantiation, and the QF case is analogous and omitted for brevity.

5.1 Memory Footprint

The memory footprint of DFF is influenced by four primary factors: the number of stored items n, the initial fingerprint length f_0, the target load factor α, and the structural overhead of the lookup table. With fingerprint growth enabled, the target fingerprint length of a segment with local depth ℓ follows the policy in §4.5.4, *i.e.*, $f_0 + \lceil 2\log_2(\ell+1)\rceil$. Under continued expansion, the local depth grows only logarithmically with n (approximately $\ell \approx \log_2 n$), yielding an $O(\log\log n)$ additive increase in fingerprint length.

The model also accounts for directory (the lookup table) overhead. In an EH-style directory with global depth g, a segment of local depth ℓ is referenced by $2^{g-\ell}$ directory entries, each storing a 128-bit pointer payload (two 64-bit pointers in our implementation). Normalizing by the segment capacity $m\cdot b$ and load factor α, the amortized bits-per-key is approximated as:

$$\text{bits per entry} = \frac{f + \lceil 2\cdot\log_2(log_2(n)+1)\rceil + \frac{128\times(g-\ell)}{mb}}{\alpha} \tag{3}$$

Under a uniform hash, splits keep segments approximately balanced, so most segments have ℓ close to g and thus $2^{g-\ell}$ stays a small constant. Therefore, when amortized over a segment's $m\cdot b$ slots (and normalized by the load factor α), the directory contributes only a small bits-per-key overhead under typical segment sizes—typically below 0.1 bits-per-key in our experiments (§6.8).

5.2 False Positive Rate

We now derive an upper bound on the false positive rate (FPR) of DFF under the InfiniFilter-style fingerprint-growth policy (Equation 2). Let the directory (lookup table) have global depth g, and let each segment have local depth $\ell \le g$. We refer to segments with the same local depth ℓ as generation-ℓ segments, since each split increases a segment's local depth by one.

Consider the filter when the maximum local depth is g. For an item stored in a segment of local depth ℓ, $(g-\ell)$ hash-prefix bits are consumed by routing beyond that segment's prefix. Under our growth policy, the target fingerprint length is $f(\ell) = f_0 + \lceil 2\log_2(\ell+1)\rceil$, so the effective number of distinguishing bits at depth g is $f_{\text{eff}}(\ell) = f(\ell) - (g-\ell)$. The overall FPR can be written as $\mathbb{E}[\epsilon] \approx \alpha\sum_{\ell=0}^{g} w_\ell \cdot 2^{-f_{\text{eff}}(\ell)}$, where w_ℓ is the fraction of items stored in depth-ℓ segments. Under balanced EH-style splitting, the population at depth ℓ grows geometrically, yielding $w_\ell \approx 2^{\ell-g-1}$. Substituting w_ℓ and $f_{\text{eff}}(\ell)$ gives

$$\mathbb{E}[\epsilon] \approx \alpha\cdot 2^{-f_0-1}\sum_{\ell=0}^{g}\frac{1}{(\ell+1)^2}. \tag{4}$$

Since $\sum_{\ell\ge 0}(\ell+1)^{-2}$ converges to $\frac{\pi^2}{6}$ (the Basel problem), $\mathbb{E}[\epsilon]$ remains bounded by a constant proportional to $\alpha\cdot 2^{-f_0}$, independent of the final scale. This matches the asymptotic stability achieved by InfiniFilter and Aleph Filter and is consistent with the $\Omega(\log\log n)$ lower bound for dynamic approximate membership [38].

5.3 Time Cost

We evaluate the time complexity of operations in DFF by considering the segment-local cost and the directory routing overhead. Since each operation performs exactly one directory lookup to locate the target segment and then executes the native backend operation inside that segment, the routing overhead is a strict $O(1)$: computing $idx(x)$ uses a constant number of bit operations and

reading one LUT entry is a single array access. Therefore, the asymptotic complexity is determined by the backend segment.

For the CF-based instantiation, query and deletion inspect a constant number of candidate buckets (two buckets with b slots), hence they take expected $O(1)$ time. The insertion cost depends on the segment load factor α_i. Following the standard uniformity approximation used in prior CF analyses [45], a bucket is full with probability approximately α_i^b, so a bucket probe succeeds with probability $1 - \alpha_i^b$. If q_{α_i} denotes the number of probed buckets until the first non-full bucket is found, then q_{α_i} is geometrically distributed and satisfies

$$\mathbb{E}[q_{\alpha_i}] = (1 - \alpha_i^b) \cdot 1 + \alpha_i^b \cdot (\mathbb{E}[q_{\alpha_i}] + 1), \tag{5}$$

which yields $\mathbb{E}[q_{\alpha_i}] = 1/(1 - \alpha_i^b)$. To express the amortized insertion cost over a growth phase where a segment's load increases from 0 to a threshold $\alpha_{\max}$, we average this expectation over α:

$$\bar{Q}(\alpha_{\max}) = \frac{1}{\alpha_{\max}} \int_0^{\alpha_{\max}} \frac{1}{1 - x^b}\, dx. \tag{6}$$

For any fixed threshold $\alpha_{\max} < 1$ (*e.g.*, 0.90–0.95 in our configuration), the integrand is bounded by $1/(1 - \alpha_{\max}^b)$ for all $x \in [0, \alpha_{\max}]$, hence $\bar{Q}(\alpha_{\max}) \le 1/(1 - \alpha_{\max}^b) = O(1)$. Therefore, as long as each segment maintains a constant load-factor cap bounded away from 1, the amortized insertion cost remains constant. Combining this with the $O(1)$ directory routing overhead, DFF preserves the expected $O(1)$ time complexity of the underlying static filter for insert, query, and delete, up to a constant additive cost for LUT routing.

6 Performance evaluation

6.1 Experimental Settings

Datasets: The evaluation is performed on synthetic datasets and two highly skewed real-world datasets, as described below.

- *Synthetic Dataset:* We utilize pre-generated 64-bit distinct integers from random number generators as items to construct synthetic datasets.
- *YCSB:* We use the Yahoo! Cloud Serving Benchmark (YCSB) [10], a widely adopted open-source framework for evaluating key-value and cloud-serving workloads, to generate a skewed workload trace. Specifically, we generate 5,000,000 keys using a Zipfian distribution with parameter $\theta = 0.99$, and evaluate YCSB workloads C (read-only), D (read-latest), and F (read-modify-write). Since we evaluate an AMQ, we treat both READ and UPDATE as query operations. As a result, YCSB A/B/C (which only differ in the read/update ratio) collapse to the same filter-level workload, so we only report C, D, and F.
- *CAIDA:* We use the CAIDA network trace from a commercial backbone link [7], which exhibits a heavy-tailed flow popularity distribution. For each packet, we pack the source–destination IPv4 address pair into a 64-bit unsigned integer key and extract the first 1,000,000 keys as the dataset.

Baselines: We compare the Dynamic Flat Filter instantiated with Cuckoo and Quotient filters (referred to as **DFF-CF** and **DFF-QF**, respectively) and their fingerprint growth variants (referred to as **DFF-CF(FG)** and **DFF-QF(FG)**, respectively) against several other algorithms, including the Dynamic Bloom filter (**DBF**) [20], Dynamic Cuckoo filter (**DCF**) [8], Elastic Bloom filter (**EBF**) [46], Logarithmic Dynamic Cuckoo Filter (**LDCF**) [47], Bamboo filter (**BBF**) [44], InfiniFilter (**IFF**) [11] and Aleph Filter (**AF**) [12]. In our experiments, **DFF-CF/QF** disable fingerprint growth and keep fixed-length fingerprints, redundantly storing routing MSB bits in each fingerprint for fast equality-based matching (maximizing throughput), whereas **DFF-CF/QF(FG)** store the routing MSB prefix as per-segment metadata and apply the fingerprint growth policy in Equation 2 to stabilize the FPR

under growth. We have excluded the Consistent Cuckoo Filter [32, 33], which utilizes consistent hashing rings, from our comparison due to its lower performance in locating corresponding buckets.
Parameter Settings: Unless otherwise specified, we use unified default settings. The initial capacity is set to 2^{16} slots with an initial fingerprint length of $f_0 = 16$ bits (inclusive of the three metadata bits for QF), resulting in an initial filter size of 128 KB. We insert from 10×2^{16} up to 80×2^{16} items to exercise resizing, and per-segment expansion is triggered at a 90% load factor for all methods that support a load-factor threshold (DFF-CF/QF, BBF, DCF, LDCF, IFF, AF), while DBF and EBF follow their native expansion policies.
Implementation and Platforms: We implement DFF and all baselines in C++ to eliminate language-specific overheads and evaluate them under identical settings. We also run the original Java implementations of IFF and AF as a sanity check and observe consistent trends with our C++ ports, so we report only the C++ results in the figures. All implementations (including the original Java IFF/AF baselines) are available in our GitHub repository [9]. All algorithms were compiled with clang++ 21.1.5 (-O3) and evaluated on an Ubuntu 22.04 server with two Intel(R) Xeon(R) Gold 5317 3.60GHz CPUs and 384GB RAM.

6.2 Parameter Evaluation

DFF exposes a single framework parameter: the segment size m. Instantiations may also inherit filter-specific parameters. For instance, CF introduces the maximum eviction count M_e, whereas QF adds none. Accordingly, we evaluate how m and M_e affect the CF-based DFF. The QF case is analogous and omitted for brevity.

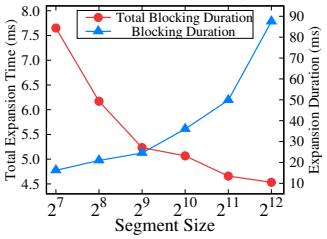


Fig. 4. Effect of segment size m.

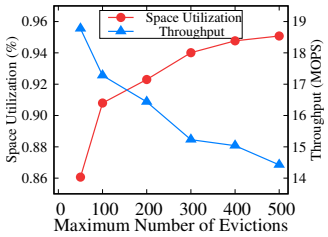


Fig. 5. Effect of maximum eviction count M_e.

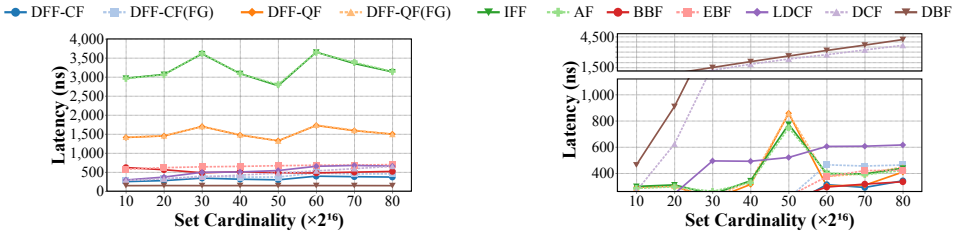
Impact of segment size m: Figure 4 shows how single-segment expansion time and the filter's total blocking time during expansion vary with segment size m. Smaller segments are associated with shorter single-segment expansion but longer total durations due to more frequent expansions.

Impact of maximum eviction count M_e: The maximum number of evictions, denoted as M_e, determines the number of attempts an item can make to find an empty slot. As shown in Figure 5, a higher value of M_e can alleviate hash conflicts, thereby resulting in higher space utilization. However, this may reduce insertion throughput due to the increased attempts during insertion.

Accordingly, in subsequent evaluations, we use $m = 2^{12}$ for all DFF variants and segment-based baselines to balance per-event blocking time and expansion frequency, and set $M_e = 500$ for CF-based algorithms to achieve high space utilization. All baselines utilize the same initial memory.

6.3 Operation Performance

Insertion. Figure 7(a) and Figure 6(a) show that DFF-CF achieves the highest insertion throughput, improving by 25.1% on average over the next-best baseline (BBF), while exhibiting P99 latency slightly higher than DBF. The latency gap is expected: CF-based designs may incur multiple kickouts in rare cases, whereas DBF—built on a Counting Bloom Filter—performs a fixed number of counter updates per insert, yielding lower tail latency but also lower steady-state throughput. DFF-QF ranks in the middle and trails CF-based methods since QF insertions slow down at high occupancy due to



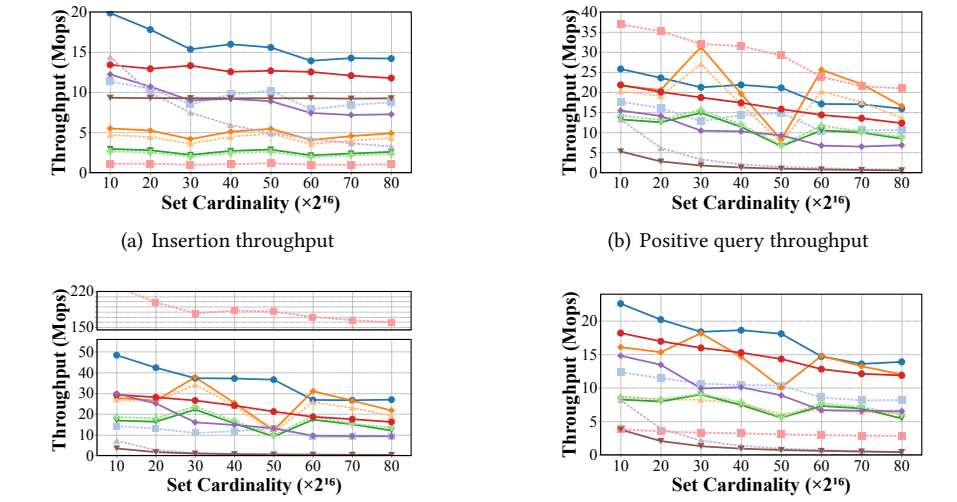
(a) Insertion latency (P99)

(b) Positive query latency (P99)

(c) Negative query latency (P99)

(d) Deletion latency (P99)

Fig. 6. 99th-percentile latency of filter operations.



(a) Insertion throughput

(b) Positive query throughput

(c) Negative query throughput

(d) Deletion throughput

Fig. 7. Throughput of filter operations.

longer runs, yet still outperforms the global-resizing QF baselines IFF/AF, especially in P99 latency, likely because longer runs in a single large QF more often span multiple cache lines as it scales. By comparison, DFF-CF(FG) and DFF-QF(FG) achieve P99 latency comparable to DFF-CF and DFF-QF, but exhibit lower throughput, as expansion incurs extra work under fingerprint growth.

Query. Figures 7(b)–7(c) and Figures 6(b)–6(c) show that DFF-CF, DFF-QF, and DFF-QF(FG) rank among the top methods for both positive and negative queries: on average, their throughput is second only to EBF, and they improve over the next-best baseline BBF by 6.8%–26.1%. In terms of tail latency, DFF-CF achieves the lowest P99 latency for positive queries, reducing it by 20.7% compared with BBF, and for negative queries, it is only slightly behind EBF. However, EBF's high query throughput comes at a substantial memory cost: it maintains a large amount of elastic

fingerprint metadata and often requires up to 100× more memory than DFF (§6.8), which is typically impractical. By contrast, DFF attains near-EBF throughput and P99 latency while remaining memory efficient. We also observe a noticeable spike around 50×2^{16} for QF-based methods (DFF-QF/DFF-QF(FG)/IFF/AF), where performance degrades as the filters approach high occupancy and QF lookups suffer more from long runs than CF at high load factors. Finally, certain chain-based baselines (DBF/DCF) exhibit a more pronounced latency increase under growth, since they may probe multiple sub-filters per lookup.

Deletion. Figure 7(d) and Figure 6(d) show that DFF-CF achieves the highest deletion throughput, improving by 18.8% over the next-best BBF, while maintaining comparable P99 latency to BBF. DFF-QF exhibits slightly worse throughput and latency than DFF-CF, consistent with QF's lower efficiency for updates at high occupancy. By comparison, DFF-CF(FG) and DFF-QF(FG) achieve lower deletion throughput and higher P99 latency than their non-FG counterparts, because deletions under fingerprint growth must examine all candidate slots and select a safe match (§4.5.3). Similar to queries, we observe a noticeable spike around 50×2^{16} for QF-based methods (DFF-QF/DFF-QF(FG)/IFF/AF), where performance degrades as the filters approach high occupancy and deletion becomes more sensitive to long runs.

6.4 Maximum Operation Blocking Duration

A key motivation of DFF's segment-based design is to enable fine-grained resizing and thus bound worst-case operation latency. To quantify worst-case stalls during growth, we report the maximum operation blocking duration for insertions in Figure 8. In each run, we measure the latency of every insertion and use the maximum insertion latency as the blocking duration, since the longest insertion typically corresponds to an expansion event.

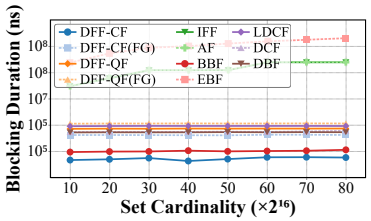


Fig. 8. Maximum operation blocking duration.

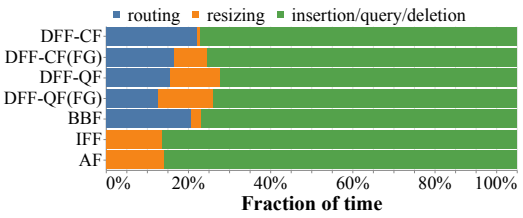


Fig. 9. Time breakdown of total execution time.

We observe that DFF-CF blocks for only about 0.05 ms in the worst case, reducing the next-best baseline BBF by up to 59.5%, and remains stable as the dataset grows. DFF-CF(FG), DFF-QF, and DFF-QF(FG) show similarly stable behavior, and even the slowest variant, DFF-QF(FG), blocks for only about 1 ms. In contrast, global-resizing methods (EBF/IFF/AF) increase rapidly with scale: from 10×2^{16} to 80×2^{16} items, blocking duration grows by roughly 10×, reaching about 2 s for EBF and 0.25 s for IFF/AF at the largest scale, and would continue to rise with further growth.

6.5 Operation Time Breakdown

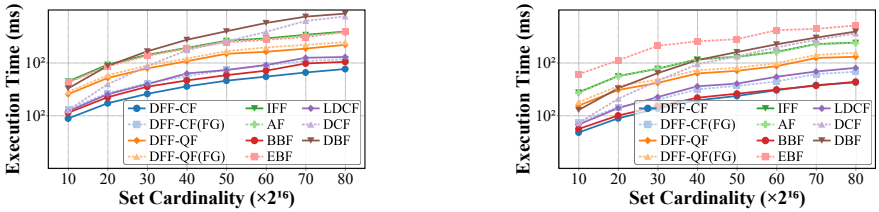
To understand where DFF's performance gains come from, we break down end-to-end execution time and compare DFF with representative baselines (BBF/IFF/AF). In this experiment, we insert 5,000,000 elements, query them, and then delete them. We report the fraction of total time spent in (i) routing (mapping an item to its target segment), (ii) resizing/maintenance (*e.g.*, segment splits and directory updates), and (iii) regular filter operations (in-segment insert/query/delete). For IFF

and AF, which rely on global resizing and do not perform segment routing, we report only (ii) and (iii). Figure 9 shows the resulting time fractions.

We observe that all DFF variants and BBF spend only about 20% of total time on routing, as expected, since both designs support constant-time segment localization. For resizing, DFF-CF is particularly efficient, accounting for only about 1% of the total time, while the other DFF variants spend about 8%–12%. This gap stems from DFF-CF's fixed-length fingerprint setting, where expansion can be implemented by duplicating the source segment and clearing half of the fingerprints in each resulting segment, avoiding reinsertion. In contrast, QF-based variants and FG-enabled variants must perform reinsertion during expansion, leading to higher resizing cost. Finally, IFF and AF show a resizing fraction similar to DFF-QF(FG), consistent with their QF-based design with fingerprint growth.

6.6 Execution Time Under Mixed Operations

To validate DFF under realistic interleaved insert/query/delete traces, Figure 10 reports total execution time for mixed-operation workloads with insertion:query:deletion ratios of 3:9:1 and 9:3:1, representing query-intensive and insertion-intensive settings, respectively. Under both mixes, DFF-CF achieves the lowest execution time among CF-based designs, delivering a 1.33× speedup over BBF (the next-best baseline) in the query-intensive workload and a 1.08× speedup in the insertion-intensive workload. Among QF-based designs, DFF-QF and DFF-QF(FG) achieve 1.50×–1.82× speedups on average over AF, the strongest QF-based baseline.



(a) Execution time (i:q:d=3:9:1)

(b) Execution time (i:q:d=9:3:1)

Fig. 10. Execution time of different workloads.

6.7 False Positive Rate (FPR)

To measure FPR, we insert from 10×2^{16} to 80×2^{16} items and, at each scale, issue the same number of negative queries using randomly generated keys not in the inserted set. As shown in Figure 11, the FPR of LDCF, DFF-CF, DFF-QF, BBF, and DCF grows roughly linearly with scale (from $\approx 0.1\%$ to $\approx 1\%$). LDCF is slightly higher, while the latter four curves nearly overlap, which matches our expectation under the equal fingerprint-length configuration. For CF with b slots per bucket and f-bit fingerprints, $\epsilon_{\mathrm{CF}} \approx \frac{2b}{2^f} = \frac{8}{2^f}$ (with $b = 4$). For QF, $\epsilon_{\mathrm{QF}} \approx 2^{-r}$ where r is the remainder length, and under our parameterization $f = r + 3$ (since QF uses 3 bits for metadata), so $\epsilon_{\mathrm{QF}} \approx 2^{-r} = \frac{8}{2^{r+3}} = \frac{8}{2^f}$, yielding the same FPR as CF.

Correspondingly, fingerprint-growth-based designs—including DFF-CF(FG), DFF-QF(FG), IFF, and AF—maintain a stable, flat FPR as the dataset grows. In our experiments, IFF/AF keep FPR at about 0.038%, while DFF-CF(FG) and DFF-QF(FG) achieve a slightly lower FPR of about 0.027% (a 29.4% reduction over IFF/AF), at the cost of slightly higher memory footprint (§6.8). Compared with their non-FG counterparts, DFF-CF(FG) and DFF-QF(FG) reduce average FPR by 92.6% (with the gap widening as the dataset grows), while consuming 37% more memory (§6.8).

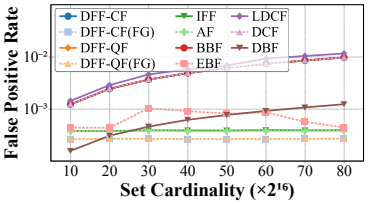


Fig. 11. FPR (DFF-CF/QF, BBF, and DCF overlap).

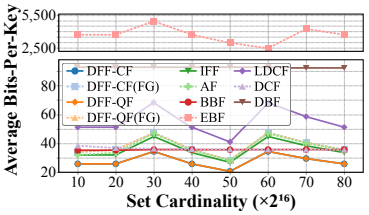


Fig. 12. Memory footprint.

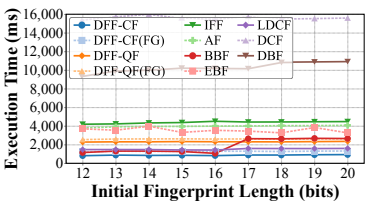
6.8 Memory Footprint

We plot the average bits-per-key (total memory footprint divided by inserted keys) to quantify memory overhead under the same scaling setting as other experiments (Fig. 12). The reported bits-per-key includes all metadata (*e.g.*, DFF's lookup table). We observe that all methods fluctuate within a bounded range, indicating near-linear memory growth with the dataset size. Notably, DFF-CF and DFF-QF incur the lowest memory footprint at about 27.8 bits-per-key on average, reducing the next-best BBF (35.7) by 22.1%. This value is consistent with the configured fingerprint length and the expected load factor dynamics. Expansion is triggered at a 90% load factor, after which each of the two resulting filters has an expected load factor of about 45%. Therefore, the expected load factor over time is approximately $\frac{45\%+90\%}{2} = 67.5\%$. Accounting for this occupancy, the effective bits-per-key for occupied slots is $27.8 \times 0.675 = 18.765$, which is close to the 16-bit fingerprint length used in our experiments within measurement variance.

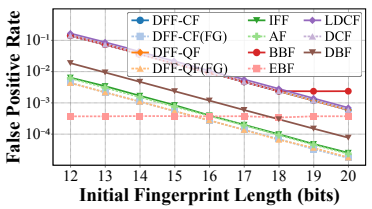
For FG-based methods, including DFF-CF(FG), DFF-QF(FG), IFF, and AF, the additional overhead follows the expected $O(\log \log n)$ trend. Accordingly, DFF-CF(FG) and DFF-QF(FG) are close to IFF/AF, averaging 38.1 bits-per-key, which is 37.1% higher than DFF-CF/QF. This is consistent with expectations since FG variants increase the fingerprint length gradually with the dataset size. In contrast, EBF incurs an extremely high memory footprint due to its elastic-fingerprint metadata, averaging about 3550 bits-per-key (roughly 100× higher than all DFF variants), which is impractical in typical deployments.

6.9 Fingerprint Length Sensitivity

To address the concern that filter performance and accuracy may vary with the initial fingerprint length (*i.e.*, the target bits-per-key), Figure 13 evaluates DFF and all baselines across a range of initial fingerprint lengths ($f_0 \in [12, 20]$ bits). For each setting, we run a 5,000,000-operation trace and follow the same evaluation methodology as §6.6 for operation performance (using the query-intensive mix i:q:d=3:9:1) and as §6.7 for FPR measurement.

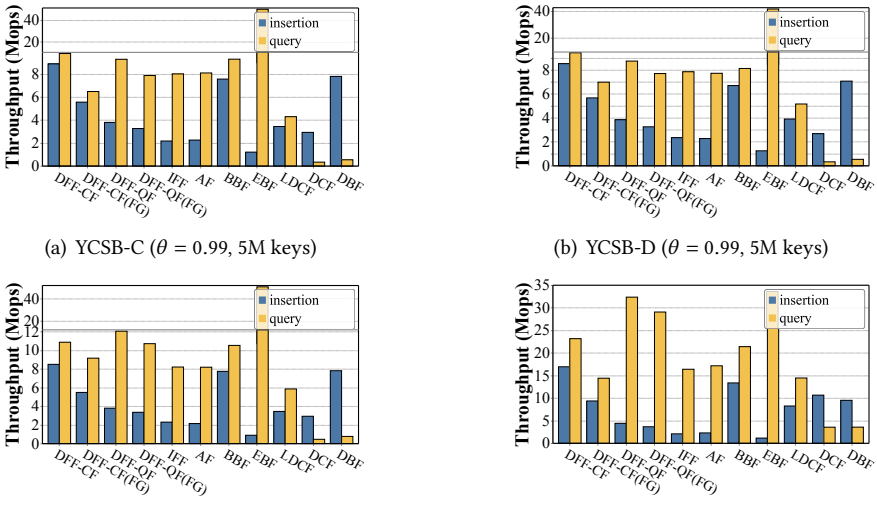


(a) Execution time (i:q:d=3:9:1)



(b) FPR (DFF-CF/QF, BBF, and DCF overlap)

Fig. 13. Sensitivity to initial fingerprint length.



(a) YCSB-C (θ = 0.99, 5M keys)

(b) YCSB-D (θ = 0.99, 5M keys)

(c) YCSB-F (θ = 0.99, 5M keys)

(d) CAIDA (1M keys)

Fig. 14. Throughput on skewed datasets (YCSB with Zipfian θ = 0.99 and CAIDA).

We find that DFF's total execution time is largely insensitive to the initial fingerprint length. Across the sweep, DFF-CF reduces total execution time by at least 22.8% compared with the next-best baseline BBF, while DFF-CF(FG) is slightly slower than DFF-CF. Among QF-based designs, DFF-QF(FG) reduces total execution time by at least 33.5% compared with the next-best QF-based baseline AF. For FPR, DFF follows the same trend as most baselines: it decreases roughly exponentially with fingerprint length, and the relative ordering is consistent with §6.7.

6.10 Throughput on Skewed Datasets

We evaluate throughput on skewed datasets, including YCSB workloads C/D/F and the CAIDA trace (§6.1). Figure 14 reports the results. We observe that EBF achieves the highest query throughput on these traces, but also the lowest insertion throughput. Moreover, EBF's throughput advantage comes with extremely high memory footprint (§6.8), which is typically impractical. Under a reasonable memory footprint, DFF variants consistently achieve the best or near-best throughput across these skewed datasets: on YCSB (C/D), DFF-CF improves insertion throughput over the next-best baseline BBF by 17.4%–37.3% and query throughput by 5.4%–16.1%. On more skewed workloads such as YCSB-F (read-modify-write) and CAIDA, DFF-QF and DFF-QF(FG) achieves higher query throughput than its CF counterparts, likely due to QF's better spatial locality under hot-key reuse, albeit with lower insertion throughput.

7 Discussion

The DFF framework decouples dynamic resizing from filter logic to ensure $O(1)$ scalability and a stable false positive rate (FPR). However, this architecture introduces specific constraints regarding security, CF variant integration, and configuration.

Backend Selection and Integration Constraints.

Our experiments show that CF-based and QF-based DFF achieve identical memory footprint and FPR under the same fingerprint-length configuration (§6.7, §6.8); thus, the backend choice in practice is mainly driven by workload characteristics. DFF-QF benefits from QF's data locality and is preferable for disk-resident or cache-sensitive settings; we also observe higher throughput

than DFF-CF under extremely heavy-tailed workloads such as YCSB-F and CAIDA (§7). With fingerprint growth enabled, DFF-QF(FG) is particularly attractive: it achieves a near-constant FPR while incurring only slightly worse throughput and tail latency than DFF-QF. In contrast, DFF-CF is better suited for write-intensive, random-access workloads due to its balanced insert/query throughput, and for tail-latency-sensitive deployments since each operation checks only a constant number of candidate fingerprints.

Additionally, DFF introduces unique constraints when adopting advanced, block-based variants of CF as backend. For example, Vacuum or Morton Filter [5, 45] requires large contiguous memory regions for balancing data locality and memory utilization. These constraints can be managed by choosing an appropriate segment size. Enlarging the initial segment size simultaneously improves fingerprint randomness (by delaying directory growth) and accommodates the large contiguous memory needs of block-based variants. This presents a tunable trade-off: larger segments ensure stability and compatibility for complex Cuckoo variants, but users must balance this against the slight increase in blocking time during expansion splits.

Hashing Cost. DFF uses a single hash per key for both segment localization and fingerprint derivation, since expansion and routing must follow the same hash-prefix used by stored fingerprints (thus we cannot use two independent hashes). As a result, DFF does not incur extra hash computations compared to the baselines, and the remaining routing cost is dominated by memory accesses (consistent with Figure 9).

Adversarial Workloads. DFF avoids global rehashing to preserve stable tail latency by relying on deterministic hash-prefix routing. However, in adversarial settings, an attacker aware of the hash function can craft keys with shared prefixes to repeatedly trigger splits within a single segment. This vulnerability can be mitigated by adopting a cryptographic hash (*e.g.*, SipHash [1]) to obfuscate the mapping. Although this incurs a modest per-operation overhead, it preserves the asymptotic complexity while making targeted collision attacks computationally infeasible.

Operational Concurrency. DFF leverages its segmented architecture to enable high parallelism by allowing operations on distinct segments to proceed independently. Global synchronization occurs only during directory doubling, briefly blocking updates to adjust pointer entries. Outside these events, concurrency control is delegated to the backend. For instance, Cuckoo Filter segments can use fine-grained bucket locks for low-contention updates, whereas Quotient Filters typically rely on coarser locking. As a result, the framework introduces negligible synchronization overhead under multi-threaded workloads.

8 Conclusion

This paper introduces the Dynamic Flat Filter (DFF), a dynamic AMQ framework that empowers fingerprint-based filters to scale while preserving constant-time insertions, queries, and deletions, and stabilizing the false positive rate as capacity grows. We demonstrated ease of adoption by integrating DFF with Cuckoo and Quotient Filters, each requiring fewer than 60 lines of code to modify. Experimental results on both real-world and synthetic datasets confirm that DFF significantly outperforms existing dynamic filters in terms of query throughput, attaining at least 1.33× the overall throughput on query-intensive workloads and reducing worst-case insertion latency by up to 59.5% compared with the best prior dynamic filter.

Acknowledgments

The work of Yang Du is supported by the National Natural Science Foundation of China (NSFC) under Grant 62572337. The work of He Huang and Yu-E Sun is supported by NSFC under Grant 62332013. The work of Guoju Gao is supported by NSFC under Grant 62472298.

References

[1] Jean-Philippe Aumasson and Daniel J Bernstein. 2012. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*. Springer, 489–508.

[2] Baruch Awerbuch and Christian Scheideler. 2006. Towards a scalable and robust DHT. In *Proc. of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*. 318–327.

[3] 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.

[4] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[5] Alex D Breslow and Nuwan S Jayasena. 2020. Morton Filters: Fast, Compressed Sparse Cuckoo Filters. *The VLDB Journal* 29, 2-3 (2020), 731–754.

[6] Hayoung Byun and Hyesook Lim. 2021. Learned FBF: Learning-Based Functional Bloom Filter for Key–Value Storage. *IEEE Trans. Comput.* 71, 8 (2021), 1928–1938.

[7] CAIDA. 2019. The CAIDA Anonymized 2019 Internet Traces. http://www.caida.org/data/overview/.

[8] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. 2017. The dynamic cuckoo filter. In *Proc. of IEEE ICNP*. 1–10.

[9] Source Code. 2026. The source codes of Dynamic Flat Filter and other related algorithms. https://github.com/Snowflyt/dynamic-flat-filter-paper.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*. 143–154.

[11] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proc. ACM Manag. Data* 1, 2 (2023), 1–27.

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

[13] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proc. of ACM SIGMOD*. 365–378.

[14] Niv Dayan, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Ofer Frishman, Evgeni Ginzburg, Igal Maly, et al. 2021. The end of Moore's law and the rise of the data processor. *Proc. VLDB Endow.* 14, 12 (2021), 2932–2944.

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

[16] Navid Eslami and Niv Dayan. 2024. Memento Filter: A Fast, Dynamic, and Robust Range Filter. *Proc. ACM Manag. Data* 2, 6 (2024).

[17] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.* 4, 3 (1979), 315–344.

[18] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proc. of ACM CoNEXT*. 75–88.

[19] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking* 8, 3 (2000), 281–293.

[20] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. 2009. The dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering* 22, 1 (2009), 120–133.

[21] Shankui Ji, Yang Du, He Huang, Yu-E Sun, Jia Liu, and Yapeng Shu. 2025. PipeFilter: Parallelizable and Space-Efficient Filter for Approximate Membership Query. *IEEE Transactions on Knowledge and Data Engineering* 37, 5 (2025).

[22] Zhaodong Kang, Jin Xu, Wenqi Wang, Jie Jiang, Shiqi Jiang, Tong Yang, Bin Cui, and Tilman Wolf. 2021. Coloring embedder: Towards multi-set membership queries in web cache sharing. *IEEE Transactions on Knowledge and Data Engineering* 34, 12 (2021), 5664–5680.

[23] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

[24] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. 1999. Web caching with consistent hashing. *Computer Networks* 31, 11-16 (1999), 1203–1213.

[25] Téo Lemane, Paul Medvedev, Rayan Chikhi, and Pierre Peterlongo. 2022. Kmtricks: efficient and flexible construction of bloom filters for large sequencing data collections. *Bioinformatics Advances* 2, 1 (2022), vbac029.

[26] Ruiyuan Li, Xiang He, Yingying Sun, Jun Jiang, You Shang, Guanyao Li, and Chao Chen. 2025. Spatio-Temporal Keyword Query Processing Based on Key-Value Stores. *Data Science and Engineering* 10, 1 (01 Mar 2025), 98–116.

[27] Yuhang Li, Rong Gu, Chengying Huan, Zhibin Wang, Renjie Yao, Chen Tian, and Guihai Chen. 2025. HotPrefix: Hotness-Aware KV Cache Scheduling for Efficient Prefix Sharing in LLM Inference Systems. *Proc. ACM Manag. Data* 3, 4 (2025), 1–27.
[28] Witold Litwin. 1980. Linear hashing: a new tool for file and table addressing. In *Proc. of VLDB*. VLDB Endowment, 212–223.
[29] Feifan Liu, Rong Gu, Meng Li, Haipeng Dai, Baohan Wang, Jie Tao, and Dian Shen. 2025. Hourglass: An Adaptive Range Filter with Lightweight Hybrid Encoding. *Proc. ACM Manag. Data* 3, 4 (2025).
[30] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable learned bloom filters for data streams. *Proc. VLDB Endow.* 13, 12 (2020), 2355–2367.
[31] Zirui Liu, Xian Niu, Wei Zhou, Yisen Hong, Zhouran Shi, Tong Yang, Yuchao Zhang, Yuhan Wu, Yikai Zhao, Zhuochen Fan, and Bin Cui. 2025. Extendible RDMA-Based Remote Memory KV Store with Dynamic Perfect Hashing Index. In *Proc. of IEEE ICDE*. 1745–1758.
[32] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard TB Ma, Xueshan Luo, and Bangbang Ren. 2019. The consistent cuckoo filter. In *Proc. of IEEE INFOCOM*. 712–720.
[33] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard TB Ma, Xueshan Luo, and Bangbang Ren. 2021. A capacity-elastic cuckoo filter design for dynamic set representation. *IEEE Transactions on Network and Service Management* 18, 4 (2021), 4860–4874.
[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 *Proc. of ACM SIGMOD*. 2071–2086.
[35] Sajad Faghfoor Maghrebi and Niv Dayan. 2025. Sphinx: A Succinct Perfect Hash Index for x86. *Proc. VLDB Endow.* 18, 11 (2025), 4424–4437.
[36] Camille Marchet and Antoine Limasset. 2023. Scalable sequence database search using partitioned aggregated Bloom comb trees. *Bioinformatics* 39, Supplement_1 (2023), i252–i259.
[37] Kenneth Odoh. 2025. UltraFilter: A Privacy-Preserving Bloom Filter. In *Proc. of ACM WWW*. 1224–1228.
[38] Rasmus Pagh, Gil Segev, and Udi Wieder. 2013. How to Approximate a Set without Knowing Its Size in Advance. In *Proc. of IEEE FOCS*. 80–89.
[39] Guannan Pan, Yongchao Zhang, and Qingjun Xiao. 2023. Bucket-Level Elastic Cuckoo Filter Based on Consistent Hashing with High Memory Efficiency. In *Proc. of IEEE ICNP*. 1–11.
[40] Prashant Pandey, Martín Farach-Colton, Niv Dayan, and Huanchen Zhang. 2024. Beyond Bloom: A Tutorial on Future Feature-Rich Filters. In *Proc. of ACM SIGMOD*. 636–644.
[41] Uzma Sattar, Talha Naqash, Muhammad Raheel Zafar, Kashif Razzaq, and Faisal bin Ubaid. 2013. Secure DNS from amplification attack by using modified bloom filters. In *Proc. of IEEE ICDIM*. 20–23.
[42] Chenxu Wang, Tony TN Miu, Xiapu Luo, and Jinhe Wang. 2017. SkyShield: A sketch-based defense system against application layer DDoS attacks. *IEEE Transactions on Information Forensics and Security* 13, 3 (2017), 559–573.
[43] Hancheng Wang, Haipeng Dai, Shusen Chen, Meng Li, Rong Gu, Youyou Lu, Chengxun Wu, Jiaqi Zheng, Lexi Xu, and Guihai Chen. 2025. Parallel Wormhole Filters: High-Performance Approximate Membership Query Data Structures for Persistent Memory. *IEEE Transactions on Parallel and Distributed Systems* 36, 11 (2025), 2229–2246.
[44] Hancheng Wang, Haipeng Dai, Meng Li, Jun Yu, Rong Gu, Jiaqi Zheng, and Guihai Chen. 2022. Bamboo filters: Make resizing smooth. In *Proc. of IEEE ICDE*. 979–991.
[45] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. 2019. Vacuum Filters: More Space-efficient and Faster Replacement for Bloom and Cuckoo Filters. *Proc. VLDB Endow.* 13, 2 (2019), 197–210.
[46] Yuhan Wu, Jintao He, Shen Yan, Jianyu Wu, Tong Yang, Olivier Ruas, Gong Zhang, and Bin Cui. 2021. Elastic bloom filter: deletable and expandable filter using elastic fingerprints. *IEEE Trans. Comput.* 71, 4 (2021), 984–991.
[47] Fan Zhang, Hanhua Chen, Hai Jin, and Pedro Reviriego. 2021. The logarithmic dynamic cuckoo filter. In *Proc. of IEEE ICDE*. 948–959.
[48] Yongchao Zhang, Qingjun Xiao, Chenyang Guo, Guannan Pan, Jun Huang, Kun He, Yu Miao, and Wenjin Li. 2025. Bucket-Level Elastic Cuckoo Filter for Dynamic Set Membership Query and Encoded Set Operations. *IEEE Transactions on Networking* (2025), 1–20.
[49] Zichen Zhu, Yanpeng Wei, Ju Hyoung Mun, and Manos Athanassoulis. 2025. Mnemosyne: Dynamic Workload-Aware BF Tuning via Accurate Statistics in LSM trees. 3, 3 (2025).

Received October 2025; revised January 2026; accepted February 2026