Diva: Dynamic Range Filter for Var-Length Keys and Queries

Navid Eslami
navideslami@cs.toronto.edu
University of Toronto
Toronto, Canada

Ioana O. Bercea
bercea@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

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

ABSTRACT

Range filters are compact probabilistic data structures that answer approximate range emptiness queries. They are used in many domains, e.g., in key-value stores, to quickly rule out the existence of keys in a given query range and avoid having to search for them in storage. However, all existing range filters exhibit at least one of the following shortcomings: (1) they do not provide robust false positive rate and performance guarantees, (2) they do not support variable-length keys and query ranges, and (3) they do not allow dynamic operations such as insertions, deletions, or expansions.

We introduce Diva, the first range filter to address all the above challenges simultaneously. Diva learns the dataset's distribution by sampling keys and storing them in a cache-efficient trie. It compresses the keys in-between samples by removing their longest common prefix and truncating their suffixes while leaving enough bits in the middle (i.e., an infix) to allow differentiating between the keys in the sorted order. It stores infixes in constant time dynamic data blocks, which it splits to handle insertions and expansions. It processes a range query by traversing the trie and checking for the inclusion of infixes in the target query range.

We show, theoretically and empirically, that Diva achieves a false positive rate on par with the state of the art on real-world datasets while supporting dynamicity and variable-length queries and keys.

PVLDB Reference Format:
Navid Eslami, Ioana O. Bercea, and Niv Dayan. Diva: Dynamic Range Filter for Var-Length Keys and Queries. PVLDB, 18(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/n3slami/Diva.git.

1 INTRODUCTION

What is a Filter? A filter is a memory-efficient probabilistic data structure that answers whether a query key exists in a given set. Its compactness allows it to fit in a higher level of the memory hierarchy than the set it represents, making it fast to query. A filter never returns a false negative but may return a false positive with a probability called the *False Positive Rate (FPR)* that depends on its memory footprint. Due to these qualities, filters are used in many applications to avoid redundant disk reads [21] and network hops [10] when a query happens to target a nonexistent key.

Range Filters and Applications. Traditional filters answer membership queries for a single key. In contrast, a range filter accepts two keys as the end-points of a range and determines if at least one key in the set is in-between them. As with a traditional filter, a range filter does not return false negatives but may return false positives. Range filters are used in many domains, especially in the context of LSM-Trees [38], to avoid accessing files that do not contain any keys within a range predicate, significantly boosting performance [2, 12, 23, 29, 33, 37, 46, 51]. They also aid in preventing redundant I/Os to SQL tables [2] and B-Tree indices [23], and they have potential applications in social web analytics [16] and replication in distributed key-value stores [44].

Range Filtering Goals. The ideal general-purpose range filter must simultaneously support (G1) the lowest possible FPR under a stringent memory budget, (G2) range queries of any length, (G3) variable-length keys, (G4) dynamic modification operations such as insertions, deletions, expansions, and contractions, and the best possible (G5) query and (G6) construction performance.

Design Contentions. Every existing range filter only fulfills a subset of these goals [2, 12, 17, 23, 24, 29, 33, 37, 46, 47, 51]. In fact, almost none of the existing ones attain (G3) or (G4) [12, 17, 24, 29, 33, 37, 46, 47]. Moreover, Goswami et. al. have proven an information-theoretic lower bound on the memory footprint of range filters [27], stating that it is impossible for any range filter to achieve (G1) and (G2) at the same time. However, this lower bound only holds for worst-case datasets, meaning that it may still be possible to achieve (G1) and (G2) for "common" datasets. As such, we pose the following research question: *Is it possible to design a range filter that simultaneously fulfills all six goals for common datasets?* This paper presents an affirmative answer.

Core Contribution: Diva. We introduce Diva, the first range filter to support dynamic operations, variable-length queries and keys, and high performance, all at the same time. Diva learns the dataset's distribution by sampling keys and storing them in a cache-efficient trie. For all keys in-between two samples of the trie, Diva removes the longest common prefix. It also truncates their suffixes while keeping enough bits in the middle of each key (i.e., an infix) to differentiate them in most cases, thus achieving (G1). At the same time, the trie separates dense and sparse regions of the key space. This allows for handling short range queries over densely populated regions and long range queries over sparse regions, thus meeting (G2). By discretizing all keys into fixed-length infixes without the use of hashing, Diva achieves (G3). This discretization also allows for storing the infixes of adjacent groups of keys within a constant time data structure called an Infix Store, fulfilling (G5). As Diva derives infixes without hashing and stores them in the original sorted order of the keys, it does a single efficient sequential pass over the dataset during construction, making it the fastest range

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Table 1: Definitions of terms and symbols.

Symbol	Definition
$q = [q_l, q_r]$	An inclusive query range.
ϵ	Target FPR, i.e., probability of a false positive.
L	Average key length.
$(a)_2$	Binary value represented by a.
T	Number of keys between two samples.
m^i_{shared}	Length of the longest common prefix of the i-th and $(i+1)$-th samples, in bits.
m_{infix}	Length of the infixes.
$m^i_{\text{redundant}}$	Number of redundant bits corresponding to the i-th and $(i+1)$-th samples.
α	Load factor of the Infix Stores.
x_q	Quotient of the infix x.
x_r	Remainder of the infix x.

filter to construct. It therefore attains (G6). Diva handles dynamic updates by splitting Infix Stores, thereby meeting (G4).

Additional Contributions.

- We quantify the necessary conditions on the data distribution for Diva to achieve a low FPR and memory footprint while supporting variable-length range queries.
- We mathematically prove Diva's low FPR and high performance properties.
- We empirically evaluate Diva against other range filters in static and dynamic settings. We also conduct end-to-end experiments on top of WiredTiger [36], a popular B-Tree-based key-value store.

2 PROBLEM ANALYSIS

This section shows that no current range filter satisfies all of the six goals outlined in Section 1. In-depth summaries of these filters are presented in [23, 42].

Range Filtering Definitions. A range filter represents a set S of keys coming from a universe of size u. Given a range query of the form $q = [q_l, q_r]$, the filter checks if the range is empty by answering whether $q \cap S = \emptyset$ with an FPR of at most ϵ, where $0 < \epsilon < 1$. The top three rows of Table 1 summarize terms describing the range filtering problem throughout the paper.

Memory Lower Bound. A range filter is *Robust* if it guarantees an FPR of at most ϵ for any dataset. A non-robust range filter often drops all information about the keys' lower-order bits. This can lead to a high FPR when range queries predicate over these lower-order bits. It is known that any robust range filter supporting queries of length up to R must use at least $\log_2 \frac{R}{\epsilon} - O(1)$ *Bits per Key (BPK)* [27], meaning that answering longer range queries requires more memory. Intuitively, this is because a robust range filter supporting longer queries must carry more information about which areas of the key space are empty.

Robustly Achieving (G1), (G2), and (G3) is Impossible. In most applications, filters are allotted a stringent memory budget of 8-16 BPK to fit in memory (with a higher budget, one may as well store the full keys). If we rearrange the above lower bound in terms of range query length R and plug in 16 BPK as the memory budget and $\epsilon = 0.01$ as the target FPR, we find that a robust range filter can only answer range queries of length at most 512. Such short query lengths limit the applicability of the filter.

Table 2: Diva is the first range filter to simultaneously support variable-length queries and keys, as well as dynamicity.

Filter	Robust FPR (G1)	Semi-Robust FPR (G1)	Var.-Len. Queries (G2)	Var.-Len. Keys (G3)	Dynamic (G4)
`SuRF`			✓	✓	
`Rosetta`	✓	✓			
`REncoder`					
`bloomRF`					
`Proteus`					
`SNARF`		✓	✓		
`Oasis+`		✓	✓		
`Grafite`	✓	✓			
`Memento`	✓	✓			✓
`Diva`		✓	✓	✓	✓

The lower bound also implies that a robust range filter cannot support variable-length keys. With variable-length keys, there can be infinitely many possible keys within a range, meaning $R = \infty$. Plugging $R = \infty$ into the lower bound, we find that infinite memory is needed to support variable-length keys. In sum, it is impossible to attain all of (G1), (G2), and (G3) with a robust range filter.

Indeed, none of the existing robust range filters [17, 23, 33] support variable-length queries (G2) or keys (G3), as they assume range query lengths bounded by R. Rosetta [33] stores prefixes of keys based on length in a hierarchy of Bloom filters of depth $\lceil \log_2 R \rceil$, with lower levels storing longer prefixes. Grafite [17] encodes N integer keys in a bitmap of size $\frac{NR}{\epsilon}$ by mapping each to a bit using a locality-preserving hash function and setting it to 1. Memento filter [23] splits each key into a prefix and a $\lceil \log_2 R \rceil$-bit suffix, and it hashes each prefix to compute a fingerprint key to a compact hash table and stores its corresponding suffix in it. In general, issuing longer range queries forces each of these filters to issue more Bloom filter probes (in Rosetta), check more bits (in Grafite), or check more fingerprints (in Memento filter). If the range query length exceeds the intended value of R used to construct these filters, the FPR grows beyond ϵ and approaches 1.

Semi-Robust FPR Guarantee. This paper observes that, despite the aforementioned impossibility, one can achieve (G1), (G2), and (G3) by providing a semi-robust FPR guarantee: *Any query must be answered with an FPR of at most ϵ when the dataset comes from a "well-behaved distribution."* Intuitively, a well-behaved distribution is one for which the cumulative distribution function is smooth. This property is commonly satisfied in practice since input datasets typically follow well-known, smooth distributions (e.g., Uniform, Normal, Zipfian, Power Law, Poisson). We formally define well-behaved distributions and show that Diva provides the above FPR guarantee for them in Section 4.

Interestingly, the learning-augmented filters [8, 31] SNARF [46] and Oasis+ [12] fulfill (G1) and (G2) with the same semi-robust FPR guarantee. They do so by fitting a linear spline model to the keys' CDF, using the model to map each key to a bit in a large bitmap, and setting the bit to 1. These filters handle variable-length queries by leveraging the monotonicity of this mapping and searching the bits corresponding to the query for a 1. There is no formalism or proof of the above FPR guarantee in their respective publications, though one could adapt Diva's proof in Section 4 to them.

Table 3: A comparison of the existing range filters' time and space complexities assuming an FPR of ϵ, range query length R, and N keys of average length L. For SuRF and Proteus, z refers to the number of internal nodes in the trie, while m denotes the length of the fingerprints stored at the leaves. For REncoder, k is its number of hash functions, which is roughly an $O(\log \frac{1}{\epsilon})$ value. For SNARF, B refers to the block size. For Memento and Diva, α refers to the load factor. The operation costs are measured as the expected number of cache misses. The construction analysis assumes that the filters are constructed on sorted keys. The two range query columns focus on empty and non-empty range queries, respectively. The filters annotated with * do not provide mathematical bounds on their memory consumption. Therefore, following [23], we give a conservative estimate of their memory footprint to enable a comparison.

Filter	Construction	Insert	Delete	Range Query (-)	Range Query (+)	BPK
`SuRF` [51]	$O(NL)$	-	-	$O(L)$	$O(L)$	$10 + \frac{10z}{N} + m + o(1)$
`Rosetta` [33]	$O(N \log \frac{R}{\epsilon})$	$O(\log \frac{R}{\epsilon})$	-	$O(\log R)$	$O(\log \frac{R}{\epsilon})$	$1.44 \cdot \log_2 \frac{R}{\epsilon}$
`REncoder` [24, 47] *	$O(Nk)$	k	-	k	k	$O(k + \log \frac{1}{\epsilon})$
`bloomRF` [37] *	$O(N(L - \log N))$	$O(L - \log N)$	-	$O(L - \log N)$	$O(L - \log N)$	$\approx 1.2 \cdot \log_2 \frac{R}{\epsilon}$
`Proteus` [29]	$O(N(L + \log \frac{1}{\epsilon}))$	-	-	$O(L)$	$O(L + \log \frac{1}{\epsilon})$	$\frac{10z}{N} + 1.44 \cdot \log_2 \frac{1}{\epsilon}$
`SNARF` [46]	$O(N)$	$O(\log N + B)$	$O(\log N + B)$	$O(\log N)$	$O(\log N)$	$2.4 + \log_2 \frac{1}{\epsilon}$
`Oasis+` [12]	$O(N)$	-	-	$O(\log N)$	$O(\log N)$	$\approx 2.4 + \log_2 \frac{1}{\epsilon}$
`Grafite` [17]	$O(N \log N)$	-	-	3	3	$2 + \log_2 \frac{R}{\epsilon} + o(1)$
`Memento` [23]	$O(N)$	$O(1)$	$O(1)$	$1 - 4$	$1 - 4$	$\frac{1}{\alpha}(3.125 + \log_2 \frac{R}{\epsilon})$
`Diva` [Static]	$O(N)$	-	-	$O(\log L)$	$O(\log L)$	$3.19 + \log_2 \frac{1}{\epsilon}$
`Diva` [Dynamic]	$O(N)$	$O(\log L)$	$O(\log L)$	$O(\log L)$	$O(\log L)$	$1.19 + \frac{1}{\alpha^2}(2 + \log_2 \frac{1}{\epsilon})$

Nevertheless, SNARF and Oasis+ do not meet (G3), (G4), and (G5). They assume fixed-length keys when learning the distribution, not attaining (G3). Although SNARF provides insertion and deletion APIs, they are prohibitively slow. This is because SNARF (and Oasis+) encode their bitmaps using the Elias-Fano scheme [22, 25] and partition them into blocks of B entries to save space and speed up decoding. SNARF's insertion and deletion APIs are slow since they rewrite an entire block and change its size, potentially increasing it to $O(N)$ over time. Oasis+ disallows dynamic operations since it prunes away empty regions of the key space to improve its FPR. Thus, both do not achieve (G4). They are also slow to query due to using binary search, thereby not meeting (G5).

Challenges of Attaining Dynamicity (G4). Range filters exhibit an intrinsic contention between support for fast dynamic operations and memory efficiency. Many existing range filters optimize for memory by operating as Bloom filters in their core, hashing keys into a bitmap and setting bits from 0s to 1s [24, 29, 33, 37, 47]. As with standard Bloom filters, such filters cannot support deletes (by resetting a bit back to 0) or expansions (by remapping the 1s to a larger bitmap) without introducing false negatives [23].

Other range filters utilize compact encoding schemes (e.g., succinct tries [51] or Elias-Fano [12, 17, 46]). Such formats are difficult to update, as they tightly pack their data to avoid storing pointers and offsets. This necessitates changing their entire representation in memory to make room for new insertions.

As of today, Memento filter [23] is the only filter that supports insertions and deletions in expected constant time. However, since it doubles its size to expand, it wastes as much as 50% of its capacity.

Diva overcomes this contention between memory efficiency and dynamicity by establishing a one-to-one mapping between keys and the metadata it stores to facilitate deletions, similarly to Memento filter and other expandable filters [5, 18, 19, 23]. Furthermore, It slightly overprovisions memory to absorb insertions and avoids wasting memory by expanding in small increments.

Contention between Supporting Variable-Length Queries (G2) and Query Speed (G5). It is an open question whether achieving fast operations while supporting variable-length range queries is possible. Grafite [17] and Memento filter [23] are the only filters that provide constant time queries. They do so by localizing partitions of size R (i.e., the maximum range query length) of the key space to the same memory region. It is unclear how to achieve a similar localization with variable-length range queries. As such, Grafite and Memento filter do not fulfill (G2). All other filters that attain (G1) and (G2), i.e., SNARF [46] and Oasis+ [12], use predecessor search to handle queries, which requires super-constant time [6, 7]. In practice, these filters use binary search to compute predecessors. Diva alleviates this contention by employing a y-Fast trie [48], leading to faster predecessor searches.

Construction Speed (G6). Existing range filters extensively use hashing [17, 23, 24, 29, 33, 37, 47], floating point operations [12, 46], or trie traversal [29, 51] during construction. Such operations lead to high CPU and cache miss costs. In contrast, Diva makes limited use of hashing, avoids floating point operations, and has good cache locality, attaining the best construction speed and (G6).

Other Range Filters. Proteus [29] employs a SuRF [51] instance with a tunable height and a Bloom filter storing key prefixes of a tunable length. It co-tunes these structures based on a sample of the query workload. While it performs well when the workload remains the same, it does not bound the FPR in the face of workload shifts. As it also inherits the limitations of SuRF and Bloom filters discussed earlier, it does not address any of the goals.

REncoder [24, 47] and bloomRF [37] are variants of Rosetta [33] that improve its speed by co-locating bits representing similar prefixes in a single bitmap. However, in doing so, they assign the same FPR to all levels in the hierarchy, breaking the robust FPR guarantee. Since they inherit Rosetta's other limitations, they do not attain the other goals either.

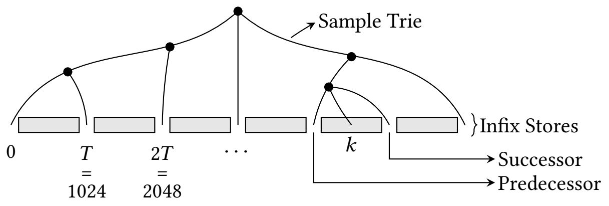

Figure 1: Diva learns the dataset distribution by storing every T-th key as a sample in a trie. It manages all keys between two adjacent samples in an Infix Store.

Summary. Table 2 summarizes the goals each range filter attains, and Table 3 outlines their operation costs and memory footprint. They demonstrate that no range filter achieves all of (G1), (G2), (G3), and (G4). They also highlight the question of how fast a range filter can be while meeting these goals, i.e., (G5) and (G6).

3 DIVA

We present Diva: the first range filter to simultaneously support (G1) a low FPR and memory consumption, (G2) arbitrarily sized range queries, (G3) variable-length keys, (G4) dynamic updates, deletes, and resizability, and high (G5) query and (G6) construction performance. Diva picks every T-th key in the ordered key set as samples and stores them in a cache-efficient trie to approximate the key distribution. It inserts all keys between two samples into a compact data structure called an *Infix Store*, as shown in Figure 1. For all keys in each Infix Store, Diva removes the longest common prefix. It also removes the variation in the keys' length by truncating a suffix from each while leaving enough bits to differentiate the keys and guarantee a given FPR. This FPR guarantee holds when the dataset follows a "well-behaved" distribution (as proven in Section 4). Figure 2 illustrates the truncation and mapping of keys into an Infix Store.

Diva processes a range query by searching for the query's endpoints in its trie. A range query intersecting at least one sample in the trie immediately returns a positive since that sample represents a key within the range. In contrast, a range query that falls between two adjacent samples searches the corresponding Infix Store and returns a positive if at least one overlapping key exists. For ease of exposition, we first describe a static version of Diva in this section. We generalize it to dynamic key sets in Section 3.5. Table 1 summarizes the terms used to describe Diva.

3.1 Sampling Keys and Deriving Infixes

The static variant of Diva requires the input to arrive in sorted order. This requirement is commonly satisfied in practice. For instance, many database tables and indices are structured as sorted column files [3, 4, 13, 26, 45], B-Trees [14, 43], or LSM-Trees [15, 20, 21]. If the input is unsorted, users must sort it before construction.

Diva samples every T-th key from the ordered key set and inserts them into its trie. The trie approximates the distribution of the keys. The reason is that adjacent samples lexicographically close to each other correspond to denser regions of the key space, while faraway samples correspond to sparser regions. Thus, the trie "learns" the data distribution. The smaller T is, the more accurate the approximate distribution becomes, yet the larger the trie and the memory footprint grow. We quantify the relationship between T and the approximation accuracy in Section 4. We have empirically found

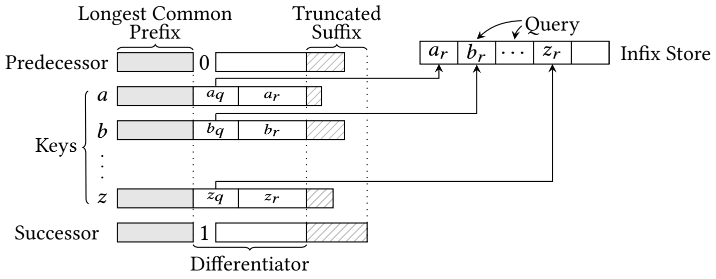

Figure 2: For all keys in-between a pair of adjacent samples in the trie, Diva removes the longest common prefix and truncates the longest possible suffix while still ensuring that there are enough bits left to distinguish between the keys.

that $T = 1024$ provides good all-round accuracy and performance with little memory overhead ($\approx 1\%$ of the total memory).

Removing the Longest Common Prefix. Consider a pair of consecutive trie samples. We refer to them as the *Predecessor* and *Successor* of the keys between them. Figure 1 shows an example of predecessor and successor for a key k. For all keys in-between such a pair of samples, Diva removes their longest common prefix since it can be inferred from the trie and is thus redundant.

Keys in the denser regions of the key space have longer common prefixes. Therefore, removing the longest common prefixes saves more memory in these regions. As we will see shortly, Diva exploits this and encodes the keys in such regions at a higher resolution by also storing their lower-order bits, losing less information. This enables accurately answering fine-grained queries over dense regions while supporting coarse-grained queries over sparse regions.

Given the i-th and $(i + 1)$-th samples as the predecessor and successor, we denote the length of their longest common prefix, in bits, by m^i_{shared}. Figure 3-A shows an example where the longest common prefix of the predecessor and successor is $m^i_{\text{shared}} = 4$ bits long. If the samples have different lengths, Diva treats the shorter one as having trailing zeros to match the length of the longer one.

Deriving Infixes. To further curb memory footprint, Diva also truncates suffixes for all keys between two samples. What remains of each key is called an *Infix* since it is a sequence of adjacent bits in the middle of the original key. All infixes in the filter have the same length, denoted by m_{infix}. Due to the removed common prefix, infixes represent less significant bits of the keys in dense regions of the key space and more significant bits in sparser ones. Truncating keys into fixed-length infixes discretizes the range between two samples while removing the variation in length of the keys. As we will see, Diva supports range queries over these infixes by correspondingly discretizing the query boundaries and checking for the inclusion of infixes between them. The fixed length of the infixes allows for quickly checking for inclusion using integer arithmetic.

The length of the infixes determines the filter's FPR. In particular, Diva guarantees an FPR of ϵ by using infixes of length $m_{\text{infix}} = \lceil \log_2 \frac{2T}{\epsilon} \rceil$ bits. As we will show in Section 3.2, Diva succinctly encodes each of these infixes using only $3 + \lceil \log_2 \frac{1}{\epsilon} \rceil$ bits, translating to one byte for an FPR of $\approx 3\%$. This FPR guarantee holds for any dataset sampled from a "well-behaved" distribution. This property is suitable for a practical range filter since datasets in practice often follow common distributions, such as the Normal distribution. We formally prove this FPR guarantee in Section 4.

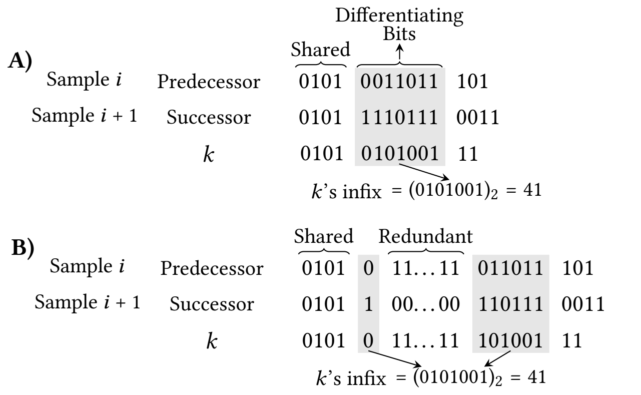
A)
Differentiating Bits
Shared
Sample i Predecessor 0101 0011011 101
Sample $i+1$ Successor 0101 1110111 0011
k 0101 0101001 11
k's infix = $(0101001)_2$ = 41
B)
Shared Redundant
Sample i Predecessor 0101 0 11…11 011011 101
Sample $i+1$ Successor 0101 1 00…00 110111 0011
k 0101 0 11…11 101001 11
k's infix = $(0101001)_2$ = 41

Figure 3: Diva derives key k's infix by removing the m^i_{shared} common prefix bits and the $m^i_{\text{redundant}}$ bits based on its predecessor and successor, and also truncating its suffix.

Figure 3-A shows an example with keys of varying lengths and $m_{\text{infix}} = 7$. As $m^i_{\text{shared}} = 4$, Diva derives k's infix as the 7 bits following its first 4 bits, i.e., the string $(0101001)_2$.

Distinguishing Infixes. There is a common scenario in practice where infixes derived with the method described above have bits that do not contribute to filtering. Intuitively, if the considered predecessor and successor are still close to each other after removing their longest common prefix, the discretized key space between them becomes smaller than desired. As a result, infixes of keys in-between can become less distinguishable, reducing filtering accuracy. This phenomenon occurs when many consecutive bits after the first differentiating bit in the predecessor and successor are all 1s and 0s, respectively. In particular, in well-behaved distributions such as Normal and Zipfian, there is at least one such bit on average due to the randomness in the samples after their longest common prefix. If not removed, these bits can increase the FPR by 2× compared to the case where infixes are differentiated well. The predecessor and successor in Figure 3-B represent an example.

Diva prevents this by identifying the longest matching sequence of 1s from the predecessor and 0s from the successor after their first differentiating bit. We denote the length of this sequence for the i-th and $(i+1)$-th samples by $m^i_{\text{redundant}}$. Diva removes the bits corresponding to this sequence from each key between the samples. These bits are redundant, as they do not carry useful information for comparing the keys between the i-th and $(i+1)$-th samples beyond what the first differentiating bit provides. Diva derives the infix of each key as its first differentiating bit concatenated with its first $m_{\text{infix}} - 1$ bits after its redundant bits. Intuitively, each of these bits doubles the size of the discretized space between two samples, allowing to better differentiate the infixes in-between. The following lemma formalizes this intuition, arguing that the removal of the redundant bits fully addresses the differentiation issue:

Lemma 3.1. *After removing the $m^i_{\text{redundant}}$ bits, there are at least $\frac{T}{\epsilon}$ and at most $\frac{2T}{\epsilon}$ different and valid values that infixes of keys between two samples can take.*

Diva is able to reconstruct the redundant bits removed from an infix. It does so by recomputing $m^i_{\text{redundant}}$ from its predecessor and successor in the trie. It adds $m^i_{\text{redundant}}$ bits between the infix's first and second bits, each equal to the negation of its first bit. This results in the original sequence of bits in the key since the redundant bits of an infix always equal the negation of its first bit.

Figure 3-B shows an example derivation of a key k's infix with $m_{\text{infix}} = 7$. Here, the longest common prefix of the predecessor and successor is $m^i_{\text{shared}} = 4$ bits long. After removing the $m^i_{\text{redundant}}$ bits, Diva appends the next 6 bits in k (i.e., 101001) to the first differentiating bit (i.e., 0), yielding the infix $(0101001)_2 = 41$.

Infix Uniformity. The samples in the trie behave similarly to the boundaries of an equi-depth histogram. When the dataset's distribution is well-behaved, the higher-order bits of the samples capture the overall shape of the distribution and the lower-order bits of the keys in-between behave like noise. Thus, keys falling into the same bin and are almost uniformly distributed, implying that their infixes are also uniformly distributed. We formally prove this property in Section 4. This uniformity allows infixes to be treated as hash digests. We will use this property to design an efficient data structure for storing the infixes in-between a pair of samples.

3.2 Infix Stores

An Infix Store is a random-access array of slots that leverages the uniformity of the infixes in-between two samples to efficiently and succinctly store them. We describe a static Infix Store variant and extend it to support dynamic operations in Section 3.5 by utilizing techniques inspired by the Rank-and-Select Quotient Filter [41].

Quotienting. An Infix Store applies Knuth's quotienting technique [30] to its infixes, splitting them into *Quotients* and *Remainders*. Specifically, it takes the $\lceil \log_2 \frac{2}{\epsilon} \rceil$ least-significant bits of an infix x as its remainder x_r while taking the rest of its bits as its quotient x_q. Figure 4-A illustrates this split for an infix $x = (11011)_2$. Splitting the infixes in this way allows an Infix Store to succinctly encode infixes using little extra metadata.

Storing Quotients. An Infix Store employs a bitmap, called the `occupieds` bitmap, to represent every possible quotient. The i-th bit in this bitmap is set to 1 if an infix with a quotient equal to i exists and is set to 0 otherwise. This bitmap is T bits long since there are between $\frac{T}{2}$ and T many possible quotients.[1] One can show this by applying Lemma 3.1 and removing the $\lceil \log_2 \frac{2}{\epsilon} \rceil$ least-significant bits belonging to remainders from the possible infixes to derive the range of all possible quotients. Figure 4-B shows an example of this bitmap. Here, since the Infix Store contains the infix x in Figure 4-A, the bit at offset 3 of the `occupieds` bitmap is set to 1.

Storing Remainders. An Infix Store places infix remainders in its array of slots. As there are exactly $T-1$ keys in-between two samples, an Infix Store allocates $T-1$ slots in this array, each $\lceil \log_2 \frac{2}{\epsilon} \rceil$ bits wide, to accommodate their remainders. It stores remainders of infixes sharing the same quotient in a set of contiguous slots called a *Run*. Runs are stored in increasing order of their quotients. Runs are 1-2 slots long in expectation since the infixes are uniformly distributed, as described at the end of Section 3.1, and have between $\frac{T}{2}$ and T distinct quotients.

An Infix Store employs a $(T-1)$-bit bitmap with one bit per slot, called the `runends` bitmap, to delimit runs in an Infix Store. The

[1]One can optimize the memory consumption of the filter by storing less than T bits in this bitmap whenever there are less than T possible quotients. Since the infixes are uniformly distributed, this saves, on average, 0.25 BPK of memory. However, we have opted for a T-bit bitmap to simplify the design and implementation of Infix Stores.

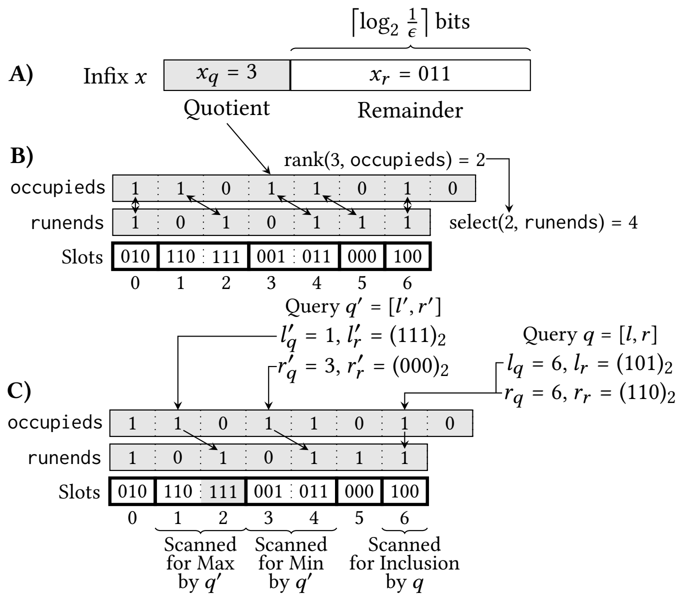

Figure 4: Diva splits an infix into a remainder and a quotient (Part A). It encodes the quotient in the occupieds bitmap while storing the remainder in contiguous runs in the array of slots, delineated using thick lines (Part B). Diva handles a query by locating the corresponding runs using rank and select operations, followed by scanning them (Part C). The remainders that satisfy the queries are highlighted.

i-th bit of this bitmap is 1 if the i-th slot in the Infix Store is the last slot of a run and is 0 otherwise.

Figure 4-B shows an example of runs, depicted as boxes, and how the `runends` bitmap delimits them. Here, the infix x from Figure 4-A has the third smallest quotient among the quotients of infixes in the Infix Store. As such, its remainder is placed in the third run from the left.

In total, the `occupieds` and `runends` bitmaps consume 2 bits per slot in the array of slots, or equivalently, 2 bits per infix.

Matching Invariant. Each quotient, i.e., a 1 in the occupieds bitmap, is associated with exactly one run in an Infix Store. Since runs are in increasing quotient order, a *Matching Invariant* holds: the run associated with the i-th 1 in the `occupieds` bitmap ends at the slot corresponding to the i-th 1 in the `runends` bitmap. Figure 4-B illustrates this invariant with bidirectional arrows.

Locating Runs. Diva locates an infix's run by first checking the bit corresponding to its quotient in the `occupieds` bitmap. If it is 0, then the infix does not have a run. Otherwise, it leverages the one-to-one correspondences in the matching invariant to locate the end of its run, allowing for processing queries via a right-to-left scan of an average of 1-2 slots. Following a similar process, Diva is able to reconstruct all the infixes by concatenating each quotient in the `occupieds` bitmap with the remainders in its run.

Diva speeds up locating an infix's run by employing rank and select primitives [41]. Formally, let rank(i, B) be the number of 1s before the i-th bit in a bitmap B and select(i, B) be the position of the i-th 1 bit in B. Diva finds the last slot in an infix x's run by evaluating select(rank(x_q, `occupieds`), `runends`). In other words, it determines the run number it must jump to by computing rank over the `occupieds` bitmap and uses the result to locate the corresponding `runends` bits by computing select. Figure 4-B shows an example of this derivation with $x_q = 3$. Here, Diva determines that x has a run by checking the third bit in the `occupieds` bitmap. It then computes rank(x_q, `occupieds`) = 1 and select(1, `runends`) = 4, concluding that x's run ends in Slot 4. Diva employs specialized hardware instructions to efficiently apply rank and select operations, as shown in Section 3.6.

Infix Store Range Queries. An Infix Store processes a range query over infixes $q = [l, r]$ by finding the quotients and remainders of the endpoints and considering the following two cases:

If the range spans a single quotient ($l_q = r_q$), Diva checks if a run corresponding to that quotient exists. If not, Diva returns a negative, as no infix could be in the query range. Otherwise, it goes to the end of the run and scans it from right to left until it reaches either a new run or the start of the Infix Store. Diva determines if it has reached a new run using the `runends` bitmap. During this scan, Diva searches for a remainder between the remainders of the endpoints l_r and r_r. If found, it returns a positive and a negative otherwise. This scan is very efficient, as the expected size of a run is 1-2 slots. Query q in Figure 4-C is an example. Here, $l_q = r_q = 6$ and the bit at position 6 of the `occupieds` bitmap is one. Thus, Diva searches the relevant run for a remainder between $l_r = (101)_2$ and $r_r = (110)_2$. Since rank(6, `occupieds`) = 3 and select(3, `runends`) = 6, Diva checks the slots from Slot 6 backward until it exhausts the run. It returns a negative, as it does not find a remainder in the desired range.

If the range spans multiple quotients ($l_q < r_q$), Diva checks if any quotient strictly between the endpoint quotients exists using the occupieds bitmap. If there is such a quotient, Diva answers with a positive, as all of the infixes in its run are strictly in the query range. Otherwise, it checks if any remainder in the left endpoint's run is larger than its remainder l_r, or if any remainder in the right endpoint's run is smaller than its remainder r_r. If either condition holds, Diva reports a positive and a negative otherwise. Query q' in Figure 4-C shows an example. Here, Diva first checks the range of bits $[l'_q + 1, r'_q - 1] = [2, 2]$ in the `occupieds` bitmap for ones. As there is no bit set to 1 in that range, Diva scans the remainders in l'_q and r'_q's runs and compares them to l'_r and r'_r. As the former run has a remainder equal to $l'_r = (111)_2$, Diva returns a positive.

False positives occur when query and key infixes collide, which is most likely when queries follow the dataset's distribution. Diva's trie learns this distribution, enabling a low FPR in this difficult setting and potentially a better FPR in other workloads.

Infix Store Point Queries. Point queries are equivalent to specialized range queries having equal endpoints. They are thus handled by following the first case of the range query algorithm.

3.3 Trie Choice

Diva can use any data structure supporting predecessor and successor searches as its trie. This data structure must also associate a pointer pointing to an Infix Store with each sample. The samples and the extra pointers result in a negligible overhead of $O(L) + 64$ bits per sample, where L is the average key length. This translates to a total overhead of $\frac{O(L)+64}{T} \approx 0.06 + O(L/T)$ BPK. Using a traditional trie [9, 32, 34] for this purpose yields poor performance for predecessor and successor queries, as it incurs $O(L)$ cache misses for tree traversal due to pointer chasing.

***y*-Fast Tries.** Instead of a traditional trie, one can employ a y-Fast trie [48] to achieve an $O(\log_2 L)$ number of cache misses for predecessor and successor queries. A y-Fast trie enables this by partitioning the ordered set of keys into groups of $\approx L$ consecutive keys and storing each group in a balanced binary search tree. It efficiently accesses these groups by storing their boundary keys in a binary trie and doing predecessor/successor searches over it.

The nodes in this binary trie are represented as a hash table storing all prefixes of the boundary keys. Each inner node without a left (resp. right) child stores a pointer to the predecessor (resp. successor) of the minimum (resp. maximum) key in its subtree, allowing for fast predecessor and successor searches. This binary trie finds the predecessor/successor of a key by determining its longest prefix with a node in the trie via binary search. If the found node is a leaf node, it is returned as the answer. Otherwise, the predecessor/successor pointers are followed to answer the query.

A y-Fast trie processes a predecessor/successor query by doing a predecessor/successor search over its binary trie to find the corresponding binary search tree and searching in it.

Searching the binary trie and the associated binary search tree each incurs $O(\log_2 L)$ cache misses, resulting in $O(\log_2 L)$ cache misses for a predecessor/successor query. This cost applies out-of-the-box to Diva's queries since searching an Infix Store incurs only a constant number of cache misses due to its small size, leaving the search cost of the trie as the main bottleneck.

A y-Fast trie supports amortized $O(\log_2 L)$ insertions and deletions by merging and splitting the binary search trees to maintain a size of $\approx L$ for each and updating its binary trie.

Wormhole. As our y-Fast trie implementation, we use Wormhole [49], a y-Fast trie with a high fanout. We extended Wormhole with bidirectional iterators, allowing to quickly traverse the trie both forwards and backwards. We also disabled prefetching in Wormhole since the trie tends to be small and cache resident in our context, resulting in improved search performance.

3.4 Construction

The static variant of Diva is constructed using an efficient bulk-loading procedure where a sorted set of keys is scanned once. Here, Diva inserts every T-th key into the trie. Since it encounters the keys in-between in increasing order, it also sees their infixes in increasing order. During this scan, Diva sets the bits in the `occupieds` bitmap corresponding to the quotients it sees to 1 while sequentially creating the runs by placing remainders in the slots. Whenever it encounters a new quotient or fills the Infix Store, it marks the end of the current run in the `runends` bitmap by setting the bit corresponding to the last filled slot to 1.

This process incurs $O(N)$ cache misses for scanning the sorted key set and inserting infixes into Infix Stores, as well as $O\left(N \cdot \frac{\log_2 L}{T}\right)$ cache misses for inserting every T-th key as a sample into the trie, totaling to $O\left(N \cdot \left(1 + \frac{\log_2 L}{T}\right)\right)$ cache misses. Since $\frac{\log_2 L}{T} \ll 1$, we can rewrite the cost as $O(N)$. The left-most column in Table 3 compares the construction times of the filters and the penultimate row expresses this cost for Diva.

Practically, Diva leverages the hardware prefetcher to the fullest during bulk-loading, as all its memory accesses are sequential. Its CPU overhead is also minimal, as it makes no use of hashing, save for the hashing of the trie, allowing Diva to enjoy a much faster construction time than its competitors.

3.5 Dynamicity

Diva generalizes to dynamic key sets, providing full support for insertions, deletions, and growing datasets.

Dynamic Infix Stores. Diva modifies its Infix Stores to support dynamic insertions and deletions using three techniques: 1) it overprovisions slots in each Infix Store to absorb new insertions, 2) it evenly spaces out the runs using a linear mapping function to uniformly place them in the array of slots according to their quotients, and 3) it handles insertions and deletions by shifting the runs to the right or left to create space for remainders, similarly to Robin-Hood Hashing [11].

Inserting and deleting infixes may cause Infix Stores to overflow or underflow. Diva handles such cases by resizing the array of slots in the Infix Store and updating its linear mapping function to maintain the even spacing of the runs.

Sampling New Keys. Diva maintains its distribution model by sampling new keys and updating its trie, "splitting" large Infix Stores in the process. Specifically, with probability $\frac{1}{T}$, Diva inserts a new key as a sample into the trie, and with probability $\left(1 - \frac{1}{T}\right)$, it inserts the key's infix into its Infix Store. This choice of probabilities ensures a high-resolution distribution model that keeps Infix Stores at an average size of T, keeping runs short and controlling the FPR. Randomization is essential here since we would like to store full keys in the trie to approximate the key distribution closely. Diva can only access full keys during insertions, so it must sample them on the fly and cannot split Infix Stores along their median like B-Trees.

Splitting and Merging Infix Stores. When sampling a new key, Diva splits the Infix Store it would have otherwise fallen into and sends the infixes of smaller and larger keys to separate Infix Stores. Splitting Infix Stores in this way keeps them small and fast to update. Analogously, Diva merges two neighboring Infix Stores when deleting the sample between them from the trie.

Variable-Length Infixes. Inserting new samples into the trie brings the predecessors and successors of the keys closer together. As a result, old infixes may not carry the required fine-grained information to differentiate their keys due to longer common prefixes and an increased number of redundant bits, making them shorter. Yet, Diva stores as much information as possible about new keys by inserting them as full-length infixes to control the FPR. Diva encodes these variable-length infixes in the same Infix Store by padding them to the same length with unary padding consisting of 0s delimited by a 1, similarly to recent expandable filters [5, 18, 19, 23]. Padded infixes are one bit longer than full-length infixes because they have the delimiter bit, requiring slots that are one bit wider in the Infix Stores and incurring a memory overhead of 1 BPK. However, the shorter infixes (and fingerprints in expandable filters) cause the FPR to increase logarithmically with the data size [19, 23]. One can maintain a stable FPR at all times by widening the slots in the Infix Stores by $O(\log_2 \log_2 N)$ bits after N insertions to store progressively longer infixes, similarly to InfiniFilter's Widening Regime [18, 19, 39].

Queries. Point and range queries are processed similarly to the static case, with the only difference being that the missing bits from shorter infixes are treated as wildcards.

3.6 Optimizations

Evaluating rank and select. Diva efficiently evaluates these functions over a bitmap B by employing specialized CPU instructions. In the case of rank(i, B), Diva reads the first $\lceil \frac{i}{64} \rceil$ words in B and applies POPCNT to count the number of 1s before the i-th bit. Similarly, it computes select(i, B) by scanning B word-by-word and keeping a running count of the 1s to locate the word containing the i-th 1. Once found, Diva applies the x86 instructions PDEP and LZCNT to calculate the bit's position, as described in [41].

Fractional Multiplication. The dynamic variant of Diva described in Section 3.5 uses a linear mapping to evenly space out the infixes of each Infix Store. Such a remapping entails a multiplication by a fractional factor. Naïvely evaluating this function using floating-point arithmetic is slow. Instead, Diva simulates fixed-point arithmetic using integers, yielding a significant speed-up.

4 THEORETICAL ANALYSIS AND RESULTS

We prove Diva's FPR guarantees and show that it has low operation costs. The last row in Table 2 summarizes this section's results.

Setting. Formally, we consider the dataset's keys to be sampled independently from a distribution with a cumulative distribution function (CDF) F and probability density function (PDF) f. For our analysis, we treat string keys as real numbers in the range $[0, 1)$ by interpreting their binary representation as a fractional binary value. For example, we interpret the strings $(1000)_2$ and $(1100)_2$ as the real numbers $(0.1000)_2 = 0.5$ and $(0.1100)_2 = 0.75$.

Infix Probability. Diva returns a false positive when a query and an input key from the distribution map to the same infix in the same Infix Store. We bound the probability of this event by calculating the probability that a specific infix exists.

Intuitively, Diva's Infix Stores partition the key space into shorter ranges. Diva discretizes each of these ranges by deriving infixes, further partitioning it into $\frac{2T}{\epsilon}$ equal-width sub-ranges, one for each possible infix. Figure 5 illustrates this partitioning for the k-th Infix Store. The probability that a particular input key maps to a specific infix is equal to the probability that it lies in the corresponding sub-range, which depends on the distribution's CDF. We denote this probability for infix n in the k-th Infix Store as $P_{k,n}$. We can guarantee an FPR of at most ϵ by showing that $P_{k,n} \le \frac{\epsilon}{N}$ and applying a union bound to all keys in the dataset.

Example: Uniform Distribution. To give intuition why $P_{k,n}$ is small for well-behaved distributions, we first discuss the case where the input distribution is uniform, i.e., $F(x) = x$. Here, $P_{k,n}$ is exactly equal to the length of its infix's sub-range. That is, it is equal to $\frac{\epsilon}{2T} \cdot l_k$, where l_k is the length of the range spanned by the k-th Infix Store, as shown in Figure 5. Since there are $\frac{N}{T}$ many Infix Stores, l_k is in expectation $\frac{T}{N}$. Thus, the probability of an input key landing in this sub-range is equal to $\frac{\epsilon}{2N}$, which is less than $\frac{\epsilon}{N}$.

Well-Behaved Distributions. In general, the CDF of the distribution could make $P_{k,n}$ arbitrarily large, depending on how abruptly it changes within that particular sub-range. Nevertheless, we can make a similar argument as in the uniform case for *well-behaved* distributions, defined as:

Definition 4.1 (Well-Behaved Distribution). Let f' be the derivative of f. A distribution is well-behaved if, for $\delta = T \cdot \frac{\log N}{N}$, a constant $\alpha \in [0, 1/4]$ exists such that for all x,

$$\delta^{\alpha} \le f(x) \le \delta^{-1+4\alpha}, \quad f'(x) \le \frac{1}{3} \cdot \delta^{-\alpha}.$$

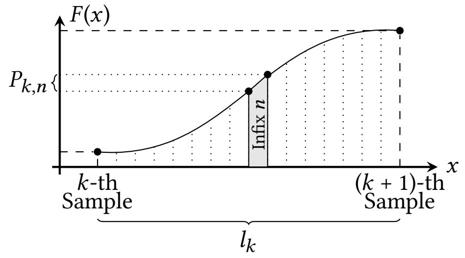

Figure 5: Infix Stores partition the key space into short ranges. Each of these ranges is partitioned into equal-width sub-ranges, each of which is condensed into an infix.

The conditions above make well-behaved distributions "smooth." That is, the condition on f keeps the distribution's CDF from radically changing, while the condition on f' limits the acceleration of change. Together, they control how much the CDF can climb in an infix's sub-range, which is $P_{k,n}$, as shown in Figure 5. We remark that the bounds on these quantities grow polynomially in N, thus becoming easier to satisfy as the dataset grows. For example, f' is allowed to take values as high as $\frac{\delta^{-\alpha}}{3} \ge \frac{\delta^{-1/4}}{3} \approx \frac{N^{1/4}}{3}$. Intuitively, this is because $P_{k,n}$ corresponds to a smaller sub-range with larger N, so our analysis can afford less smooth distributions.

Many natural distributions are well-behaved, as one can verify by plugging their PDFs into Definition 4.1. For example, the uniform distribution is trivially well-behaved since it has $f(x) = 1$ and $f'(x) = 0$. Normal distributions with a standard deviation of $\sigma \gtrapprox \frac{3}{\sqrt{\pi} \cdot N^{1/5}}$ and power law distributions (which generalize the Zipfian distribution) with an exponent of $\lambda \le \frac{\log_2 N}{4}$ are also well-behaved. Note that σ and λ often satisfy these conditions.

One can use the properties of well-behaved distributions in an argument similar to that of the uniform distribution to show that:

Theorem 4.2. *If the input distribution is well-behaved, then, with high probability over Diva's samples, in the static (dynamic) case, a particular infix exists with (expected) probability at most ϵ.*

FPR. We now use Theorem 4.2 to bound Diva's FPR when the dataset comes from a well-behaved distribution.

Theorem 4.3. *Suppose that the input distribution is well-behaved. Then the static variant of Diva returns a false positive for a point query q with probability at most ϵ. Moreover, it returns a false positive for a range query $[q_l, q_r]$ with a probability of at most 2ϵ.*

Proof. Consider the point query q. Since we are analyzing the probability of a false positive, we can suppose that q is not in the dataset and thus does not equal any sample in Diva's trie. Therefore, Diva checks if an infix corresponding to q exists to answer the query. By Theorem 4.2, such an infix exists with a probability of at most ϵ, implying an FPR of at most ϵ for q.

Now, consider the range query $[q_l, q_r]$. As before, we can suppose that this is an empty range, implying that it includes none of the samples in the trie and thus falls in a single Infix Store. Denoting the infixes corresponding to q_l and q_r by l and r, one can see that there must be no other infix in the Infix Store that is strictly between l

and r, as it would cause the filter to return a true positive. Thus, Diva can only mistakenly return a false positive if some infix equals either l or r. By applying Theorem 4.2, one can show that this happens with a probability of at most 2ϵ. □

One can analyze the dynamic variant of Diva by following a proof similar to that of Theorem 4.3. More specifically, since the filter is initially constructed using the static bulk-loading method, it starts out with point and range query FPRs of ϵ and 2ϵ. After inserting N new keys into the filter with no distribution shifts, one can use an analysis similar to that of InfiniFilter [19] to show that:

Theorem 4.4. *The dynamic variant of Diva returns a false positive for point query q with expected probability at most $\epsilon/2 \cdot (\log_2 N + 2)$. Moreover, it returns a false positive for a range query $[q_l, q_r]$ with expected probability at most $\epsilon \cdot (\log_2 N + 2)$.*

Memory Footprint. Both Diva variants keep a $\frac{1}{T}$ fraction of the keys in their trie. For each, they also store a total of 96 bits of metadata, representing whether the sample is a partial sample, the size of the Infix Store to its right, the number of infixes it stores, and a pointer to its array. Assuming that the keys have an average length of L bits, the trie consumes a total of $\frac{(96+L)\cdot N}{T}$ bits.

In the static variant, each Infix Store has T slots. It consumes T bits for the `occupieds` bitmap, one bit per slot for the `runends` bitmap, and $\log_2 \frac{2}{\epsilon}$ bits for each slot. Since there are a total of N/T Infix Stores, the total memory footprint of the Infix Stores is $N/T \cdot (T\cdot(3+\log_2 \frac{1}{\epsilon}))$ bits. Summing up the trie and Infix Store costs, noting that $T = 1024$, and dividing by N results in a memory footprint of $\approx 3.09 + \frac{L}{1024} + \log_2 \frac{1}{\epsilon}$ BPK.

In contrast, in the dynamic variant, an Infix Store with n infixes uses at most $\frac{n}{\alpha^2}$ slots. Since there are a total of $\frac{T-1}{T} \cdot N$ infixes, this translates to a total of $\frac{T-1}{T\cdot\alpha^2} \cdot N \approx \frac{1}{\alpha^2} \cdot N$ slots. Moreover, each slot is one bit wider than the static case to accommodate the unary counter. As there are an expected of N/T Infix Stores, this results in a total memory footprint of $\approx 1.09 + \frac{L}{1024} + \frac{1}{\alpha^2} \cdot (2 + \log_2 \frac{1}{\epsilon})$ BPK.

Performance. Each of Diva's operations searches for the predecessor and successor of the query endpoints in the trie. As Diva uses Wormhole as its underlying trie, and since Wormhole searches for an L-bit key using at most $O(\log_2 L)$ cache misses, this search incurs a total of $O(\log_2 L)$ cache misses. Moreover, as the `occupieds` and `runends` bitmaps are $T = 1024$ and $\frac{T}{\alpha} \approx 1078$ bits long, and since a typical cache line is 512 bits long, applying rank and select operations on them incurs a constant number of cache misses. Finally, reading/modifying the slots in the Infix Store incurs a single cache miss in expectation. Thus, Diva incurs an expected of $O(\log_2 L + 1) = O(\log_2 L)$ cache misses. Notice that when keys are fixed-length, the number of cache misses becomes constant.

5 EVALUATION

We evaluate Diva against existing range filters in static and dynamic settings in Sections 5.1 and 5.2, respectively. We also conduct end-to-end evaluations on top of WiredTiger [36], a popular B-Tree-based key-value store, in Section 5.2. We implement and open source a new and extensible range filter benchmarking framework and use it to conduct our experiments.

Platform. We run experiments on a Fedora 39 machine with an Intel Xeon w7-2495X processor (4.8 GHz) with 24 cores and 48 hyper-threads. Our machine has a 1920 KB L1, a 48 MB L2, and a 45 MB L3 cache, along with 64 GBs of main memory. It also has two SK Hynix 512 GB PC611 M.2 2280 80mm SSDs, which are used in the end-to-end experiments only.

5.1 Static Evaluation

Baselines. We compare Diva to all existing range filters except for bloomRF [37], as its implementation is closed-source. As most other range filters only support integer keys, we specialize a version of Diva for integers to enable a fair comparison while still benchmarking the general-purpose version of Diva that supports variable-length keys. We implement both these versions in `C++`. All other baselines we compare to are implemented in `C/C++` as well. We compile all filters using `gcc-13`.

Integer Datasets. Following prior work [12, 17, 23, 24, 29, 33, 37, 46, 47, 51], we employ the following synthetic and real-world [28, 35] datasets for our evaluation over integer data:

- Uniform: 200M uniformly sampled 64-bit integers.
- Normal: 200M 64-bit integer samples from $\mathcal{N}(2^{63}, 2^{50})$.
- Books: 200M popularity scores for books on Amazon. This dataset is heavily skewed, containing many more lower ratings than higher ones.
- OSM: 200M geocoordinates from the Open Street Map. It features several densely populated regions in the key space.

Later in Experiment 3, we adapt some of these workloads to include variable-length keys.

Query Workloads. For integer datasets, we sample a start key x and a length R to generate the range $[x, x + R - 1]$. We vary R to showcase the effect of range lengths on filter performance and FPR. For synthetic workloads, we choose the starting key x by sampling from the same distribution as the dataset. For real workloads, we sample x from the dataset and subsequently remove it from the set. We generate point queries by using $R = 1$.

Our workloads issue empty range queries, and we measure the FPR of each filter by dividing the number of positive results by the size of the query batch. We also conduct a separate experiment gauging performance for non-empty queries. Our benchmarks focus on filter CPU times. All workloads issue a total of 10M queries.

Experiment 1: Integer FPR vs. Query Size Tradeoff. The first and second rows of Figure 6 depict the FPR of our baselines with a memory budget of 10 and 16 BPK, respectively. Here, the x-axes vary the query sizes. The bottom row of Figure 6 compares the filters' query latencies for the experiments of the second row. We omit query latency measurements for the first row since they are similar. Figure 6 has both solid and dotted plots for Proteus. In the solid plots, Proteus is tuned using a sample of the queries, while in the dotted plots, it is tuned with range queries with uniformly distributed endpoints, simulating a workload shift. Such workload shifts do not impact the other range filters. We tune Rosetta and Memento filter to assume a maximum range query length of $R = 128$ to keep their FPR for short range queries low and their memory footprint in line with the other filters. If one keeps the memory footprint constant and tunes Rosetta for longer ranges, its query speed deteriorates. Tuning Memento filter for longer ranges yields faster queries at the expense of the FPR.

The FPRs of robust range filters, i.e., Rosetta, Grafite, and Memento filter, approach one as the range query length increases.

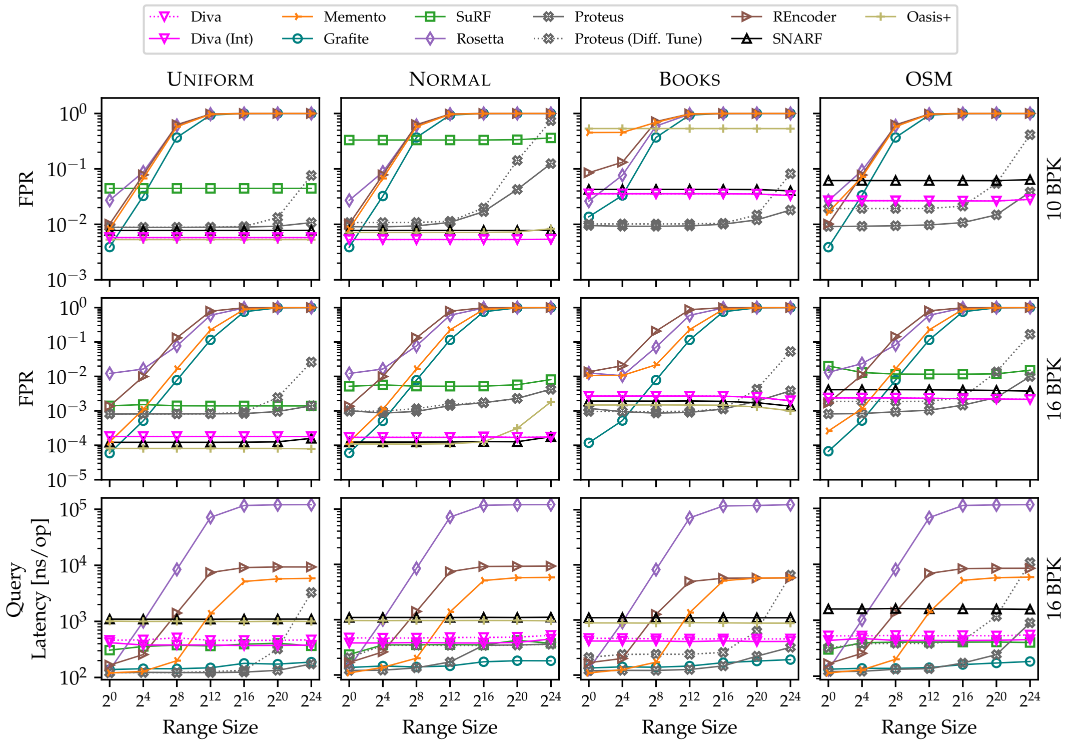

Figure 6: Diva provides the best balance between FPR and query latency for any workload across range query sizes.

As mentioned in Section 2, this is due to the filters issuing extra probes in their internal structures. This also increases the number of cache misses in Rosetta and Memento filter, slowing down range queries by orders of magnitude. Similarly, REncoder's FPR and speed rapidly deteriorate with longer queries since it issues more Bloom filter probes.

While SuRF approximately matches Diva's query performance, its FPR is at least an order of magnitude higher across experiments, as it maximally truncates the keys in its trie. Furthermore, SuRF is unable to curb its memory footprint under some datasets. For instance, in Figure 6, SuRF is missing from the BOOKS column since it requires at least 21 BPK to create a trie that differentiates all of the dataset's keys, exceeding the allotted memory budget.

SNARF and Oasis+ match Diva's FPR but exhibit $\approx 2\times$ slower queries due to their slower binary searches. They also do not support variable-length keys and dynamic operations. Moreover, in some scenarios, such as in the OSM dataset, Diva achieves better FPRs than SNARF by a factor of $\approx 2\times$. This is due to floating point errors in SNARF's distribution model. We exclude Oasis+ from the OSM column since its construction takes more than 12 hours.

Proteus' FPR is slightly lower than Diva's under the BOOKS and OSM datasets (due to some redundancy of the keys carrying over into Diva's infixes) but is worse by orders of magnitude under the UNIFORM and NORMAL datasets, as the latter workloads. Moreover, Proteus' dotted plots show that in the face of a workload shift, its FPR shoots up for medium to long queries, resulting in orders of magnitude higher FPRs than Diva. Although Proteus achieves faster queries than Diva in the absence of workload shifts, it becomes slower than Diva by an order of magnitude in the face of a workload shift as it performs more Bloom filter probes.

As shown in Figure 6, any filter that outperforms Diva in terms of FPR has significantly slower queries and vice-versa. Hence, Diva provides the best balance between FPR and query speed across the board for variable-length range queries. Moreover, as shown by the range size 2^0, Diva is also competitive for point queries. As we show in subsequent experiments, Diva is also more general-purpose than its competitors since it supports variable-length keys and dynamic insertions, deletions, and expansions.

Experiment 2: Integer FPR vs. Memory Tradeoff. Figure 7 depicts the FPRs of the baselines as we vary the memory budget while fixing the query length to $R = 2^{10}$. Due to its semi-robust FPR guarantee, Diva achieves a low FPR across the board, closely matching the best competitors. Although Proteus, SNARF, and Oasis+ exhibit better filtering under certain datasets, they have higher FPRs under others. Moreover, unlike Diva, none support variable-length keys and dynamicity. Because Diva's memory footprint does not depend on R and its metadata overhead is small, it is still operational under stringent memory budgets as low as 8 BPK (unlike SuRF or the robust range filters).

Experiment 3: String FPR vs. Memory Tradeoff. We now experiment with datasets and workloads containing variable-length keys. We use the following datasets, each consisting of 200M keys:

- NORMAL: the length of the key, in bytes, is chosen at random from the set $\{8, 16, 32, 64, 128, 256\}$, and the first 64 bits of the keys follow the normal distribution $\mathcal{N}(2^{63}, 2^{50})$ while the other bits are chosen uniformly at random,
- ENWIKI: English words extracted from Wikipedia titles [1].

Using the ENWIKI dataset without modification results in a high FPR for all filters. This is an artifact of the erratic distribution of the English language, which features many "short and stepwise skews." To address this problem, we compress the keys using Hu-Tucker coding [50], an order-preserving variant of Huffman coding, to "flatten" the distribution and make it more well-behaved

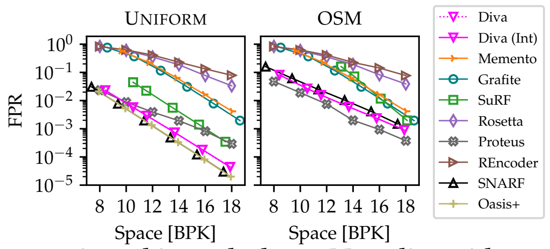

Figure 7: Diva achieves the best FPR scaling with memory while being competitive under stringent memory budgets.

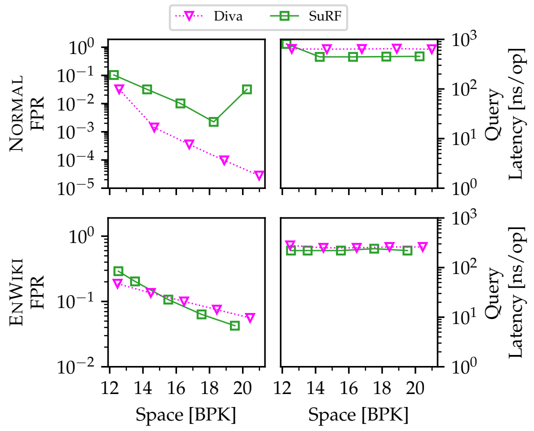

Figure 8: Diva guarantees competitive FPRs and comparable query times to SuRF when operating on strings.

before constructing the filters. We generate queries by sampling and removing pairs of adjacent keys from the dataset.

Figure 8 evaluates the FPR of Diva and SuRF, the only filters that support string keys, under various memory budgets. SuRF creates a shallow truncated trie to differentiate the keys in these workloads. This, in conjunction with early termination of empty queries in the upper levels of the trie, allows SuRF to achieve marginally faster queries than Diva by 35%. However, its maximal truncation of suffixes causes SuRF's FPR to be worse than Diva by as much as two orders of magnitude under the NORMAL dataset. While SuRF has a lower FPR for large memory budgets under the ENWIKI dataset due to it ensuring that keys are differentiated, it is worse by 1.5× for stringent budgets, which is the setting in which filters are typically most useful. In contrast, Diva truncates suffixes to different degrees for different regions of the key space, storing finer-grained information for denser regions that improve the FPR.

Experiment 4: Non-Empty Query Performance. Until now, we only issued empty queries in our evaluations. We now benchmark non-empty query latency in Figure 9. The left subfigure varies the memory budget while fixing the range query length to $R = 2^{10}$, and the right subfigure varies the query size while fixing the memory budget to 16 BPK. We only consider the UNIFORM integer dataset, as other datasets yield similar results. Henceforth, Rosetta's hierarchy depth and Memento filter's suffix size are tuned to the longest range query length to enable their best performance.

Diva is faster than all range filters, except for Grafite and Memento filter, across all memory budgets and range query lengths. This is because the other filters incur many cache misses due to using binary search or checking for existing keys in Bloom filters and tries. The closest other range filter to Diva's speed, SuRF, is slower

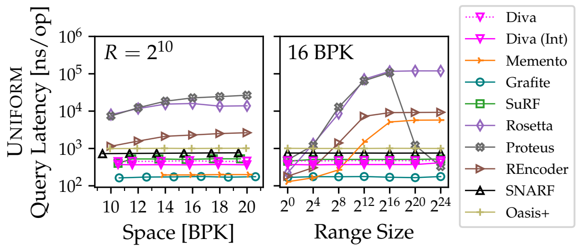

Figure 9: Diva brings competitive performance to the table when processing non-empty range queries.

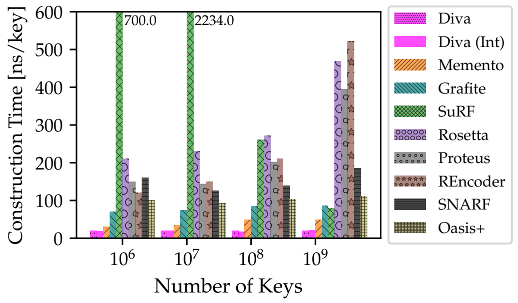

Figure 10: Diva has the fastest construction time among range filters with a large margin.

by as much as 35%, as it has to traverse all the levels in its trie down to a leaf node for a non-empty query. Grafite and Memento filter achieve their speed by localizing similar keys to the same memory region, forgoing support of variable-length keys and queries, both of which are supported by Diva. Rosetta, REncoder, and Proteus incur more cache misses as the memory budget or query length grows since they check more bits in their Bloom filters. In contrast, Diva's query complexity is independent of the memory budget and query size, yielding consistently fast queries.

We have also experimented with mixed workloads, but exclude the plots, as all filters behave similarly to Figure 9, except for Proteus, which behaves similarly to the last row of Figure 6.

Experiment 5: Construction Times. Figure 10 presents the construction times of the range filters on integer datasets of varying sizes under a memory budget of 16 BPK. We use the UNIFORM dataset, though the dataset choice has little influence on filter construction times (except for Oasis+). Diva is significantly faster to construct than all other range filters, with the closest range filter being 2.7× slower. This gap is due to Diva's sequential memory access pattern in the input array and its Infix Stores (which leverages the hardware prefetcher) and due to it only hashing the sample keys in its y-Fast trie, which comprise ≈ 0.1% of all the keys.

5.2 Dynamic Evaluation

We now turn to evaluate Diva under dynamic workloads.

Baselines. We compare Diva to Rosetta [33], REncoder [24, 47], SNARF [46], and Memento filter [23], as they are the only filters that provide insertion APIs. Here, we implement an improved version of Memento filter that employs Aleph filter's techniques [18, 40] to support infinite expansions. In the presence of deletions, we compare Diva to SNARF and Memento filter since they are the only other filters that have deletion APIs. Finally, we integrate Diva and Memento filter into WiredTiger [36] to conduct an end-to-end

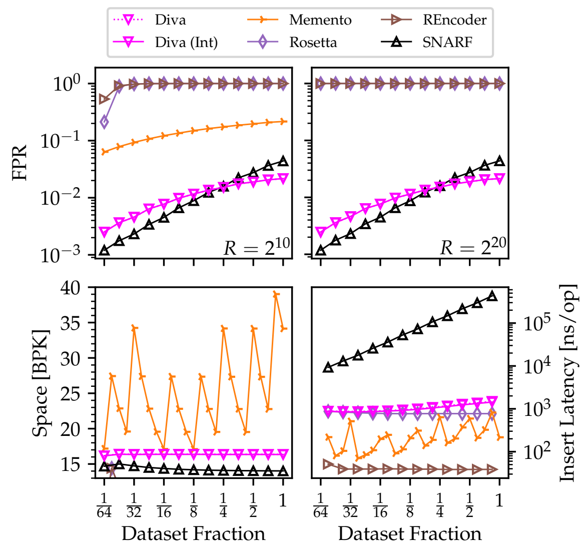

Figure 11: Diva exhibits the best FPR scaling with insertions. It is also the only filter to simultaneously provide fast insertions and a stable space consumption across expansions.

evaluation on a popular B-Tree-based key-value store. By default, the memory budget of each filter is set to 16 BPK.

Datasets and Workloads. We consider the Books dataset containing 200M keys. We construct the filters on a randomly chosen $\frac{1}{64}$ fraction of this dataset and insert the remaining keys in a random order. We issue queries of length $R = 2^{10}$ (and $R = 2^{20}$ in Experiment 6) with endpoints sampled from the same distribution and collect measurements as the dataset grows. Other datasets result in similar measurements.

Experiment 6: FPR, Space, and Insert Latency. Figure 11 evaluates the baselines' FPR, space, and insert latency as new keys are inserted. The x-axis shows the fraction of the dataset ingested. The top row plots the filters' FPR, with its left and right subfigures issuing shorter and longer range queries. The left and right subfigures of the bottom row present the baselines' memory footprint and insertion latency. We measure the memory footprint of the baselines, as some deviate from the user-defined memory budget.

Since Rosetta and REncoder are based on Bloom filters, they are not expandable. As such, new insertions increase the proportion of bits set to 1s in their Bloom filters, causing their FPR to quickly converge to one, making them useless as filters.

SNARF's initial FPR is better than Diva by $\approx 2.1\times$, as it does not have to pay the extra cost of storing unary counters. However, its FPR deteriorates with insertions, becoming worse than Diva by more than $2\times$. This is because SNARF does not update its distribution model. This is also what leads to the slight drop in its memory footprint. In contrast, Diva maintains an accurate distribution model by inserting new samples into its trie. Moreover, SNARF has a linearly deteriorating insertion speed with dataset size due to its linearly increasing block sizes, causing slower insertions than Diva by as much as three orders of magnitude.

Since Memento filter does not support variable-length queries, its FPR is higher than Diva by $\approx 25.3\times$ in the top-left subfigure of Figure 11. Memento filter is missing from the top-right subfigure since it requires more than 16 BPK of memory to support these

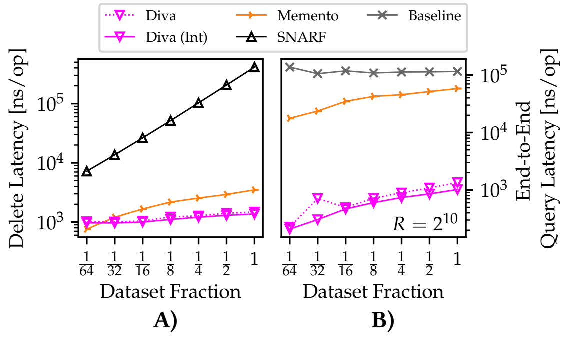

Figure 12: Diva achieves the best deletion and end-to-end query performance under dynamic workloads.

longer queries. Although Memento filter initially has faster insertions than Diva, it becomes similar to Diva as the the dataset grows due to shifting more slots in its hash table. Moreover, its memory consumption exceeds the user-defined budget at times. This is because it expands to twice its original size to accommodate new insertions, wasting as much as 50% of its capacity until it fills up again. In contrast, Diva expands its Infix Stores in small increments, maintaining a constant memory footprint and avoiding wastage.

In sum, Diva maintains the lowest FPR for range queries of any length across expansions while supporting fast insertions. It is also the only dynamic filter that respects its memory budget.

Experiment 7: Deletions. Figure 12-A measures the deletion performance of the filters that provide a deletion API as the dataset grows. It shows the latency of issuing 500K deletions each time the dataset size doubles. Figure 12-A demonstrates that Diva exhibits the fastest deletions, beating Memento filter and SNARF by as much as $2.54\times$ and $300\times$. SNARF's deletion speed deteriorates with insertions since its blocks grow, forcing it to rewrite larger blocks. Memento filter also slows down since the number of slots it has to shift increases. In contrast, Diva controls the amount of shifting and rewriting it does by splitting Infix Stores.

Experiment 8: End-to-End Query Latency. We integrate Diva and Memento filter with WiredTiger and measure the end-to-end empty range query latency. We exclude SNARF from this experiment since it is unsuitable for dynamic workloads due to its inefficient update APIs. Figure 12-B measures end-to-end query latency each time the dataset size doubles. Each key in this workload is associated with a 504-byte value, resulting in 512-byte key-value pairs being stored on disk. We configure WiredTiger with a buffer pool that is 1% of the data size on disk. We trade some of this memory for the range filter to draw a fair comparison. The curve labeled "Baseline" in Figure 12-B represents WiredTiger without a filter.

Diva speeds up WiredTiger's query processing by as much as three orders of magnitude. Moreover, since it supports variable-length queries, it beats Memento filter's query latency by $\approx 85\times$.

6 CONCLUSION

We introduced Diva, the first range filter to simultaneously attain the six range filtering goals. We showed, both theoretically and empirically, that Diva provides support for variable-length queries and keys with excellent FPR, dynamicity, and performance.

REFERENCES

[1] 2025. *EnWiki*. https://dumps.wikimedia.org/enwiki/latest/

[2] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive range filters for cold data: avoiding trips to Siberia. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1714–1725. https://doi.org/10.14778/2556549.2556556

[3] Apache. 2024. *Apache Cassandra*. https://cassandra.apache.org/_/index.html

[4] Apache. 2024. *Apache Druid*. https://druid.apache.org/

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

[6] Paul Beame and Faith E. Fich. 1999. Optimal bounds for the predecessor problem. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing* (Atlanta, Georgia, USA) *(STOC '99)*. Association for Computing Machinery, New York, NY, USA, 295–304. https://doi.org/10.1145/301250.301323

[7] Paul Beame and Faith E. Fich. 2002. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.* 65, 1 (Aug. 2002), 38–72. https://doi.org/10.1006/jcss.2002.1822

[8] Ioana O. Bercea, Jakob Bæk Tejs Houen, and Rasmus Pagh. 2024. Daisy Bloom Filters. In *19th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2024) (Leibniz International Proceedings in Informatics (LIPIcs))*, Hans L. Bodlaender (Ed.), Vol. 294. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:19. https://doi.org/10.4230/LIPIcs.SWAT.2024.9

[9] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 521–534. https://doi.org/10.1145/3183713.3196896

[10] Andrei Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1 (11 2003). https://doi.org/10.1080/15427951.2004.10129096

[11] Pedro Celis, Per-Ake Larson, and J. Ian Munro. 1985. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. 281–288. https://doi.org/10.1109/SFCS.1985.48

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

[13] Google Cloud. 2024. *BigQuery Enterprise Data Warehouse*. https://cloud.google.com/bigquery/

[14] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (jun 1979), 121–137. https://doi.org/10.1145/356770.356776

[15] Alex Conway, Abhishek Gupta, Vijay Chidambaran, Martin Farach-Colton, Rick Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: closing the bandwidth gap for NVMe key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 4, 15 pages.

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

[17] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2023. Grafite: Taming Adversarial Queries with Optimal Range Filters. arXiv:2311.15380 [cs.DS]

[18] Niv Dayan, Ioana Bercea, and Rasmus Pagh. 2024. Aleph Filter: To Infinity in Constant Time. arXiv:2404.04703 [cs.DB] https://arxiv.org/abs/2404.04703

[19] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proc. ACM Manag. Data* 1, 2, Article 140 (jun 2023), 27 pages. https://doi.org/10.1145/3589285

[20] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 365–378. https://doi.org/10.1145/3448016.3457273

[21] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (oct 2021), 32 pages. https://doi.org/10.1145/3483840

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

[23] Navid Eslami and Niv Dayan. 2024. Memento Filter: A Fast, Dynamic, and Robust Range Filter. *Proc. ACM Manag. Data* 2, 6, Article 244 (Dec. 2024), 27 pages. https://doi.org/10.1145/3698820

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

[25] R.M. Fano. 1971. *On the Number of Bits Required to Implement an Associative Memory*. MIT Project MAC Computer Structures Group. https://books.google.ca/books?id=07DeGwAACAAJ

[26] MariaDB Foundation. 2024. *MariaDB Server: The Innovative Open Source Database*. https://mariadb.org/

[27] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. 2015. Approximate Range Emptiness in Constant Time and Optimal Space. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Diego, California) *(SODA '15)*. Society for Industrial and Applied Mathematics, USA, 769–775.

[28] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).

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

[30] Donald Knuth. 1973. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley. 391–392 pages.

[31] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[32] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[33] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2071–2086. https://doi.org/10.1145/3318464.3389731

[34] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[35] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

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

[37] Bernhard Mößner, Christian Riegger, Arthur Bernhardt, and Ilia Petrov. 2022. bloomRF: On Performing Range-Queries in Bloom-Filters with Piecewise-Monotone Hash Functions and Prefix Hashing. arXiv:2207.04789 [cs.DB]

[38] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (01 Jun 1996), 351–385. https://doi.org/10.1007/s002360050048

[39] Rasmus Pagh, Gil Segev, and Udi Wieder. 2013. How to Approximate A Set Without Knowing Its Size In Advance. arXiv:1304.1188 [cs.DS] https://arxiv.org/abs/1304.1188

[40] Rasmus Pagh, Gil Segev, and Udi Wieder. 2013. How to Approximate a Set without Knowing Its Size in Advance. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science* (2013), 80–89. https://api.semanticscholar.org/CorpusID:10365891

[41] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 775–787. https://doi.org/10.1145/3035918.3035963

[42] 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* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 636–644. https://doi.org/10.1145/3626246.3654681

[43] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems* (3 ed.). McGraw-Hill, Inc., USA.

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

[45] Snowflake. 2024. *The Snowflake AI Data Cloud*. https://www.snowflake.com/en/

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

[47] 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). 2036–2049. https://doi.org/10.1109/ICDE55515.2023.00158

[48] Dan E. Willard. 1983. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inform. Process. Lett.* 17, 2 (1983), 81–84. https://doi.org/10.1016/0020-0190(83)90075-3

[49] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-memory Data Management. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 18, 16 pages. https://doi.org/10.1145/3302424.3303955

[50] J. M. Yohe. 1972. Algorithm 428: Hu-Tucker minimum redundancy alphabetic coding method [Z]. *Commun. ACM* 15, 5 (May 1972), 360–362. https://doi.org/10.1145/355602.361319

[51] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 323–336. https://doi.org/10.1145/3183713.3196931