To Adapt or Not to Adapt, That is the Ski Question

YUVARAJ CHESETTI, Northeastern University, USA
PRASHANT PANDEY, Northeastern University, USA

Adaptive filters, unlike traditional filters, update their representation upon detecting a false positive to avoid repeating the same error in the future. Adaptive filters require an auxiliary structure, typically much larger than the main filter and often residing on slow storage, to facilitate adaptation. Each adaptation requires a disk I/O, creating overhead that must be amortized across repeated false positives. On highly skewed or adversarial workloads, this overhead is well amortized over repeated false positives. However, on uniform random workloads, adaptive filters experience noticeable performance degradation as false positives do not repeat often, making adaptivity an unnecessary overhead.

In this paper, we address this fundamental limitation in adaptive filters by establishing a connection to the online ski rental problem, a classic online decision problem where one must repeatedly choose between renting or buying skis without knowing future usage. We demonstrate that adapting a false positive is equivalent to "buying", while accepting the false positive without adaptation corresponds to "renting". Using the online model, we theoretically analyze the cost of adaptivity and develop an adaptivity strategy to determine when to adapt a false positive to avoid unnecessary overheads.

We introduce the SkiQF, an online adaptive filter that provides robust false positive rate guarantees through adaptivity while maintaining consistent, high performance across diverse query distributions. The SkiQF makes no assumptions about the query distribution and is 2-competitive, i.e., its total I/O never exceeds twice the optimal for any workload. In a database system using a filter to avoid unnecessary disk accesses, the SkiQF achieves up to 1.78× higher throughput compared to the AdaptiveQF. Compared to traditional filters, the SkiQF achieves up to 1.45× higher throughput on skewed workloads and up to 10× higher throughput on adversarial workloads, demonstrating robustness and resilience. We also introduce the HybridSkiQF, a variant that detects the underlying query distribution and chooses an adaptivity strategy to minimize I/O. Additionally, it dynamically adjusts its strategy in response to distribution drift. When it detects repeating false positives, the HybridSkiQF switches to immediate adaptation, like a standard adaptive filter, avoiding the overhead of the SkiQF's delayed adaptivity when immediate adaptation is optimal. The HybridSkiQF achieves up to 1.28× and 1.79× higher throughput than the SkiQF and traditional filter respectively.

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

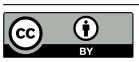
Additional Key Words and Phrases: Filters; Adaptive; Dictionary data structure; Databases

ACM Reference Format:
Yuvaraj Chesetti and Prashant Pandey. 2026. To Adapt or Not to Adapt, That is the Ski Question. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 244 (June 2026), 27 pages. https://doi.org/10.1145/3802121

1 Introduction

A ***filter*** (e.g., Bloom [5], quotient [4, 35], cuckoo [18]) compactly represents a set and supports approximate membership testing, trading accuracy for space. It allows one-sided errors bounded by a configurable false positive rate (FPR). Filters have long served as the fundamental building block in large-scale data systems with scarce memory and tolerable one-sided errors. They compactly

Authors' Contact Information: Yuvaraj Chesetti, Northeastern University, Boston, MA, USA, chesetti.y@northeastern.edu; Prashant Pandey, Northeastern University, Boston, MA, USA, p.pandey@northeastern.edu.



ACM 2836-6573/2026/6-ART244
https://doi.org/10.1145/3802121

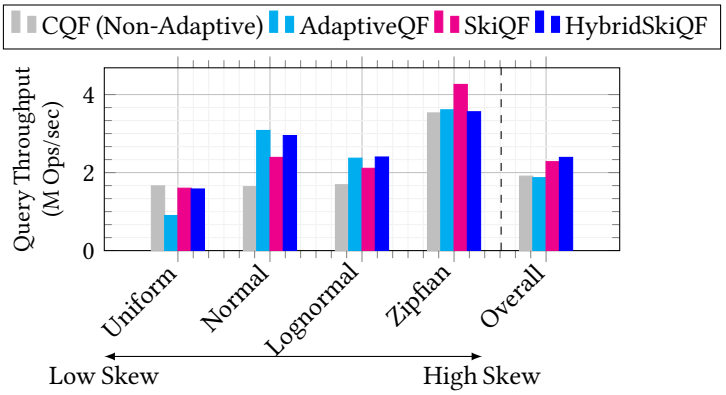


Fig. 1. Comparing query throughput of an adaptive (AdaptiveQF [41]) and a traditional non-adaptive (counting quotient filter (CQF) [35]) filter in a database across varying workload skewness. The overall throughput measures the aggregate throughput (harmonic mean) across all workload distributions. The overhead of adaptivity degrades performance on low-skew workloads. Increasing skew initially benefits from adapting recurring false positives, but extreme skew introduces many one-off false positives that negate adaptivity gains. The SkiQF and HybridSkiQF, the online adaptive filters we introduce in this paper, achieve consistent performance across all skew levels through online adaptation strategies. (Higher is better.)

represent datasets in memory, avoiding unnecessary disk accesses for non-existent items. Filters have seen wide usage in numerous applications such as databases [12, 14, 40], networking [7, 22, 42], computational biology [6, 11], storage systems [15, 17, 21] and machine learning [38].

Traditional filters, however, guarantee a bounded false positive rate (FPR) *only* when queries are drawn uniformly at random from the universe of items. That is, the FPR guarantee only applies to a single query, not to an arbitrary sequence of queries. Consequently, the observed false positive rate can far exceed theoretical bounds for workloads with repeated false positive items, such as skewed or adversarial distributions. This degradation occurs because traditional filters treat each query independently, never learning from or adapting to previous false positives.

For example, filters are commonly employed in database indexes to avoid unnecessary disk I/Os. The filter is kept in main memory and used to skip disk accesses whenever it returns a negative response. However, false positives still trigger unnecessary disk reads. In workloads with repeated queries for non-existent keys, these false positives accumulate and degrade performance, often rendering the filter obsolete. This problem is particularly severe for write-optimized databases (RocksDB [40], SplinterDB [12]) that rely on filters to mitigate the high read amplification inherent in storing multiple sorted runs on disk.

Recently introduced ***adaptive filters*** [3, 29, 32, 41] overcome this fundamental limitation in traditional filters by updating their representation when encountering false positives, preventing their recurrence. As a result, adaptive filters offer a theoretically-bounded false positive rate for any arbitrary sequence of queries. Such robust guarantee makes adaptive filters appropriate for workloads with skewed or adversarial distributions.

Overheads of adaptivity. A key limitation of adaptive filters is that it is impractical to build one that operates only in main memory. Bender et al. [3] show that any adaptive filter that guarantees a bounded false positive rate for n items from a universe of size u must use $O(\min(n\log n, n\log\log u))$ bits of space, making the filter impractical to fit in main memory. However, this does not eliminate practical solutions. One can build an adaptive filter by splitting it into two parts: a *main-memory structure* that uses the same amount of space as a traditional filter, and a *disk-resident structure* that

is consulted only on adaptations. The disk-resident structure, called the *reverse map*, is a dictionary mapping fingerprints (lossy item representations in the filter) to original items, enabling the filter to adapt efficiently. Since each adaptation accesses the reverse map, false positives now incur additional I/Os compared to traditional filters.

The additional I/Os to access the reverse map is the overhead incurred by an adaptive filter to provide a theoretically-bounded false positive rate for any arbitrary sequence of queries. On skewed and adversarial (where false positives are intentionally replayed) workloads, the additional I/O overhead is well amortized over repeated false positives. However, many practical workloads feature distributions where items rarely repeat: e.g., items drawn uniformly randomly from the universe, or skewed distributions with long tails where most items appear only once or twice despite a few high-frequency items. On these workloads, system developers are forced to choose between a traditional filter for higher performance or an adaptive filter for robustness against adversarial attacks.

Problem. This performance variability creates a critical challenge for system developers who must choose between traditional and adaptive filters without knowing workload characteristics, which are often not known in advance and are subject to sudden and unpredictable changes.

Figure 1 shows the performance trade-off between traditional and adaptive filters across workloads of varying skewness. Traditional filters outperform adaptive filters by $\sim 1.8\times$ on uniform random workloads where false positives rarely repeat. With moderate skew (normal and lognormal), adaptive filters achieve $1.4 - 1.8\times$ better performance. At extreme skew (Zipfian with $\theta = 0.99$), both filter types converge to similar performance.

Research goal. *Our goal is to develop a filter that provides robust accuracy guarantees while maintaining consistent, high performance across all workload distributions, eliminating the need to predict workload characteristics in advance.*

To achieve this research goal, we must design an ***online adaptive filter*** that guarantees robust performance across all distributions. This requires making real-time adaptation decisions without knowledge of future queries. Before performing an adaptation, we must determine whether the cost of adaptation can be amortized over repeated false positives. This requires quantifying the overheads of false positives and adaptations, then designing an online algorithm that decides whether the adaptation overhead is likely to yield a net benefit through prevented false positives.

Challenges in designing an online adaptive filter. There are several critical challenges in designing an online adaptive filter. First, filters are employed across a wide range of applications, such as security, networking, databases, distributed systems, genomics, each with unique set hardware/software constraints and therefore different cost models. Second, the filter must provide strong theoretical bounds despite unpredictable workloads, enabling system developers to adequately provision resources and ensure reliability. Third, the filter must not add non-trivial overheads to the strict space and CPU budgets of traditional filters, requiring negligible overhead for any online decision.

We show that this online decision of deciding whether to adapt or not maps directly to the classic ***ski rental problem*** [25], where a skier must daily choose between *renting* skis (recurring small cost) or *buying* them (one-time large cost) without knowing how many more days they will ski. The ski rental problem models scenarios requiring cost minimization under uncertainty: choosing between an expensive one-time purchase that eliminates future costs or continuing with smaller recurring payments. This framework has been applied to snoopy caching [24, 25], TCP acknowledgment [23], cloud spending optimization [37], and elastic caching [28].

In the adaptation context, the analogy is precise: adapting a false positive corresponds to buying (one-time cost preventing recurrence), while accepting the false positive without adaptation corresponds to renting (repeated cost per occurrence). This mapping enables us to leverage ski rental algorithms with theoretically-bounded costs for adaptation decisions.

Analyzing adaptivity and false positive costs. To apply the ski rental framework, we empirically analyze and model the I/O costs of false positive queries and their adaptation. In the context of large-scale databases, filters are maintained in memory and used to prevent disk accesses for negative queries. In the disk-based setting, I/Os dominate the overall application performance. To analyze algorithm costs, we employ the disk access model [1] (DAM), measuring the overall cost of an algorithm in terms of the number of block transfers between memory and disk.

Modelling the I/O costs allows us to apply the *deterministic ski-rental algorithm* to adaptive filters. This ski-rental algorithm follows a simple strategy: rent until the break-even day—when accumulated cost would exceed the purchase price on the next day —and then commit to buying. For adaptive filters, this translates to adapting when the cost of repeated false positives matches the adaptation cost. We express this break-even point through α, the ratio of reverse map lookup cost (C_{RM}) to database lookup cost (C_{DB}). Since adapting (buying) a false positive requires a database lookup to detect the false positive, the overall cost of an adaptation is $C_{DB}+C_{RM}$. Similarly, incurring a false positive and not adapting (renting) is a database lookup, which costs C_{DB}. Therefore, the break-even point is $\lfloor \frac{C_{DB}+C_{RM}}{C_{DB}} \rfloor = 1+\lfloor \alpha \rfloor$.

The parameter α varies widely across different application scenarios. In simple configurations such as single-node databases, both the database and reverse map use identical storage implementations on the same medium, yielding $\alpha \approx 1$. However, architectural choices significantly affect α value. In distributed settings, the database can reside on fast local storage while the reverse map uses cheaper remote storage, α can far exceed 1. Conversely, in distributed databases where lookups require network round-trips but the reverse map is cached locally, α can drop below 1. In security-sensitive applications, the cost of a false positive can be huge and α can approach 0.

Our analysis reveals clear adaptation thresholds based on α. For $\alpha > 1$, adapt on the $(1+\lfloor \alpha \rfloor)^{\text{th}}$ false positive occurrence (Theorem 1). For $0.618 \leq \alpha < 1$, adapt on the second occurrence, and for $\alpha < 0.618$, adapt immediately (Theorem 5).

The analysis also establishes the *competitive ratio* of our deterministic ski-rental strategy for online adaptivity—that is, the worst-case ratio of the online algorithm's cost to that of an optimal offline algorithm. Our strategy achieves a competitive ratio of $\min\left(\frac{2+\alpha}{1+\alpha}, 1+\alpha, 2\right)$. In contrast, immediate adaptation always has a competitive ratio of $(1+\alpha)$, making it less competitive for $\alpha > 0.618$. For example, in typical deployments where $\alpha \approx 1$, immediate adaptation is 2-competitive while our strategy is 1.5-competitive. Furthermore, since the deterministic ski-rental algorithm is known to be optimally competitive, no online adaptivity algorithm can achieve a better competitive ratio.

The SkiQF. In this paper, we implement the Ski-Adaptive Filter (SkiQF): an online-adaptive filter that employs the deterministic ski-rental algorithm to make adaptivity decisions at runtime. The SkiQF is built on top of the AdaptiveQF [41], a state-of-the-art adaptive filter, augmenting it to track false positive repetitions and adapting only when repetitions exceed a configurable threshold. This selective adaptation enables the SkiQF to provide the adaptivity benefits on skewed workloads, while also matching traditional filter performance on uniform random workloads where adaptivity would be wasteful.

The SkiQF uses no additional main-memory compared to the AdaptiveQF. The SkiQF modifies the metadata scheme to count false positive repetitions within the existing slots of the AdaptiveQF. When the count of a false positive reaches the configured threshold, the SkiQF overwrites the slots used for counting with extension bits that fix the false positive. While the SkiQF does incur a tiny CPU overhead from maintaining this counter, this cost is negligible compared to the reduction in I/O overhead from unnecessary adaptations.

As shown in Figure 1, the SkiQF achieves 1.78× higher throughput than the AdaptiveQF on uniform random workloads (matching traditional filters) and 1.18× higher throughput on extremely skewed

workloads. On normal and lognormal distributions, it achieves a 11-23% throughput reduction versus AdaptiveQF—an expected trade-off within its guaranteed 2-competitive bound. Overall, the SkiQF has 1.19× higher aggregate throughput across the four different workload distributions.

The HybridSkiQF. The SkiQF's optimal competitive ratio across any arbitrary sequence of queries comes with a deliberate trade-off: a slightly lower throughput on workloads when adapting immediately is optimal. This trade-off occurs because the SkiQF makes no assumptions about the workload distribution, forcing it to avoid over-adapting by delaying adaptations until absolutely necessary.

However, in many practical applications, it is reasonable to assume that workloads follow a consistent distribution. If false positives were mostly repeating in the past, future false positives are also likely to repeat, making immediate adaptations more efficient. This motivates us to introduce the HybridSkiQF. The HybridSkiQF detects if false positives are repeating and decides to either adapt all future false positives immediately like an adaptive filter, or not adapt at all like a traditional filter. Additionally, the HybridSkiQF dynamically adjusts its adaptation strategy to handle drift in the workload distribution. Note that unlike the SkiQF, the HybridSkiQF trades worst-case guarantees for higher average-case performance under stable distributions.

Similar to the SkiQF, the HybridSkiQF matches the performance of a traditional filter on a uniform random workload. On skewed workloads (normal and lognormal), the HybridSkiQF has a 28% higher throughput compared to the SkiQF, matching the performance of AdaptiveQF. Overall across all workloads, the HybridSkiQF improves upon the AdaptiveQF throughput by 27%.

Our contributions.

(1) We model the performance of adaptive filters integrated in databases in terms of I/O costs and draw a connection between minimizing overall I/O and the classic online ski rental problem. We employ the model to determine adaptivity strategy to ensure the overall I/O performed remains 2-competitive for diverse system configurations and cost ratios.
(2) We introduce the SkiQF, *the first online adaptive filter*. The SkiQF overcomes the key performance trade-off of adaptive filters, offering consistent performance across query workloads of various distributions by bounding the overall I/O with 2-competitive bound.
(3) We introduce the HybridSkiQF, a variant of the SkiQF with higher performance on skewed workloads where immediate adaptation is beneficial. The HybridSkiQF monitors the workload and periodically updates its adaptivity strategy, switching between adapting immediately or never adapting. While it deviates from strong theoretical guarantees of the SkiQF, it offers higher performance when workloads are stable and predictable.
(4) We perform an extensive experimental evaluation, integrating the filters with a production-ready database (WiredTiger) and measuring system throughput. We test the filters with workloads of distributions with varying skew, including adversarially generated ones. Our experiments show that the online adaptive filters achieve consistent, high performance across all workloads. Further, we run microbenchmarks in a main-memory only setting, showing that adaptive filters can be augmented with online-adaptivity policies with no space overhead.

2 Background

In this section, we provide an overview of filters, adaptivity and the ski rental problem.

2.1 Filters

A filter is a compact, approximate set representation that supports membership queries with one-sided errors: queries for items in the set always return true (no false negatives), while queries for items not in the set may incorrectly return true with bounded probability ϵ (false positives). Allowing one-sided errors enables filters to save space. Modern filters require $n\log\varepsilon^{-1} + \Omega(n)$ bits: quotient

filters use $n\log\varepsilon^{-1}+2.125n$ bits [35] and cuckoo filters use $n\log\varepsilon^{-1}+3n$ bits [18]—typically just 1-2 bytes per item for 0.1-1% false positive rates. On the other hand, Bloom filters use $1.44n\log\varepsilon^{-1}$, beating modern filters only when ε approaches one—an impractical scenario. In contrast, representing a set exactly without errors requires at least $\Omega(n\log u)$ bits, where u is the universe size [9].

Traditional filters provide weak guarantees. Traditional filters do not change their representation in response to false positives, always returning the same response for repeated false positive queries. Their theoretically-bounded false positive rate is only guaranteed for a single query and not an arbitrary sequence of queries. Consequently, when workloads exhibit skew or adversarial patterns—where certain items repeat disproportionately—the observed false positive rate can approach 1: a single frequently-queried false positive can render the filter obsolete. Traditional filters are also vulnerable to timing attacks. Attackers can identify false positives by measuring query latency differences between false positives and true negatives, then deliberately repeat these queries to degrade system performance [41].

Fingerprint filters. State-of-the-art filters such as quotient [36] and cuckoo [18] are fingerprint-based. They compactly store short, lossy fingerprints exactly in a table and false positives occur only due to hash collisions between keys not in the set with those in the set. Fingerprint filters employ ***quotienting*** [27, 34] to compactly store fingerprints in a table. In quotienting, a p-bit fingerprint $h(x)$ is divided into two parts: the first q bits $h_0(x)$ is called the ***quotient*** and the remaining $r=p-q$ bits $h_1(x)$ is called the ***remainder***. The quotient is stored implicitly and only the remainder is stored explicitly to save space. Quotient filters [4, 35, 36] use Robin Hood hashing (a variant of linear probing) to store the remainders in a linear table. To resolve soft collisions (i.e., when two fingerprints share the same quotient but have different remainders) it uses 2-3 extra metadata bits to resolve collisions and efficiently perform queries.

2.2 Adaptivity

Recently introduced adaptive filters [3, 29, 32, 41] provide stronger guarantees regarding false positives rates compared to traditional filters. Adaptive filters update their representation upon encountering a false positive in order to avoid repeated errors.

Strongly-adaptive filters. Bender et al. [3] formally define a filter to be *strongly adaptive* if the upper bound on the number of false positives returned by the filter on any sequence of n queries is tightly concentrated around $(\epsilon\cdot n)$. More specifically, it will never return more than $\epsilon n+O(\sqrt{n\epsilon\log n}+\log n)$ false positives with high probability.

Adaptive filter designs. Several strongly adaptive filters have been proposed: the broom filter [3][1], the telescoping filter [29], and the AdaptiveQF [41]. All employ fingerprint-based designs. Fingerprint filters efficiently support adaptations by eliminating collisions that cause false positive repetitions. The broom filter and AdaptiveQF adapt by lengthening colliding fingerprints until the collision resolves. The telescoping filter [29] employs multiple hash functions and cycles through hash functions to eliminate collisions.

To our knowledge, no strongly adaptive Bloom filter [5] variants exist, and we believe none are forthcoming. The fundamental obstacle is that Bloom filters cannot unset bits without introducing false negatives, preventing the updates required for adaptation. This limitation also precludes deletions, enumeration, counting, and resizing.

Furthermore, not all adaptive fingerprint filters achieve strong adaptivity. Reviriego et al. [39] demonstrate that adversarial query sequences can force the adaptive cuckoo filter [32]—a fingerprint

[1]The broom filter is a purely theoretical structure, and the AdaptiveQF [41] is its practical implementation.

filter based on cuckoo filters [18]—to adapt continuously without successfully eliminating false positives.

Choosing AdaptiveQF as the base filter. We build the SkiQF atop the AdaptiveQF due to its strong adaptive guarantees and efficient adaptation mechanism. Although both the telescoping filter [29] and AdaptiveQF provide strong adaptivity, the AdaptiveQF's highly efficient I/O operations result in significantly higher system throughput [41]: up to 10× faster insertions and 2-3× faster performance on adversarial query workloads. While we demonstrate our online adaptive policies on the AdaptiveQF, they generalize to all strongly adaptive fingerprint filters, including potential future designs. This is because our techniques require only the ability to tag fingerprints with false positive repetition counts, which any fingerprint filter can support through value association.

2.3 Design of adaptive filters

Bender et al. [3] prove that strongly adaptive filters for n items from universe size u require at least $\Omega(\min(n \log\log u, n \log n))$ bits—approaching the dataset size itself, making pure in-memory storage impractical. To overcome this, adaptive filters employ a two-tier architecture: a compact in-memory structure (comparable to traditional filters) that handles membership queries, while a larger external-memory index stores information to support efficient adaptations. The external index is accessed only during adaptations, not positive queries. This separation enables efficient query processing despite the large lower bound on space, adding only minimal I/O overhead for false positive adaptations.

I/O trade-offs of adaptivity. The performance benefit due to adaptive filters is dependent on the workload skew, as external-memory adaptation costs must be amortized by preventing future false positives. Adaptation benefits skewed [8, 13] and adversarial workloads [29, 32, 41], where eliminating repeated false positives saves multiple disk accesses. However, adaptive filters can degrade the overall system performance when false positives rarely repeat. For example, uniform random queries from large universes seldom repeat, making adaptation wasteful. Similarly, extremely skewed workloads with a long tail of one-off queries negate adaptation benefits as the overhead of adapting non-repeating queries outweighs gains from adapting frequent ones.

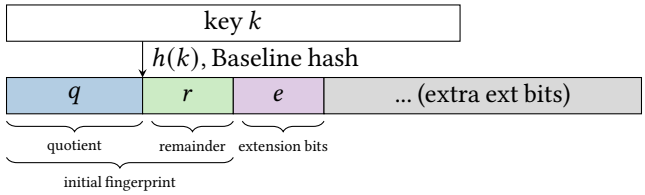
Our goal is to develop a filter that combines the strengths of both adaptive and non-adaptive approaches while mitigating their respective weaknesses. Specifically, we seek a filter that maintains adaptive filters' robust accuracy guarantees—bounding false positive rates even under adversarial or highly skewed query patterns—while delivering consistent, predictable performance across the entire spectrum of workload distributions. By achieving this balance, the filter would eliminate the need for system designers to predict workload characteristics in advance or risk choosing a suboptimal filter type.

2.4 Ski Rental Problem

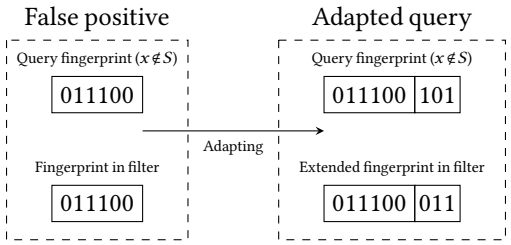
In the ski rental problem, a skier must choose between buying skis for B units or renting for R units per day. Renting costs less initially, but buying provides unlimited future use for a one-time fee. The optimal strategy depends on the total skiing days d. For $d \leq \lfloor B/R \rfloor$ days, renting each day for total cost Rd is optimal in the offline setting. For longer periods, buying on day one for cost B is optimal. The challenge is that the skier cannot predict their total skiing days in advance.

Online decision algorithms, such as those designed for the ski rental problem, are evaluated through *competitive analysis* where an algorithm's cost is compared against an optimal offline algorithm that knows the entire input sequence. The competitive ratio of an online algorithm on a given input is the ratio of its cost to that of the optimal offline algorithm. More specifically, an algorithm is α-competitive if its cost never exceeds α times the offline optimal across all possible inputs.

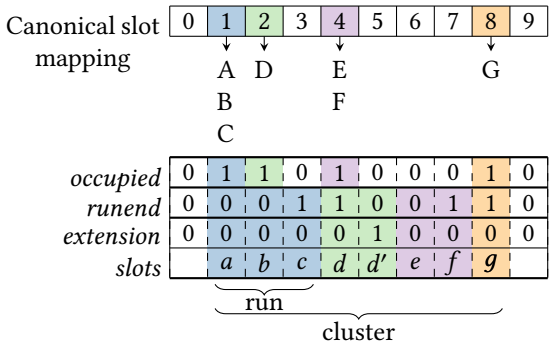
The ski rental problem has a well-known 2-competitive algorithm. The skier rents until the break-even day $\lfloor B/R \rfloor$, then buys if they continue skiing. For $d \leq \lfloor B/R \rfloor$ days, the algorithm pays Rd,



(a) Baseline hash and extension bits.



(b) Extending fingerprints resolves collisions. The query on the left is a false positive as the fingerprint of a key not in the set matches a fingerprint in the filter. Extracting more bits from the filter breaks the collision and resolves the false positive. The AdaptiveQF extends fingperprints in increments of r ($r=3$ in the above figure) bits, enabling it to efficiently store extension bits in its r-bit sized slots. Consequently, each set of extracted extension bits decreases the probability of a future false positive for that fingerprint by 2^{-r}.



(c) The AdaptiveQF uses quotienting to store the fingerprints. We denote each key in uppercase and their corresponding remainder in lowercase. The fingerprint for D is extended and adds extension bits d'.

Fig. 2. Variable-length fingerprints and AdaptiveQF schema. The figures illustrate how variable-length fingerprints are employed to resolve false positives due to hash collisions and the metadata schema used by the AdaptiveQF to store variable-length fingerprints.

which matches the optimal. For longer durations, it pays $R\lfloor B/R \rfloor + B \leq 2B$, at most twice the optimal buying cost. This guarantees 2-competitiveness. No deterministic algorithm can achieve better than 2-competitiveness. However, randomized algorithms can achieve $e/(e-1) \approx 1.58$ competitive ratio in expectation [24].

3 The AdaptiveQF

We now describe the AdaptiveQF [41], a state-of-the-art adaptive filter that serves as the foundation for our online adaptive filters, SkiQF and the HybridSkiQF.

The AdaptiveQF is composed of two parts: a small main-memory filter that answers membership queries, and a disk-based *reverse map* to support efficient adaptations. For ease of discussion, we use

AdaptiveQF to refer to only the main-memory filter and the reverse map as a disk-resident structure that is consulted during adaptations.

3.1 Filter design

We now describe the structure of the main-memory filter in AdaptiveQF.

Variable-length fingerprints. The AdaptiveQF is a fingerprint filter (described in Section 2). As a reminder, false positives happen due to hash collisions—when a queried key not in the set has the same fingerprint as a key in the set. The AdaptiveQF adapts to false positives by using variable-length fingerprints to eliminate fingerprint collisions. Initially, fingerprints are of a fixed length and consist of the lower order bits in the key's hash. When a fingerprint causes a false positive, the AdaptiveQF extends the fingerprint to differentiate it from the query fingerprint, eliminating the fingerprint collision and adapting to the false positive. The bits added to the initial fingerprint to resolve the false positive collision are referred to as ***extension*** bits. Figure 2a shows how fingerprints are generated and Figure 2b shows how extending fingerprints resolves false positives.

Invertible hash functions. The AdaptiveQF uses an invertible hash function [19, 20, 30, 31] to ensure distinct keys map to distinct hash values ($h(x) \neq h(y)$ for $x \neq y$). The key insight here is that since two distinct inputs $x \neq y$ differ somewhere in their invertible hash outputs $h(x) \neq h(y)$, sampling bits from the hash output to construct filter fingerprints guarantees that a sufficiently large sample will capture a distinguishing bit. We use a standard permutation-based implementation [30] widely used in systems requiring invertible hash functions.

Note that although invertible hash values are as large as input keys, the AdaptiveQF only stores prefixes of the hash values as fingerprints. In the extreme case where every key in the universe is queried, the filter would expand all fingerprints to full length and become a perfect hash table, which is the best any data structure can achieve in that scenario.

Quotienting. The AdaptiveQF uses quotienting [27], a compact hash table design common in membership structures [16, 35]. It maintains 2^q slots, each storing an r-bit remainder. A key x's fingerprint splits into a q-bit *quotient* $h_0(x)$ (determining the canonical slot) in the table and an r-bit *remainder* $h_1(x)$ (stored tag). When the canonical slot is occupied, Robin Hood hashing (below) resolves collisions. Since fingerprints can be extended, the AdaptiveQF also needs to store *extension bits* added from adaptations. These are stored in adjacent slots following the remainder.

Robin Hood hashing. The AdaptiveQF employs Robin Hood hashing [10, 26], an ordered linear probing scheme maintaining the invariant: if $h(a) < h(a')$, then the fingerprint of a precedes that of a'. During insertion, existing fingerprints move right to preserve ordering. Keys sharing a canonical slot form contiguous *runs*. Sequential runs displaced from their canonical slots form *clusters*. Figure 2c shows three keys mapped to Slot 1 forming a 3-slot run, while entries for Slots 2 and 4 are displaced rightward, creating a cluster with Slot 8.

Metadata bits. The AdaptiveQF uses 2 metadata bits per slot to help with locating the start of a run: the *occupied* and *runend* bits. The *occupied* bit of a slot is set if there is at least one fingerprint in the filter for which it is the canonical slot. In Figure 2c, for example, the *occupied* bit is set to 1 only for Slots 1, 2, 4, and 8. The *runend* bit is set for a given slot if it contains the last remainder in a given run. In Figure 2c, the *runend* bit is set to 1 for Slots 3, 4, 7 and 8. To identify extension slots, the AdaptiveQF also adds a *extension* metadata bit to each slot. The *extension* bit of a slot is set if if stores extension bits. In Figure 2c, the slots storing the extension bits d' for key D have its *extension* bit set.

Queries. The AdaptiveQF processes membership queries by locating the queried key's run, then scanning its remainders and extension bits. It returns *YES* if the fingerprint matches, *NO* otherwise.

To accelerate queries, the AdaptiveQF divides the filter into blocks of 64 slots with metadata bits. Each block maintains occupied and runend bits where the i^{th} runend bit corresponds to the run containing the i^{th} occupied bit. This correspondence enables constant-time run location via *RANK* and *SELECT* primitives [35]. Since runs can overflow across blocks, each block stores an 8-bit offset counting slots occupied by prior blocks' runs. These offset slots are excluded from the *RANK*and *SELECT* operations as they contain data from previous blocks.

3.2 Adaptivity

This section describes the overall process of adapting false positives.

False positive feedback. The AdaptiveQF cannot detect false positives on its own, instead it needs feedback from the database. The database detects a false positive when it cannot find a key for which the filter returned a positive result. When this happens, the database notifies the AdaptiveQF, triggering an adaptation.

Adapting false positives. The AdaptiveQF first locates the fingerprint that collided with the query key. Using the fingerprint, the AdaptiveQF performs a lookup in the reverse map—an external-memory index mapping fingerprints to keys—to retrieve the key corresponding to the fingerprint. The key's hash is computed and extension bits are extracted from it to avoid the collision with the query key. The extension bits are then stored next to the remainder, making space by shifting items if necessary. In practice, the AdaptiveQF reserves space to store extension bits as it operates below maximum load factors for efficiency.[2] While adaptations increase the load factor, strong adaptivity ensures extremely slow growth. Additionally, the AdaptiveQF supports resizing when load factor becomes high enough that reserve slots are sparse and expensive to locate.

3.3 Reverse map

We now discuss the the reverse map design of the AdaptiveQF. The reverse map enables fingerprint-to-key lookups, avoiding full table scans during adaptations. However, it is the primary performance bottleneck in adaptive filters as it is disk resident. Below, we discuss design decisions in AdaptiveQF that minimize this overhead.

Reverse map design. The reverse map is a dictionary mapping the fingerprints in the filter to their corresponding keys. Since it contains as many entries as the dataset size, it is stored on disk as a separate database table or secondary index. Note that the additional disk overhead is relatively small compared to the primary database since it only needs to store fingerprint-key pairs and not full row/tuples.

In AdaptiveQF, the reverse map is a dictionary mapping (fingerprint, rank) pairs to keys, where rank is position of the fingerprint in the run. Rank of a fingerprint does not change upon subsequent insertions. The following sections detail how this rank-based indexing enables efficient adaptivity and updates.

Adaptations. The reverse map is populated during insertions. There are no updates needed in the reverse map during adaptations because the rank of an adapted fingerprint does not change when it is adapted. In contrast, telescoping [29] and adaptive cuckoo filters [32] require multiple on-disk updates per operation to the reverse map because they use multiple hash functions and relocate fingerprints, affecting multiple entries. As an example, consider the reverse map entries for a run with three fingerprints and their corresponding ranks: $(a,1),(b,2),(c,3)$. If the fingerprint b is extended to bb', the extended fingerprint's corresponding key can be retrieved from the reverse map entry $(b,2)$. Thus, the reverse map does not need to be updated during adaptations.

[2]Insert cost is $O(\ln n/(\alpha-\ln\alpha-1))$ [4], where n is the number of entries and α is the load factor. Most quotient filters operate at $\alpha \in [0.90,0.95]$.

Inserts and Deletes. Updates to the database (and filter) must also update the reverse map. Inserts require only a single append operation in the reverse map because new fingerprints receive new ranks without affecting existing entries. Deletions require updating multiple entries as subsequent ranks must be decremented, but this overhead remains acceptable because run lengths stay small due to hashing. For example, consider the reverse map entries for a run with three entries: $(a,1),(b,2),(c,3)$. Inserting a new key d requires updating the reverse map with a new entry $(d,4)$. Deleting b then results in keys $(c,3)$ and $(d,4)$ being updated in the reverse map to $(c,2)$ and $(d,3)$.

4 Adaptivity as a Ski Rental Problem

In this section, we model the costs of adaptivity as a ski rental problem and analyze when to adapt.

Modelling as independent ski rental instances. We model the online adaptivity problem by dividing the query workload into multiple *instances* of the ski rental problem, one for each unique query key. The goal of the model is to minimize the overhead of adaptations and the cost of incurring false positives for each instance. We can safely ignore instances with query keys having a correct response on the first attempt, as the filter does not need to change its response in these instances. For each instance, the input is a sorted list of timestamps of when the key was queried, and the filter decides whether to adapt or not at each timestamp without knowing how many more future timestamps are in the input.

We can analyze the overall cost of adaptations and false positives by aggregating each instance (individual keys) independently. Each instance is independent, as the AdaptiveQF never forgets an adaptation; therefore, adapting in one instance does not revert adaptations in other instances. While it is possible for an adaptation in one instance to adapt another for free, this only lowers the overall cost. Thus, summing the costs of each group independently provides an upper bound on overall cost.

Disk access model. We employ the disk access model (DAM) [1] to analyze the cost of adaptations and false positives. The disk access model divides memory into a hierarchy consisting of a fast but limited-in-size *main memory* and a slower but arbitrarily large *external memory*. The CPU can operate only on data present in the main memory, and data is transferred between the main memory and external memory in *blocks* of B elements. Since block transfers dominate the total running time, the DAM model measures an algorithm's performance by counting the number of total *block transfers*.

Deterministic ski-rental algorithm. The deterministic ski-rental algorithm rents until day $\lfloor B/R \rfloor$ and buys on the next day, where B and R are the costs of buying and renting respectively. For adaptivity, we specify B and R in terms of C_{RM} and C_{DB} , the cost of a reverse map and database lookup respectively in the DAM model. As the cost of a false positive (renting) is a database lookup, $R = C_{DB}$. Adapting (buying) also includes a database lookup to detect the false positive and a reverse map lookup, so $B = C_{DB} + C_{RM}$. Thus, the break-even day is $\lfloor B/R \rfloor = 1 + \lfloor C_{RM}/C_{DB} \rfloor$.

Competitive ratio. We now analyze the worst-case competitive ratio for online adaptivity based on the deterministic ski rental algorithm for different values of C_{DB} and C_{RM}. We analyze the competitive ratio in terms of $\alpha = C_{RM}/C_{DB}$, the ratio of the cost of a reverse map lookup to a database lookup. Since the input sequence can be divided into multiple independent instances of the ski rental problem, we can analyze the competitive ratio for a single ski rental instance with an input sequence of length k.

Theorem 1. *If $\alpha \geq 1$, i.e., the cost of querying the reverse map is greater than a database lookup, the optimal deterministic strategy is to adapt immediately after the false positive repeats $\lfloor \alpha \rfloor$ times and is 2-competitive.*

Proof. The proof follows the structure used to show that the deterministic ski rental algorithm is 2-competitive. Let k be the length of the final input sequence. If $k \leq \lfloor \alpha \rfloor$, this algorithm pays a total cost of $k \cdot C_{DB}$. This is optimal (1-competitive), as adapting once incurs a cost of $C_{DB} + C_{RM}$, which is $\geq k \cdot C_{DB}$.

For $k > \lfloor \alpha \rfloor$, this algorithm pays a total cost of $\lfloor \alpha \rfloor C_{DB} + (C_{DB} + C_{RM})$, representing the total cost of incurred false positives and the final adaptation. For this case, the optimal offline cost is $C_{DB} + C_{RM}$, i.e., adapt on the first occurrence and avoid the $\lfloor \alpha \rfloor C_{DB}$ cost from repeated false positives. In this case, the algorithm is 2-competitive as the competitive ratio is:

$$
\begin{aligned}
&\frac{\lfloor \alpha \rfloor C_{DB} + (C_{DB} + C_{RM})}{C_{DB} + C_{RM}} \\
&= \frac{\lfloor \alpha \rfloor C_{DB}}{C_{DB} + C_{RM}} + 1 \\
&\leq \frac{\alpha C_{DB}}{C_{DB} + C_{RM}} + 1 \\
&\leq \frac{C_{RM}}{C_{DB} + C_{RM}} + 1 && \text{as } \alpha = C_{RM}/C_{DB} \\
&\leq 2
\end{aligned}
$$

□

Lemma 2. *For $\alpha < 1$, the optimal offline strategy is to not adapt if $k = 1$, and adapt immediately if $k > 1$.*

Proof. We first analyze the offline optimal cost. For $k = 1$, the offline optimal cost is C_{DB} as it avoids the reverse map lookup. For $k \geq 2$, the optimal strategy is to adapt on the first occurrence for a cost of $(C_{DB} + C_{RM})$. Any algorithm that adapts later incurs at least two false positives and would have a cost $\geq 2C_{DB}$. Since $C_{DB} + C_{RM} = (1 + \alpha)C_{DB} < 2C_{DB}$, the optimal strategy is to adapt immediately for a cost of $C_{DB} + C_{RM}$.

□

Lemma 3. *For $\alpha < 1$, adapting immediately achieves a competitive ratio of $(1 + \alpha)$.*

Proof. For $k = 1$, the offline optimal cost is C_{DB} from Lemma 2. Since adapting immediately costs $C_{DB} + C_{RM}$, the competitive ratio for $k = 1$ is $(C_{DB} + C_{RM})/C_{DB} = (1 + \alpha)$. For $k > 1$, this algorithm is optimal (1-competitive) as it immediately adapts. Thus, adapting immediately has a worst-case competitive ratio of $(1 + \alpha)$. □

Lemma 4. *For $\alpha < 1$, adapting on the second false positive achieves a competitive ratio of $(2 + \alpha)/(1 + \alpha)$.*

Proof. For $k = 1$, the offline optimal cost is C_{DB} from Lemma 2. Since this algorithm does not adapt, it is optimal and 1-competitive for $k = 1$. For $k > 1$, this algorithm incurs a cost of $C_{DB} + (C_{DB} + C_{RM}) = 2C_{DB} + C_{RM}$ as it adapts on the second occurrence. The offline optimal cost from Lemma 2 is $C_{DB} + CRM$. The competitive ratio for this case is therefore $\frac{2C_{DB} + C_{RM}}{C_{DB} + C_{RM}} = \frac{2+\alpha}{1+\alpha}$. Since $(2 + \alpha)/(1 + \alpha) > 1$, adapting on the second false positive has a worst-case competitive ratio of $(2 + \alpha)/(1 + \alpha)$. □

Theorem 5. *For $\alpha < 1$, the optimal strategy is to adapt immediately when $\alpha < 0.618$ and adapt on the second occurrence otherwise.*

Proof. By equating $(1 + \alpha) = \frac{2+\alpha}{1+\alpha}$, the worst-case competitive ratios for adapting immediately(Lemma 3) and on the second occurrence repsectively(Lemma 4), we get a threshold value of $\alpha = 0.618$ that indicates which strategy to use: adapt immediately if $\alpha < 0.618$ and on the second occurrence otherwise.

To complete the proof, we must also eliminate strategies that allow for more than 2 false positives and adapt on the third false positive. Adapting on the third occurrence or later has a worst-case competitive ratio $\geq (3 + \alpha)/(1 + \alpha)$. This ratio is always greater than $(2 + \alpha)/(1 + \alpha)$, the worst-case

competitive ratio achieved by adapting on the second occurrence. Therefore, we can eliminate algorithms that adapt on the third false positive occurrence or later. □

Theorem 5 shows that when $\alpha = 1, C_{DB} = C_{RM}$, the optimal ski-rental algorithm is 1.5-competitive by adapting on the second false positive occurrence. In contrast, adapting immediately would be 2-competitive.

False positive rate. As we use the ski-rental algorithm on a strongly adaptive filter, we can bound the number of false positives observed for a sequence of queries.

Lemma 6. *A strongly adaptive filter augmented with the ski rental algorithm never returns more than* $(1+\lceil\alpha\rceil)(n\epsilon + O(\sqrt{n\epsilon \log n} + \log n))$ *false positives with high probability.*

Proof. From [3], a strongly adaptive filter never returns more than $(n\epsilon + O(\sqrt{n\epsilon \log n} + \log n))$ false positives with high probability. From Theorem 1 and Theorem 5, using a ski rental algorithm for adapting never repeats a false positive more than $(1+\lceil\alpha\rceil)$ times: for $\alpha \geq 1$ and $\alpha < 0.618$, the ski rental algorithm allows $1+\lfloor\alpha\rfloor \leq 1+\lceil\alpha\rceil$ occurrences, and for $0.618 \leq \alpha \leq 1$, it allows $2 = 1+\lceil\alpha\rceil$ occurrences. Therefore, a strongly adaptive filter with a ski rental algorithm will never return more than $(1+\lceil\alpha\rceil)(n\epsilon + O(\sqrt{n\epsilon \log n} + \log n))$ false positives with high probability. □

Lemma 6 implies that the ski rental filter is not strongly adaptive and may return more false positives compared to a strongly adaptive filter. For most common cases ($\alpha = 1, C_{DB} = C_{RM}$) the overhead is a factor of 2. However, this trade-off is intended: the online adaptive filter minimizes the total I/O cost of false positives and adaptations and therefore offers higher throughput(Section 6).

5 Implementation

This section describes the design and implementation of the SkiQF, a filter that extends the AdaptiveQF to implement online adaptivity described in Theorems 1 and 5. The SkiQF keeps track of false positive repetitions, adapting when the false positive repeats for more than τ times, where τ denotes the ***adaptivity threshold***. We then describe the HybridSkiQF, a variant of the SkiQF that offers higher performance than the SkiQF on workloads where adapting immediately is optimal. The HybridSkiQF monitors the workload to detect false positive repetitions and chooses between immediate adaptation or no adaptation based on the observed workload characteristics.

5.1 SkiQF

The SkiQF reuses the schema of the AdaptiveQF filter to keep track of false positive repetitions and adapts a false positive query only if crosses the adaptivity threshold (τ).

Determining τ, the adaptivity threshold. The value of τ depends on the ratio $\alpha = C_{RM}/C_{DB}$, where C_{RM} and C_{DB} are the cost of a reverse map lookup and a database lookup respectively. From Theorems 1 and 5:

$$\tau = \begin{cases} \lfloor\alpha\rfloor & \text{if } \alpha \geq 1, \text{ (adapt on } 1+\lfloor\alpha\rfloor \text{ occurrence)} \\ 1 & \text{if } 0.618 \leq \alpha < 1, \text{ (adapt on second occurrence)} \\ 0 & \text{if } \alpha < 0.618, \text{ (adapt on first occurrence)} \end{cases}$$

In the simplest case, we expect $\alpha = 1$ when both the reverse map and the database use the same underlying data structure, implementation and storage device. In complex deployments where the value of C_{DB} and C_{RM} may vary over time and α may significantly deviate from 1, the values of C_{DB} and C_{RM} can be tracked through periodic sampling and averaging of lookup latencies. When the values of C_{DB} and C_{RM} vary over time and τ is updated, the SkiQF resets all fingerprint collision

occupied	0	1	1	0	1	0	0	0	1	0
runend	0	0	0	1	1	0	1	0	1	0
extension	0	0	0	0	0	0	0	0	0	0
slots		*a*	*b*	*c*	*d*	*e*	*f*		*g*	

(a) Initial state of the SkiQF. The filter has not returned any false positives so far.

occupied	0	1	1	0	1	0	0	0	1	0
runend	0	0	0	1	1	1	0	1	1	0
extension	0	0	0	0	0	1	0	0	0	0
slots		*a*	*b*	*c*	*d*	×	*e*	*f*	*g*	

(b) A query for key that maps to fingerprint d is a false postive. The filter makes space for a slot immediately next d to indicate that a false positive was seen. This step incurs no additional I/O to the reverse map. The metadata bits (*runend and extension*) are both set to indicate that this is a collision-counter slot.

occupied	0	1	1	0	1	0	0	0	1	0
runend	0	0	0	1	1	0	0	1	1	0
extension	0	0	0	0	0	1	0	0	0	0
slots		*a*	*b*	*c*	*d*	*d′*	*e*	*f*	*g*	

(c) The above false positive query has appeared $\tau+1$ times, where τ is the adaptivity threshold. Since it exceeds the adaptivity threshold, the filter adapts the entry for d. The filter queries the reverse map to find the key D corresponding to d and adds extension bits d'. The collision-counter slot is converted to an extension slot.

Fig. 3. Online adaptivity in the SkiQF. The SkiQF extends the AdaptiveQF using the existing metadata scheme to also support counting the number of false positive repetitions per fingerprint. Once this count exceeds τ (adaptivity threshold), the SkiQF overwrites the collision-counter slots with extension bits.

counters to provide competitive guarantees within periods where τ remains constant. Since τ is typically stable in practice, this suffices for real deployments.

Fingerprint collision counter. The SkiQF uses *collision-counter slots* in the filter to keep track of false positive repetitions. A collision-counter slot is located immediately next to a fingerprint and stores in binary the number of false positive collisions of that fingerprint. Collision-counter slots for a fingerprint are allocated only if the fingerprint was involved in at least one false positive, so fingerprints not involved in a false positive query do not use additional slots. When a false positive is detected, the SkiQF checks the fingerprint's counter value, adapting the false positive if the count equals τ. If the fingerprint is adapted, the collision-counter slots are overwritten and used for extension bits. Otherwise, the counter is incremented, allocating a collision counter slot for the fingerprint if it does not exist.

Identifying slots using metadata bits. The slots of the SkiQF contain either a remainder, extension bits, or a collision counter. The SkiQF utilizes the *runend* and *extension* bits to differentiate between the three types of slots. For each entry, the filter stores the remainder, extension bits and collision-counter slots in that order. A remainder, which always occupies exactly one slot, has its *runend* bit set to 1 if it is the last item of a run and 0 otherwise. The *extension* bit of a slot containing a remainder is always set to 0. Extension slots have their *extension* bit set to 1 and their *runend* bit set to 0. Collision counter slots have both their *extension* and *runend* bits set to 1.

Queries and adaptations. Query in the SkiQF are similar to those in the AdaptiveQF, using the *RANK* and *SELECT* mechanism of the AdaptiveQF to locate runs. When performing the *RANK* and *SELECT*, a bitwise *AND* operation on the *runend* with a negation of the *extension* bits allows the filter to ignore slots with extension or counter slots.

Adaptations are also similar to those in AdaptiveQF: the SkiQF queries the reverse map and adds extension bits to eliminate the false positive, except that the SkiQF adapts only if the counter equals τ.

While adapting, counter slots are converted to extension slots, effectively resetting the counter for the extended fingerprint. Note that, if $\tau = 0$, the filter behaves like the AdaptiveQF and adapts immediately.

5.2 HybridSkiQF

The HybridSkiQF is a variant of the SkiQF that monitors the workload to determine if false positives are repeating, adapting all false positives immediately if they are, and forgoing adaptations otherwise. To handle workload drift, the HybridSkiQF periodically updates its adaptivity strategy. Thus, the HybridSkiQF avoids the performance overhead of the SkiQF when false positives are repeated, while also avoiding the overheads of adaptivity when the workload is uniform random.

Adaptivity Policy. The HybridSkiQF revisits its adaptivity decision at the start of each *decision window*. Each decision window spans a fixed number of negative queries (false positives and true negatives). Within a decision window, the HybridSkiQF keeps track of two types of false positive queries: *repeated false positives* and *corrected false positives*. Repeated false positives are those that the HybridSkiQF has already observed in the current window, while corrected false positives are those for which the HybridSkiQF returned a correct true negative response due to a prior adaptation.

At the end of a decision window, the HybridSkiQF uses the following heuristic to determine whether to adapt or not: if the count of either repeated false positives or corrected false positives exceeds their expected value under a uniform random distribution, the underlying distribution is skewed and requires false positives to be adapted immediately. To detect workload drift without overreacting to transient fluctuations, the filter maintains a weighted moving average of both metrics across windows and uses these averaged values for adaptivity decisions. Further, we choose the decision window size w to be just large enough such that the observed false positives do not exceed their thresholds with high probability, which can be determined through concentration bounds.

To keep track of repeated false positives, the HybridSkiQF uses an additional, tiny Bloom filter called the *repeat-detector filter*. The HybridSkiQF inserts observed false positive queries into the repeat-detector filter, counting it as repeated if it is already present in the repeat-detector filter. The repeat-detector filter is reset between decision phases.

Since the HybridSkiQF only inserts false positive queries, the repeat-detector filter only needs a tiny amount of space. The repeat-detector filter has a false positive rate of ε and must be sized to handle $w \cdot \epsilon$ insertions, the expected number of false positives in a decision window of size w in a HybridSkiQF with a false positive rate of ϵ. Assuming that we use the optimal number of hash functions for the Bloom filter, the repeat-detector filter would need $\left(\ln 2 \cdot w \cdot \epsilon \cdot \log(\frac{1}{\varepsilon})\right)$ bits [5].

Repeated false positive threshold (τ_R). The threshold for repeated false positives, τ_R, is set to the number of repeated false positive queries expected in a workload with a uniform random distribution. For a decision window of size w, the expected number of repetitions detected for a uniform random workload would then be $(w \cdot \epsilon \cdot \varepsilon)$ — among the w queries, approximately $(w \cdot \epsilon)$ will be false positives returned by the HybridSkiQF, and of these, $(w \cdot \epsilon \cdot \varepsilon)$ will falsely appear as repetitions due to the repeat-detector filters's own false positive rate.

Detecting corrected false positives. To detect a corrected false positive, the HybridSkiQF checks if a query returned a true negative result due to an adaptation. The filter determines this by checking that the query fingerprint matched with an initial fingerprint in the filter, but ultimately returned a true negative result as the extension bits did not match.

Corrected false positive threshold (τ_C) . Similar to τ_R, we estimate the expected value of τ_C on a uniform random workload as the threshold to detect if a workload is skewed. A query will register as a corrected false positive if it collides with any fingerprint disregarding extension bits and the fingerprint happens to be extended. The probability of the first event occuring is ϵ and the conditional probability the second event happens given the first occuring is f, the fraction of adapted fingerprints

in the filter. Thus, the probability that a fingerprint collides with an extended fingerprint is $(\epsilon \cdot f)$, and the expected value of τ_C for a window of size w is $(w \cdot \epsilon \cdot f)$.

Example configuration. Consider a HybridSkiQF with 2^{27} slots with 8-bit remainders (178 MB, $\epsilon = 2^{-8}$). We choose $w = 1e6$ to balance statistical confidence against space overhead (larger w requires more repeat-detector space) and responsiveness to workload drift (smaller w detects changes faster). With a decision window size of $w = 1e6$, the repeat-detector expects $1e6 \cdot 2^{-8} = 3906$ inserts. Using a Bloom filter with a false positive rate of $\varepsilon = 0.0001$ and 4 hash functions, the repeat-detector filter occupies 8KB of space. Under a uniform random workload, the HybridSkiQF expects $w \cdot \epsilon \cdot \varepsilon = 4$ repeated false positives. Similarly, if the filter has performed a total of 1,250,000 adaptations so far, the fraction of slots containing adapted fingerprints is $f = 1250000/2^{27} = 0.0093$. For $w = 1e6$, the HybridSkiQF would expect $w \cdot \epsilon \cdot f = 3.6$ repeated false positives. We set both thresholds τ_R and τ_C to thrice their expected values to ensure high-confidence detection of non-uniform workloads.

Load factor. Unlike the SkiQF and AdaptiveQF, the load factor of the HybridSkiQF does not increase on uniform random workloads. The SkiQF allocates collision-counter slots even when adaptations aren't beneficial, and the AdaptiveQF creates extension slots for non-repeating queries. Since filter operations cost more as the load factor increases [4], the SkiQF and AdaptiveQF filter must be sized appropriately to maintain filter performance. On skewed workloads, all three filters will have the same load factor.

Tradeoffs compared to the SkiQF. The SkiQF makes no assumptions about the workload and provides a 2-competitive guarantee by applying the deterministic ski-rental algorithm per query. The HybridSkiQF trades this worst-case guarantee for higher average-case performance by assuming that the workload distribution is largely stable. It periodically classifies the workload as skewed or uniform and matches the better-performing strategy: adapting immediately (like an adaptive filter) or not adapting at all (like a traditional filter). When the workload distribution drifts, the HybridSkiQF detects the change and switches strategies accordingly.

This design gives the HybridSkiQF two advantages: it matches the SkiQF on uniform workloads where no false positives repeat, and outperforms it on skewed workloads where immediate adaptation is optimal. However, its binary adapt-all-or-nothing strategy cannot selectively adapt individual false positives. On workloads where only a few false positives repeat frequently among many that do not—such as extremely skewed distributions with long tails—the SkiQF outperforms the HybridSkiQF by adapting only the recurring false positives while ignoring the rest. Additionally, during distribution drift, the HybridSkiQF may briefly lag behind the SkiQF, as it requires a full decision window to detect the change, whereas the SkiQF reacts to each false positive independently.

6 Experiments

We evaluate the performance of SkiQF and HybridSkiQF through full system benchmarks and microbenchmarks. We perform system benchmarks by integrating the filters in WiredTiger [33], a disk-based B-Tree key-value store. We also perform microbenchmarks to study the various internal metrics of the filters and their performance implications. Our experiments and implementation of the filters are available as an open source repository[3].

We compare our proposed online adaptive filters against two baselines: the counting quotient filter(CQF) [35] (non-adaptive) and the AdaptiveQF [41] (adaptive). The AdaptiveQF is the state-of-the-art adaptive filter with strong guarantees and minimal adaptivity overhead, and offers superior performance compared to other adaptive filter implementations. The counting quotient filter (CQF) is the state-of-the-art fingerprint filter and requires the lowest space $(1.053(n\log(1/\epsilon) + 2.125n + o(n))$ bits) for a given false positive rate ϵ compared to other fingerprint filters (cuckoo $1.053(n\log(1/\epsilon) +$

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

Metric	B-Tree	B^ϵ-Tree
Insert Performance		
Throughput (Keys/s)	4540	9005
Relative Throughput	49%	99%
Query Performance		
Throughput (MOps/sec)	1.04	0.91
Reverse map avg. latency (C_{RM}) (ns)	121567	158882
Ratio α	1.0007	1.306

Table 1. I/O overhead of reverse map on insert and query throughput. We measure the insert overhead as relative throughput based on the baseline with no reverse map and WiredTiger as the database (9083 Keys/s insert throughput). The query throughput is measured on a workload consisting entirely of negative queries constructed from a uniform random distribution. The B-Tree reverse map uses WiredTiger [33], while the B^ϵ-tree uses SplinterDB [12] and both are configured with a 64MB main-memory buffer.

$3n+o(n)$) and vector quotient filter $1.075(n\log(1/\epsilon)+2.91n+o(n))$ [16, 36]. In I/O-bound database workloads, disk accesses dominate performance, making the false positive rate the critical metric for traditional filters rather than in-memory query speed.

6.1 Experimental setup

We evaluate query workloads using an on-disk WiredTiger [33] database paired with an in-memory filter containing the database keys. The filter prevents negative queries — queries for non-existent keys — from triggering unnecessary disk accesses. Following prior filter evaluations [16, 41], we generate workloads consisting entirely of negative queries. This isolates the filter's impact on database performance, as filters only accelerate negative queries. Each query first checks the in-memory filter. Only positive filter responses proceed to query WiredTiger on disk. When WiredTiger fails to find a key (confirming a false positive), it notifies the filter to trigger an adaptation according to the filter's adaptation policy.

Database. The database instance contains a single 2-column table with 8-byte keys and 8-byte values, representing standard 64-bit identifiers used in practice. The table consists of 120M key-value pairs. All 120M key-value pairs are generated uniformly at random from the universe $[0,2^{64}-1]$, occupying 2.2 GB on disk.

Note that our results generalize to other key and value sizes. Since filter false positive rates depend on the number of keys present in the set and not the universe size, larger keys do not affect the filter's impact on database performance. Similarly, value size is irrelevant as our workloads only contain negative queries.

Reverse Map. The adaptive filter variants (SkiQF, HybridSkiQF, and AdaptiveQF) use a separate WiredTiger instance as the reverse map. This instance is configured with a 4MB buffer pool (in-memory cache), and stores a single table mapping fingerprints to 8-byte integer keys. Since fingerprints can repeat in a filter, the offset of the fingerprint within its run (as described in Section 3) is used to differentiate between duplicate fingerprints. This means that the reverse map is only accessed in read-only mode and no updates are required during adaptations.

Table 1 reports insert overhead and query performance for two reverse map indexes: a B-Tree (WiredTiger [33]) and a write-optimized B^ϵ-tree (SplinterDB [12]). We measure insert throughput by inserting 15 million 8-byte keys and values drawn from a uniform random distribution. Query throughput is measured over one million negative queries, also drawn from a uniform random distribution. The table also reports the average reverse map latency and $\alpha = C_{RM}/C_{DB}$.

The B-Tree achieves higher query throughput compared to the B^ϵ-tree (1.04 MOps/sec vs. 0.91 MOps/sec) due to its lower latency (121 μs vs. 158 μs), but incurs 51% higher overhead on inserts. The write-optimized B^ϵ-tree maintains nearly equivalent insert throughput (99%) compared to the baseline with no reverse map at the cost of slightly slower reverse map queries. Both structures have similar adaptivity thresholds, with $\lfloor \alpha \rfloor = 1$ (i.e., SkiQF adapts on the second repeated false positive). Since the B-Tree's α ratio is closer to its floor value, it presents a more challenging case and is thus the focus of our experiments. However, our approach generalizes to any reverse map and any value of α.

Database and buffer pools. The adaptive filters use a 64MB buffer pool for the database WiredTiger instance and an additional 4MB buffer pool for the reverse map instance, totaling 68MB. To maintain fairness, we configure the non-adaptive filter (CQF) with a 68MB database buffer pool, ensuring all filters use the same total memory. The 64MB database buffer pool represents approximately 6.25% of the total database size.

Filter configurations. All filters employed in our evaluation are fingerprint filters. They are configured to operate at 90% load factor and initialized with 2^{27} slots. Each slot is 8-bit, thereby guaranteeing a maximum false positive rate of $\epsilon = 2^{-8}$. As reported in Table 1, our reverse map and database setup has a cost ratio of $\lfloor \alpha \rfloor = 1$. Therefore, we configure the SkiQF to adapt a false positive on its second occurrence. The HybridSkiQF is configured with a decision window size of 1 million queries with a smoothing factor of 0.7 and configured with a 8KB repeat-detector Bloom filter with 4 hash functions (max false positive rate of 0.001 for 3092 inserts).

System specification. All the experiments were run on an Intel(R) Xeon(R) 6710E CPU @ 2.40GHz with two NUMA nodes, 64 cores and 96MB L3 cache per node. The machine has 1TB of DRAM running Linux kernel 5.14 and a Dell DC NVMe PM9A3 RI U.2 with 3.84TB total capacity. All experiments were run using a single thread.

6.2 Robustness across workloads distributions

In this experiment, we evaluate the database performance using the various filters across query workloads of varying skewness. The goal of this experiment is to show that the online adaptive filters introduced in this paper (SkiQF and HybridSkiQF) provide robust performance guarantees independent of the skewness in the workload distribution. Figure 4a shows the instantaneous throughput, measured at intervals of 1% of the query workload, while Figure 4b shows the overall (aggregate) throughput.

Query distributions. We evaluate all the filters on workloads containing keys drawn from both skewed and uniform random distributions. Each query workload consists of 100 million queries, generated using the following distributions:

- **Uniform:** The keys are drawn uniformly at random from the universe of keys($[0, 2^{64}-1]$).
- **Normal:** Frequencies of keys follow a normal distribution. 5 million samples are drawn from a normal distribution ($\mu = 0.0, \sigma = 1.0$) to be frequency weights.The weights are shifted to be positive and normalized to add up to the desired number of queries, which in our tests is 100 million. A random integer is assigned to each frequency weight to generate the final distribution.
- **Lognormal:** Frequencies of keys are generated in the similar manner as before using a lognormal distribution($\mu = 0.0, \sigma = 1.0$). A lognormal distribution is one where the logarithms of the values follow a normal distribution.
- **Zipfian:** The keys are drawn from a Zipfian distribution with $\alpha = 0.99$, which simulates the Zipfian distribution parameter used in the YCSB benchmark [13].

The workload consists of entirely negative queries, i.e., keys that are not present in the input set. The queries are generated by sampling from the universe using the above distributions and discarding keys that are present in the dataset.

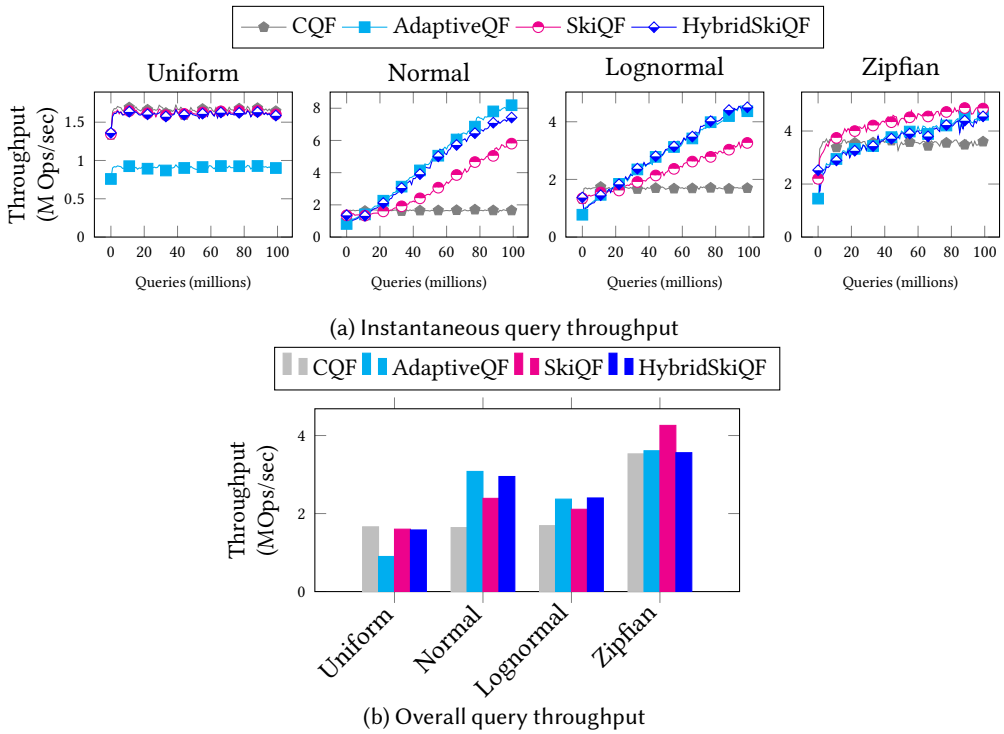


Fig. 4. Database query throughput with filters across various distributions. (Higher is better.)

Uniform Random (No skew). On uniform random workloads, the SkiQF (1.60 MOps/sec) and the HybridSkiQF (1.58 MOps/sec) achieve the highest query throughput. For uniform random workload, false positives seldom repeat and adaptations in the AdaptiveQF therefore contribute towards overhead without any performance improvements. The SkiQF and the HybridSkiQF are able to avoid adapting by detecting that false positives are not repeating. These filters have similar performance as the CQF (1.66 MOps/sec). As adaptations only contribute towards overhead, the AdaptiveQF has the lowest throughput (0.90 MOps/sec). The HybridSkiQF also achieves the highest query throughput without an increase in its load factor, while the load factor of the other adaptive filters (AdaptiveQF, SkiQF) increases slowly over the duration of the workload (Figure 10).

Normal and Lognormal (Medium skew). On the normal and lognormal workloads, which are skewed workloads, the HybridSkiQF (2.95 MOps/sec for normal, 2.40 MOps/sec for lognormal) offers similar performance to the AdaptiveQF (3.08 MOps/sec for normal, 2.37 MOps/sec for lognormal). The SkiQF has a slightly lower throughput compared to the AdaptiveQF but is within its 1.5-competitive bound (2.39 MOps/sec for normal, 2.11 MOps/sec for lognormal). The difference in performance is due to the SkiQF following the deterministic ski-rental algorithm and adapting a false positive on the second occurrence given the derived α value. In contrast, the HybridSkiQF and the AdaptiveQF adapt all false positives immediately and avoid extra false positives.

Zipfian (High skew). On the Zipfian workload (with $\alpha = 0.99$), the SkiQF (4.26 MOps/sec) offers the highest throughput. The HybridSkiQF (3.56 MOps/sec) and the AdaptiveQF (3.60 MOps/sec) perform the next best. Both filters adapt false positives immediately. This strategy proves suboptimal for this workload: while some false positives repeat frequently and benefit from adaptation, a long tail of one-off false positives also exists. Adapting these non-repeating false positives creates overhead that

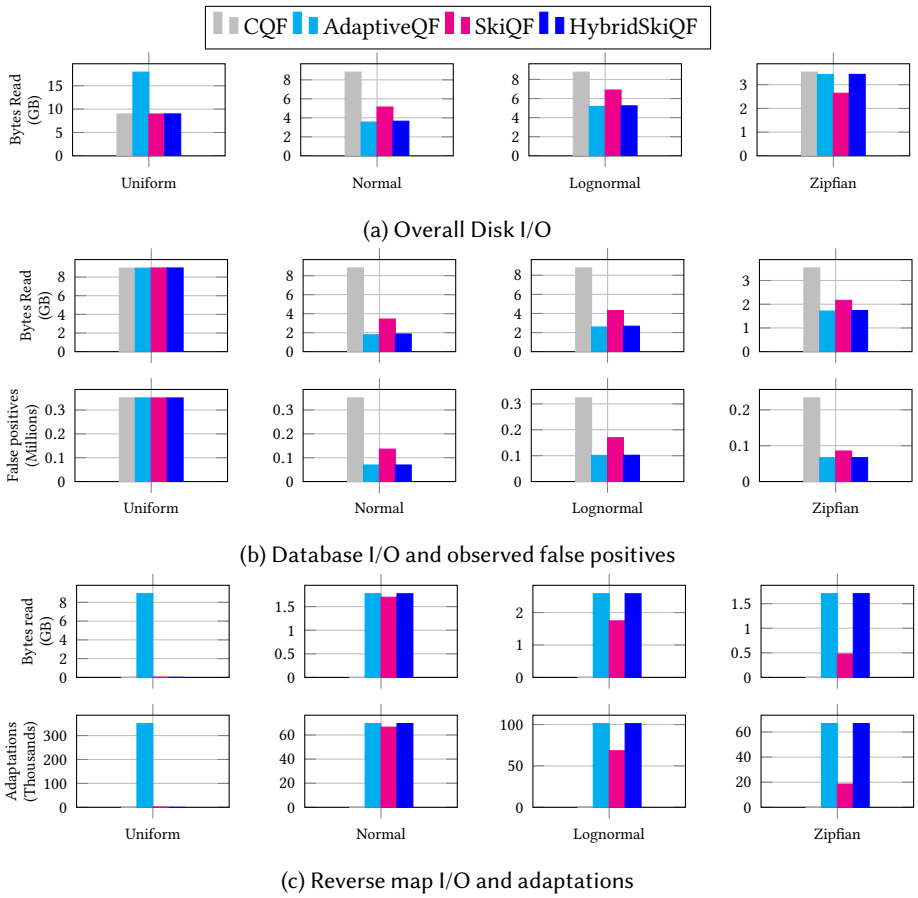


Fig. 5. Disk I/O, false positives and adaptations breakdown. (Lower is better for all.)

reduces the AdaptiveQF and HybridSkiQF to similar throughput as the CQF (3.53 MOps/sec). The SkiQF avoids this problem by adapting only repeated false positives.

Database and reverse map I/O breakdown. We also measure the total I/O performed by the database and the reverse map while running the query workloads to validate the theoretical bounds presented in Section 4. The I/O measurements were retrieved from the statistics provided by the WiredTiger instances, for the database and the reverse map. Figure 5a plots the total I/O performed in bytes during the query workloads. The total I/O reported in Figure 5a correlates strongly with the query throughput reported in Figure 4. This is expected, as database performance is bottlenecked on disk I/Os; therefore, the more the I/O performed, the lower the throughput. Figure 5b and Figure 5c breakdown the I/O cost across the database and the reverse map.

On the workload with a uniform random distribution, all the filters return the same amount of false positives and therefore incur equal database I/O costs. However, the AdaptiveQF incurs additional I/Os to adapt the false positives, resulting in twice the overall I/O cost compared to the other filters.

On workloads with normal and lognormal distributions, the CQF has the highest database I/O cost due to repeated false positives. Among the adaptive filters, the SkiQF ends up having twice the database I/O cost compared to the AdaptiveQF and HybridSkiQF, as they adapt on the second false positive occurrence.

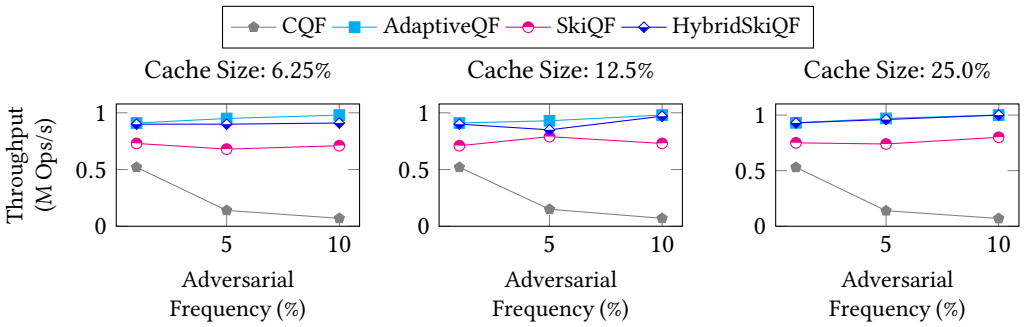


Fig. 6. Database query throughput of the filters on adversarial query workloads. (Higher is better.)

On the workload with Zipfian distribution, the SkiQF has the lowest overall I/O cost. It only adapts recurring false positives and avoids adapting the long tail of one-off false positives. Despite the CQF returning an orders-of-magnitude higher number of false positives, it only has twice the database I/O cost compared to all the adaptive filters. This is because the vast number of false positive queries are repeated, causing their index pages to be cached inside the buffer pool of the database.

6.3 Adversarial workload

In this benchmark, we evaluate whether online adaptive filters (SkiQF and HybridSkiQF) provide the same adversarial robustness as strongly-adaptive filters (AdaptiveQF). Our benchmark is similar to the one proposed by Wen et al. [41].

Motivation. We simulate an attacker that aims to degrade database system performance by exploiting false positive queries. The attacker issues random queries and identifies false positives through timing analysis—false positives access disk and exhibit higher latency compared to true negatives that are returned after a negative response from in-memory filter. After collecting these false positives, the attacker replays them to force disk I/Os and degrade performance. To bypass caching, the attacker spaces these replays to ensure cache entries expire between repetitions.

Test Setup. We evaluate the adversarial benchmark across varying cache sizes (6.25%, 12.5%, 25.0% of dataset size) and adversarial query proportions (1%, 5%, 10%). An x% adversarial frequency means the attacker replaces x% of queries with known false positives. The attacker cycles through false positives, removing any that have been adapted. The filter is initialized with $2^{27} = 134M$ slots containing 8-bit remainders and filled to 90% load factor. The query workload consists of 40M uniformly random 8-byte unsigned integers.

Results summary. We report adversarial benchmark results in Figure 6. The online adaptive filters are able to guarantee robustness against adversarial workloads. Across all configurations, the AdaptiveQF and HybridSkiQF have the best performance, between $0.90-1.01$ MOps/sec. The SkiQF is able to maintain its 1.5-competitiveness compared to AdaptiveQF, with its performance ranging from $0.68-0.75$ MOps/sec. The CQF is specifically susceptible to the adversarial attack, with performance deteriorating as the adversarial frequency increases (0.54 MOps/sec for 1%, 0.15 MOps/sec at 5%, 0.07 MOps/sec at 10%). Furthermore, the CQF does not benefit from using a larger cache, as the adversary can always issue enough random queries to flush the cache.

6.4 Shifting workload distributions

We evaluate the performance of the online adaptive filter when the query distribution changes during a workload. We construct the workload with distribution shifts based on prior literature [2, 43]. Specifically, we test the filters against workloads with phases, each drawing queries from a different query

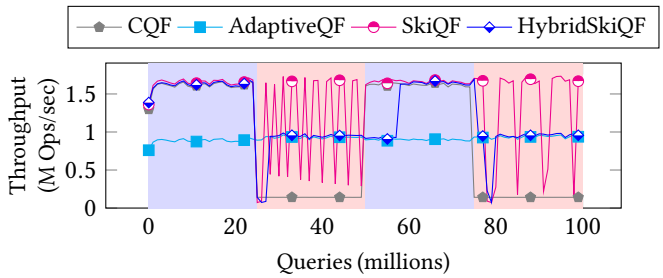


Fig. 7. Database query throughput under workload drift. The workload alternates between uniform random queries (blue) and adversarial queries (red). HybridSkiQF can detect distribution drifts and adjusts its adaptivity policy online, matching the CQF (non-adaptive filter) during uniform phases and AdaptiveQF during adversarial phases. (Higher is better.)

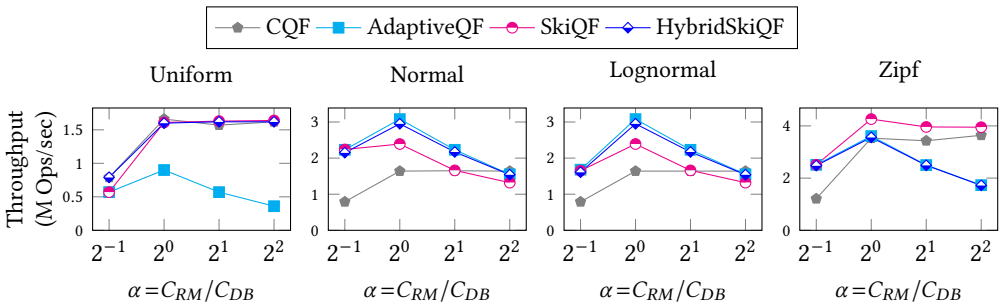


Fig. 8. Database query throughput of the filters for varying α, the ratio of the reverse map (C_{RM}) and database cost (C_{DB}). (Higher is better.)

distribution. We evaluate the filters against a workload of 100 million queries divided into 4 phases of 25 million queries each. Starting with a uniform random phase, the workload alternates between uniform random workloads and adversarially generated workloads (5% adversarial query rate).

Figure 7 shows the instantaneous query throughput (throughput to perform 1 million operations). During the uniform random phases (shaded blue in Figure 7), the SkiQF and HybridSkiQF match CQF's performance (≈ 1.66 MOps/sec), while the AdaptiveQF's throughput is 45% lower (0.90 MOps/sec) due to unnecessary adaptations. When the workload switches to adversarial (shaded in red), the CQF's throughput drops to 0.14 MOps/sec while the HybridSkiQF matches the AdaptiveQF. Note that the HybridSkiQF incurs a brief detection lag at each distribution shift, as it requires a full decision window to identify the change and update its policy.

6.5 Varying external-memory costs

In this experiment, we study the performance of the filters across different values of α, the ratio of the cost of the reverse map to the database. As reported in Table 1, our base experimental setup has an α value of 1. We simulate different values of α by introducing an artificial delay using a thread sleep. The delays added are in increments of the median latency reported in Table 1. For example, to get an α value of $1/2$, we introduce a delay of the average latency (130μs) to the database lookup, and for $\alpha = 2$, the sleep is added to the reverse map lookup.

The experiment uses a similar setup as the robustness study (Section 6.2), and we plot the query throughput in Figure 8. Both the online adaptive filters avoid the overhead of adapting in the uniform random workload, matching the performance of the traditional filter. For skewed workloads, the SkiQF has the best or close to best performance for each experiment run, and is flexible enough to

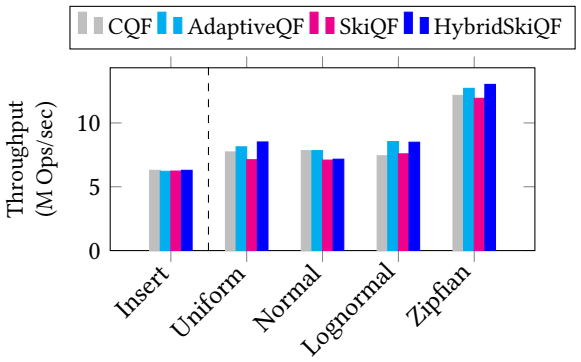


Fig. 9. Main-memory insertion and query throughput of various filters. The adaptive filters were queried after adapting at 95% load factor, the non-adaptive filter was queried at 90% load factor. (Higher is better.)

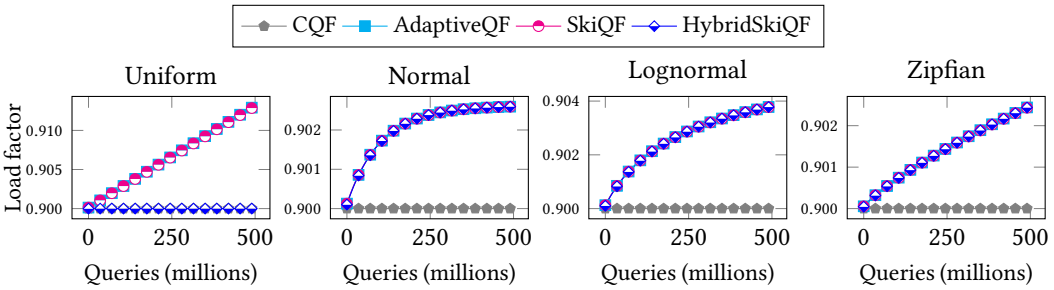


Fig. 10. Load factor across various distributions. (Lower is better, y-axis starts at 0.90.)

operate at different values of α. The throughput of the AdaptiveQF decreases as α increases as the cost of querying the reverse map increases. When $\alpha = 1$, the performance of the AdaptiveQF on the uniform random workload represents the downsides of adapting immediately (2× lower throughput on uniform random workloads) as opposed to the second adaptation. When adapting immediately is optimal (skewed workloads with $\alpha \leq 1$), the HybridSkiQF is the best performing filter, with a 28% higher throughput compared to the SkiQF.

6.6 Microbenchmarks

In the microbenchmarks, we evaluate the filters in main memory. We initialize all filters with 2^{27} slots, 8-bit remainders, and fill them to 90% load factor using uniform random keys.

Filter-only insert and query throughput. This experiment evaluates the in-memory performance of filter operations. Although disk I/O dominates false positive costs, in-memory performance remains critical for true negative queries. We run a query workload of 500 million of various distributions. Our throughput measurements exclude database access and adaptation cost (reverse map accesses).

Our experiments show that the online adaptive filters introduce negligible overhead to in-memory operations. Figure 9 shows similar performance for all filters (traditional, adaptive, and online adaptive) at 7.06-8.51 MOps/sec for most workloads. The Zipfian workload achieves higher throughput, likely due to its low unique query count keeping relevant filter slots cached. Insertion throughput measurements, conducted without database or reverse map updates, confirm similar in-memory performance across all filters.

Load factor. Figure 10 plots the filter's load factor across various query distributions. On skewed query workloads (normal, lognormal, Zipfian), all adaptive filters increase load factor at similar rates. The SkiQF achieves similar load factors to AdaptiveQF but performs different numbers of adaptations. Each false positive uses a slot in the filter to track repetitions, regardless of adaptation. This is clearest

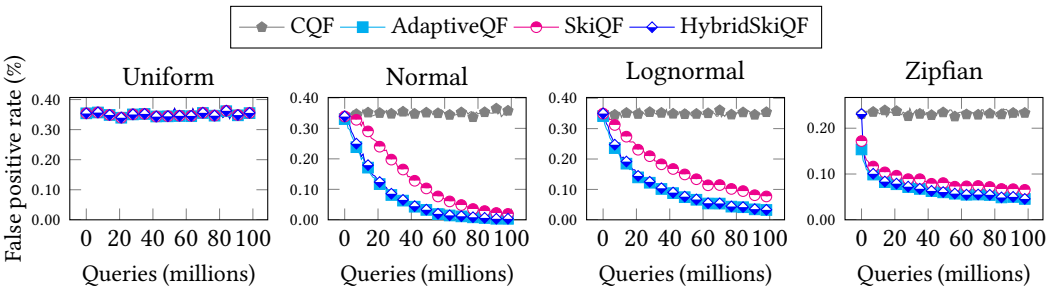


Fig. 11. False positive rate across various distributions. (Lower is better.)

on uniform workloads, where SkiQF matches AdaptiveQF's load factor despite performing nearly zero adaptations. The HybridSkiQF matches CQF's load factor on uniform workloads. The CQF maintains 90% load factor across all workloads since it never adapts.

False positive rate. Figure 11 shows false positive rates across query distributions. We measure the instantaneous false positive rate per million queries (1% of the 100M workload). All filters begin at the configured rate $\epsilon = 2^{-8} = 0.0039$. For the skewed workloads, as the number of adaptations increase, the false positive rate decreases. The SkiQF's rate decreases slightly slower as it delays adaptation until the break-even point. On the other hand, the CQF's false positive rate is significantly higher compared to the adaptive filters as it incurs repeated false positives.

Space Usage. The CQF has the lowest space usage at 162.13 MB of main memory. The AdaptiveQF and SkiQF use 178.15 MB, an overhead of about 8-9% compared to the CQF, which is due to the additional metadata bit used to keep track of extension slots. The HybridSkiQF occupies 178.16MB, a negligible 0.008% (16KB) space overhead due to the repeat-detector Bloom filter. The space usage of the SkiQF and HybridSkiQF is same as the AdaptiveQF because they have the same underlying structure without any added space overhead.

7 Discussion and conclusion

The SkiQF and HybridSkiQF present an approach that addresses a key limitation of adaptive filters, the slowdown in performance due to the overhead of adaptivity. Previously, applications had to choose between the benefits of adaptivity—lowered false positive rates on skewed workloads and protection against adversarially generated workloads—and performance on commonly occurring workloads with low or no repeating false positives.

By modelling and analyzing when to adapt as a ski rental problem, we take a principled approach to designing an adaptive filter that has the benefits of adaptivity, while also matching the performance of non-adaptive filters when adaptivity is not needed. Our experiments, which consist of workloads with query distributions of varying skewness, show that the SkiQF and HybridSkiQF improve the performance of the AdaptiveQF by 21% and 27% respectively, while demonstrating robustness by never performing worse than the CQF, a non-adaptive filter. Our microbenchmarks also show that the SkiQF and HybridSkiQF implement the ski rental algorithm with no overhead.

With its robust performance, we expect the SkiQF and HybridSkiQF to be the go-to filter data structures in modern databases. The SkiQF provides strong guarantees and can react to changes in the underlying query distribution immediately. The HybridSkiQF trades robustness for performance and works well for workloads that follow a consistent pattern. Our evaluation shows that filters in databases can support adaptivity without performance overhead.

8 Acknowledgments

This research was funded in part by NSF grant OAC 2517201, 2513656.

References

[1] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127. doi:10.1145/48529.48535

[2] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1626–1639. doi:10.1145/3514221.3526121

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

[4] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2011. Don't Thrash: How to Cache Your Hash on Flash. In *3rd USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'11, Portland, OR, USA, June 14, 2011*, Irfan Ahmad (Ed.). USENIX Association. https://www.usenix.org/conference/hotstorage11/dont-thrash-how-cache-your-hash-flash

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

[6] Phelim Bradley, Henk C Den Bakker, Eduardo PC Rocha, Gil McVean, and Zamin Iqbal. 2019. Ultrafast search of all deposited bacterial and viral genomic data. *Nature biotechnology* 37, 2 (2019), 152–159.

[7] Andrei Z. Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Math.* 1, 4 (2003), 485–509. doi:10.1080/15427951.2004.10129096

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[9] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark N. Wegman. 1978. Exact and Approximate Membership Testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho (Eds.). ACM, 59–65. doi:10.1145/800133.804332

[10] Pedro Celis, Per-Åke Larson, and J. Ian Munro. 1985. Robin Hood Hashing (Preliminary Report). In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*. IEEE Computer Society, 281–288. doi:10.1109/SFCS.1985.48

[11] Justin Chu, Sara Sadeghi, Anthony Raymond, Shaun D Jackman, Ka Ming Nip, Richard Mar, Hamid Mohamadi, Yaron S Butterfield, A Gordon Robertson, and Inanc Birol. 2014. BioBloom tools: fast, accurate and memory-efficient host species sequence screening using bloom filters. *Bioinformatics* 30, 23 (2014), 3402–3404.

[12] Alex Conway, Martin Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proc. ACM Manag. Data* 1, 1 (2023), 46:1–46:27. doi:10.1145/3588726

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. doi:10.1145/1807128.1807152

[14] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 79–94. doi:10.1145/3035918.3064054

[15] Biplob K. Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proceedings of the 2010 USENIX Annual Technical Conference, USENIX ATC 2010, Boston, MA, USA, June 23-25, 2010*, Paul Barham and Timothy Roscoe (Eds.). USENIX Association. https://www.usenix.org/conference/usenix-atc-10/chunkstash-speeding-inline-storage-deduplication-using-flash-memory

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

[17] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2012. The TokuFS Streaming File System. In *4th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'12, Boston, MA, USA, June 13-14, 2012*, Raju Rangaswami (Ed.). USENIX Association. https://www.usenix.org/conference/hotstorage12/workshop-program/presentation/esmet

[18] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, Aruna Seneviratne, Christophe Diot, Jim Kurose,

Augustin Chaintreau, and Luigi Rizzo (Eds.). ACM, 75–88. doi:10.1145/2674005.2674994

[19] Horst Feistel. 1973. Cryptography and Computer Privacy. *Scientific American* 228, 5 (May 1973), 15–23. doi:10.1038/scientificamerican0573-15

[20] Geoffrey Irving. 2012. Inverse of a hash function. https://naml.us/post/inverse-of-a-hash-function/

[21] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, Jiri Schindler and Erez Zadok (Eds.). USENIX Association, 301–315. https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen

[22] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. 2009. LIPSIN: line speed publish/subscribe inter-networking. In *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain, August 16-21, 2009*, Pablo Rodriguez, Ernst W. Biersack, Konstantina Papagiannaki, and Luigi Rizzo (Eds.). ACM, 195–206. doi:10.1145/1592568.1592592

[23] Anna R. Karlin, Claire Kenyon, and Dana Randall. 2001. Dynamic TCP acknowledgement and other stories about e/(e-1). In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing* (Hersonissos, Greece) *(STOC '01)*. Association for Computing Machinery, New York, NY, USA, 502–509. doi:10.1145/380752.380845

[24] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. 1990. Competitive Randomized Algorithms for Non-Uniform Problems. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*. Society for Industrial and Applied Mathematics, USA, 301–309.

[25] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel Dominic Sleator. 1988. Competitive Snoopy Caching. *Algorithmica* 3 (1988), 77–119. doi:10.1007/BF01762111

[26] Don Knuth. 1962. NOTES ON "OPEN" ADDRESSING. (1962). https://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf

[27] Donald E. Knuth. 1973. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison Wesley, Reading, MA.

[28] Ravi Kumar, Todd Lipcon, Manish Purohit, and Tamás Sarlós. 2025. Linear Elastic Caching via Ski Rental. In *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org. https://vldb.org/cidrdb/2025/linear-elastic-caching-via-ski-rental.html

[29] David J. Lee, Samuel McCauley, Shikha Singh, and Max Stein. 2021. Telescoping Filter: A Practical Adaptive Filter. 60:1–60:18 pages. doi:10.4230/LIPICS.ESA.2021.60

[30] Heng Li. 2015. Invertible Hash Function Implementation. https://gist.github.com/lh3/59882d6b96166dfc3d8d. GitHub Gist. Accessed: 2026-02-02.

[31] Michael Luby and Charles Rackoff. 1988. How to Construct Pseudorandom Permutations from Pseudorandom Functions. *SIAM J. Comput.* 17, 2 (1988), 373–386. doi:10.1137/0217022

[32] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2020. Adaptive Cuckoo Filters. *ACM J. Exp. Algorithmics* 25 (2020), 1–20. doi:10.1145/3339504

[33] MongoDB. 2024. *WiredTiger Storage Engine*. https://www.mongodb.com/docs/manual/core/wiredtiger/

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

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

[36] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1386–1399. doi:10.1145/3448016.3452841

[37] Krishna P. N. Puttaswamy, Thyaga Nandagopal, and Murali S. Kodialam. 2012. Frugal storage for cloud file systems. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 71–84. doi:10.1145/2168836.2168845

[38] Brandon Reagen, Udit Gupta, Robert Adolf, Michael Mitzenmacher, Alexander M. Rush, Gu-Yeon Wei, and David Brooks. 2018. Weightless: Lossy weight encoding for deep neural network compression. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJpXxgaIG

[39] Pedro Reviriego, Alfonso Sánchez-Macián, Salvatore Pontarelli, Shanshan Liu, and Fabrizio Lombardi. 2022. Attacking Adaptive Cuckoo Filters: Too Much Adaptation Can Kill You. *IEEE Trans. Netw. Serv. Manag.* 19, 4 (2022), 5224–5236. doi:10.1109/TNSM.2022.3182906

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

[41] Richard Wen, Hunter McCoy, David Tench, Guido Tagliavini, Michael A. Bender, Alex Conway, Martin Farach-Colton, Rob Johnson, and Prashant Pandey. 2024. Adaptive Quotient Filters. *Proc. ACM Manag. Data* 2, 4 (Sept. 2024), 192:1–192:28. doi:10.1145/3677128

[42] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. 2009. BUFFALO: bloom filter forwarding architecture for large organizations. In *Proceedings of the 2009 ACM Conference on Emerging Networking Experiments and Technology, CoNEXT 2009, Rome, Italy, December 1-4, 2009*, Jörg Liebeherr, Giorgio Ventre, Ernst W. Biersack, and Srinivasan Keshav (Eds.). ACM, 313–324. doi:10.1145/1658939.1658975

[43] Xinjing Zhou, Xiangpeng Hao, Xiangyao Yu, and Michael Stonebraker. 2025. Tiered-Indexing: Optimizing Access Methods for Skew. *VLDB J.* 34, 4 (2025), 45. doi:10.1007/S00778-025-00928-6

Received October 2025; revised January 2026; accepted February 2026