Adaptive Quotient Filters

RICHARD WEN, University of Maryland, USA
HUNTER MCCOY, University of Utah, USA
DAVID TENCH, Lawrence Berkeley National Labs, USA
GUIDO TAGLIAVINI, Rutgers University, USA
MICHAEL A. BENDER, Stony Brook University, USA
ALEX CONWAY, Cornell Tech, USA
MARTIN FARACH-COLTON, New York University, USA
ROB JOHNSON, VMware Research, USA
PRASHANT PANDEY, University of Utah, USA

Filters trade off accuracy for space and occasionally return false positive matches with a bounded error. A fundamental limitation in traditional filters is that they do not change their representation upon seeing a false positive match. Therefore, the maximum false positive rate is only guaranteed for a single query, not for an arbitrary set of queries. We can improve the filter's performance on a stream of queries, especially on a skewed distribution, if we can adapt after encountering false positives.

Adaptive filters, such as telescoping quotient filters and adaptive cuckoo 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.

However, existing adaptive filters are not practical for two main reasons. First, they offer weak adaptivity guarantees, meaning that fixing a new false positive can cause a previously fixed false positive to come back. Secondly, the sub-optimal design of the auxiliary structure results in adaptivity overheads so substantial that they can actually diminish overall system performance compared to a traditional filter.

In this paper, we design and implement the AdaptiveQF, the first practical adaptive filter with minimal adaptivity overhead and strong adaptivity guarantees, which means that the performance and false-positive guarantees continue to hold even for adversarial workloads. The AdaptiveQF is based on the state-of-the-art quotient filter design and preserves all the critical features of the quotient filter such as cache efficiency and mergeability. Furthermore, we employ a new auxiliary structure design which results in considerably low adaptivity overhead and makes the AdaptiveQF practical in real systems.

We evaluate the AdaptiveQF by using it to filter queries to an on-disk B-tree database and find no negative impact on insert or query performance compared to traditional filters. Against adversarial workloads, the AdaptiveQF preserves system performance, whereas traditional filters incur 2× slowdown from adversaries representing as low as 1% of the workload. Finally, we show that on skewed query workloads, the AdaptiveQF can reduce the false-positive rate 100× using negligible (1/1000th of a bit per item) space overhead.

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

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

Authors' Contact Information: Richard Wen, rwen1@umd.edu, University of Maryland, USA; Hunter McCoy, hunter@cs.utah.edu, University of Utah, USA; David Tench, dtench@pm.me, Lawrence Berkeley National Labs, USA; Guido Tagliavini, guido.tag@rutgers.edu, Rutgers University, USA; Michael A. Bender, bender@cs.stonybrook.edu, Stony Brook University, USA; Alex Conway, ajc473@cornell.edu, Cornell Tech, USA; Martin Farach-Colton, martin@farach-colton.com, New York University, USA; Rob Johnson, robj@vmware.com, VMware Research, USA; Prashant Pandey, ppandey@cs.utah.edu, University of Utah, USA.

ACM 2836-6573/2024/9-ART192
https://doi.org/10.1145/3677128

ACM Reference Format:
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 (SIGMOD), Article 192 (September 2024), 28 pages. https://doi.org/10.1145/3677128

1 Introduction

Filters [11, 41, 46, 71] are a go-to data structure in systems builders' toolkits. Filters maintain a compact representation of a set of items, saving space by allowing a small ***false-positive rate*** ε: a membership query to a filter for set S returns YES for any $x \in S$ and returns NO with probability at least $1-\varepsilon$ for any $x \notin S$.

Filters are powerful because allowing false-positives dramatically reduces the space required to store S. For example, if we are required to answer queries on S with no errors, then the size of a data structure is at least $\log\binom{u}{n} = \Omega(n\log(u/n))$ bits, where u is the size of the universe [19]. In contrast, modern filters have size $n\log(1/\varepsilon)+cn$, where c is between 2 and 3 [9, 71]. This means that, for typical false-positive rates around 1% to 0.1%, a filter can store one or two bytes of information per item, no matter how large the universe. This bound is tight up to lower-order terms in that any filter requires at least $n\log(1/\varepsilon)$ bits [19]. Filters have been extensively used to compactly summarize a set of items in networks, storage systems, machine learning, computational biology, and other areas [2, 13, 16, 27, 33, 34, 40, 44, 50–52, 74, 75, 78, 86, 87, 91, 93, 94, 96].

Types of problems. The following problem settings offer challenges for traditional filters and opportunities for improvement:

- **Static YES/NO lists.** Given a set Y of YES items and a set N of NO items chosen from a universe U, build a data structure that answers YES to any query for an item in Y, NO for any query for an item in N, and answers NO with probability at least $1-\varepsilon$ for any other item in U.
- **Dynamic YES/NO lists.** This is similar to the static YES/NO list problem, except that the sets Y and N may be updated dynamically.
- **Skewed query distributions.** In some settings, the frequency distribution of queries may be highly skewed. In such settings, the observed false-positive rate of the filter can be very far from the expected rate, ε. For example, if all the queries are for a single item, the observed false-positive rate will be 0 or 1, but not in between. Avoiding repeated mistakes can reduce the false-positive rate of a filter, or equivalently, reduce the filter size needed to achieve some target error rate. In summary, filters that ignore the skew may perform arbitrarily poorly, whereas filters that exploit the skew can outperform the lower bounds.
- **Adversarial queries.** In this problem, the goal is to design a filter that guarantees that the fraction of queries from an adversary that are false positives is at most ε, even when the queries are chosen by an adversary that is trying to cause the filter to return as many false positives as possible. Here we assume the attacker can detect when a query results in a false positive and can repeat queries arbitrarily. This is a more general case of the skewed-query distribution.

Prior work. Prior work has considered each of these problems separately, and consequently has developed distinct approaches to solving each of them. Chazelle et al. [22] describe ***Bloomier filters***, which encode static YES/NO lists. Bloomier filters also support a limited form of dynamicity—they support moving items between Y and N but not adding or deleting items. Tripunitara and Carbunar [89] introduce ***cascading Bloom filters*** to solve the static YES/NO list problem, and these are used in many systems [25, 55, 64, 82, 83]. Reviriego et al. [80] proposed an extension of the ***static xor filter*** to support the static YES/NO list problem. Li et al. [59] proposed the ***seesaw counting filter*** for the dynamic YES/NO list problem, specifically in the context of detecting malicious URLs. Mitzenmacher et al. [62] proposed ***adaptive cuckoo filters*** to solve the skewed-query-distribution problem. Bender

et al. [7] define the notion of an adaptive filter, which offers strong guarantees on the number of false positives that an application will see, even with a skewed or even adversarial query distribution, and present the broom filter, which meets their definition. Bender et al. [6] analyzed the performance of ***broom filters*** [7] on queries that obey Zipfian distributions. Lee et al. [56] proposed ***telescoping filters*** to address the skewed query distribution problem.

This paper. We argue that all of these problems can be naturally solved, with comparable space and better performance than the prior special-purpose solutions, by what we call a ***monotonically adaptive filter***, which is a filter that never forgets a false positive. Furthermore, we show how to build fully dynamic monotonically adaptive filters from quotient filters [70]. We design, build, and evaluate a fully dynamic monotonically adaptive filter, the AdaptiveQF, and show that it outperforms several prior solutions to these problems. Finally, we prove lower bounds on the space required to solve the yes/no list problem, showing that the AdaptiveQF is space-optimal.

Like adaptive filters, monotonically adaptive filters can ***adapt***, i.e. they can update their state to correct false positives. Bender et al. defined what it means for a filter to be adaptive: every query has a probability of at most ε of returning a false positive, independent of the outcome of all prior queries [7]. Adaptivity is a very strong property: it guarantees that after n queries—even adversarially generated queries—the upper bound on the number of false positives is tightly concentrated around εn. Specifically, the system will see at most $\varepsilon n + O\left(\sqrt{\varepsilon n \log n} + \log n\right)$ false positives with high probability.

Adaptive filters require two things: feedback about their false positives and auxiliary data to correct them. For example, if an adaptive filter is used by an application to avoid database lookups for non-existent items, then the application can inform the filter that a query for an item x was a false positive if the subsequent database query returned that x is not present in the database. Adaptive filters also need an auxiliary structure to store information to support adaptation. Bender et al. showed that this auxiliary information is necessary and in fact must be quite large: the total size of an adaptive filter on a set S essentially must be large enough to store S [7]. The trick is to break the filter into two parts, a small in-memory component that is accessed on every query and a large auxiliary structure that is accessed only during adaptations and hence can reside in slower storage. Note that all proposed adaptive filters have this overall structure. In some applications, such as when the filter is in front of a database, the database may be able to serve as the auxiliary structure, so that the total storage requirements of the system remain essentially unchanged. See Bender et al. for more discussion [7].

What makes monotonically adaptive filters special is that, when they adapt, their false-positive set only shrinks. Prior proposed adaptive filters were not monotonic: fixing one false positive could cause other elements to become false positives. Even filters that meet Bender et al.'s strong definition of adaptivity need not be monotonic. For example, Bender et al.'s broom filter periodically rotates its hash function, at which point it forgets all the false positives it corrected under the old hash function.

Fingerprint filters, such as the quotient filter [70], are good candidates for building practical monotonically adaptive filters because they store a set S by compactly storing the set $h(S) = \{h(x) \mid x \in S\}$, where h is a hash function and $h(x)$ is called the ***fingerprint*** of x. A query for y simply checks whether $h(y) \in h(S)$, so the only source of false positives is fingerprint collisions. Fingerprint filters support a false-positive rate of ε on a set of size n by using $\log(n/\varepsilon)$-bit fingerprints and typically store the first $\log n$ bits of each fingerprint implicitly so that the per-item space is $\log(1/\varepsilon) + O(1)$ bits. To make a fingerprint filter monotonically adaptive, we need only to be able to eliminate fingerprint collisions. To do so, we can use a hash function h that outputs a large number of bits and initially store the first $\log(n/\varepsilon)$ bits of $h(x)$ for each $x \in S$, where $n = |S|$. Whenever we discover a false positive, i.e. a query y whose fingerprint matches a fingerprint for some $x \in S$, we modify the filter to store a longer fingerprint for x until the collision disappears. In fact, Kopelowitz et al. [54] show

that this approach is not only natural but necessary: a space-efficient fingerprint filter must have variable-length fingerprints to be adaptive.

We implement a prototype fully dynamic monotonically adaptive filter, the AdaptiveQF, that uses only $(1+o(1))n\log(1/\varepsilon)+O(n)$ space. We demonstrate experimentally that it outperforms existing purpose-built solutions for skew distribution and YES/NO list workloads in terms of space efficiency, insertion speed, and query speed. We also show a space lower bound for the static YES/NO list problem and that the AdaptiveQF meets the space lower bound up to low-order terms.

The challenge is to store and update these variable-length fingerprints efficiently in terms of space and time. Prior theoretical proposals for building adaptive filter have had complex mechanisms for managing variable-sized fingerprints [7]. We propose a simple scheme for implementing variable-sized fingerprints within the AdaptiveQF. Even though adapting requires extending a fingerprint by only two bits in expectation [7], the AdaptiveQF simplifies fingerprint management by *over-adapting*, i.e. fingerprints grow by multiples of $\log(1/\varepsilon)$ bits. Over-adaptation could cause the filter to use too much space and over-minimize the false-positive probability below ε. However, in practice, this is not an issue.

Our results. We evaluated the AdaptiveQF in isolation and as a component of larger systems. The high-level summary of our findings is that the AdaptiveQF can speed up query throughput by delivering far fewer false positives than non-adaptive filters. We compared the performance of the AdaptiveQF to that of two other adaptive filters, the telescoping quotient filter (TQF) [56] and the adaptive cuckoo filter (ACF) [62]. We also compared it to two non-adaptive filters, the quotient filter (QF) [71] and the cuckoo filter (CF) [41].

(1) In a disk-based database, the AdaptiveQF is between $10\times - 30\times$ faster than other adaptive filters (TQF, ACF) for overall insertion performance and is comparable to non-adaptive filters.
(2) In a disk-based database, the AdaptiveQF achieves between $15\% - 6\times$ faster overall query performance than non-adaptive filters (QF, CF) for adversarial queries and has comparable performance for uniform-random query workloads.
(3) The AdaptiveQF is dynamic (i.e. support deletes and resizability) but still achieves similar or better space usage compared to purpose-built solutions for the static YES/NO list problem.
(4) The AdaptiveQF has negligible performance overhead compared to the filter on which it is based.
(5) The AdaptiveQF preserves all the critical features of the quotient filter such as mergeability, resizeability, and bulk insertions.

In summary, the adaptivity overhead is minimal in the AdaptiveQF compared to non-adaptive filters. It is able to substantially improve overall system performance in scenarios where disk accesses incur a large cost. Furthermore, it matches or beats the performance of custom-built solutions for static YES/NO list problems.

2 Filters and Applications

In this section, we give an overview of general-purpose filters and adaptive filters. We then describe applications that can benefit from adaptive filters and review existing purpose-built filters for these applications. We divide these applications in two broad categories:

(1) Applications using traditional filters on skewed workload patterns where adaptive filters can help.
(2) Applications that use purpose-built solutions where the cost of certain false positives is very high.

2.1 General-purpose filters

For decades, the Bloom filter [11] was essentially the only available filter, but Bloom filters are suboptimal in terms of space usage, running time, and data locality, and they support a bare-bones set of operations (insert and lookup). The Bloom filter has inspired numerous variants [1, 12, 18, 33, 43, 61, 76, 77].

The counting Bloom filter (CBF) [43] supports deletes at the cost of space. The blocked Bloom filter [76] provides better cache locality than the standard Bloom filter but it comes at a higher false-positive rate.

The quotient filter (QF) [8, 36, 38, 68, 71] is a fingerprint filter. It stores fingerprints using Robin Hood hashing [20]. It divides the fingerprint into two parts, higher order $\log(n)$-bits as the quotient and lower order $\log(1/\varepsilon)$-bits as the remainder. The quotient bits are used to locate a slot in the table, and the remainder bits are stored in that slot. It supports insertion, deletion, lookups, enumeration, resizing, and merging. The counting quotient filter (CQF) [71], improves upon the performance of the quotient filter and adds variable-sized counters to count items using asymptotically optimal space, even in large and skewed datasets.

The cuckoo filter [41] also stores fingerprints but uses cuckoo hashing instead of Robin Hood hashing. The Morton filter [14] is a variant of the cuckoo filter that is designed to speed up insertion using optimizations designed for hierarchical-memory systems.

2.2 Strongly adaptive filters

A *strongly adaptive filter* modifies its state so that if a false positive is repeated, the probability that it is still a false positive is at most ε. Bender et al. [7] introduce the broom filter and the notion of strong adaptivity used in this paper. The broom filter is based on the quotient filter, but supports variable-length fingerprints to adapt to (and correct) false positives. Lee et al. [56] introduce the telescoping filter, which is also built using the quotient filter [71]. The telescoping filter is strongly adaptive but avoids directly extending fingerprints. They change the remainder (or the tag) stored in the filter by using a different lower-order $\log(1/\varepsilon)$-bits. Additionally, they maintain a table to record which $\log(1/\varepsilon)$-bits they have used as a remainder in the filter for the adapted items. As the filter adapts, the size of this table grows (so in this sense their fingerprints are variable-length and strong adaptivity is possible). The adaptive cuckoo filter of Mitzenmacher et al. [62] is adaptive in the sense that it changes its representation in response to false positives, but is not strongly adaptive. See Section 2.3.

2.3 Filters for skewed query distributions

A common and important application of filters is their use in key-value stores based on Log-Structured Merge Trees (LSMs) and B^ε-trees [15, 67]. In these key-value stores, filters are used to avoid performing multiple expensive disk accesses per query [3, 21, 29, 45, 63, 81, 84]. In some database systems, this type of key-value store is used as the storage engine [3, 29, 45, 81].

In an LSM tree, data is stored in SSTables, which are static, sorted arrays of key-value pairs. The SSTables are organized into levels L0, L1, ..., where L0 is the smallest and holds the most recently written data. Each subsequent level is larger by a factor of g, where g is a configuration parameter. As data is written, SSTables are moved down the structure and are merged into each other according to a chosen compaction policy. In general, at any point in time, a given key can be present in an SSTable on any level (even multiple SSTables per level in some variants), so queries need to check at least one SSTable on each level, which is expensive. As a result, in almost all practical systems, each SSTable has a corresponding filter so that queries only read data from the SSTable when the key is present there or due to rare false positives from the filter. Note that in this application, queries to SSTables on smaller levels are often negative since most of the data is stored in the larger levels. However, the smaller levels may contain recent updates, so they cannot be skipped. Thus, filters in the smaller levels see frequent negative queries [32].

One challenge to applying adaptive filters to LSMs is that the SSTables are typically static and most LSM trees store their filters in their SSTables. However, this is not necessary or universal. For example, SplinterDB stores its filters separately from the data they cover [29]. An LSM could even store adaptive versions of its filters only in memory. The adaptivity information would be lost on a crash, meaning that false positives might increase after a crash, but this should be rare enough to be insignificant.

Query workloads to database systems commonly follow a power-law or otherwise skewed distribution [28, 65]. As a result, state-of-the-art benchmark suites YCSB, TPC-C, and TPC-E incorporate data skew into most of their workloads [23, 30, 47, 58]. In fact, many systems have tried to mitigate or even exploit the effects of data skew [5, 24, 37, 66, 95]. A skewed database query workload also results in a skewed workload for filters. Moreover, as noted above, many of the queries will be negative. Adaptive filters, such as AdaptiveQF, can outperform non-adaptive filters on skewed workloads by eliminating repeated false positives on frequently accessed keys.

2.4 Filters for Yes/No list problems

Detecting Malicious URLs. Malicious websites pose a major threat to internet users. For example, merely visiting a malicious URL may cause a user's web browser to be hijacked [88]. Since URLs are long [49] and abundant [85], an effective way for a router to block malicious URLs is to store them as the yes list of a filter [59].

However, it is important not to block legitimate URLs that are false positives [35], so every positive response of the malicious-URL filter must be verified [57, 60], which is expensive. This additional overhead imposed on false positive (safe) URLs is especially undesirable when the URL is *important*. For instance, a false positive may block access to a voter registration webpage, or emergency weather information, whereas slowing the loading of other false-positive pages is relatively benign.

One way to address this variability in false positive cost is to store important false positives in a no list, so that they are never blocked and so they do not pay the URL-verification penalty. Chazellete et al. [22] introduced the Bloomier filter which solves the yes/no list problem. Li et al., [59] present the Seesaw Counting Filter (SSCF), which implements a yes/no list filter specifically for the malicious URL blocking problem. Reviriego et al. [80] present the Integrated Filter which also implements a no list. Both focus on the case where the no list is static and known ahead of time. The SSCF has an extension for adding no list items dynamically, but it is not guaranteed to prevent false positives by doing so and can also introduce false negatives.

URL requests may also vary in frequency, and these frequencies may even change over time. However, the existing filters literature on this problem does not consider this generalization. For this work we restrict our focus to the standard assumption that a static set of high-priority elements must never be false positives.

Certificate Revocation Lists. In the TLS PKI (Transport Layer Security Public Key Infrastructure [48]), browsers should check whether a certificate has been revoked before trusting connections authenticated by the certificate. Traditionally this was done via a "pull" approach, i.e., browsers would check with a central repository of revoked certificates when they established a connection. More recent work has sought to move to a "push" model, where browsers receive frequent updates to the list of revoked certificates, so the browser can perform a purely local check when it establishes a new connection.

Larisch, et al., [55] proposed CRLite, which uses cascading Bloom filters to store the set of revoked certificates at the client. They observed that, in the case of TLS certificates, the universe is a small finite set and known at construction time. They can build a cascade of Bloom filters to exactly represent the set of revoked certificates. In the cascading Bloom filter, each subsequent Bloom filter contains false positive set from the earlier Bloom filter until the false positive set is small enough to be stored exactly in a hash table. A central system would periodically push updates to this list to browsers. The updates are encoded as bitwise deltas on the original filters. When the space of certificates grows too large, so that they need to resize the filters, then they have to transmit new filters from scratch.

De Bruijn graph traversal. In computational biology, de Bruijn graphs (DBGs) are at the heart of numerous genomic sequence analysis pipelines [70, 72]. In a de Bruijn graph, each node is a k-length

subsequence (of the DNA bases, "A", "C", "G", and "T") from the underlying biological samples, and two nodes are connected via an edge if they share a $(k-1)$-length subsequence. Analyses traverse DBGs during assembly, error correction, "contig" detection, and numerous other applications.

De Bruijn graphs are often large enough that they do not fit in the memory. Numerous methods have been proposed to exploit their special structure for compression. One of the main tricks is to take advantage of the fact that each node has at most 4 incoming edges and 4 outgoing edges (one for each base that can be prepended or appended to the node). Thus a traversal can query for the existence of each edge, so we can represent the DBG using an (approximate) set data structure that supports only membership queries, i.e. a filter. In this application, false positives in the filter result in extra edges in the graph. To avoid the false edges, Chikhi and Rizk [26] proposed to store the de Bruijn graph in a cascading Bloom filter as the set of queries is known in advance. Each Bloom filter stores the false positives from querying the earlier Bloom filter using all possible queries during the dBg traversal.

2.5 Attacking adaptive filters

An adversary who is able to issue queries to a Bloom filter, and detect when the filter returns a false positive, can eventually force the filter to give false positives on nearly every query. It simply issues queries until it observes a false positive and then repeats the query that induced the false positive. Such an attack works on any filter that does not change its representation in response to false positives. Since adaptive filters do change their representations in response to false positives, some of them are more robust to attacks by adversaries.

Reviriego et al. [80] demonstrate that an adversary who is able to issue queries to an adaptive cuckoo filter, and detect when the filter returns a false positive, can eventually force the filter to give false positives on nearly every query. The attack exploits the property that adaptive cuckoo filters will revert to the initial fingerprint for some element in the represented set after a certain number of adaptations. After finding an ***adaptation loop***, a sequence of queries that when queried in order will force the filter to 1) yield multiple false positives and 2) revert to its initial state at the end of the loop, the adversary is able to replay this adaptation loop indefinitely, forcing the filter to return many false positives. The attack succeeds even when the adversary does not have exclusive query access to the filter (that is, other processes/users may query the filter at any time), though the time to complete an attack increases in this case.

Further, Kopelowitz et al. [54] show that any variant of the adaptive cuckoo filter, or indeed any space-efficient filter with fixed-length fingerprints, can be forced by such an adversary to suffer a high proportion of false positives. Filters that are adaptive according to Bender et al.'s definition [7], such as the broom filter and the telescoping adaptive filter [56], are provably robust against this type of adversary because they guarantee a false-positive rate of ε for any sequence of queries, even those generated by an adversary.

3 AdaptiveQF design

In this section, we describe the high-level schema of our solution, without worrying about how to encode this design in a small number of bits. The encoding is described in subsequent sections.

3.1 High-level design

The AdaptiveQF builds on the idea of fingerprint filters, which store a set S by storing the set of ***fingerprints*** $h(S) = \{h(x) \mid x \in S\}$. The basic idea behind the AdaptiveQF is that, initially, we store only enough bits of each fingerprint to ensure a false positive rate of ε, i.e. we store a set F of fingerprints, where each fingerprint is actually a *prefix* of $h(x)$, for some $x \in S$. A query for y returns yes if some fingerprint in F is a prefix of $h(y)$. When we discover a false positive, i.e., an item $y \notin S$ such that some fingerprint $f \in F$ is a prefix of $h(y)$, we increase the length of f until it is no longer a prefix of $h(y)$.

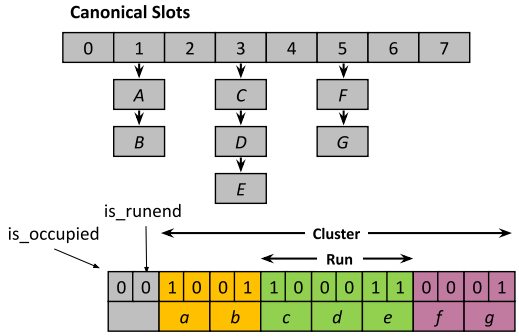


Fig. 1. The quotient filter [71] structure. The upper part shows the logical structure. The lower part shows the encoding of the logical structure in the quotient filter. It uses two metadata bits per slot. All items that share the same canonical location are stored together in a ***run***. A sequence of items without any empty slot is called a ***cluster***. Note: the items are showed in upper case in the canonical representation and the remainders corresponding to the items in the slots are showed in lower case.

The main issue that arises is: how can we extend a fingerprint f in our filter without knowing the full hash $h(x)$ of which it is a prefix? To solve this problem, all adaptive filters maintain a reverse map that maps fingerprints in F back to their full hashes (or even the original keys). Notably, the need to store full hashes means this map will be much larger than the filter, possibly too large to fit in fast storage. Thus, we would like to minimize how often this reverse map needs to be updated and/or queried.

At the very minimum, one insert needs to be done to the reverse map for each insert to the filter. In addition, one reverse map query needs to be made for each adaptation. However, since adaptations are responses to false positives, a disk access is done at this point anyway, and adapting ensures that the offending query will not cause another disk access in the future. Thus, the AdaptiveQF maintains its fast performance on general queries. Ideally, we would access the reverse map at no other time.

In the following sections, we describe how we store and update variable-length fingerprints efficiently and how we maintain the reverse map with little overhead.

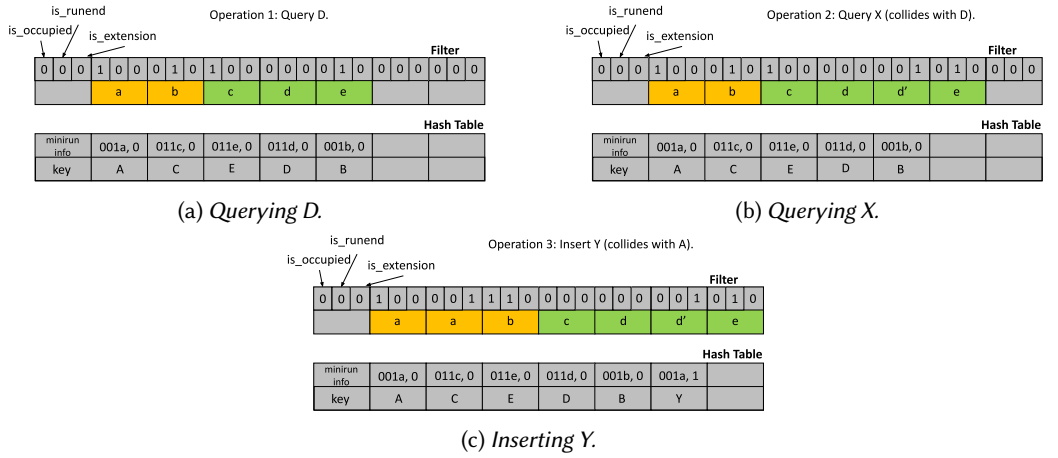
3.2 Quotient filter

We build the AdaptiveQF using the quotient filter [71]. The quotient filter has the ability to associate small variable-length values with fingerprints. We exploit this feature to extend the fingerprint size to adapt. Using the quotient filter as the underlying filter helps retain advantages, such as good cache-locality, deletion, resizability, enumerability, mergeability, etc., that the quotient filter has over other filters. In this section, we give an overview of Pandey et al.'s quotient filter [71]. Later in Section 4, we explain how we modify the quotient filter schema to build the AdaptiveQF.

The quotient filter (QF) stores an approximation of a multiset $S \subseteq \mathcal{U}$ by maintaining a compact, lossless representation of the multiset $h(S)$, where $h : \mathcal{U} \to \{0,...,2^p - 1\}$ is a hash function that maps items from the universe $\mathcal{U}$ to a p-bit fingerprint. To handle a multiset of up to n distinct items while maintaining a false-positive rate of at most ε, the QF sets $p = \log_2 \frac{n}{\varepsilon}$ (see [8] for the analysis).

The quotient filter uses Robin-Hood hashing [20] to store the fingerprints compactly in a table. It consists of an array Q of 2^q ***slots*** and a hash function h mapping items from a multiset to p-bit integers, where $p \geq q$. Robin-Hood hashing is a variant of linear probing in which we try to place an item a in slot $h(a)/2^{p-q}$, but shift items down when there are collisions to create empty space. Robin-Hood hashing maintains the invariant that, if $h(a) < h(a')$, then a will be stored in an earlier slot than a'.

The quotient filter divides $h(x)$ into its first q bits, ***quotient*** $h_0(x)$, and its remaining r bits, ***remainder*** $h_1(x)$. Together, the quotient and remainder form the ***fingerprint*** of x. The quotient filter



(a) *Querying D.* (b) *Querying X.*

(c) *Inserting Y.*

Fig. 2. AdaptiveQF block diagram and the reverse map. It shows the changes to the filter and reverse map during queries, insertions, and adaptations. **(a)** Hashing D gives 011dd'd", and 011 is the quotient of the 4th bucket. The run begins at the 4th slot, and matching remainder d is found in this run in slot 5. Filter returns YES. **(b)** X hashes to 011dxx'. Filter finds 011d and returns a false positive YES. Thus, look up (011d, 0) in hash table to obtain D. Hash D to get d'. Insert d' in the next slot and mark extension bit. **(c)** Y hashes to 001a. Find the run for bucket 001. Look for remainder *a* in the run. Having found it, add y to the end of the minirun by shifting everything to the right. Add Y to reverse map as the second fingerprint of the minirun.

maintains an array Q of 2^q r-bit slots, each of which can hold a single remainder. When an element x is inserted, the quotient filter attempts to store the remainder $h_1(x)$ at index $h_0(x)$ in Q (which we call x's ***canonical slot***). If that slot is already in use, then the quotient filter uses Robin hood hashing to find the next available empty slot to store $h_1(x)$. All the items that share the same canonical slot are stored together in a ***run***, and a sequence of runs stored contiguously with no empty space is called a ***cluster***. During an insert operation, the next available empty slot is found at the end of the cluster. If an item lands at the start of the cluster then all the items in that cluster must be shifted to create space (see Figure 1).

The quotient filter also maintains 2 bits of additional metadata (*is_occupied* and *is_runend*) per slot in order to determine which slots are in use and the canonical slot of each remainder stored in Q. When an item is inserted into the canonical slot, the *is_occupied* bit for that slot is set to 1. The *is_runend* bit is set to one for every slot that contains the last remainder in a run. Please refer to Pandey et al. [71] for further details.

4 AdaptiveQF Implementation

The AQF uses the same fundamental structure as the QF. Here we describe how we modify the QF schema to support adaptivity.

4.1 False positives

First, let us understand how false positive queries occur in the QF. The QF stores the fingerprints compactly and exactly in the table. Therefore, a false positive occurs due to a hash collision while computing the fingerprint. That is, suppose there exist two distinct items x and y such that $x \in S$ but $y \notin S$. If x and y share the same fingerprint, then a query for y will result in a false positive.

A QF guarantees a false-positive rate [1] of $\varepsilon = 2^{-r}$, where r is the number of remainder bits, for a set of items drawn uniformly at random from the universe $\mathcal{U}$. However, if the items are drawn from some arbitrary distribution then there is no guarantee on the false positive rate. For example, if a query set consists of a single item that happens to be a false positive, then the false positive rate of the filter for that set will be 1.

4.2 Adapting to false positives

The AdaptiveQF can update its representation on every false positive query, ensuring that the false-positive probability remains $\leq \varepsilon$, even on repeated queries that were previously false positives.

The AdaptiveQF adapts by giving an additional slot to the fingerprint responsible for the false positive, extending it by the next r bits in its full hash, known as an ***extension***. Multiple extensions can be added if the resulting fingerprint still produces a false positive on the given query, with the probability that the fingerprint incurs a false positive decreasing by a factor of 2^{-r} with each extension. The ***fingerprint*** of an item in the AdaptiveQF now refers to its quotient, remainder, and any extensions that have been added to it in the filter.

Recall that the QF has an overhead of 2.125 bits per item. We introduce an additional overhead bit, the *is_extension* bit, to differentiate between slots storing remainders and extensions, bringing the total overhead to 3.125 bits per slot. Slots with an unmarked *is_extension* bit are treated as usual. A marked *is_extension* bit indicates that the slot contains an extension of the previous remainder.

Miniruns and the reverse map. To extend a fingerprint in the filter, we need the original key that fingerprint represents. We extend the fingerprint for a given key by adding additional bits derived from the full hash of that key. Thus, we maintain an on-disk ***reverse map*** from fingerprints to keys.

Recall that the reverse map is not specific to the AdaptiveQF and is needed by any adaptive filter to retrieve the additional information necessary for adaptation. However, because the AdaptiveQF adapts by appending to fingerprints rather than reshuffling them as done in the telescoping filter [56] and the adaptive cuckoo filter [62], the bits of the fingerprint that existed prior to adaptation stay the same. Most notably, the quotient and remainder, which every fingerprint starts with, are fixed from the moment of insertion. We use this fact to construct a reverse map that does not need to be accessed in response to natural shifting in the filter during insertions.

Suppose two items x and y share a quotient and remainder. Because runs are sorted by quotient, and fingerprints within a run are sorted by remainder, it follows that the fingerprints of x, y, and any other items with the same quotient-remainder pair are stored contiguously in the AdaptiveQF. Let us call this group of fingerprints a ***minirun*** and their shared quotient-remainder pair their ***minirun ID***. Even when an item does not share a quotient-remainder pair with any other item, we still consider its fingerprint to be contained in a minirun of length 1. The ***minirun rank*** of an item is the rank of its fingerprint within its minirun. Our reverse map will map a given minirun ID to a list of all inserted keys with that minirun ID. We order the keys in this list according to the order in which their fingerprints appear in the minirun.

To insert a key-value pair into the database, we start by inserting its fingerprint into the filter. We locate the slot in which the fingerprint belongs and insert it into the back of its associated minirun, if one already exists. Thus, its minirun rank k is the length of that minirun prior to the insertion, or zero if there was no such minirun. In the reverse map, we map the kth element under this minirun ID to the key of the key-value pair we are inserting. Finally, we insert the key-value pair into the database.

When querying for a key in the database, we always start by querying the filter for its fingerprint, obtaining its minirun rank k if the fingerprint is present. If the filter returns a positive, we then query the reverse map for the kth key under this minirun ID. The query is detected to be a false positive

[1] The false-positive rate is defined as the ratio of the number of false positives reported over the total number of queries in the set.

if that key differs from the queried key. If so, we use the key provided by the reverse map to adapt the offending fingerprint in the filter, which we can do without modifying the reverse map. Otherwise, we query the database as usual to obtain the value for the queried key.

Figure 2 shows the state of the AdaptiveQF and reverse map during insertions and adaptations.

The reverse map as a database. So far, we have acted as though the reverse map and the database are two separate data structures, both on disk. This would imply that the reverse map incurs additional disk accesses during every insertion and positive query. We call this the ***split*** reverse map setup.

However, we can merge the database and reverse map into a single key-value store, eliminating this cost. Because the reverse map allows us to uniquely identify the original key of any fingerprint in the filter, we can simply store any relevant values next to these keys in the reverse map. That is, instead of the reverse map mapping from fingerprints to keys and the database mapping from keys to values, we use the reverse map as a single mapping from fingerprints to key-value pairs. In effect, the reverse map replaces the database. This does not require any additional queries over the conventional key to value mapping – after all, the filter is always queried before the database, so we already have the fingerprint for any queried item by the time we turn to the database. We call this the ***merged*** reverse map setup.

For our on-disk experiments, we use the merged setup with all adaptive filters. However, note that this optimization leads to the items in the database being stored in hash order. Thus, the merged setup no longer supports range queries. For applications that wish to use range queries, the split setup would suffice. We evaluate the overhead of using the split setup later.

Counters. Like the counting quotient filter upon which it is based, the AdaptiveQF supports efficiently storing duplicate items by maintaining an optional variable-length counter with each fingerprint. In the CQF, each item uses one slot for its remainder, followed by 0 or more slots encoding the number of times that fingerprint is present in the filter. The CQF encodes the counter such that singleton fingerprints use zero additional slots for their count, so the CQF is no less space efficient than a non-counting quotient filter even when the set contains no duplicates. However, because the CQF has only two metadata bits, encoding the counters is fairly complex.

Since the AQF has three metadata bits, we can use a much simpler encoding. The AQF has three types of slots—remainders, extensions, and counters—and we only need to distinguish between the two types of "extra" slots that can follow a remainder: extension slots and counter slots. In our encoding, both extension and counter slots have the *is_extension* bit set, and we use the *is_runend* bit to indicate whether the slot holds an extension or a counter. This is safe because we indicate the end of a run by setting the *is_runend* bit on only the first slot of the last fingerprint in the run, which always stores the remainder – any other slots of that fingerprint are free to use their own *is_runend* bit to indicate whether they are extension or counter slots.

It's worth noting that in both the CQF and the AdaptiveQF, use of this counter is not restricted to storing counts specifically. It can be used to optionally attach any kind or number of associated values next to any fingerprint in the filter.

4.3 Dynamic Yes/No List Problem

We can extend AdaptiveQF to the dynamic yes/no list problems as follows. First, we extend the filter to store an extra bit with each fingerprint, i.e. we extend each slot with one extra bit. We then store all elements of both Y and N in the filter, performing adaptations to eliminate any fingerprint collisions that occur during insertion and using the extra bit to record which set each fingerprint belongs to. We can now add new items to Y and N in the same way: add the item to the filter, performing any adaptation necessary to eliminate fingerprint collisions and tagging the items with their origin. We can delete items by simply deleting them from the filter. Deleting a fingerprint f may mean that we can shorten other fingerprints in the filter that we extended because they collided with f. Finding any such eligible

fingerprints is easy and efficient because the AdaptiveQF stores fingerprints sorted lexicographically, so all the fingerprints that can be shortened will be stored in a contiguous run of slots containing f.

Like the quotient filter, the AdaptiveQF also supports growing and shrinking. The data structure described in this section operates by initially provisioning the table with enough slots to hold a certain number of elements and adaptivity bits. In the case of the static AdaptiveQF filter described in Section 5.1, when sizes n and m of the YES list and NO list, respectively, are known ahead of time, we can predict the total space cost with high confidence, as per Theorem 2. This tight allocation of the table is what allows us to match the lower bound from Theorem 7.

Alternatively, if the YES and NO list sizes are not known in advance, we can instead fix two upper bounds $\hat{n}$ and $\hat{m}$ on their maximum allowed size, and construct the following *dynamic* YES/NO filter: we allocate our table as a function of $\hat{n}$ and $\hat{m}$ (instead of n and m), and perform all the insertions into the NO list and queries of YES list elements dynamically, as they are needed. The closer n and m get to $\hat{n}$ and $\hat{m}$, respectively, the closer the space cost will be to the space lower bound for a static filter on these NO list and YES list sets.

4.4 Skewed and Adversarial Workloads

The basic AdaptiveQF structure is monotonically adaptive, i.e. it never repeats a false positive. The cost of never forgetting false positives is that, over time, the AdaptiveQF needs more and more slots to hold adaptivity information. Like the regular quotient filter, the cost of an insert into the AdaptiveQF is $\Theta(\log n/(1-\alpha)^2)$ w.h.p., where n is the number of slots and α is the fraction of slots that are currently in use [10]. Thus, as the AdaptiveQF adapts, α approaches 1 and insertions performance can fall off a cliff. In static and dynamic YES/NO-list problems, this can be mitigated by making the filter large enough to accommodate the anticipated number of items.

However, for skewed and adversarial workloads, we can recover space used by adaptation, ensuring that the total space used by the filter remains constant over time. This compromises monotonicity but still ensures that the number of false positives from any sequence of k queries is very close to εk w.h.p.

The basic idea is to periodically rebuild the filter with a new hash function. Rebuilding the filter puts the attacker back into the position of attacking a filter about which he has no information. Thus we can drop any adaptivity information after the rebuild. In other words, when we do the rebuild, each item will consume a single slot.

So, for example, we can build the filter, say, 10% larger than necessary, run the filter until the extra space is consumed by adaptations, and then rebuild. Furthermore, we can de-amortize the rebuild process. See Bender et al. for details [7].

5 Static YES/NO List Bounds

In this section, we prove a space lower bound for solving the *static* YES/NO list problem and we show that we can use AdaptiveQF to build an optimal solution to the static YES/NO list problem, up to low order terms. Importantly, existing practical solutions to this problem (namely the Seesaw Counting Filter [59], and the Bloomier Filter [22]) are either not always correct solutions to the problem, or their space cost is at least constant factor away from the lower bound.

Let U be a finite universe of elements, and let Y and N be subsets of U, with $Y \cap N = \emptyset$. Let $\varepsilon \in (0,1)$. A ***YES/NO filter*** supports queries with the following guarantees: (i) every query for $y \in Y$ must answer YES, (ii) every query for $z \in N$ must answer NO, and (iii) every query for $x \notin Y \cup N$ answers YES with probability at most ε. Notice that YES/NO filters are static data structures. Although it's possible to consider a dynamic version where the elements of either the YES or NO set (or both) are inserted and deleted dynamically, in this section we do not study this scenario.

Throughout this section we will let $n = |Y|$ be the size of the YES list, $m = |N|$ be the size of the NO list set, and $u = |U|$ be the size of the universe.

5.1 Upper Bound for Yes/No Filters

Next we give an upper bound for the static yes/no list problem, based on the AdaptiveQF. We maintain the notation of the previous section, and let $\mu := \varepsilon m/n$ be the number of no false positives per yes element. Consider the following implementation, which we refer to as the yes/no AdaptiveQF.

(1) Create an AdaptiveQF F with capacity for n elements, and $\mathcal{A}(n,m,\varepsilon)$ bits reserved for adaptivity, where
$$\mathcal{A}(n,m,\varepsilon) := (1+o(1))n\log(1+\mu)+O(n).$$
(2) Insert into F every element from Y.
(3) Query F on each element from N.
(4) If F becomes full at any point before all queries are done, fail.
(5) Return F.

Importantly, the queries in the final step fix all false positives from N. The resulting filter satisfies the requirements of a yes/no filter.

Proposition 1. *The yes/no AdaptiveQF uses* $(1+o(1))n\log(\max\{1/\varepsilon, m/n\})+O(n)$ *bits of space.*

Proof. The space cost is the sum of the space reserved for the remainders, plus the per-slot metadata bits, plus the space reserved for adaptivity bits. This is a total of $n\log(1/\varepsilon)+O(n)+\mathcal{A}(n,m,\varepsilon)$ bits. Observe that

$$\mathcal{A}(n,m,\varepsilon) = \begin{cases} O(n) & \text{if } \mu \le 1 \\ (1+o(1))n(-\log(1/\varepsilon)+\log(m/n))+O(n) & \text{otherwise} \end{cases}$$

Notice that $\mu \le 1$ if and only if $1/\varepsilon \ge m/n$. Hence,

$$\begin{aligned} &n\log(1/\varepsilon)+O(n)+\mathcal{A}(n,m,\varepsilon) \\ &\qquad = \begin{cases} n\log(1/\varepsilon)+O(n) & \text{if } 1/\varepsilon \ge m/n \\ (1+o(1))n\log(m/n)+O(n) & \text{otherwise} \end{cases} \\ &\qquad = (1+o(1))n\log(\max\{1/\varepsilon, m/n\})+O(n). \end{aligned}$$

□

Notice that the construction of the yes/no AdaptiveQF may fail if the space initially reserved is insufficient. The following theorem, the central result of this section, establishes that failure is unlikely.

Theorem 2. *Suppose* $\omega((\log^{3/2} n)/\sqrt{n}) \le \mu \le 2^{o(n)}$. *Then, the number of adaptivity bits added to the yes/no AdaptiveQF is at most* $\mathcal{A}(n,m,\varepsilon)$ *with probability* $1-1/\mathrm{poly}(n)$. *In particular, the probability that the construction succeeds is* $1-1/\mathrm{poly}(n)$.

In the rest of this section, we prove this theorem. Let A be the number of adaptivity bits needed after all the elements of N are queried in step (5). We want to show that $A \le \mathcal{A}(n,m,\varepsilon)$ with probability $1-1/\mathrm{poly}(n)$.

Let h be fingerprint hash function. For any $x,y \in U$, let $\mathrm{lcp}(x,y)$ be the longest common prefix of $h(x)$ and $h(y)$. We decompose A into two parts: Let A_1 and A_2 be the number of adaptivity bits added in steps 3 and 4, respectively. For any $y \in Y$, let $A_1(y)$ and $A_2(y)$ be the number of adaptivity bits added in step 3 and 4, respectively, to the fingerprint of y. Then, $A = A_1 + A_2$, and $A_i = \sum_{y \in Y} A_i(y)$.

Lemma 3. *We have* $A_1 = O(n)$ *with probability* $1-1/\mathrm{poly}(n)$.

Proof. This follows directly from Lemma 9 in Bender et. al. [7]. □

We will use the following basic result about the distribution of the maximum of a collection of independent geometric random variables.

Lemma 4 ([39]). *Let $X_1,\ldots,X_k$ be independent geometric random variables with parameter $1/2$. Let $M_k := \max_i X_i$. Then,*

$$0 \le \mathbb{E}[M_k] - \log(e) H_k \le 1,$$

where H_k is the k-th harmonic number.

Lemma 5. $\mathbb{E}[A_2] \le n(1+\log(e)+\log(1+\mu))$

Proof. We say that $z \in N$ has a ***soft collision*** with $y \in Y$ if the baseline fingerprints of y and z match, that is, $\mathrm{lcp}(y,z) \ge \log(n/\varepsilon)$.

Fix an arbitrary $y \in Y$. We claim that $\mathbb{E}[A_2(y)] \le 1+\log(e)+\log(1+\mu)$. Let C be the (random) numbers of $z \in N$ that have a soft collision with y. For each z that has a soft collision with y, let $F(z) := \mathrm{lcp}(y,z) - \log(n/\varepsilon)$; this is the smallest number of adaptivity bits that y must have to fix a false positive z. Then,

$$A_2(y) = \max_{z \text{ has a soft collision with } y} F(z).$$

Observe that $F(z)$ is a geometric random variable with parameter $1/2$. Because all fingerprints are independent, the $F(z)$'s are independent. This is true even if C is known. Thus, $A_2(y)$ conditioned on $C=k$ is identically distributed as the maximum of k independent geometric random variables with parameter $1/2$. Call this maximum M_k. Then,

$$\mathbb{E}[A_2(y) \mid C=k] = \mathbb{E}[M_k].$$

For $k=0$, we have $\mathbb{E}[A_2(y) \mid C=k] = 0$. For $k \ge 1$,

$$\begin{aligned}
\mathbb{E}[A_2(y) \mid C=k] &\le 1+\log(e)H_k && \text{(by Lemma 4)} \\
&\le 1+\log(e)(1+\ln k) && \text{(as } H_k \le 1+\ln k \text{ for } k \ge 1) \\
&= 1+\log(e)+\log k.
\end{aligned}$$

Hence, $\mathbb{E}[A_2(y) \mid C] \le 1+\log(e)+\log(1+C)$. Then,

$$\begin{aligned}
\mathbb{E}[A_2(y)] &= \mathbb{E}[\mathbb{E}[A_2(y) \mid C]] && \text{(by the tower rule)} \\
&\le \mathbb{E}[1+\log(e)+\log(1+C)] \\
&\le 1+\log(e)+\log(1+\mathbb{E}[C]). && \text{(by linearity of expectation and Jensen's inequality)}
\end{aligned}$$

Notice that $C = \sum_{z \in N} C(z)$, where $C(z)$ is an indicator random variable that is 1 exactly when z has a soft collision with y. Recall that $\mathbb{E}[C(z)] = \varepsilon/n$. By linearity of expectation, $\mathbb{E}[C] = \varepsilon m/n = \mu$, and, finally,

$$\mathbb{E}[A_2(y)] \le 1+\log(e)+\log(1+\mu).$$

This concludes the proof of the claim. The lemma follows by summing over all $y \in Y$, and using linearity of expectation. □

To simplify notation, let $\mu = \varepsilon m/n$ be the mean number of queries that have a soft collision with any fixed $x \in S$.

Lemma 6. *Suppose $\omega((\log^{3/2} n)/\sqrt{n}) \le \mu \le 2^{o(n)}$. Then, $A_2 \le (1+o(1))\,\mathbb{E}[A_2]$ with probability $1-1/\mathrm{poly}(n)$.*

Proof sketch. Recall that $A_2 = \sum_{y \in Y} A_2(y)$. The proof is divided into two parts. First, we show that the random variables $A_2(y)$ are negatively associated (NA). Roughly speaking, this is because when some query from N is a false positive due to fingerprint match with $y \in Y$, then the new adaptivity bits make following queries less likely to cause a false positive on y. Though intuitive,

proving that the $A_2(y)$'s are NA is challenging because we can't directly apply standard theorems on negative association [90].

For each $y \in Y$, let $G(y)$ be the number of $z \in N$ such that $\mathrm{lcp}(y,z) \geq \log(n/\varepsilon) + A_1(y)$, that is, elements that can cause a false positive due to a fingerprint match with y (and only with y). The argument rests on the following four properties of the random variables involved:

(1) The $G(y)$'s are NA random variables.
(2) Conditioned on the $G(y)$'s, the $A_2(y)$'s are independent.
(3) The probability that $A_2(y)$ is large is a non-decreasing function of the $G(y)$. Formally, for every ℓ, $\Pr[A_2(y) \geq \ell \mid G(y) = k]$ is a non-decreasing function of k.
(4) Conditioned on $G(y)$, the random variable $A_2(y)$ is independent of the $G(y')$'s with $y' \neq y$.

Once negative association of the $A_2(y)$'s is established, we are almost in the conditions of the Chernoff-Hoeffding inequality for the sum $\sum_{y \in Y} A_2(y)$. Unfortunately, there is one hypothesis that is not met, namely they $A_2(y)$'s are not deterministically bounded—in the worst case, an unbounded number of adaptivity bits may need to be added to some element y. This, however, is unlikely, as $A_2(y) = O(\log(\mu + n))$ with probability $1 - 1/\mathrm{poly}(n)$; this is by a Chernoff bound, and a tail bound on the geometric distribution. We can put this observation to work and circumvent the boundedness requirement of Chernoff-Hoeffding, using the following truncation trick: We define $A_2'(y) := \min\{A_2(y), O(\log(\mu+n))\}$, and apply the Chernoff-Hoeffding bound on the truncated sum $A_2' := \sum_{y \in Y} A_2'(y)$. Once concentration around the mean is established on A_2', we conclude the proof by showing that, with high probability, no truncation is actually done, so the analysis on A_2' applies to A_2 most of the time. Specifically:

(1) $A_2 = A_2'$ with probability $1 - 1/\mathrm{poly}(n)$;
(2) $\mathbb{E}[A_2'] \leq \mathbb{E}[A_2]$.

□

Proof of Theorem 2. The construction of the filter succeeds if and only if $A \leq \mathcal{A}(n,m,\varepsilon)$. Since $A = A_1 + A_2$, and by Lemma 3, Lemma 5 and Lemma 6 we have $A \leq O(n) + (1+o(1))(n(1+\log(e) + \log(1+\mu))) = O(n) + (1+o(1))n\log(1+\mu) = \mathcal{A}(n,m,\varepsilon)$, with probability $1 - 1/\mathrm{poly}(n)$. □

5.2 Lower Bound for Yes/No Filters

A lower bound for this problem was sketched out in Reviriego et al. [79], but without a rigorous proof. Moreover, the lower bound as stated in that work is hard to compare against our upper bound using AdaptiveQF; here, we give an equivalent but more condensed lower bound.

Theorem 7. *Suppose $u \geq c(n^2/\varepsilon + m^2)$, for some large enough constant $c > 0$, and $\varepsilon \leq 1/2$. Then, the number of bits used by a static yes/no filter is at least*

$$n\log\left(\max\left\{\frac{1}{\varepsilon}, \frac{m}{n}\right\}\right) + \log(e)\min\{\varepsilon m, n\} + O(1).$$

Before diving into the proof let us briefly discuss the lower bound. Dividing by n, we have that a yes/no filter uses at least

$$\log(\max\{1/\varepsilon, m/n\}) + O(1) = \log(1/\varepsilon) + \log(\max\{\varepsilon m/n, 1\}) + O(1)$$

bits per yes element. For comparison, traditional filters have an information-theoretical lower bound of $\log(1/\varepsilon) + O(1)$ bits per element. This can be interpreted as follows: When building a yes/no filter, we need to (i) record the n yes list elements while ensuring that at most an ε fraction of all other elements are incorrectly reported as present, and to (ii) record the m no elements. To accomplish (i) we need at least $\log(1/\varepsilon)$ bits per element, just like a regular filter. The number of additional bits

needed to accomplish (ii) depends on $\mu := \epsilon m/n$, which is the number of NO false positives per YES element. When $\mu \leq 1$, only a small constant number of extra bits per NO element are needed; when $\mu > 1$, $\log(\mu)$ extra bits per NO element are needed. We have omitted the proof for space but the complete proof can be found in the full version of the paper [92].

6 Evaluation

The goal of the evaluation is to answer the following questions regarding the performance of the AdaptiveQF:

(1) How does the insertion and query performance of the AdaptiveQF compare to other adaptive and non-adaptive filters?
(2) How well does the AdaptiveQF improve overall database performance, compared to other adaptive and non-adaptive filters?
(3) How much space does the AdaptiveQF use to adapt?
(4) How does the AdaptiveQF compare to prior solutions to the YES/NO list problem?
(5) How does the false positive rate in the AdaptiveQF change during a dynamic workload?
(6) How fast can two AdaptiveQF instances be merged?

6.1 Results summary

We compare the AdaptiveQF [2] against two state-of-the-art adaptive filters, the telescoping adaptive filter (TQF) [56] and the adaptive cuckoo filter (ACF) [62]. We also include two non-adaptive filters, the quotient filter (QF) [71] and the cuckoo filter (CF) [42], as baselines to understand the overheads and benefits of adaptivity. The quotient and cuckoo filter are chosen as baselines as these are the filters upon which the adaptive filters used in our evaluation are developed.

We found that adaptivity is an extremely efficient way to reduce the false-positive rate of a filter. For example, on a Zipfian query workload, the AdaptiveQF is able to reduce the false-positive rate by about 100× for an additional cost of less than 1/1000th of a bit per item. In contrast, a non-adaptive filter would need 7 bits per item to achieve the same false-positive rate reduction.

Absent any system, the AdaptiveQF has comparable space usage to the other filters. For example, the AdaptiveQF uses more space than the cuckoo filter, but only by 1%. Most notably, the AdaptiveQF performs at par with the quotient filter on which it is based, indicating little to no overhead for its adaptivity. On the other hand, the adaptive cuckoo filter and telescoping adaptive filter are significantly slower than their respective non-adaptive counterparts.

However, when the cost of a false positive is increased by including an on-disk database, the benefits of adaptivity become apparent. For example, when used to filter queries from a fixed dataset, the AdaptiveQF is able to learn the query set, seeing 10× fewer false positives over 200 million queries than the quotient filter and the cuckoo filter. This resulted in 4-7× faster queries. Furthermore, using the AdaptiveQF to filter queries in a B-tree databases had query throughput that was impervious to an adversarial query workload, whereas the throughput of non-adaptive filters dropped about 2× with the inclusion of an adversary representing a mere 1% of queries.

Although the benefits of maintaining a low false positive rate during queries are shared by all three adaptive filters, the inclusion of a database reveals that the AdaptiveQF is much faster than other adaptive filters during insertions. The systems using the TQF and the ACF slow down significantly as the filters fill up due to frequent modifications of their on-disk backing stores. Between 85-90% fullness, the AdaptiveQF averages 5× the insertion throughput of the ACF and 30× that of the TQF.

Other results. The AdaptiveQF matches the rate of change of false-positive rate during queries from a real-world dataset compared to other adaptive filters. The AdaptiveQF solves dynamic

[2] https://github.com/splatlab/adaptiveqf

YES/NO problems with changing sets. The AdaptiveQF supports fast merges and bulk insertions. Finally, despite being dynamic, the AdaptiveQF achieves similar or better space usage compared to purpose-build solutions for the static YES/NO problem.

6.2 Experimental setup

One challenge we face is that the filters do not all support the same false-positive rates. Thus, we pick a target false-positive rate and configure each filter to get as close as possible to the target false-positive rate without sacrificing performance. This is in accordance with the prior research on evaluating filters [41, 70, 73]. Our target false-positive rate is 2^{-9} ($\approx 0.2\%$) which is the commonly used in most practical system configurations [42, 71].

We configure the quotient filter-based filters (AQF, TQF, and QF) with 9-bit remainders. We use 12-bit fingerprints and blocks of size 4 in the cuckoo filter-based filters (ACF, CF). This results in all filters having a false-positive rate of 2^{-9}. We use MurmurHash2 [4] as the hash function for all filters. The AdaptiveQF does not require any special properties in the hash function compared to other filters.

Machine specification. All experiments were run on an Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz with 96 MiB L3 cache. The machine has 1 TB of memory, 64 CPUs, and 4 TB of SSD-based local instance storage, 64-bit platform. We restrict our runs to a single core.

6.3 Microbenchmarks

We evaluate the performance of the filters in RAM. We create the filters with 2^{27} (134M) slots, which makes them substantially larger than the L3 cache on the machine where experiments are performed. We fill each filter to 90% load factor[3] and report the performance of the filter as a function of load factor. Although all of the filters evaluated in our benchmarks support up to 95% load factor, we restrict them to 90% in order to give them room to store any additional data needed to adapt.

Speed. We evaluate adaptive filter performance on two fundamental operations: insertions and lookups. We evaluate insertions on uniformly random 64-bit keys, and lookups on both uniform-random and Zipfian distributions [31]. In the Zipfian distribution, we use a Zipfian coefficient of 1.5 and a universe size of 10 million items. We do not count the time required to generate the input to the filters, only the time to insert and query items in the filters. This way we only measure the differences in filter performance. We perform 200 million queries for both query distributions. The numbers reported for both insertions and queries are the average of 5 trials.

We perform the benchmarks in isolation of any overheads from the reverse maps. For the adaptive filters, which use reverse maps to obtain keys for adaptation, we pick valid arbitrary keys that will suffice in order to simulate having the reverse map present. We still measure hashing as part of the filters' performance because it is done independently of the reverse map and database.

Figure 3 shows the throughput of adaptive and non-adaptive filters for insertions and queries. The AdaptiveQF is based on the counting quotient filter. AdaptiveQF is not slower than the quotient filter, but it is slightly faster during both insertions and queries, indicating that the overhead of adaptivity in the AdaptiveQF is minimal. Increased query speed may also be attributed in part to the slightly lower false positive rate resulting from adaptation. The CF has the highest insertion throughput among all the filters, which is consistent with previous research [41, 70, 73]. In exchange, the quotient filters offer fast resizing, mergeability, and efficient variable-length counters/values.

In contrast, there is a noticeable overhead in the ACF compared to the CF when it comes to queries due to the need to hash a given query multiple times. The CF can use a single hash function to obtain both the index and the tag of an item simultaneously. On the other hand, the tag of an item in the ACF depends on the location of the tag. This means that the ACF must first apply a hash to calculate

[3]Load factor is the ratio of the number of occupied slots over the total number of slots in the filter

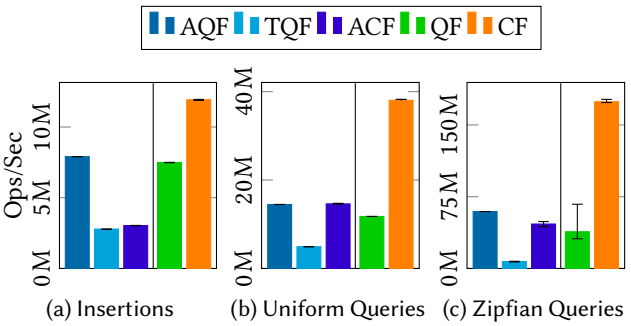


Fig. 3. Micro operation throughput of filters absent any external system. Adaptive filters are compared on the left, while nonadaptive filters are shown as reference on the right.

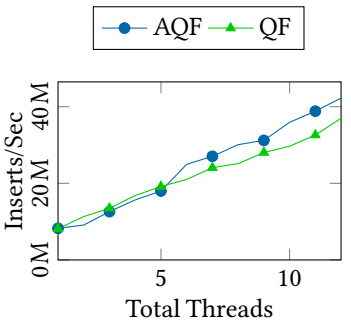


Fig. 4. Parallel insertion throughput scaling up with number of threads. Filters built with 2^{26} slots.

the indexes of an item, and only then apply additional hashes to search for the tag. Similarly, the TQF also sees a lower throughput due to the additional overhead of applying arithmetic coding to encode and decode hash selectors when making queries. Aside from queries, both the ACF and TQF have additional overhead during insertions for the same reason. As items naturally shift and move during insertions, the tags stored need to be rehashed in order to reflect their new locations.

When switching to Zipfian queries, all five filters benefit. Since the filters are easily large enough to overwhelm the machine's L3 cache, skewing the queries allows the cache to be more effective. The adaptive filters can also maintain a significantly lower false positive rate than the non-adaptive filters. However, the extra overhead in the ACF and TQF from the use of hash selectors puts a cap on how fast their queries can be, so the decreased false positive rate and increased cache friendliness have limited benefit. The QF ends up having high variance in its Zipfian query speed. This is a result of the high impact of locality in the QF, which uses linear probing in contrast to the CF. When popular items fall in small clusters, queries are fast, but if they are in larger clusters, query performance slows down. The AdaptiveQF does not see the same variance since it quickly adapts to any Zipfian distribution regardless of its locality.

Space. We evaluate the space efficiency of the filters by measuring the actual space needed to store items. We report the space efficiency at 90% load factor. This is space usage prior to any adaptation, so each filter contains the same number of items and uses the same number of slots. Table 2 shows the empirical space usage and false-positive rate of different filters in these experiments. The space reported in the table is only the filter space. It does not include the space required by the reverse hash map. The AdaptiveQF has $\sim 8-9\%$ space overhead compared to the non-adaptive quotient filter.

Parallelism. The AdaptiveQF preserves thread safety from the counting quotient filter. It divides the slots in blocks of 4096 slots each and uses a lightweight spin lock for each block to avoid corruption. During an insertion or an adaptation, each thread first acquires two locks on consecutive blocks, the block in which the item hashes and the next one. Two consecutive locks helps to avoid any corruption in case the shifting of items overflows into the next block.

It is also possible to execute mixed operations concurrently using two modifications. First, locks would also have to be acquired during queries, which would not be necessary if insertions and queries are performed in separate phases. Second, if the database being used also supports concurrent inserts, the lock acquired during filter inserts would need to be held until the database insert is finished. This is to ensure the items in the same minirun are also inserted into the database in the same order as they are inserted into the filter. However, this is only necessary if there are mixed inserts and adaptations

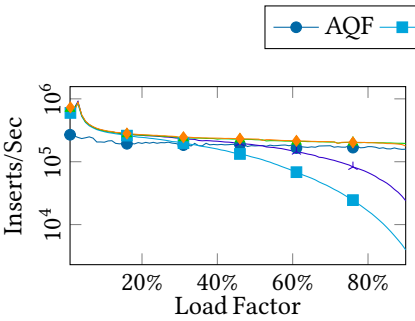


Fig. 5. System insert throughput as filter fills up.

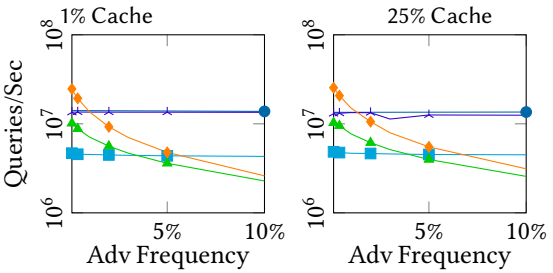


Fig. 6. Query throughput with varying cache size and frequency of adversarial queries.

Filter	Size (log)	Map Inserts	Map Updates	Map Queries
AQF	20	943718	0	0
TQF	20	943718	3608887	3356560
ACF	20	947815	584829	584829
AQF	24	15099494	0	0
TQF	24	15099494	56697889	52650676
ACF	24	15103591	9336669	9336669

Table 1. Number of reverse map accesses during insertions. The TQF and ACF make additional updates and queries in maintaining the reverse map. All filters of a given size were filled to 90% load.

Filter	−log(FPR)	Space (MB)
AQF	9	203.610
TQF	9	218.104
ACF	9	201.402
QF	9	186.818
CF	9	201.401

Table 2. Empirical space usage for same false positive rate. All filters were created with 2^{26} slots.

Filter	CAIDA	Shalla
AQF	31.6M	15.8M
TQF	8.4M	6.7M
ACF	34.1M	18.3M
QF	24.1M	16.8M
CF	119.2M	62.2M

Table 3. Queries per second on real-world datasets after 2^{26} inserts.

being done concurrently; in an insert-only workload, items in a minirun are identical in the filter until an adaptation happens, so the order of insertions into the database does not matter.

Figure 4 shows insertion throughput of the AdaptiveQF in isolation, as a function of the number of threads used, to demonstrate that the AdaptiveQF itself maintains good parallelism. We also show the performance of the QF for comparison. For this experiment, we use a filter of size 2^{26}, and we configure the locks to span 2^{16} slots each. Therefore, there are 2^{10} locks and contention is low. We vary from 1 thread to 12 threads in the increments of 2. Both the AdaptiveQF and QF show almost linearly scaling with the increasing number of threads, with the AdaptiveQF being slightly faster.

6.4 System benchmarks

In this section, we evaluate the performance of AdaptiveQF as a front-end filter to a disk-based B-tree database. We create an instance of the disk-based B-tree by using the B-tree implementation from SplinterDB [29]. For these tests, we disable the B^ϵ-tree structure and its accompanying filters in SplinterDB, and use it as a filter-less on-disk dynamic B-tree. Because filters are frequently used

alongside databases too large to fit in memory, storing data on disk using the B-tree is representative of real-world database systems.

We create an in-memory filter together with an on-disk map holding uniformly distributed keys with randomly generated values that represents a database. For the non-adaptive filters, the database holds the full set of keys-value pairs. For the adaptive filters, the database instead maps the fingerprints stored in the filter to their associated key-value pairs based on the optimization of using the reverse map as the database described in Section 4.

To perform an insertion, a key is first inserted into the in-memory filter, and then the key-value pair is inserted into the database. For non-adaptive filters, the key-value pair is inserted into the database directly. For adaptive filters, a fingerprint is obtained when inserting into the filter, and then the fingerprint-key-value triple is inserted into the database, mapping the fingerprint to the key-value pair.

To perform a query, the key is queried in the filter. If the filter returns "negative," then that key is not in the database and no disk query is performed. If the filter returns a "positive," the key and any corresponding data are retrieved from the database. The non-adaptive filters do this by directly querying the database for the key, and a false positive occurs if the database was not able to find that key. The adaptive filters instead query the database for the fingerprint found in the filter, and a false positive occurs if the key stored in the database does not match the key that was queried for. The adaptive filters can then use the returned key to adapt the filter so that the queried key no longer returns "positive."

Insertion performance. Figure 5 shows insertion throughput of the database as a function of the filter load factor. For this experiment, we create filters with 2^{25} slots and insert keys from a uniform random distribution until the filters are 90% full. At 1% progress intervals, we record the amount of time taken and calculate the insertion throughput over that interval.

The system has similar performance when using the AdaptiveQF compared to the non-adaptive QF and CF filters. This shows that there is little to no overhead of using the adaptive filter on the insertion performance of the system.

Since insertions into the B-tree are the main bottleneck, all 5 filters start with roughly equal insertion throughput. However, the ACF and TQF fall off over time. This is due to the cost of maintaining the reverse map. Table 1 shows the number of additional accesses to the reverse map done by the adaptive filters. As fingerprints are inserted into the AdaptiveQF, no entries for previous insertions need to be modified in the reverse map. As the ACF fills up, it needs to do a large number of kick outs. Since the tag being stored to represent an item changes depending on its location in the filter, it is not sufficient to simply move a tag when performing a kick out. Instead, every kick out requires an expensive query to the backing map so that a new fingerprint can be hashed. The frequency of kick outs increases with load factor. When a fingerprint is inserted into the TQF, it may cause other fingerprints to shift. The reverse map implemented with the TQF is based on location – keys are stored alongside their fingerprints and thus need to shift with them. The constant shifting of fingerprints induces many additional reverse map accesses.

Adversarial query performance. In Figure 6, we measure the effect of a query-only adversary on system throughput. Even if the overall query distribution is uniform, an attacker can artificially skew the distribution by skewing their own queries. An adversary can detect the latency difference between negative and positive queries (including false positives), and even without knowledge of the actual insertion set, record a list of positive queries. They can then repeat these queries to intentionally induce I/Os. Even in a system with a cache, the adversary needs only collect enough false positives to overload the cache, then proceed to cycle between these queries to render the cache ineffective.

In this experiment, we configure a filter of size 2^{26} and insert 90% that many random 32-byte key-value pairs. Then, we perform 200M queries. The first 100M warm up the cache and give the adversary time to collect false positives. We then measure the average query throughput over the next 100M

queries. We vary both the cache size and the frequency of adversarial queries over different trials. We do experiments on cache sizes of 1%, 3%, 6%, 12%, and 25% the size of the input dataset. Figure 6 shows only the results of the minimum and maximum cache sizes. There are marginal improvements to the performance of the non-adaptive filters when provided a larger cache. However, even with the cache holding 25% of the dataset, an adversary representing less than 1.5% of the total queries can cause query throughput for the entire system to drop by 2× that of normal operation, which is already lower than that of the AdaptiveQF. This increases to 3× with 3% adversarial influence and up to 10× with 10% adversarial influence.

It is intuitive to expect that a large cache would be able to offset the effects of an adversary. However, these experiments show that increasing the size of the cache has a disproportionately low impact. The adversary induces disk accesses by cycling through known false positives which it hopes are out-of-cache. The chance that a false positive is out of cache equals the proportion of key-value pairs that do *not* fit in cache. That is, by increasing the cache size 25×, we do not decrease the effectiveness of the adversary by 25×, but rather by 25%. A vast majority of of adversarial queries (75%) will still be out-of-cache. Moreover, with the cache being a page cache, the adversary only needs to collect one false positive from each page (or at least from most pages) in order to effectively cycle the entire dataset through cache. In conclusion, any reasonably-sized cache is largely ineffective against this particular kind of query adversary.

The AdaptiveQF offers high and consistent query performance irrespective of the frequency of adversarial queries. Even without adversarial queries the AQF has comparable query performance to the non-adaptive filters. But in the presence of adversarial queries it can offer up to an order of magnitude higher query performance.

Merged vs. split reverse map. As discussed in Section 4, the reverse map and database can be merged into a single data structure so that reverse map inserts and queries do not incur additional overhead. This makes the database unable to perform range queries. The split reverse map and database setup, however, does support range queries. Table 4 compares the two setups to show the overhead of using the split setup in the case that one would like to use range queries. The insertion throughput is halved due to needing to insert into both the database and reverse map individually. However, query throughput is affected by only about 1% due to the infrequency of false positives on general workloads.

6.5 Adaptivity rate for real-world datasets

In application benchmarks, we use real-world datasets to evaluate the rate of change of false positive rate and space usage in adaptive filters in the presence of queries.

To evaluate the false positive rate over time, we first construct all three adaptive filters and fill them to 90% load factor. We then construct a query set that will be performed over time, and the filters will adapt to the false positive queries. We also construct multiple independent query sets from the Zipfian distribution, which we use to compute the instantaneous false positive rate. The filters do not adapt while measuring the instantaneous false positive rate.

We perform a total of 3 million queries when the filters adapt to measure the rate of change of false positive rate and the space usage. We compute the instantaneous false positive rate and space usage after every 1% of queries. To compute the instantaneous false positive rate, we construct 100 independent query sets from a Zipfian distribution. During the false positive computation, we turn off the adaption in the filters. Therefore, filters only adapt during normal queries and do not adapt while computing the false positive rate.

We use three different datasets for application workloads. The first dataset is synthetic and generated from the Zipfian distribution (with Zipfian constant 1.5) on a universe of size 1 billion. The second dataset is CAIDA passive traces [17], a set of anonymized network traces collected by the

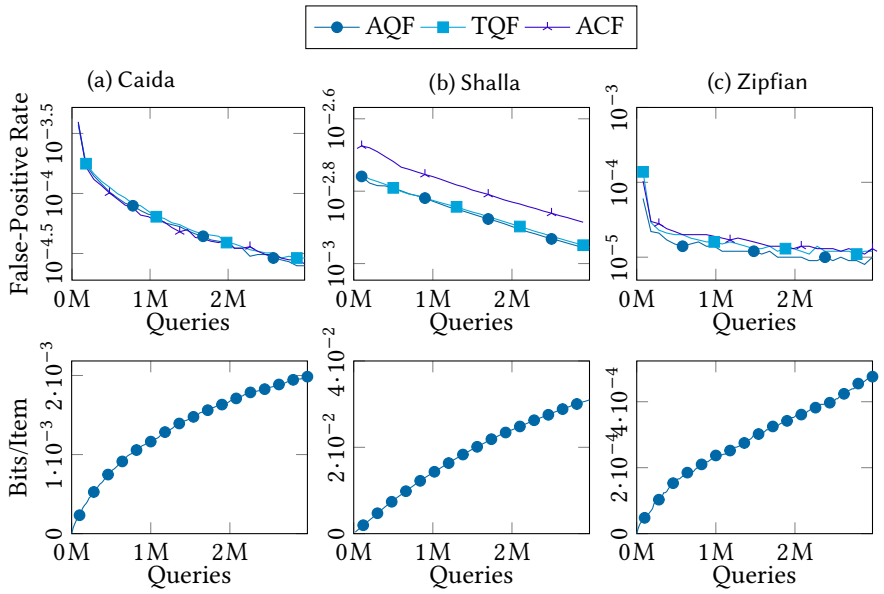


Fig. 7. False-positive rate (FPR) and additional space usage over time. The adaptive filters are able to almost immediately reduce their FPR by up to 100× on a skewed workload. The AdaptiveQF is able to achieve a marginally lower or equivalent FPR at negligible space overhead.

Center for Applied Internet Data Analysis between April 2008 and January 2019. The third database, the Shalla [53] block list, is a list of about 3 million malicious URLs compiled by Shalla Secure Services. For our experiments, we perform insertions and queries from the Shalla list.

Figure 7 shows the rate of change of false positive rate and space during queries in adaptive filters. The false positive rate immediately drop for all three filters. This is because the filters adapt to hot items early in the query sequence. Later on, the filters adapt to infrequent items, each of which brings a smaller drop in the false positive rate.

The drop in false positive rate over time is similar for all three adaptive filters for the Caida and Shalla datasets. These two datasets are not very skewed, and therefore the strong adaptivity advantage of AdaptiveQF over the TQF and ACF is not especially apparent.

On Zipfian queries, the false positive rate for all three filters drops equally. However, over time the false positive rate in the AdaptiveQF drops to lower than the TQF and ACF. This shows that the strong adaptivity guarantees in the AdaptiveQF lead to a lower false-positive rate over time. The TQF and ACF do not adapt completely the first time they encounter the false positive, which can result in subsequent false positive results when colliding with the already adapted key.

The space usage (in bits/item) increases over time slightly in the AdaptiveQF, because adaptation involves using additional slots for some fingerprints. However, the rarity of false positives means this additional space is negligible, and false positives will become even more rare over time.

For actual query throughput, we list the numbers in Table 3. This includes costs incurred by occasional queries to the database. The AdaptiveQF has comparable query throughput to both the AQF and QF. All filters benefit from cache-friendliness induced by the skewed distribution of CAIDA's queries. However, the AdaptiveQF and ACF see more improvement than the QF due to their ability to adapt to the most popular false positive queries in the distribution.

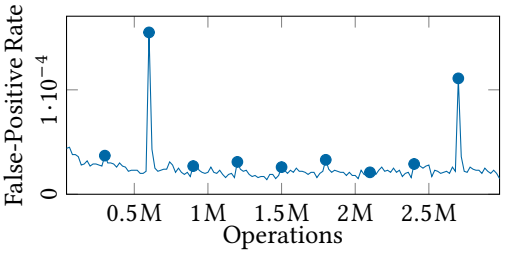


Fig. 8. False-positive rate over time on dynamic workloads. Churning points are marked.

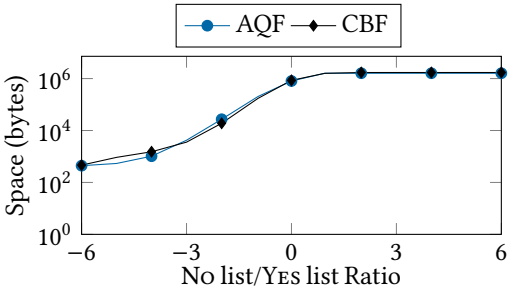


Fig. 9. Space usage with varying ratio of NO list to YES list. Log of NO/YES ratio is provided on the x-axis.

6.6 Dynamic workloads

Apart from static workloads, we also evaluate the change of false positive rate in the AdaptiveQF in the presence of deletions, insertions, and queries over time. This simulates the real-world use cases where the items in the YES list change over time. We do not include other adaptive filter implementations, TQF and ACF, in this experiment as they do not support deletes.

Like with Figure 7, we compute instantaneous false-positive rates after every 1% of queries. At 10% intervals, we delete and replace 20% of the items. To compute the instantaneous false-positive rate, We use 1 million queries from the same Zipfian distribution without adapting.

Figure 8 shows the false positive rate over time in the presence of deletions, insertions, and queries. Every 10% of the operations, we introduce a massive churn in which 20% of the items in the filter are replaced. There are a couple of spikes in the false positive rate that coincide with the churns. They are caused when one of the inserted items causes a popular query to become a false positive. But the filter quickly adapts to the new item, and the false positive rate once again drops very low. TQF and ACF are excluded from these experiments as those implementations do not support deletions.

In these experiments, we lose strong adaptivity. This is a deliberate choice and not a limitation of the AdaptiveQF. We can support strong adaptivity in the presence of updates to the NO list and YES list by associating a small value to the fingerprints as described in Section 4.3. Strong adaptivity can be preserved in the presence of deletions by setting an item's counter to zero instead of deleting the item completely. We chose in these experiments not to preserve strong adaptivity in order to demonstrate the filter's ability to maintain a low false positive rate in dynamic environments and because the highly dynamic nature of these experiments would make the extra space usage impractical.

6.7 Merge and bulk load performance

Filters are often used to build inverted text indexes on genomics data [69], where they are merged with other filters during compaction. Therefore, mergeability is a critical feature in filters for easy adoption in database systems.

The AdaptiveQF supports efficient merging by the same means as the CQF, since we do not store any auxiliary hash encoding information. In contrast, merging in the TQF and ACF is not straightforward due to the hash selectors obscuring the original keys. To evaluate the merge performance of the AdaptiveQF, we use an in-memory hash table as the reverse map because we just want to evaluate the filter's merging speed. Note that merging two reverse maps is easy, because minirun lists sharing an ID can be concatenated, so long as miniruns in the filter are also concatenated during merging.

We also evaluate bulk loading in the AdaptiveQF, where the entire list of items is known. We find that the raw execution time of merging and bulk inserting is extremely low. For bulk loading, we would prepare by first sorting the items in hash order.

Reverse Map Setup	Inserts	Queries
Merged	2.32×10^5	1.843×10^7
Split	1.12×10^5	1.819×10^7

Table 4. Operations per second using the merged vs split reverse map setup. Tests were run using filters of size 2^{26}. In the split setup, reverse map and database inserts are independent, so insertion has half the speed. However, due to the infrequency of false positives, querying is only 1-2% slower.

Operation	Time/insert (μs)
Insert into filter	0.520092
Insert into half-size filter	0.353332
Merge two half-size filters	0.039147
Sort in hash order (qsort)	0.348060
Bulk insert	0.019569

Table 5. Average latency for random inserts; inserting into two filters of half the size and then merging; and sorting items beforehand followed by bulk inserting.

Table 5 shows the merge and bulk-build performance of the AdaptiveQF. Normal and bulk inserts were done on filters of size 2^{26} until 90% load factor. Merging was done between two filters of size 2^{25}. Merging two existing filters is 13x faster than constructing a full filter from scratch, thanks to the fingerprints in the smaller filters already being sorted. Similarly, bulk loading, in which we sort and then insert using a specialized insertion procedure, is about 70% faster than random insertions. For sorting we used the C standard qsort function. Specialized sorting functions for a given situation may be even faster. Merging is slower than bulk loading due to comparing quotient-remainder pairs between the two merged filters, as well as an overhead in identifying runs when stepping through the filter.

6.8 Space comparison to static Yes/No solution

Figure 9 shows the space usage of CRLlite [55], a custom-built and static yes/no list solution based on the cascading Bloom filter, and the AdaptiveQF. The space usage of the AdaptiveQF while being dynamic is always smaller or similar to CRLite. For the evaluation, we fix the aggregate size of the no list and yes list to 1 million items and evaluate the space with changing ratio of the no list and yes list.

6.9 Non-adaptive filter additional space

The adaptive filters have higher space usage (due to the overhead of adaptivity) compared to the non-adaptive filters. Therefore, we performed an experiment where we configured the QF and CF with a higher number of bits to give them extra space and lower false-positive rate. With extra space, the uniform query performance of the CF increases by 1%, and the Zipfian query performance increases by 0.3%. Similar performance gains are seen for the QF. Therefore, even with extra space and a lower false-positive rate, the CF-based system is 20% slower compared to the AdaptiveQF-based system.

7 Conclusion

We introduce AdaptiveQF in this paper. The AdaptiveQF is the first strongly adaptive filter which supports high throughput operations using single-hashing and quotienting. Using the adaptive filters in the system we can increase the overall system throughput by avoiding repeated unnecessary accesses to the backing stores (or other slower storage). The strongly adaptive filter guarantees consistently low false-positive rate over time on dynamic workloads.

Traditional filters have been the go-to data structure for over five decades. However, traditional filters lose their benefits in the presence of real-world skewed and adversarial workloads. Today's applications can benefit from practical adaptive filters that offer strong theoretical guarantees and high performance independent of the data distribution in order to quickly and efficiently perform complex analyses on large-scale data.

Acknowledgments

This research is funded in part by NSF grant OAC 2339521.

References

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom filters. *Journal of Information Processing Letters* 101, 6 (2007), 255–261.

[2] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. 2014. Storage management in AsterixDB. *Proceedings of the VLDB Endowment* 7, 10 (2014), 841–852.

[3] Apache. [n. d.]. Cassandra. http://cassandra.apache.org.

[4] Austin Appleby. 2016. SMHasher source code in C++. https://github.com/aappleby/smhasher

[5] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in Parallel Query Processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Snowbird, Utah, USA) *(PODS '14)*. Association for Computing Machinery, New York, NY, USA, 212–223. https://doi.org/10.1145/2594538.2594558

[6] Michael A. Bender, Rathish Das, Martín Farach-Colton, Tianchi Mo, David Tench, and Yung Ping Wang. 2021. Mitigating False Positives in Filters: to Adapt or to Cache?. In *Proc. 2nd Symposium on Algorithmic Principles of Computer System (APoCS)*.

[7] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. 2018. Bloom Filters, Adaptivity, and the Dictionary Problem. In *Proc. 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. Paris, France, 182–193.

[8] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kaner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the VLDB Endowment* 5, 11 (2012).

[9] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. 2012. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1627–1637.

[10] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *PVLDB* 5, 11 (2012), 1627–1637.

[11] Burton H. Bloom. 1970. Space/time Trade-offs in Hash Coding With Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[12] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting Bloom filters. In *European Symposium on Algorithms (ESA)*. Springer, 684–695.

[13] 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.

[14] Alex D Breslow and Nuwan S Jayasena. 2018. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1041–1055.

[15] Gerth Stølting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 546–554.

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

[17] CAIDA. 2016. Anonymized Internet Traces 2016. https://www.caida.org/catalog/datasets/passive_dataset/

[18] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Christian A Lang, and Kenneth A Ross. 2010. Buffered Bloom Filters on Solid State Storage.. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 1–8.

[19] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. 1978. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*. 59–65.

[20] Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (FOCS)*. 281–288.

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 205–218.

[22] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 30–39.

[23] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *ACM Sigmod Record* 39, 3 (2011), 5–10.

[24] Long Cheng, Spyros Kotoulas, Tomas E. Ward, and Georgios Theodoropoulos. 2014. Robust and Skew-Resistant Parallel Joins in Shared-Nothing Systems. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management* (Shanghai, China) *(CIKM '14)*. Association for Computing Machinery, New York, NY, USA,

1399–1408. https://doi.org/10.1145/2661829.2661888
[25] Rayan Chikhi and Guillaume Rizk. 2013. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology* 8, 1 (2013), 1.
[26] Rayan Chikhi and Guillaume Rizk. 2013. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology* 8, 1 (2013), 22.
[27] 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.
[28] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. 2009. Power-law distributions in empirical data. *SIAM review* 51, 4 (2009), 661–703.
[29] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. {SplinterDB}: Closing the Bandwidth Gap for {NVMe}{Key-Value} Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 49–63.
[30] 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*. 143–154.
[31] Bernat Corominas-Murtra and Ricard V Solé. 2010. Universality of Zipf's law. *Physical Review E* 82, 1 (2010), 011102.
[32] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.
[33] Biplob Debnath, Sudipta Sengupta, Jin Li, David J Lilja, and David HC Du. 2011. BloomFlash: Bloom filter on flash-based storage. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*. 635–644.
[34] Biplob K Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory.. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.
[35] Kyle Deeds, Brian Hentschel, and Stratos Idreos. 2020. Stacked filters: learning to filter by structure. *Proceedings of the VLDB Endowment* 14, 4 (2020), 600–612.
[36] Peter C. Dillinger and Panagiotis (Pete) Manolios. 2009. Fast, All-Purpose State Storage. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software* (Grenoble, France). Springer-Verlag, Berlin, Heidelberg, 12–31. https://doi.org/10.1007/978-3-642-02652-2_6
[37] Jennie Duggan, Olga Papaemmanouil, Leilani Battle, and Michael Stonebraker. 2015. Skew-Aware Join Optimization for Array Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/2723372.2723709
[38] Gil Einziger and Roy Friedman. 2016. Counting with TinyTable: Every Bit Counts!. In *Proceedings of the 17th International Conference on Distributed Computing and Networking* (Singapore, Singapore) *(ICDCN '16)*. Association for Computing Machinery, New York, NY, USA, Article 27, 10 pages. https://doi.org/10.1145/2833312.2833449
[39] Bennett Eisenberg. 2008. On the expectation of the maximum of IID geometric random variables. *Statistics & Probability Letters* 78, 2 (2008), 135–143. https://www.sciencedirect.com/science/article/pii/S0167715207002040
[40] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2012. The TokuFS Streaming File System. In *Proc. 4th USENIX Workshop on Hot Topics in Storage (HotStorage)*. Boston, MA, USA.
[41] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. 75–88.
[42] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 75–88.
[43] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)* 8, 3 (2000), 281–293.
[44] Martin Farach-Colton, Rohan J. Fernandes, and Miguel A. Mosteiro. 2009. Bootstrapping a hop-optimal network in the weak sensor model. *ACM Transactions on Algorithms* 5, 4 (2009).
[45] Google, Inc. 2015. LevelDB: A fast and lightweight key/value database library by Google. http://github.com/leveldb/, Last Accessed May 16, 2015.
[46] Thomas Mueller Graf and Daniel Lemire. 2020. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *ACM J. Exp. Algorithmics* 25, Article 1.5 (March 2020), 16 pages. https://doi.org/10.1145/3376122
[47] Trish Hogan. 2009. Overview of tpc benchmark e: The next generation of oltp benchmarks. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 84–98.
[48] Russell Housley, Warwick Ford, William Polk, and David Solo. 1999. *Internet X. 509 public key infrastructure certificate and CRL profile*. Technical Report.
[49] InternetLiveStats.com. 2022. *Google search statistics*. https://www.internetlivestats.com/google-search-statistics/

[50] Shaun D Jackman, Benjamin P Vandervalk, Hamid Mohamadi, Justin Chu, Sarah Yeo, S Austin Hammond, Golnaz Jahesh, Hamza Khan, Lauren Coombe, Rene L Warren, et al. 2017. ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter. *Genome research* 27, 5 (2017), 768–777.

[51] 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 *Proc. 13th USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, USA), Jiri Schindler and Erez Zadok (Eds.). 301–315.

[52] 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: Write-Optimization in a Kernel File System. *Transactions on Storage—Special Issue on USENIX FAST 2015* 11, 4 (2015), 18:1–18:29.

[53] Shalla Secure Services KG. [n. d.]. Shalla's Blacklists. http://www.shallalist.de/index.html

[54] Tsvi Kopelowitz, Samuel McCauley, and Ely Porat. 2021. Support optimality and adaptive cuckoo filters. In *Workshop on Algorithms and Data Structures*. Springer, 556–570.

[55] James Larisch, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. 2017. CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers. In *2017 IEEE Symposium on Security and Privacy (SP)*. 539–556. https://doi.org/10.1109/SP.2017.17

[56] David J. Lee, Samuel McCauley, Shikha Singh, and Max Stein. 2021. Telescoping Filter: A Practical Adaptive Filter. 204 (2021), 60:1–60:18. https://doi.org/10.4230/LIPIcs.ESA.2021.60

[57] Allen Leng. 2022. 1 in 2 visitors abandon a website that takes more than 6 seconds to load. https://digital.com/1-in-2-visitors-abandon-a-website-that-takes-more-than-6-seconds-to-load/

[58] Scott T Leutenegger and Daniel Dias. 1993. A modeling study of the TPC-C benchmark. *ACM Sigmod Record* 22, 2 (1993), 22–31.

[59] Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Siqiang Luo, Rong Gu, Tong Yang, and Guihai Chen. 2022. Seesaw Counting Filter: An Efficient Guardian for Vulnerable Negative Keys During Dynamic Filtering. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 2759–2767. https://doi.org/10.1145/3485447.3511996

[60] Steve Lohr. [n. d.]. For Impatient Web Users, an Eye Blink Is Just Too Long to Wait. *The New York Times* ([n. d.]). https://www.nytimes.com/2012/03/01/technology/impatient-web-users-flee-slow-loading-sites.html

[61] Guanlin Lu, Biplob Debnath, and David HC Du. 2011. A Forest-structured Bloom Filter with flash memory. In *Proceedings of the 27th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–6.

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

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

[64] Nima Mousavi and Mahesh Tripunitara. 2019. Constructing cascade bloom filters for efficient access enforcement. *Computers & Security* 81 (2019), 1–14. https://doi.org/10.1016/j.cose.2018.09.015

[65] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[66] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (feb 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[67] Patrick O'Neil, Edward Cheng, Dieter Gawlic, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[68] 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*. Society for Industrial and Applied Mathematics, 823–829.

[69] Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. 2018. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell systems* 7, 2 (2018), 201–207.

[70] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics* 33, 14 (2017), i133–i141.

[71] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 775–787.

[72] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics* 34, 4 (2017), 568–575.

[73] Prashant Pandey, Alex Conway, Joe Durie, Michael A Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 International Conference on Management of Data*. 1386–1399.

[74] Prashant Pandey, Shikha Singh, Michael A Bender, Jonathan W Berry, Martín Farach-Colton, Rob Johnson, Thomas M Kroeger, and Cynthia A Phillips. 2020. Timely Reporting of Heavy Hitters using External Memory. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1431–1446.

[75] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. 2012. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences* 109, 33 (2012), 13272–13277.
[76] Felix Putze, Peter Sanders, and Johannes Singler. 2007. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*. 108–121.
[77] Yan Qiao, Tao Li, and Shigang Chen. 2014. Fast Bloom Filters and Their Generalization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 25, 1 (2014), 93–103.
[78] Brandon Reagen, Udit Gupta, Robert Adolf, Michael M Mitzenmacher, Alexander M Rush, Gu-Yeon Wei, and David Brooks. 2017. Weightless: Lossy weight encoding for deep neural network compression. *arXiv preprint arXiv:1711.04686* (2017).
[79] Pedro Reviriego, Alfonso Sánchez-Macián, Stefan Walzer, and Peter C. Dillinger. 2021. Approximate Membership Query Filters with a False Positive Free Set.
[80] Pedro Reviriego, Alfonso Sánchez-Macián, Stefan Walzer, and Peter C. Dillinger. 2021. Approximate Membership Query Filters with a False Positive Free Set. https://doi.org/10.48550/ARXIV.2111.06856
[81] RocksDB [n. d.]. RocksDB. https://rocksdb.org/, Last Accessed Oct. 15, 2022.
[82] Roye Rozov, Ron Shamir, and Eran Halperin. 2014. Fast lossless compression via cascading Bloom filters. *BMC Bioinformatics* 15 (2014).
[83] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. 2014. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithms for Molecular Biology* 9, 1 (2014), 1–10.
[84] ScyllaDB. [n. d.]. ScyllaDB. https://www.scylladb.com/.
[85] Securelist.com. 2022. . https://securelist.com/kaspersky-security-bulletin-2021-statistics/105205/
[86] Brad Solomon and Carl Kingsford. 2016. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology* 34, 3 (2016), 300.
[87] Henrik Stranneheim, Max Käller, Tobias Allander, Björn Andersson, Lars Arvestad, and Joakim Lundeberg. 2010. Classification of DNA sequences using Bloom filters. *Bioinformatics* 26, 13 (2010), 1595–1600.
[88] Bo Sun, Mitsuaki Akiyama, Takeshi Yagi, Mitsuhiro Hatada, and Tatsuya Mori. 2016. Automating URL blacklist generation with similarity search approach. *IEICE TRANSACTIONS on Information and Systems* 99, 4 (2016), 873–882.
[89] Mahesh V. Tripunitara and Bogdan Carbunar. 2009. Efficient Access Enforcement in Distributed Role-Based Access Control (RBAC) Deployments. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies* (Stresa, Italy) *(SACMAT '09)*. Association for Computing Machinery, New York, NY, USA, 155–164. https://doi.org/10.1145/1542207.1542232
[90] David Wajc. 2017. Negative Association - Definition, Properties, and Applications.
[91] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. 16:1–16:14.
[92] 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. *CoRR* abs/2405.10253 (2024). https://doi.org/10.48550/ARXIV.2405.10253 arXiv:2405.10253
[93] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A Bender, et al. 2017. Writes wrought right, and other adventures in file system optimization. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–26.
[94] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2016. Optimizing Every Operation in a Write-Optimized File System. In *Proc. 14th USENIX Conference on File and Storage Technologies (FAST)*.
[95] Wangda Zhang and Kenneth A. Ross. 2022. Exploiting Data Skew for Improved Query Performance. *IEEE Transactions on Knowledge and Data Engineering* 34, 5 (2022), 2176–2189. https://doi.org/10.1109/TKDE.2020.3006446
[96] Benjamin Zhu, Kai Li, and R Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*. 1–14.

Received January 2024; revised April 2024; accepted May 2024