Beyond Bloom: A Tutorial on Future Feature-Rich Filters

Prashant Pandey
ppandey@cs.utah.edu
University of Utah
USA

Martín Farach-Colton
martin@farach-colton.com
New York University
USA

Niv Dayan
nivdayan@cs.toronto.edu
University of Toronto
Canada

Huanchen Zhang
huanchen@tsinghua.edu.cn
Tsinghua University
China

ABSTRACT

Filters, such as Bloom, quotient, and cuckoo, save space by maintaining an approximate representation of a set and occasionally returning false positives. Filters play a critical role in building modern data-intensive applications and are used across various domains such as databases, storage engines, computational biology, cybersecurity, and networks. There has been extensive research on filters in the past few decades resulting in filters with much improved performance and features. Yet modern data-intensive applications are still designed around the limitations of traditional filters resulting in complex designs and sub-optimal performance.

This tutorial aims to bring together researchers at the forefront of filter data structure research to help the database community learn about the recent advancements in the theory and practice of filters. The tutorial will cover real-world case studies of redesigning applications using the modern filter APIs to achieve simplicity and improved application performance. The tutorial will further help uncover the open research problems, both in theory and systems, and increase interaction among researchers to tackle those problems.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; **Bloom filters and hashing**.

KEYWORDS

Dictionary data structure, Filters, Membership query

ACM Reference Format:
Prashant Pandey, Martín Farach-Colton, Niv Dayan, and Huanchen Zhang. 2024. Beyond Bloom: A Tutorial on Future Feature-Rich Filters. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3626246.3654681

1 INTRODUCTION

Filters, such as Bloom [15], quotient [13, 35, 45, 46, 75, 77], and cuckoo [18, 49] filters, maintain an approximate representation of a set or a multiset. The approximate representation saves space by allowing queries to occasionally return a false positive. For a given false-positive rate ε: a membership query to a filter for set S returns present for any $x \in S$, and returns absent with probability at least $1-\varepsilon$ for any $x \notin S$. A filter for a set of size n uses space that depends on ε and n but is much smaller than explicitly storing all items of S.

Filters offer performance advantages when they fit in cache but the underlying data does not. Filters are widely used in databases, networks, storage systems, machine learning, computational biology, and other areas [6, 17, 19, 25, 26, 40, 41, 47, 50, 57, 78, 81, 88, 89, 94, 104]. For example, in databases, filters are used to summarize the contents of on-disk data, query optimization, and table joins [9, 10, 20, 29, 30, 32, 58, 59, 80, 94, 100, 101]. In networks, they are used to summarize cache contents, implement network routing, and maintain probabilistic measurements [19]. In computational biology, they are used to represent huge genomic data sets compactly [4, 5, 25, 73, 74, 76, 78, 88].

Filters can be broadly classified according to the operations they support. ***Static filters*** do not support insertions or deletions and are built on a set of items that are known at the time of construction. Examples of these filters include XOR filters [52] and Ribbon filters [44]. ***Semi-dynamic*** filters support insertions but no deletions. Such filters include the Bloom filter [15] and Prefix filter [48]. For the most part, static and semi-dynamic filters are only suitable for representing immutable data (e.g., files in an LSM-tree or keys in a join operation).

In contrast, ***dynamic*** filters, such as cuckoo and quotient filters, support both insertions and deletions. In fact, quotient filters support a wide number of features including the abilities to (1) efficiently scale out of RAM, (2) resize dynamically, (3) count the number of times each input item occurs (represent multisets), (4) handle skewed input distributions (which are common in DNA sequencing and other real-world datasets), (5) associate small values with keys, a variant of filters called ***maplets***, and (6) scale with the number of threads (i.e., achieve high concurrency). Recent advances such as InfiniFilter [35] allow dynamic filters to expand more efficiently and indefinitely. We will discuss various modern data-intensive applications that have evolved to take advantage of these features.

Recently, researchers have introduced ***adaptive filters*** [12, 63, 68, 96]. Adaptive filters, such as the adaptive quotient filter [12, 96], telescoping filter [63], and adaptive cuckoo filter [68], update their structure upon detecting a false positive to avoid repeating the same errors in the future. Furthermore, the Adaptive quotient filter is monotonically adaptive, which means that the performance and false-positive probability guarantees continue to hold even for adversarial workloads.

Range filters [1, 43, 51, 102, 103], on the other hand, are a generalization of the classic point filtering problem to allow an interval as the filter input. Currently, range filters are mainly used in databases to quickly determine whether a given key range is empty to avoid unnecessary disk I/Os for a range query.

1.1 Goals

The primary goal of the tutorial is to bring together researchers who are at the forefront of filter data structure research and make the

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06.
https://doi.org/10.1145/3626246.3654681

communities (theory, systems, and applications) aware of the recent developments. This will further help uncover the open research problems, both in theory and systems, and increase interaction among researchers to tackle those problems.

The talks from experts will help researchers understand the recent advancements in filter theory and practice. Especially, the advanced API supported by modern filters compared to traditional filters such as Bloom filters. This will make the audience aware of modern ways of using filters in their applications and how they can reap the full potential of filters to achieve performance and scalability. For example, systems developers still use Bloom filters in traditional ways leaving performance on the table. Using modern filters (such as fingerprinting-based and adaptive filters) with advanced features can help reduce development time and also simplify system pipelines.

Finally, this tutorial will help foster new collaborations between researchers doing filter data structure research and also between core data structure researchers and application/systems researchers.

1.2 Duration and format

The tutorial duration is 3 hours. The tutorial will be organized as a combination of research talks and a panel discussion. The research talks will help give a brief overview of filter data structures and state-of-the-art research across various fronts, such as maplets, expandable filters, range filters, GPU filters, etc. We will also cover the modern filter API and how researchers can integrate filters into their applications. The tutorial will cover how various application domains (machine learning, computational biology, storage, databases) use filters.

2 TYPES OF FILTERS

Filter data structures can be broadly classified into *static*, *semi-dynamic*, and *dynamic*. Static filters approximately represent a set of items that must be known before building the filter. Examples of these filters include XOR filters [52] and Ribbon filters [44]. *Semi-dynamic* filters such as Bloom [15] and Prefix filters [48] support insertions without knowing the set of keys in advance. However, the set size must be known in advance to provide a guarantee over the false positive rate. As such, semi-dynamic filters are also typically only used for representing static key sets.

In contrast, dynamic filters approximately represent a set of items that do not need to be known before the construction. Dynamic filters have seen much more advancement in recent years as applications often do not know the set of items in advance. Examples of dynamic filters are quotient filters [13, 35, 45, 46, 71, 75] and cuckoo filters [18, 49]. This tutorial will cover in detail the mechanics and trade-offs of these filters and their many variants.

A dynamic filter is known to require $n \log \varepsilon^{-1} + \Omega(n)$ bits. For example, the quotient filter uses $n\log\varepsilon^{-1}+2.125n$ bits [75] and the cuckoo filter uses $n\log\varepsilon^{-1}+3n$ bits [49]. When ε has typical values such as 2^{-8} or 2^{-16}, then the overhead of $2.125n$ bits increases the space by 25% (resp. 12.5%), compared to the a filter that uses $n\log\varepsilon^{-1}$. (Note that a Bloom filter uses $1.44n\log\varepsilon^{-1}$, so it beats a modern filter for space only in non-practical cases where ε is close to one.)

2.1 Fingerprint-based dynamic filters

Most dynamic filters are fingerprint-based. They compactly and exactly store small, lossy fingerprints in a table using ***quotienting*** [71]. In quotienting, a p-bit fingerprint $h(x)$ is divided into two parts: the first q bits $h_0(x)$ is called the ***quotient*** and the remaining $r = p - q$ bits $h_1(x)$ is called the ***remainder***. The quotient is stored implicitly and only the remainder is stored explicitly to save space.

Quotient filters [13, 45, 46, 75, 77] use Robin Hood hashing (a variant of linear probing) to store the remainders in a linear table. To resolve soft collisions (i.e., when two fingerprints share the same quotient but have different remainders) it uses 2-3 extra metadata bits[1] to resolve collisions and efficiently perform queries. The cuckoo filter [18, 49] on the other hand uses a 4-way associative table to store remainders and 3 metadata bits. It employs cuckoo hashing to resolve collisions. Each item is hashed to two locations in the table and it performs kicking to make space for the new incoming item. Other variants such as the Crate [14] and Prefix [48] filters chain hash buckets to resolve collisions.

2.2 Expandable filters

In many applications that use filters, the data size grows over time. This creates a need for filters that can expand along with the data. For example, many modern log-structured key-value stores employ an in-memory maplet to map the location of each data entry in storage [7, 21, 38, 82]. As the data size grows, the maplet must expand to map a greater number of keys and their storage locations. In the networks community, expandable filters have recently been pointed out as pivotal for supporting black lists and multicast routing [97]. In the architecture community, expandable filters have been proposed to implement page tables [87].

The difficulty common to all filters with respect to expanding is that they store hashed representations of keys rather than the original keys themselves. Hence, an original key cannot be rehashed when expanding, as done with a regular hash table. The obvious workaround is to scan the original data and construct a filter with greater capacity from scratch. However, the cost of traversing the whole data set can be prohibitive. Another possibility is to pre-allocate a very large filter in advance, but this wastes a lot of memory from the get-go and restricts the ultimate set size the filter can represent. Yet another option is to create a chain (i.e., a linked list) of filters, and to add new filters to this chain as the data grows. Some works propose to add fixed-size filters to this chain [24, 53, 54], while others propose adding filters of geometrically increasing capacities [2, 98]. Either way, this approach increases query costs as all filters along the chain potentially need to be searched during a query [2, 72, 98].

A few recent filters conceptually form a hash ring of buckets to support elastic expansion [65, 99]. The issue is that queries, deletes, and insertions all become logarithmic as a search tree has to be searched to find a given entry's bucket.

Quotient filters provide limited support for expansion: it is possible to double their capacity and sacrifice one bit from each fingerprint to uniformly map it to the expanded hash table. The problem is that the fingerprints shrink as the data grows, and this increases the false positive rate (FPR). Eventually, the fingerprint bits run out, at which point the filter returns a positive for every query, and it cannot continue expanding.

[1]The original quotient filter [13] uses 3 metadata bits, the counting quotient filter [75] uses 2.125 metadata bits, and the vector quotient filter [77] uses 2.914 metadata bits.

Inspired by a theoretical construction [72], Taffy Cuckoo filter [8] allows expanding up to a known universe size while maintaining fast queries and a stable FPR, though it does not support deletes. The core idea is to use a hash table that supports variable-length fingerprints by augmenting each slot with self-delimiting unary padding. This allows sacrificing a bit from every existing fingerprint to uniformly map it to a larger hash table during expansion, while still setting long fingerprints to new entries inserted after expansion. InfiniFilter [35] improves on these ideas by supporting deletes and expanding up to an unbounded universe size, though queries are not constant time. The more recent Aleph Filter [34] improves on InfiniFilter with a constant time guarantee on all operations.

2.3 Adaptive filters

An adaptive filter is a filter that returns True with probability at most ε for every negative query, regardless of answers to previous queries. For a dictionary using an adaptive filter, any sequence of n negative queries will result in $O(\varepsilon n)$ false positives, with high probability. This gives a strong guarantee on the number of (expensive) negative accesses that the dictionary will need to make to disk. This is true even if the queries are selected by an adaptive adversary.

Adaptive filters are a generalization of traditional filters that enable filters to be employed for a diverse set of problems. Prior work has considered each of these problems separately and consequently has developed distinct approaches based on traditional filters to solve each of them. Examples include cascading Bloom filters [91], static XOR filters [83], seesaw counting filters [64], telescoping filters [63], and adaptive cuckoo filters [68]. All these filter variants turn out to be suboptimal in terms of performance and accuracy guarantees.

Bender et al. [11, 12] 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 adversarial query distribution, and present the broom filter, which meets their definition. Bender et al. [11] analyzed the performance of ***broom filters*** [12] on queries that obey Zipfian distributions. Lee et al. [63] proposed telescoping filters to address the skewed query distribution problem. Mitzenmacher et al. [68] proposed adaptive cuckoo filters to solve the skewed-query-distribution problem. Recently, Wen et al. [96] proposed a practical implementation of the adaptive quotient filter that is monotonically adaptive and offers strong performance guarantees.

2.4 Maplets

Across applications ranging from genomics [67] to storage engines [7, 21, 28, 38, 82], it is desirable to associate a small value to each key in a filter. Such key-value filters are referred to as maplets [28]. A query to an existing key in a maplet must return the value associated with this target key and potentially a few additional arbitrary values. Furthermore, most maplets return arbitrary value/s when queried for a non-existing key. It is the responsibility of the application to deal with the noise in the query result (e.g., by doing extra work to check which is the real value). Existing maplets can be classified with respect to the average number of values returned for positive vs. negative queries, i.e., the expected positive result size (PRS) and the expected negative result size (NRS).

Bloomier filter [22] can be thought of as a XOR filter [52] that stores fixed-width values rather than fingerprints in the hash table slots. While it supports updates to values associated with existing keys, it does not support insertions of new data entries. Its PRS and NRS are both 1.

The more recent cuckoo and quotient filters can be easily transformed into maplets by extending each slot in the hash table to also store a value alongside the key's fingerprint [28, 38]. Such maplets have a PRS of $1+\varepsilon$ and a NRS of ε. They support dynamic insertions and deletions, and they can be made expandable as shown in Section 2.2. Quotient filter-based maplets are extensively used in genomic applications for building inverted string indexes [3–5, 73].

Recent storage engines have pioneered the design of dynamic maplets with PRS of 1 to bound tail latency. They do so by detecting and eliminating fingerprint collisions on the insertion path. SlimDB employs an auxiliary dictionary that stores the full keys of insertions that collided with existing fingerprints [82]. In contrast, the Pliops Delta Hash table is a maplet that resolves fingerprint collisions by succinctly encoding the indexes of the first differentiating bits among all fingerprints that coincide within a hash table bucket [39].

Some maplets require storing multiple values or even variable-sized values per key [38, 73]. Quotient filters are adept at this because their use of linear probing allows storing variable-sized entries across adjacent slots, and as many values as required within a run.

2.5 Range filters

Range filtering is a generalization of the classic point filtering problem to allow an interval as the filter input. It is also known as the ε-approximate range emptiness problem [51]: given a set S, how to determine the "emptiness" of an interval $I=[a,b]$ (i.e., $[a,b]\cap S=\emptyset$?) with a false positive probability $<\varepsilon$? Goswami et al. [51] gave a worst-case space lower bound for any data structure (i.e., a range filter) capable of answering the range emptiness queries: $\Omega(n\lg(L/\varepsilon))-O(n)$ bits, where n is the number of points in set S, and L is the maximum interval length allowed. Currently, range filters are mainly used in LSM-tree-based storage engines (e.g., RocksDB) to reduce unnecessary I/Os for range queries (e.g., `SELECT * FROM T WHERE T.year BETWEEN 2020 AND 2024`).

The Adaptive Range Filter (ARF) [1] introduced in Hekaton [43] is considered the first attempt to build a practical range filter. ARF encodes the entire integer key space using a binary tree. ARF only works well with a stable or repeating integer workloads. However, the high training overhead prevents ARF from solving the general range filtering problem efficiently.

The first general-purpose range filter is the Succinct Range Filter (SuRF) [102, 103]. SuRF stores the unique prefixes of the keys in the set using a trie structure. The trie is encoded with space close to the information-theoretic lower bound. SuRF then uses additional bits from each key's (hashed) suffix to strike trade-offs between space and false-positive rate (FPR). Although SuRF pioneers the research of practical range filters and achieves impressive speedup on RocksDB, it has major limitations. First, SuRF lacks a theoretical guarantee for the space and range query FPR. An adversarial workload (e.g., each pair of keys produces a unique long prefix) can destroy SuRF's space efficiency. Second, SuRF is a static data structure, which limits its use in applications that require updating the filter frequently.

Proteus [60] improves SuRF's design by combining the Fast Succinct Trie (FST) used in SuRF with a prefix Bloom filter. The FST in

Proteus only stores each key prefix up to a uniform length $l1$, while the Bloom filter records the existence of all the key prefixes of length $l2 > l1$. Proteus requires query samples to determine the parameters (i.e., l1 and l2) to achieve an optimal false positive rate under a memory budget. Therefore, it must maintain a query cache and rebuild itself upon a workload shift to provide robust performance.

Rosetta [66] addresses SuRF's first limitation by arranging a hierarchy of Bloom filters forming a segment tree conceptually. The Bloom filter at each level records all the key prefixes of the corresponding length. A querying range is decomposed into dyadic intervals, where the Bloom filter covering each interval is probed (recursively down the subtree if necessary) to determine the emptiness of that dyadic interval. Rosetta provides a theoretical guarantee for space and is robust against the above adversarial workload for point and short-range queries. As the querying range gets larger, Rosetta's FPR grows rapidly and eventually provides no filtering. Another major limitation of Rosetta is its high CPU overhead. The benefit of trading CPU efficiency for better FPR may diminish as modern SSDs get faster [38].

Besides Rosetta, Bloom-filter-based solutions also include REncoder [95] and bloomRF [69]. REncoder reduces Rosetta's computational overhead by leveraging the bit locality within the Bloom filters. The Sparse Numerical Array-Based Range Filters (SNARF) [92] adopts the "learned" approach to solve the range filtering problem. SNARF models the keys' cumulative distribution function (CDF) using linear splines and then maps the keys according to the estimated CDF to a sparse bit array. A range query returns false if its mapped interval in the bit arrays has no bit set. The length of the bit array controls the false positive rate of the filter.

The most recent Grafite [31] proposes a practical implementation of the algorithm introduced in Goswami et al. [51]. It hashes the keys using a hash function that preserves the locality of the keys while having a small collision probability. The hash codes are sorted and encoded using the classic Elias-Fano code. Compared to the prior solutions, Grafite exhibits a more robust performance under workloads with high correlations between keys and queries. Compared to SuRF, Grafite fixes SuRF's vulnerability to adversarial workloads but sacrifices the ability to handle non-integer keys. Grafite is also a static structure. A dynamic and expandable range filter is still an unsolved problem.

2.6 Counting filters

Counting filters generalize traditional point filters to multisets. They support INSERT, QUERY, and DELETE operations, except a query for an item x returns the number of times that x has been inserted. A counting filter may have an error rate δ. Queries return true counts with probability at least $1-\delta$. Whenever a query returns an incorrect count, it must always be greater than the true count.

The counting Bloom filter (CBF) is an early example of a counting filter. The CBF was originally described as using fixed-sized counters, which means that counters could saturate. This could cause the counting Bloom filter to undercount. Once a counter is saturated, it can never be decremented by any future delete, and so after many deletes, a counting Bloom filter may no longer meet its error limit of δ. Both issues can be fixed by rebuilding the entire data structure with larger counters whenever one of the counters saturates.

The d-left Bloom filter [16] offers the same functionality as a counting Bloom filter and uses less space, generally saving a factor of two or more. It uses d-left hashing and gives better data locality. However, it is not resizable and the false-positive rate depends on the block size used in building the data structure. The spectral Bloom filter [27] is another variant of the counting Bloom filter that is designed to support skewed input distributions space-efficiently. The spectral Bloom filter saves space by using variable-sized counters. It offers significant space savings, compared to a plain counting Bloom filter, for skewed input distributions. However, like other Bloom filter variants, the spectral Bloom filter has poor cache locality and cannot be resized.

The counting quotient filter (CQF) [75] is a space-efficient and scalable counting filter that offers good performance on arbitrary input distributions, including highly skewed distributions. The CQF is based on the quotient filter and uses a variable-length encoding for counters to achieve asymptotically optimal space for encoding counters. The variable-length encoding also enables the CQF to efficiently handle highly-skewed distributions often seen in real-world datasets.

2.7 Static filters (XOR/Ribbon)

In some applications of filters, such as log-structured merge trees (LSMs) [20, 32], where filters are used to speed up membership queries (but not successor queries), the set for which the filter is built is known ahead of time. Since static filters can, in theory, be smaller than dynamic filters, and since RAM is usually not big enough to fit all the filters needed to achieve performance in an LSM, the question becomes: are there static filters that achieve nearly $n\log\varepsilon^{-1}$ bits of space, while being reasonably fast to build and very fast to query?

Researchers have developed so-called *algebraic* filters that compute a representation of the set to be filtered. The XOR filter was the first of such filters, and it achieves $1.22n\log\varepsilon^{-1}$ bits. The XOR+ filter achieves $1.08n\log\varepsilon^{-1}+0.5n$ bits, which is better for realistic ε. The ribbon filter improves this bound to $1.005n\log\varepsilon^{-1}+0.008n$ bits, under certain assumptions and has better construction and query times. The ribbon filter is available in the LSM tree-based RocksDB. It is potentially useful when space is at an absolute premium, though its query times remain slower than the fast competing filters.

2.8 Exploiting the query distribution

A recent class of filters exploits knowledge of the query workload to improve the false positive rate and/or space. To operate, these filters require a sample of historical queries as input. They can then train a classifier to predict the likelihood of each potential key being queried and the probability of its existence in the dataset [61, 79]. Such a classifier can then be used to learn to predict a positive outcome for frequently accessed positive keys and thereby avoid having to insert them into a regular filter to save space. In contrast, stacked filters [42] exploit knowledge of frequently queried non-existing keys to insert them into a hierarchy of additional filters and thereby exponentially decrease the false positive rate when querying for them.

3 APPLICATIONS

3.1 Storage engines & databases

A storage engine is the component of a database system that lays out data on a storage device, rendering it with structure. Most modern storage engines organize data loosely in storage to optimize for

data ingestion. At the same time, they employ in-memory filters or maplets to facilitate queries. In particular, LSM-tree [70] is a write-optimized data structure that is now the core of many storage engines and KV-stores (e.g., RocksDB, Cassandra, HBase, SplinterDB, etc). It works by flushing incoming data as small sorted files to storage and gradually sort-merging them. Typically, there is a Bloom filter in memory for each file to allow point queries to skip files that do not contain the target entry. As each file is immutable once created, however, any static filter is applicable in this context. Furthermore, the requirement of LSM-trees to support range queries kick-started the line of research on range filters, which we discuss in Section 2.5.

Filters enable deeper optimization opportunities for LSM-trees [85]. Monkey [32, 33] assigns smaller false positive rates to filters of smaller files to reduce query cost from $O(\varepsilon \cdot \lg N)$ to $O(\varepsilon)$ I/Os by causing the sum of false positive rates to converge with respect to the number of files in the system. The Dostoevsky [36] and LSM-Bush [37] systems further harness this technique to compact smaller files more lazily while setting their filters lower false positive rates. By so doing, they reduce write-amplification from $O(\lg N)$ to $O(\lg\lg N)$ without harming query cost or the memory footprint [36, 37].

Yet another strand of work replaces the multiple filters of an LSM-tree with a single maplet that maps each key in the system to the file in which corresponding data entry resides. SlimDB [82] pioneered this approach to eliminate false positives using an auxiliary dictionary that resolves all fingerprint collisions. Chucky [38] reduces the memory footprint for such maplets by compressing file identifiers using Huffman coding. SplinterDB [28] employs a maplet for a collection of files rather than a filter for each individual file to significantly reduce query CPU overheads. GRF [93] is a recent global range filter for LSM-tree utilizing SNARF [92].

Circular logs are another class of recent storage engines that optimize for write ingestion even more than LSM-trees. A circular log flushes all application insertions/updates/deletes as log records into an append-only file in storage, and it occasionally garbage-collects this log to remove obsolete entries [7, 21, 39]. To efficiently find entries in this log, there is a maplet in memory to map each entry in the log. It is crucial for these maplets to support updates, deletes, and expansion to reflect modifications and additions to the underlying data. It is also crucial for these maplets to exhibit high performance and low false positive rates. Interestingly, no system that we are aware of uses maplets that meet these requirements, and so we expect recent and ongoing research on maplets to significantly impact this area in the coming years.

Filter data structures are also widely used to process selective equality joins. A common approach is to build a filter over qualified join keys from the smaller table [62]. When the larger table is scanned, we can check its join keys against this filter to preemptively discard rows with non-matching join keys in the smaller table. This helps reduce the number and sizes of join partitions to improve both CPU utilization and I/Os.

3.2 Computational biology

Filters are extensively used to represent large genomic data in the form of k-mers (length-k substrings). Representing the genomic data as k-mers enables applications to efficiently perform sequence-level searches and genomic assembly over the sequences.

Solomon and Kingsford [88] first introduced the experiment discovery problem where a search for a query set q of k-mers must return all the experiments with a fraction Θ of the set q is present. They introduced the sequence Bloom tree (SBT) [55, 88], a binary tree of Bloom filters. The leaves of the SBT represent individual sequencing experiments, and the associated Bloom filter holds the k-mers present in this experiment. The Bloom filter of an interior node n represents the (approximate) set of k-mers present in the leaves of the subtree rooted at n. Given the false positives in the Bloom filters the SBT index also has false positives in the final results.

Mantis [3–5, 73] takes an inverted-index approach to support fast and efficient sequence-level searches. Mantis proved to be smaller, faster, and exact compared to the SBT which is an approximate index. Subsequent versions Mantis improved upon the scalability and indexed up to 40K experiments from the SRA comprising of more than 100TB of sequencing data. Mantis uses the counting quotient filter [75] as a maplet to map k-mers to the collection of experiments in which the k-mer appears. Each k-mer is associated with a unique bit vector of length equal to the number of experiments. Additionally, the quotient filter allows for an exact mapping by employing fingerprints that match the original key size.

Bloom and quotient filters have been used to compactly represent de Bruijn graphs. In a de Bruijn graph, each node is a k-length subsequence from the underlying biological samples, and two nodes are connected via an edge if they share a $(k-1)$-length subsequence. Pell et al. [78] introduced a probabilistic representation of the de Bruijn graph using a Bloom filter to represent the underlying set of k-mers. Though this representation admits false positives in the edge set, they observe that this has little effect on the large-scale structure of the graph until the false positive rate becomes very high (i.e., ≥ 0.15). Building upon this probabilistic representation, Chikhi and Rizk [25] introduce an exact de Bruijn graph representation that couples a Bloom-filter-based approximate de Bruijn graph with an exact table storing critical false positive edges. Chikhi and Rizk's de Bruijn graph representation exploits the fact that there are very few edges connecting true-positive k-mers to false-positive k-mers in the Bloom filter representation of the k-mers set. Such edges are called *critical false positives*. They observe that eliminating these critical false positives is sufficient to provide an exact (navigational) representation.

Subsequently, Salikhov et al. [84] improved the memory requirements even further by replacing the exact table with a cascading Bloom filter. The cascading Bloom filter stores an approximate set using a combination of an approximate (i.e., Bloom filter-based) representation of the set and a smaller table to record the relevant false positives. This construction can be applied recursively to substantially reduce the amount of memory required to represent the original set.

deBGR [76] generalizes the filter-based de Bruijn graph representation to the weighted de Bruijn graph. deBGR representation is based upon the CQF [75] which itself provides an approximate representation of the weighted de Bruijn graph. Observing certain abundance-related invariants that hold in an exact weighted de Bruijn graph, Pandey et al. devised an algorithm that uses this approximate data representation to iteratively self-correct approximation errors in the structure. A major benefit of the deBGR approach is that weighted de Bruijn graph construction requires considerably less memory than what is required by other tools, and the *working memory* is close to the final memory required by the structure. Hence, the whole

process can be done in a space-efficient manner. The deBGR representation allows for the assembly of larger and more complicated transcriptomes on smaller and less expensive computers (machines provisioned for large transcriptome assembly often have upwards of 1TB of RAM). Moreover, given the capabilities enabled by the CQF the de Bruijn graph representation is (at least partially) dynamic, allowing k-mers to be removed from deBGR and the resulting data structure efficiently updated. This capability is important to enable simplifications of the de Bruijn graph that are typically carried out prior to assembly (e.g., tip removal and bubble popping).

3.3 Networking and cybersecurity

Filters have been extensively used in networking and cybersecurity applications [19]. 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 [90]. Users may also be actively tricked into downloading harmful material or sharing sensitive information like passwords or credit card numbers. Since URLs are long [56] and abundant [86], an effective way for a router to block malicious URLs is to store them as the YES list of a filter [64].

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. [23] introduced the Bloomier filter which solves the YES/NO list problem. Li et al. [64] present the Seesaw Counting Filter (SSCF), which implements a YES/NO list filter specifically for the malicious URL blocking problem. Reviriego et al. [83] 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. Recently, Wen et al. [96] show that the YES/NO list problem can be efficiently solved by employing adpative filters in both the static and dynamic case.

4 INTENDED AUDIENCE

The filter tutorial is intended for both core data structure and database researchers as well as the researchers building production applications in industry. Researchers working on core indexing data structures and database internals will learn the latest advances in the theory and practice of modern filters. Application developers and industry experts will learn about the latest filter API and performance on modern hardware. They will further learn about case studies from real-world use cases where feature-rich filters are employed and the kind of performance impacts they have achieved. We will also discuss use cases from computational biology and databases where the applications are redesigned around the extended API and performance of modern filters to achieve higher performance and simplicity. The proposed tutorial will enable database researchers at SIGMOD to learn about all the advancements in filters in one place.

5 PRESENTER BIOGRAPHIES

Prashant Pandey. Prashant Pandey is an assistant professor in the Kahlert school of computing at the University of Utah. Previously, he did postdocs at University of California Berkeley and Carnegie Mellon University. He obtained his Ph.D. in Computer Science at Stony Brook University in December 2018. He has worked extensively on advancing the theory and practice of data structures and using them to build high-performance and large-scale databases and file systems, computational biology tools, and anomaly detection tools for cybersecurity. He won the NSF Career Award in 2024, IEEE CS TCHPC Early Career Researchers Award in 2023, Catacosinos Fellowship in 2018, and a Best paper award at FAST 2016.

Martín Farach-Colton. Martín Farach-Colton is the Lenoard J. Shustek Chair Professor of Computer Science and Chair of the Department of Computer Science and Engineering at the Tandon School of Engineering, New York University. He is a Fellow of the ACM, IEEE, and SIAM, and a member of the Argentine National Academy of Science. His research interests span the theory and practice of data structures in storage systems. His papers have won best paper awards at the FAST and ASPLOS conferences. He founded Toketek, a performance database company that was acquired in 2015.

Niv Dayan. Niv Dayan is an Assistant Professor of Computer Science at the University of Toronto. He is broadly interested in data structures for storage applications. He was a postdoc at both Harvard and Copenhagen University, and his PhD is from the IT University of Copenhagen. He is a recipient of the "Best of SIGMOD" award in 2017.

Huanchen Zhang. Huanchen Zhang is an Assistant Professor in the IIIS (Yao Class) at Tsinghua University. His research interest is in database management systems with particular interests in indexing, data compression, and cloud databases. He received his Ph.D. from the Computer Science Department at Carnegie Mellon University. Before joining Tsinghua, he worked at Snowflake as a Postdoctoral Research Fellow. He is the recipient of the SIGMOD Jim Gray Dissertation Award (2021) and the SIGMOD Best Paper Award (2018).

6 FILTER TUTORIAL FROM SPAA 2023

One of the authors of the current proposal (Prashant Pandey) organized a workshop on filter data structures at SPAA 2023 (Symposium on Parallelism in Algorithms and Architectures). All the talks were recorded and are available at: https://prashantpandey.github.io/workshop/. The main goals of the workshop was to bring together researchers at the forefront of data structure research and help uncover the open research questions.

The format of the workshop included seven talks from experts in the field working on core filter data structures and also researchers in application domains. The talks included recent advancements in the theory and practice of filters such as static filters, infinitely-resizeable filters, maplets, adaptive filters, GPU filters, and applications of filters in computational biology.

The insights gained from the previous workshop on filters have motivated the design and proposal of the current tutorial on filters. The current tutorial involves four leading experts on filter data structures to cover the depth and breadth of research on filter data structures. The current team will present an overview of filters along with the recent advancements in the theory of filters. We will then cover the major filter variants, such as range, counting, and adaptive filters. We will also discuss how to redesign application using the modern filter API.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive range filters for cold data: Avoiding trips to Siberia. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1714–1725.

[2] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom Filters. *Inform. Process. Lett.* (2007).

[3] Fatemeh Almodaresi, Jamshed Khan, Sergey Madaminov, Michael Ferdman, Rob Johnson, Prashant Pandey, and Rob Patro. 2022. An incrementally updatable and scalable system for large-scale sequence search using the Bentley–Saxe transformation. *Bioinformatics* 38, 12 (March 2022), 3155–3163. https://doi.org/10.1093/bioinformatics/btac142

[4] Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. 2019. An Efficient, Scalable and Exact Representation of High-Dimensional Color Information Enabled via de Bruijn Graph Search. In *International Conference on Research in Computational Molecular Biology (RECOMB)*. Springer, 1–18.

[5] Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. 2020. An Efficient, Scalable, and Exact Representation of High-Dimensional Color Information Enabled Using de Bruijn Graph Search. *Journal of Computational Biology* 27, 4 (2020), 485–499.

[6] 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.

[7] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. *SOSP* (2009).

[8] Jim Apple. 2022. Stretching your data with taffy filters. *Software: Practice and Experience* (2022).

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. 53–64.

[10] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. 2021. External-memory Dictionaries in the Affine and PDAM Models. *ACM Trans. Parallel Comput.* 8, 3 (2021), 15:1–15:20. https://doi.org/10.1145/3470635

[11] 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)*.

[12] 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.

[13] 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).

[14] Ioana O Bercea and Guy Even. 2020. Fully-Dynamic Space-Efficient Dictionaries and Filters with Constant Number of Memory Accesses. *SWAT*.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

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

[22] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier filter: an efficient data structure for static support lookup tables. In *Symposium on Discrete Algorithms*.

[23] 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.

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

[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), 22.

[26] 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.

[27] Saar Cohen and Yossi Matias. 2003. Spectral Bloom filters. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 241–252.

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

[29] Alexander Conway, Martin Farach-Colton, and Philip Shilane. 2018. Optimal Hashing in External Memory. In *ICALP (LIPIcs, Vol. 107)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 39:1–39:14.

[30] 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.

[31] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2023. Grafite: Taming Adversarial Queries with Optimal Range Filters. *arXiv preprint arXiv:2311.15380* (2023).

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

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

[34] Niv Dayan, Ioana Bercea, and Rasmus Pagh. 2024. Aleph Filter: To Infinity in Constant Time. *arXiv preprint arXiv:2404.04703* (2024).

[35] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proceedings of the ACM on Management of Data* (2023).

[36] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. *SIGMOD* (2018).

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

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

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

[40] 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.

[41] 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)*.

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

[43] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.

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

[45] 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

[46] 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

[47] 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.

[48] Tomer Even, Guy Even, and Adam Morrison. 2022. Prefix Filter: Practically and Theoretically Better Than Bloom. *Proc. VLDB Endow.* 15, 7 (2022), 1311–1323. https://doi.org/10.14778/3523210.3523211

[49] 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.

[50] Martin Farach-Colton, Rohan J. Fernandes, and Miguel A. Mosteiro. 2009. Bootstrapping a hop-optimal network in the weak sensor model. *ACM Trans. Algorithms* 5, 4 (2009), 37:1–37:30.

[51] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. 2014. Approximate range emptiness in constant time and optimal space. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 769–775.

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

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

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

[55] Robert S. Harris and Paul Medvedev. 2019. Improved representation of sequence bloom trees. *Bioinformatics* 36, 3 (Aug. 2019), 721–727. https://doi.org/10.1093/bioinformatics/btz662

[56] InternetLiveStats.com. 2022. *Google search statistics*. https://www.internetlivestats.com/google-search-statistics/

[57] 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.

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

[59] 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. *ACM Trans. Storage* 11, 4 (2015), 18:1–18:29. https://doi.org/10.1145/2798729

[60] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A self-designing range filter. In *Proceedings of the 2022 International Conference on Management of Data*. 1670–1684.

[61] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. *SIGMOD* (2018).

[62] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. 2019. Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput. In *VLDB*.

[63] 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

[64] 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

[65] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard TB Ma, Xueshan Luo, and Bangbang Ren. 2019. The Consistent Cuckoo Filter. In *INFOCOM*.

[66] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[67] Hunter McCoy, Steven Hofmeyr, Katherine Yelick, and Prashant Pandey. 2023. High-performance filters for gpus. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 160–173.

[68] 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

[69] Bernhard Mößner, Christian Riegger, Arthur Bernhardt, and Ilia Petrov. 2023. bloomRF: On performing range-queries in Bloom-Filters with piecewise-monotone hash functions and prefix hashing. In *Advances in database technology: Proceedings of the 26th International Conference on Extending database Technology (EDBT)*, Vol. 26. 131–143.

[70] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* (1996).

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

[72] Rasmus Pagh, Gil Segev, and Udi Wieder. 2013. How to Approximate a Set Without Knowing its Size in Advance. In *FOCS*.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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*. ACM, 1386–1399. https://doi.org/10.1145/3448016.3452841

[78] 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.

[79] Jack Rae, Sergey Bartunov, and Timothy Lillicrap. 2019. Meta-learning neural bloom filters. In *International Conference on Machine Learning*.

[80] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[81] 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).

[82] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *PVLDB* (2017).

[83] Pedro Reviriego, Alfonso Sánchez-Macián, Stefan Walzer, and Peter C. Dillinger. 2021. Approximate Membership Query Filters with a False Positive Free Set.

[84] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. 2013. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. In *Algorithms in Bioinformatics*. Springer, 364–376.

[85] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM Design Space and its Read Optimizations. In *ICDE*.

[86] Securelist.com. 2022. . https://securelist.com/kaspersky-security-bulletin-2021-statistics/105205/

[87] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *ASPLOS*.

[88] Brad Solomon and Carl Kingsford. 2016. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology* 34, 3 (2016), 300.

[89] 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.

[90] 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.

[91] 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

[92] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: a learning-enhanced range filter. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1632–1644.

[93] Hengrui Wang, Tw Guo, Junzhao Yang, and Zhang Huanchen. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. In *SIGMOD*.

[94] 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.

[95] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. Rencoder: A space-time efficient range filter with local encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2036–2049.

[96] Richard Wen, Hunter McCoy, David Tench, Guido Tagliavini, Michael Bender, Alex Conway, Martin Farach-Colton, Rob Johnson, and Prashant Pandey. 2025. Adaptive Quotient Filters. In *Proceedings of the 2025 International Conference on Management of Data*. ACM.

[97] Yuhan Wu, Jintao He, Shen Yan, Jianyu Wu, Tong Yang, Olivier Ruas, Gong Zhang, and Bin Cui. 2021. Elastic Bloom Filter: Deletable and Expandable Filter Using Elastic Fingerprints. *IEEE Trans Comput* (2021).

[98] Kun Xie, Yinghua Min, Dafang Zhang, Jigang Wen, and Gaogang Xie. 2007. A Scalable Bloom Filter for Membership Queries. In *GLOBECOM*.

[99] Minghao Xie, Quan Chen, Tao Wang, Feng Wang, Yongchao Tao, and Lianglun Cheng. 2022. Towards Capacity-Adjustable and Scalable Quotient Filter Design for Packet Classification in Software-Defined Networks. *IEEE Open Journal of the Computer Society* (2022).

[100] 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 *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, Angela Demke Brown and Florentina I. Popovici (Eds.). USENIX Association, 1–14. https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan

[101] 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. 2017. Writes Wrought Right, and Other Adventures in File System Optimization. *ACM Trans. Storage* 13, 1 (2017), 3:1–3:26. https://doi.org/10.1145/3032969

[102] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.

[103] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct range filters. *ACM Transactions on Database Systems (TODS)* 45, 2 (2020), 1–31.

[104] 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.