Dynamic Range Filtering Beyond Worst-Case Bounds

Navid Eslami
University of Toronto

Ioana O. Bercea
KTH Royal Institute of Technology

Niv Dayan
University of Toronto

ABSTRACT

Range filters are compact 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 stretches and eventually splits to handle insertions and expansions. It processes a range query by traversing the trie and checking for the inclusion of at least one infix in the target query range.

Diva is the culmination of several years of research on filters from Orca Lab at the University of Toronto, in collaboration with KTH and Copenhagen University. This paper describes how Diva builds on this body of work, and how it addresses the limitations of prior art.

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 [22] and network hops [9] when a query happens to target a nonexistent key.

 This is a minor revision of the paper "Diva: Dynamic Range Filter for Var-Length Keys and Queries" published in PVLDB, Vol. 18, No. 11, ISSN 2150-8097. DOI: https://doi.org/10.14778/3749646.3749664.

Range Filters and Applications. 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 were originally developed to support write-optimized database indexes such as LSM-trees [42], where queries must examine multiple candidate files, many of which may not contain any keys within a query's target range [59, 37, 41, 34, 52, 11, 25, 12]. Their applicability, however, is much broader: range filters also help prevent redundant I/Os to SQL tables [1] and B-tree indexes [25, 12], and they have applications in social web analytics [17] and distributed key-value stores [50].

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, and expansions, and the best possible (G5) query and (G6) construction performance. In addition, it should support point queries at least as well as traditional point filters (e.g., Bloom [8, 48], Cuckoo [43, 26], or Quotient filters [6, 45, 46]).

Design Contentions. Every existing range filter only fulfills a subset of these goals [1, 59, 37, 54, 27, 41, 34, 52, 11, 18, 25, 12]. In fact, almost none of them attain (G3) or (G4) [37, 54, 27, 41, 34, 52, 11, 18]. Moreover, Goswami et. al. have proven an information-theoretic lower bound on the memory footprint of range filters [30], stating that it is impossible 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.

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

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.

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 can be constructed using one sequential pass, making it the fastest range filter to construct and enabling (G6). To support dynamicity (G4), Diva builds on a complementary line of work from our lab and collaborators on resizable filters [20, 19, 32, 25, 12].

2. PROBLEM ANALYSIS

This section shows that no current range filter satisfies all of the six goals outlined in Section 1. We leave in-depth summaries of these filters to [25, 47].

Range Filtering Definitions. A range filter represents a set S of keys 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, i.e., 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. 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)* [30], 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. 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.

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	✓	✓			✓
Aeris	✓	✓			✓
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 [37, 18, 25, 12] support variable-length queries (G2) or keys (G3), as they assume range query lengths bounded by R. For example, Memento filter [25], our lab's previous range filter, assumes keys are fixed-length and splits each into a prefix and a $\lceil \log_2 R \rceil$-bit suffix. It hashes a key's prefix to store its suffix within a compact, approximate hash table. Since a range query longer than R spans multiple prefixes, Memento filter must query the hash table for each constituent prefix to answer the query. This causes the FPR to compound beyond ϵ and eventually approach 1 as the number of checks increases, since each check can yield a false positive. All other robust range filters also do more checks for longer range queries, leading to a higher FPR [37, 18, 12].

Semi-Robust FPR Guarantee. We observe that despite the above impossibility result, 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 guarantee in Section 4 of our original paper [24].

Interestingly, SNARF [52] and Oasis+ [11] are learning-augmented range filters [36, 7] that 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 this model, they map each key to a bit in a large bitmap, setting it 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 Diva's proof in Section 4 of our original paper [24] applies to them as well.

Nonetheless, SNARF and Oasis+ assume fixed-length keys when learning the data distribution and therefore fail to sat-

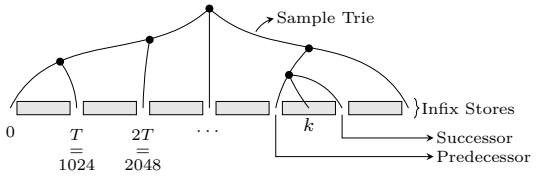

Figure 1: Diva learns the dataset's distribution by storing every T-th key as a sample in a trie. It manages all keys between two adjacent samples in an Infix Store.

isfy (G3). Moreover, both incur high query latency due to their reliance on binary search, violating (G5). Although SNARF nominally supports insertions and deletions, these operations are prohibitively slow because its bitmap is encoded using the compact but static Elias–Fano scheme [23, 28]. Oasis+ does not support dynamic updates, as it prunes empty regions of the key space to improve its false positive rate. Consequently, neither filter satisfies (G4).

Challenges of Attaining Dynamicity (G4). Many existing range filters operate as Bloom filters in their core, hashing keys into a bitmap and setting bits from 0s to 1s [37, 54, 27, 41, 34]. 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 [25, 12].

Other range filters utilize compact encoding schemes (e.g., succinct tries [59] or Elias-Fano [52, 11, 18]). Such formats are difficult to update, as they tightly pack data to avoid storing pointers and offsets. This necessitates changing their entire representation to make room for insertions.

Our prior papers, Memento filter and Aeris filter [25, 12], are the first range filters to support fast insertions, deletions, and resizability. They achieve this by building on our line of work on resizable filters [20, 19]. However, since they double in size to expand, they waste over 50% of their capacity right after expansion. Diva also builds on our prior work on resizable filters and pushes the envelope by allowing expansions by factors smaller than 2, as discussed in Section 3.4.

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 [18] and Memento filter [25] 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 [52] and Oasis+ [11], use predecessor search to handle queries, which requires super-constant time [4, 5]. In practice, these filters use binary search to compute predecessors. Diva alleviates this contention by employing a y-Fast trie [56] and ensuring that it fits in CPU caches, leading to faster predecessor searches.

Construction Speed (G6). Existing range filters use hashing [59, 37, 54, 27, 41, 34, 18, 25, 12] or floating-point operations [52, 11] extensively during construction, incurring high CPU and cache miss costs. In contrast, Diva avoids these operations and performs a single sequential pass over the data, thus supporting the fastest construction speed (G6).

Summary. Table 2 summarizes the goals each range filter meets. As shown, no preexisting range filter achieves (G1), (G2), (G3), and (G4) simultaneously. Table 2 also raises the question of how performant a range filter can be to attain (G5) and (G6) while meeting (G1)-(G4).

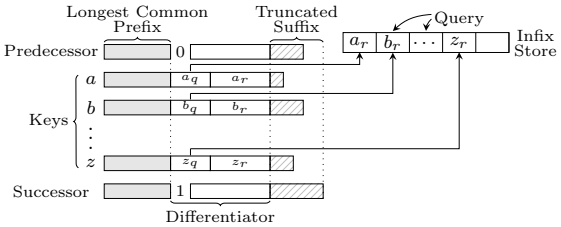

Figure 2: For all keys in-between a pair of adjacent samples in the trie, Diva removes the longest common prefix and truncates the longest possible suffix while still ensuring that there are enough bits left to distinguish between the keys.

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 homogenizes the keys' lengths by truncating a suffix from each while leaving enough bits in the middle (i.e., an infix) to differentiate the keys and guarantee a given FPR. Figure 2 illustrates the truncation and mapping of keys into an Infix Store. To answer a range query, Diva checks for the absence of overlapping keys and infixes across its trie and the relevant Infix Stores, as discussed in Section 3.3. We describe a static version of Diva in Sections 3.1 to 3.3 and generalize it to dynamic sets in Section 3.4. The bottom part of 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 [14, 2, 3, 29, 51], B-Trees [15, 49], or LSM-Trees [22, 16, 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 it 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's memory footprint grows. We find that $T = 1024$ provides good all-round accuracy and performance with little memory overhead ($\approx 1\%$ of the filter's memory). We structure the trie as a y-Fast trie [56] to achieve faster search times than traditional tries ($O(\log_2 L)$ time vs $O(L)$, where L is the search key's length). We use Wormhole [57] as our y-Fast trie implementation.

Removing the Longest Common Prefix. Consider a pair of consecutive trie samples. We refer to them as the *Predeces-*

sor 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, as 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. This 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 distributions such as Normal or Zipfian. We prove this guarantee in Section 4 of our original paper [24]. As we show in Section 3.2, Diva succinctly encodes each infix using $3 + \lceil \log_2 \frac{1}{\epsilon} \rceil$ bits, yielding an FPR vs memory tradeoff on-par with state-of-the-art point filters.

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.

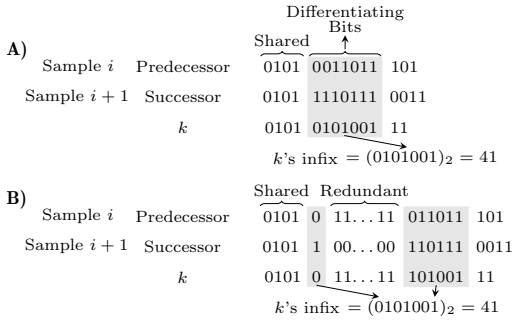

Figure 3: Diva derives key k's infix by removing the m^i_{shared} common prefix bits and the $m^i_{\text{redundant}}$ bits based on its predecessor and successor, and also truncating its suffix.

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.

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, the infixes of keys falling into the same bin are almost uniformly distributed. We formally prove this property in Section 4 of our original paper [24]. In the next section we leverage 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. It utilizes techniques inspired by Cleary hash tables [13] and Rank-and-Select Quotient Filters [45].

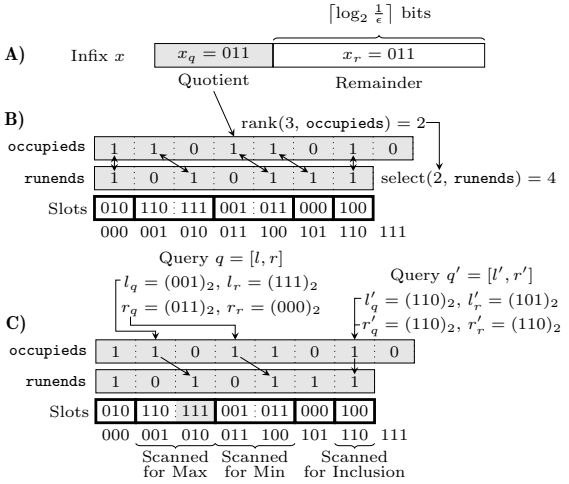

Figure 4: Diva splits an infix into a remainder and a quotient (Part A). It encodes the quotient in the `occupieds` bitmap while storing the remainder in contiguous runs in the array of slots, delineated using thick lines (Part B). Diva handles a query by locating the corresponding runs using rank and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

Quotienting. An Infix Store applies Knuth's quotienting technique [35] to its infixes, splitting them into *Quotients* and *Remainders*. Specifically, we take the $\lceil \log_2 \frac{2}{\epsilon} \rceil$ least-significant bits of an infix x as its remainder x_r and the rest of it (i.e., the $\log_2 T$ higher-order bits) as its quotient x_q. Figure 4-A illustrates this split for an infix $x = (011011)_2$.

Storing Quotients. Splitting each infix allows an Infix Store to succinctly encode its quotient within a dense bitmap called the `occupieds` bitmap. 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. For example, in Figure 4-B, infix x causes the bit at offset 3 (or 011 in binary) of the `occupieds` bitmap to be set to 1. This bitmap needs as many bits as there are possible quotients, which varies based on the predecessor and successor of the Infix Store. This amount is at most T bits since each quotient is $\log_2 T$ bits long.

Storing Remainders. Since there are exactly $T - 1$ keys in-between two samples, an Infix Store allocates an array of $T-1$ slots, each $\lceil \log_2 \frac{2}{\epsilon} \rceil$ bits wide, to accommodate their infixs' 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. Each run spans ≈ 1 slot in expectation since the infixes are almost uniformly distributed, as described at the end of Section 3.1.

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

Matching Invariant. Each run in an Infix Store is associated with exactly one 1 in the `occupieds` bitmap and one 1 in the `runends` bitmap. Since runs are stored 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, the infix could not have been inserted and thus does not have a run. Otherwise, Diva locates the end of the infix's run by employing rank and select primitives [45]. Formally, $\text{rank}(i, B)$ provides the number of 1s before the i-th bit in a bitmap B, and $\text{select}(i, B)$ gives the position of the i-th 1 bit in B. Diva locates the last slot in an infix x's run by evaluating the expression $\text{select}(\text{rank}(x_q, \texttt{occupieds}), \texttt{runends})$. 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` bit by computing select. Figure 4-B shows an example of this derivation with $x_q = 3$. Diva utilizes specialized hardware instructions [45] to apply rank and select efficiently.

Once Diva identifies the end of a run, it scans the remainders within that run from right to left until hitting the preceding run or the beginning of the Infix Store. During this scan, it reconstructs the run's original infixes by concatenating the run's quotient with each scanned remainder.

Bulk Loading. Diva bulk loads a sorted set of keys in a single sequential pass by inserting every T-th key into its trie and placing the infixes into the Infix Stores from left to right. This is significantly faster than other range filters, which incur the random access and CPU cost of hashing.

3.3 Query Processing

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.

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 two cases:

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.

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. Otherwise, it scans this run 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 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.

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 setting and potentially a better FPR in others.

Point Queries. Point queries are equivalent to specialized range queries with equal endpoints. Thus, they are handled by searching the trie and the corresponding Infix Store using the second case of the range query algorithm above.

3.4 Dynamicity

In addition to the static variant of Diva described in previous sections, we provide an equally performant design that supports dynamicity in exchange for ≈ 1 BPK more memory. This variant of Diva employs four core techniques, some of which draw on our prior works on dynamic filters. To better illustrate these techniques, we have reorganized the presentation of an Infix Store's metadata bitmaps and its array of slots in Figure 5.

1) Overprovisioning Infix Stores. Diva overprovisions slots within each Infix Store to absorb insertions, as depicted in Figure 5-A. It inserts a new infix by mapping it to the slot at the offset equal to its quotient. Doing so uniformly scatters the runs across the Infix Store while leaving space in-between them. Diva stores each infix's run in its mapped slot if there is space. If not, it makes room for the new infix by shifting other runs to the right until hitting an empty slot (and shifting to the left if there is no room to the right), similarly to Robin-Hood Hashing [10, 45]. The amount of shifting is moderate, since empty slots are spread evenly across the Infix Store.

2) Stretching Infix Stores. Once an Infix Store is 95% full, Diva expands it by a small factor (≈ 1.05×). It does this using a technique we developed for Zeno Filter [32] called "*Stretching*," whereby we stretch the distance between adjacent runs using a linear function to create more empty slots between them and accommodate more insertions.[1] For example, stretching the Infix Store in Figure 5-A results in Figure 5-B, which has two more empty slots. Stretching ensures a space utilization of at least 90% overall, as opposed to the 50% space utilization of other dynamic filters or hash tables that expand by a factor of 2 [47, 45, 20, 19, 25, 12].

3) Sampling Keys and Splitting Infix Stores. After ≈ T insertions into an Infix Store (and multiple stretching operations), the Infix Store should "*split*" in two, with a new sample in-between added to the trie. Splitting keeps runs short and queries efficient. It also refines the trie's approximation of the data's distribution, particularly for the parts of the key space that become denser.

An invariant of our design is to only store full, untruncated keys within the trie to allow for refining the key space in-between samples indefinitely. This invariant requires that only new insertions into the filter can be picked as samples,

[1]In an Infix Store with n slots, Diva maps an infix x with a quotient of x_q to the slot at offset $\lfloor x_q \cdot n/(r-l) \rfloor$, where l and r denote the value of the bits in the predecessor and successor corresponding to the quotient x_q.

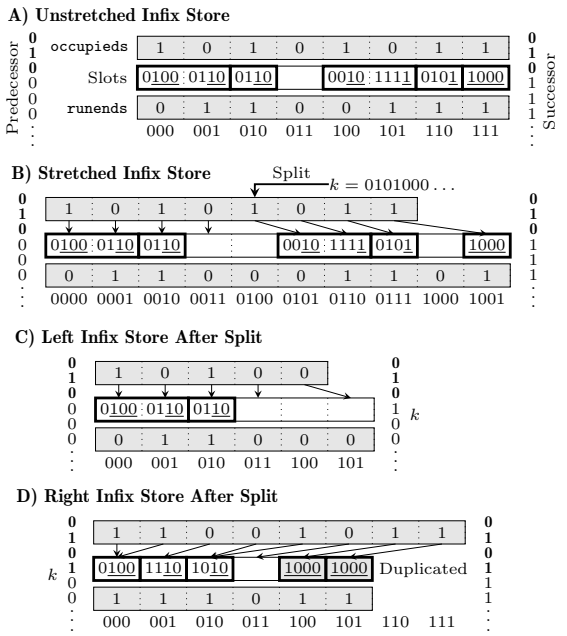

Figure 5: When an Infix Store runs out of overprovisioned space (Part A), Diva stretches it to make room for new insertions (Part B). The insertion of a new sample k into an Infix Store (Part B) splits it into a left (Part C) and a right (Part D) Infix Store. As a result of this split, old infixes may lose bits, elongating the unary paddings (underlined) and duplicating infixes with empty remainders. The predecessor and successor's longest common prefix is highlighted in bold.

since all previously inserted keys have been truncated. Yet, this potentially leads to a vulnerability: deterministically inserting every T-th key as a sample into the trie would allow an adversary to create pathologically uneven splits and degrade performance. We address this risk by randomly picking each new insertion as a sample with a probability of $\frac{1}{T}$. This strategy ensures that, in expectation, every T-th key inserted into an Infix Store splits it and is taken as a sample. As such, the trie is still kept small (≈ $\frac{1}{T}$ of the data's size). Moreover, since the samples collectively constitute a random $\frac{1}{T}$ fraction of the dataset, they lead to approximately even splits across the board, even though some splits may occasionally be uneven.

Figure 5-B illustrates an example of this procedure, which yields the left and right Infix Stores depicted in Figures 5-C and 5-D. The bit strings on the left and right of each Infix Store represent its predecessor and successor, respectively.

4) Variable-Length Infixes. Splitting an Infix Store causes the predecessor and successor of one of the new Infix Stores to have a longer common prefix. For example, the longest common prefix in Figure 5-D (shown in bold) is one bit longer than that of Figures 5-A and 5-B.

This longer common prefix takes over the higher-order bit(s) of the portion of each key previously taken as an infix, thereby shortening the original infix. To ensure each of these shorter infixes has an unambiguous quotient of the same length as before, Diva repurposes the higher-order bits of the infix's old remainder to become the lower-order bits of its quotient. Doing so splits each old run containing re-

mainders with different higher-order bits into shorter runs. Figures 5-B and 5-D present an example whereby the run previously in Slots 0101 and 0110 is split into two runs stored in Slots 000 and 001.

Despite old remainders becoming shorter, Diva still stores the infixes of new insertions with full-length remainders. This means that we truncate newer keys to a lesser extent as their region within the key space becomes denser. However, doing so necessitates storing variable-length remainders within an Infix Store. Following our work on InfiniFilter [20], Diva does this by padding each remainder to its original slot length with unary codes consisting of 0s and a delimiting 1. For example, the two-bit remainder in Slot 100 of Figure 5-A is stored with the two-bit padding 10, whereas the full-length remainder in Slot 101 is stored with a padding of 1. Due to the delimiting 1 bit, each slot must be one bit wider, adding a memory overhead of 1 BPK.

Each split reduces the length of the remainders within one Infix Store by one bit on average. Since each Infix Store is split after roughly T insertions (doubling its expected size), one can expect half of its remainders to be of full length, a quarter (from one split ago) to be shorter by one bit, an eighth (from two splits ago) to be shorter by two bits, etc. Taking a weighted average of the remainders' FPRs based on their lengths and proportion, we obtain an FPR that increases logarithmically with data size, in line with our past work on expandable filers. One can maintain a constant FPR by elongating new remainders, similarly to InfiniFilter's Widening Regime or Aleph Filters' Predictive Regime [20, 19, 44].

After many splits, some infixes' remainders may run out of bits, preventing Diva from computing an unambiguous quotient for them. Diva duplicates such infixes to account for all possible missing quotient bits and prevent false negatives, similarly to our work on Aleph Filter [19]. For example, it duplicates the right-most remainder in Figure 5-B to create the highlighted remainders in Figure 5-D in two different runs. Duplicating infixes complicates deletions, as there may be multiple duplicates to clean up. Our past work offers several methods for tackling this complexity [19].

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.

4. EVALUATION

We evaluate Diva against existing range filters in the static setting in Section 4.1. We also conduct end-to-end experiments in the dynamic setting on top of WiredTiger [40], a popular B-Tree-based key-value store, in Section 4.2. We present further experiments in both static and dynamic settings in Sections 5.1 and 5.2 of our original paper [24].

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 an 80 KB L1 cache and a 2 MB L2 cache for each core, a 45 MB shared L3 cache, and 64 GB 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.

4.1 Static Evaluation

Baselines. We compare Diva to all existing range filters except for bloomRF [41], 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. Like prior work [59, 37, 54, 27, 41, 34, 52, 11, 18, 25, 12], we employ the following synthetic and real-world [38, 33] datasets for our evaluation over integers:

- 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.
- OSM: 200M geocoordinates from the Open Street Map. It features several dense regions in the key space.

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 $R = 1$.

Each of our workloads issue a total of 10M 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 focus on filter CPU times in this section and measure end-to-end performance in the next section.

Experiment 1: Integer FPR vs Query Size Tradeoff. The top row of Figure 6 depicts the FPR of our baselines with a memory budget of 16 BPK, while the bottom row of Figure 6 compares their query latencies. Here, we vary query size on the x-axis. 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. 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. Due to similar reasons, REncoder's FPR and speed also rapidly deteriorate with longer queries.

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.

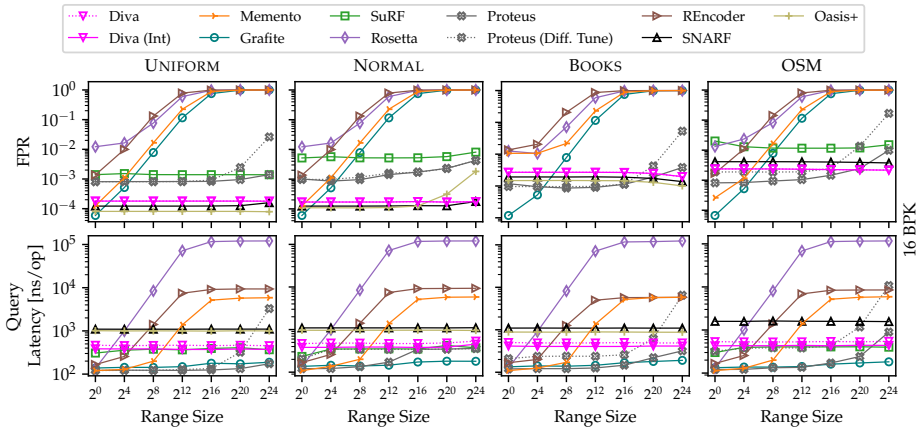

Figure 6: Diva provides the best balance between FPR and query latency for any workload across range query sizes.

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 that of Diva 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. 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 achieves all other range filtering goals.

4.2 End-to-End Evaluation

Baselines. We compare Diva to Memento filter [25] by integrating both into WiredTiger [40], a popular B-Tree-based key-value store used to serve dynamic OLTP/HTAP workloads, to conduct an end-to-end evaluation. We only consider Memento filter as a baseline, as it is the only baseline with efficient dynamic operations. By default, the memory budget of each filter is set to 16 BPK.

Dataset and Workload. We use the BOOKS dataset with 200M keys. We load a randomly chosen $\frac{1}{64}$ fraction of this dataset into WiredTiger, construct the filters, and insert the remaining keys in a random order. We issue queries of length $R = 2^{10}$ with endpoints from the same distribution and collect measurements each time the dataset size doubles.

Experiment 2: End-to-End Query Latency. Figure 7 plots end-to-end range query latency measurements, with the x-axis showing the fraction of the dataset ingested. Here, each key is associated with a 504-byte value, resulting in 512-byte key-value pairs 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 7 represents WiredTiger without a filter.

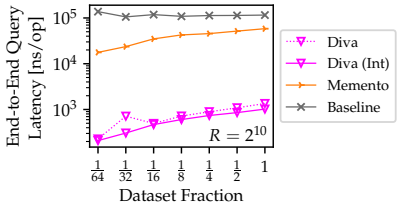

Figure 7: Diva achieves the best end-to-end query performance under dynamic workloads.

Diva speeds up WiredTiger's query processing by as much as three orders of magnitude. Moreover, since it supports variable-length queries without a deteriorating FPR, it beats Memento filter's query latency by $\approx 85\times$.

5. FUTURE DIRECTIONS

Diva opens up multiple fronts for future research. We are integrating Diva into OrcaDB, our lab's fork of RocksDB [22]. Slow range queries have long been a fundamental challenge for LSM-trees [42, 22, 59, 37, 54, 27, 41, 34, 52, 11, 18, 25, 12], and Diva offers a general-purpose path forward for real-world datasets and workloads. Beyond range queries, Diva opens new research directions in storage engines, including adaptivity [39, 55, 12] (proactively mitigating recurring false positives) and efficient range deletes [53] (marking entire key ranges as deleted while preventing access to deleted keys in storage). One remaining limitation is Diva's sensitivity to jagged data distributions, which commonly arise in natural language; however, we expect this issue can be addressed through entropy-encoding techniques [31, 58] over identical adjacent infixes.

6. ACKNOWLEDGEMENTS

This research was supported by NSERC grant #RGPIN-2023-03580.

7. REFERENCES

[1] K. Alexiou, D. Kossmann, and P.-r. Larson. Adaptive range filters for cold data: avoiding trips to siberia. *Proc. VLDB Endow.*, 6(14):1714–1725, sep 2013.

[2] Apache. Apache cassandra, 2024.

[3] Apache. Apache druid, 2024.

[4] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, STOC '99, page 295–304, New York, NY, USA, 1999. Association for Computing Machinery.

[5] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, Aug. 2002.

[6] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: how to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, July 2012.

[7] I. O. Bercea, J. B. T. Houen, and R. Pagh. Daisy bloom filters. In H. L. Bodlaender, editor, *19th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2024)*, volume 294 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, jul 1970.

[9] A. Broder and M. Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1, 11 2003.

[10] P. Celis, P.-A. Larson, and J. I. Munro. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288, 1985.

[11] G. Chen, Z. He, M. Li, and S. Luo. Oasis: An optimal disjoint segmented learned range filter. *Proc. VLDB Endow.*, 17(8):1911–1924, may 2024.

[12] Y. Chesetti, N. Eslami, H. Zhang, N. Dayan, and P. Pandey. Aeris filter: A strongly and monotonically adaptive range filter. In *Proceedings of the 2026 International Conference on Management of Data*, SIGMOD '26, page 1670–1684, New York, NY, USA, 2026. Association for Computing Machinery.

[13] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, 100(9):828–834, 1984.

[14] G. Cloud. Bigquery enterprise data warehouse, 2024.

[15] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, jun 1979.

[16] A. Conway, A. Gupta, V. Chidambaran, M. Farach-Colton, R. Spillane, A. Tai, and R. Johnson. 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, USA, 2020. USENIX Association.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[18] M. Costa, P. Ferragina, and G. Vinciguerra. Grafite: Taming adversarial queries with optimal range filters. volume 2 of *SIGMOD '24*, page 1–23, New York, NY, USA, Mar. 2024. Association for Computing Machinery.

[19] N. Dayan, I. O. Bercea, and R. Pagh. Aleph filter: To infinity in constant time. *Proc. VLDB Endow.*, 17(11):3644–3656, July 2024.

[20] N. Dayan, I. O. Bercea, P. Reviriego, and R. Pagh. Infinifilter: Expanding filters to infinity and beyond. In *Proceedings of the 2023 International Conference on Management of Data*, volume 1 of *SIGMOD '23*, New York, NY, USA, jun 2023. Association for Computing Machinery.

[21] N. Dayan and M. Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 365–378, New York, NY, USA, 2021. Association for Computing Machinery.

[22] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.

[23] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, apr 1974.

[24] N. Eslami, I. O. Bercea, and N. Dayan. Diva: Dynamic range filter for var-length keys and queries. *Proc. VLDB Endow.*, 18(11):3923–3936, July 2025.

[25] N. Eslami and N. Dayan. Memento filter: A fast, dynamic, and robust range filter. In *Proceedings of the 2025 International Conference on Management of Data*, SIGMOD '25, page 1–27, New York, NY, USA, 2025. Association for Computing Machinery.

[26] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Z. Fan, B. Ye, Z. Wang, Z. Zhong, J. Guo, Y. Wu, H. Li, T. Yang, Y. Tu, Z. Liu, and B. Cui. Enabling space-time efficient range queries with rencoder. *The VLDB Journal*, Aug 2024.

[28] R. Fano. *On the Number of Bits Required to Implement an Associative Memory.* Computation Structures Group Memo. MIT Project MAC Computer Structures Group, 1971.

[29] M. Foundation. Mariadb server: The innovative open source database, 2024.

[30] M. Goswami, A. Grønlund, K. G. Larsen, and R. Pagh. Approximate range emptiness in constant time and optimal space. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, page 769–775, USA, 2015. Society for Industrial and Applied Mathematics.

[31] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[32] H. M. Kim, N. Eslami, and N. Dayan. Zeno filter: To

infinity in tiny steps. *To Appear*, 2026.

[33] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. Sosd: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[34] E. R. Knorr, B. Lemaire, A. Lim, S. Luo, H. Zhang, S. Idreos, and M. Mitzenmacher. Proteus: A self-designing range filter. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1670–1684, New York, NY, USA, 2022. Association for Computing Machinery.

[35] D. Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley, 1973.

[36] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[37] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos. 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*, SIGMOD '20, page 2071–2086, New York, NY, USA, 2020. Association for Computing Machinery.

[38] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020.

[39] M. Mitzenmacher, S. Pontarelli, and P. Reviriego. Adaptive cuckoo filters. *ACM J. Exp. Algorithmics*, 25, Mar. 2020.

[40] MongoDB. Wiredtiger storage engine, 2024.

[41] B. Mößner, C. Riegger, A. Bernhardt, and I. Petrov. bloomrf: On performing range-queries in bloom-filters with piecewise-monotone hash functions and prefix hashing, 2022.

[42] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, Jun 1996.

[43] R. Pagh and F. F. Rodler. Cuckoo hashing. In F. M. auf der Heide, editor, *Algorithms — ESA 2001*, pages 121–133, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[44] R. Pagh, G. Segev, and U. Wieder. How to approximate a set without knowing its size in advance, 2013.

[45] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 775–787, New York, NY, USA, 2017. Association for Computing Machinery.

[46] P. Pandey, A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, and R. Johnson. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1386–1399, New York, NY, USA, 2021. Association for Computing Machinery.

[47] P. Pandey, M. Farach-Colton, N. Dayan, and H. Zhang. Beyond bloom: A tutorial on future feature-rich filters. In *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, page 636–644, New York, NY, USA, 2024. Association for Computing Machinery.

[48] F. Putze, P. Sanders, and J. Singler. Cache-, hash- and space-efficient bloom filters. In *Experimental Algorithms, 6th International Workshop, WEA 2007*, volume 4525 of *Lecture Notes in Computer Science*, pages 108–121. Springer, 2007.

[49] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 3 edition, 2002.

[50] R. Sears, M. Callaghan, and E. Brewer. Rose: compressed, log-structured replication. *Proc. VLDB Endow.*, 1(1):526–537, aug 2008.

[51] Snowflake. The snowflake ai data cloud, 2024.

[52] K. Vaidya, S. Chatterjee, E. Knorr, M. Mitzenmacher, S. Idreos, and T. Kraska. Snarf: A learning-enhanced range filter. *Proc. VLDB Endow.*, 15(8):1632–1644, apr 2022.

[53] F. Wang, D. Mo, and S. Luo. Don't forget range delete! enhancing lsm-based key-value stores with more compatible lookups and deletes, 2025.

[54] Z. Wang, Z. Zhong, J. Guo, Y. Wu, H. Li, T. Yang, Y. Tu, H. Zhang, and B. Cui. Rencoder: A space-time efficient range filter with local encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 2036–2049, 2023.

[55] R. Wen, H. McCoy, D. Tench, G. Tagliavini, M. A. Bender, A. Conway, M. Farach-Colton, R. Johnson, and P. Pandey. Adaptive quotient filters. *Proc. ACM Manag. Data*, 2(4), Sept. 2024.

[56] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

[57] X. Wu, F. Ni, and S. Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[58] J. M. Yohe. Algorithm 428: Hu-tucker minimum redundancy alphabetic coding method [z]. *Commun. ACM*, 15(5):360–362, May 1972.

[59] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 323–336, New York, NY, USA, 2018. Association for Computing Machinery.