bloomRF: On Performing Range-Queries in Bloom-Filters with Piecewise-Monotone Hash Functions and Prefix Hashing

[Extended Version]

Bernhard Mößner, Christian Riegger, Arthur Bernhardt, Ilia Petrov
Data Management Lab, Reutlingen University
[firstname].[surname]@reutlingen-university.de

ABSTRACT

We introduce bloomRF as a unified method for approximate membership testing that supports both point- and range-queries. As a first core idea, bloomRF introduces novel point- and range-queries. As a second core idea, bloomRF introduces novel prefix hashing to efficiently encode range information in the hash-code of the key itself. As a second core concept, bloomRF proposes novel piecewise-monotone hash-functions that preserve local order and support fast range-lookups with fewer memory accesses. bloomRF has near-optimal space complexity and constant query complexity. Although, bloomRF is designed for integer domains, it supports floating-points, and can serve as a multi-attribute filter. The evaluation in RocksDB and in a standalone library shows that it is more efficient and outperforms existing point-range-filters by up to 4× across a range of settings and distributions, while keeping the false-positive rate low.

1 INTRODUCTION

Modern data sets are large and grow at increasing rates [45]. To process them data-intensive systems perform massive scans that incur significant performance and resource consumption penalties. While indices may reduce the scan pressure, they are not always effective due to size or predicate selectivity concerns, or due to the high maintenance costs and workload compatibility. Filters are a class of approximate data structures that may effectively complement the workhorse data structures to reduce scans. Bloom-Filters (BFs) [3], are prominent representatives of this class that are efficient and compact. They avoid false negatives, while false positives are possible, yet the false positive rate (FPR) can be controlled by parameters such as bits/key or the number of hash functions. If a BF returns true, the search key may be present or not and the system needs to verify that through expensive scans or index-lookups. BFs only support point-lookups, i.e. is key 4711 not in the dataset.

State-of-the-Art Overview. Many algorithms and systems necessitate efficient range filtering for queries such as: are there keys between 42 and 4711 in the dataset. Classical Prefix BFs or Min/Max indices (fence pointers, ZoneMaps [34] in Neteeza or Block-Range Index [38] in PostgreSQL) can perform range-filtering, but are impractical for point queries and result in a higher FPR.

Rosetta [29], SuRF [49] and ARF [2] are some recent proposals that can handle point- and range-lookups on a unified data structure and serve as point-range filters (PRF). ARF [2] and SuRF [49] utilize tries and thus partially materialize the index at the cost of extra space. Such techniques result in increased range-filter sizes, that are reduced by trie-truncation or require tedious training/re-optimization. These yield an a posteriori/offline creation. Rosetta [29] takes a different approach, where each key is decomposed into a set of prefixes according to a dyadic interval scheme and implicit Segment-Trees [13]. The key-prefixes are maintained in a hierarchical set of BFs, one for each prefix length. Fig. 1 shows a holistic PRF positioning in the problem space according to their FPR, for different space budgets and query ranges. It is a flattened version of Fig. 11.E, where we average the FPR for 10^3-5·10^7 keys.

Problem 1: Existing point-range-filters are designed either for small or for large query ranges. Existing Point-Range-Filters are optimized for handling different query ranges sizes. While Rosetta [29] excels at relatively small ranges [2^5-2^6], SuRF [49] offers outstanding FPR for mid- and large-ranges [2^37-2^38]. On the one hand, as stated in [29] trie-truncation techniques, like the ones used in SuRF may lose effectiveness as short query ranges may fall in the scope of the truncated suffixes and thus have higher probability of being detected non-empty. On the other hand, range-lookups in Rosetta have logarithmic (sometimes linear) complexity with respect to the query range size. It may lose efficiency for longer ranges as probing a hierarchical set of BFs, implies higher memory or CPU-costs. Moreover, it is not always possible to bound the query-range size. While short-ranges seem reasonable for KV-stores, this does not apply to other systems or workloads. Besides, datatypes also have an impact: for doubles a range of 1 can be 264 in the bit representation.

Problem 2: Existing point-range-filters are offline. Existing PRF (ARF [2], SuRF [49], or Rosetta [29]) employ powerful optimizations, which require a priori the complete dataset and are therefore constructed offline. Hence, PRF cannot serve range-queries, while data is being simultaneously inserted. This limits PRF applicability
in the general case, i.e., when PRF are used standalone, when the data is too large, is streamed, or is not available in advance, etc.

The issue can be mitigated by the way PRF are integrated in larger systems. KV-stores use a main-memory delta area to absorb new data. The PRF leverage that delta and get constructed only when it gets full and thus holds the complete PRF dataset. Searching the main-memory delta is handled otherwise, e.g. through its organization (HashSkipLists or HashLinkLists in RocksDB).

While this approach is practicable in such systems, it: (a) is a property of the system integration, not of the PRF; (b) disregards the extra space for the delta; and (c) is far from optimal in general settings.

Prefix-BFs and Min/Max filters may be constructed online, but are inadequate for point-querying. Rosetta [29] may be used online per se, yet some of its optimizations require the dataset a priori.

**Problem 3: Existing Point-Range-Filters exhibit non-robust performance across a variety of workload- and data-distributions.** Existing PRF are sensitive to data and workload skew. For instance, Rosetta claims [29] to outperform SuRF by 2× on normally distributed workloads in RocksDB [30] as the suffix-truncation techniques in SuRF yield more prefix-collisions for small ranges.

**Bloom-Range-Filter (bloomRF).** We introduce bloomRF as a unified data structure, supporting approximate point- and range-membership tests that can substitute existing BFs. bloomRF operates on prefixes of keys. Firstly, bloomRF introduces novel prefix hashing (Fig. 2) to efficiently encode range information in the hash-code of the key. This information is based on certain dyadic intervals to which the key prefixes correspond. Secondly, bloomRF proposes novel piecewise-monotone hash-functions (PMHF) that preserve local order and support fast range-lookups with fewer memory accesses. PMHF place information for adjacent prefixes side by side in an overlapped bit-array such that this information can be queried with a single word access. Insertions and point-lookups (Fig. 2.A) behave much like in a BF except that in bloomRF they operate on prefixes. Range-lookups (Fig. 2.B) follow a two-path algorithm, computing the intervals along the prefix-paths for the left and the right key, and probe a tight interval-set. The area in between is probed automatically. PMHF incur fewer memory accesses, e.g. for the query [42, 43] (Fig. 2.B), H4 uses a single access to probe both points.

Our contributions are: (a) bloomRF is a unified point-range-filter that is online and can serve queries, while data is being simultaneously inserted. (b) bloomRF has constant query complexity, independent of the query range size, due to PMHF and its two-path range-lookup algorithm. bloomRF has a near-optimal space complexity, due to prefix hashing. (c) bloomRF can serve small-to-large query ranges and can handle different workload- and data-distributions. It supports integers, floating-point numbers, and can serve as a multi-attribute filter. (d) bloomRF outperforms all baselines by up to 4× across a wide range of settings. bloomRF is more efficient as it achieves better performance and FPR at lower bits/key.

**Outline.** We continue with a brief background (Sect. 2), overviewing key terms. On their basis we introduce basic bloomRF’s prefix hashing and PMHF (Sect. 3.1, 3.2), and range-lookup algorithm (Sect. 4). We present the theoretical model of basic bloomRF and compare it to the theoretical lower bound [20] in Sect. 5, 6. While basic bloomRF is simple, tuning-free, and suitable for ranges $R \leq 2^{14}$, various optimizations (Sect. 7) are needed for larger queries. We present the evaluation in Sect. 9 and conclude in Sect. 11.

### 2 BACKGROUND

We now overview well-known BFs and dyadic intervals from the perspective of bloomRF and establish several key terms.

**Bloom-Filters (BF).** Consider a set $X = \{x_1, x_2, \ldots, x_n\} \subseteq D$ of $n$ elements in a domain $D$ represented by $d$ bits, $|D| = 2^d$, e.g. $d=16$ for UINT16. We call the elements $x \in X$ keys and arbitrary elements $y \in D$ lookup keys. A BF [3] uses a bit-array of $m$ bits with positions $M = \{0, 1, \ldots, m-1\}$ and $k$ hash functions $h_i$ mapping $D$ to $M$ (i.e. $h_i : D \rightarrow M$, $i = k - 1, \ldots, 0$). Noticeably, the hash functions transform each lookup key $y \in D$ in a code of bit-array positions:

$$code(y) = \{ h_{k-1}(y), h_{k-2}(y), \ldots, h_0(y) \}.$$  

Initially all bits in the bit-array are set to zero. To insert the set of keys $X$ in a BF for each key $x \in X$ the bits of $code(x)$ are set to one. A BF performs an approximate membership test to decide if a lookup key $y \in D$ is in $X$, by checking, if all the bits of $code(y)$ are set to one. This procedure may return positive results for elements $y \notin X$, called false-positives. The ratio between false-positives and negatives is called false-positive rate.

**Dyadic Intervals (DI).** A DI is an interval whose boundaries are aligned to powers of two. They can be organized in a domain of non-negative integers represented by $d$ bits there are $2^d$ elements. For a domain represented by $d$ bits there are $2^d$ dyadic levels $\ell \in \{0, 1, \ldots, d\}$. Each DI on level $\ell + 1$ is decomposed in two DIs on level $\ell$. Thus DIs form a complete binary tree. For example, for a domain $D$ of non-negative integers with $d = 3$ bits there are $2^3 = 8$ levels: on level 0 the DIs are the points $\{0, 0\}, \{1, 1\}, \ldots, \{7, 7\}$; on level 1 are $\{0, 1\}, \{2, 3\}, \ldots, \{6, 7\}$; on level 2 are $\{0, 3\}, \{4, 7\}$; and level 3 has just $\{0, 7\}$. We show how bloomRF encodes DIs with $code(y)$ in Sect. 3.

**Prefixes.** A prefix of $y$ on level $\ell$ is the sequence of the $d - \ell$ most significant bits of $y$. These bits are accessed by a right shift by $\ell$ bits (i.e. $y >> \ell$), discarding the $\ell$ least significant bits. Thus for $\ell > t'$

$$y >> \ell = (y >> t') >> (t - \ell'),$$

i.e., a prefix of $y$ on level $\ell$ is a prefix of a prefix of $y$ on level $t'$.

Noticeably, prefixes are DIs. There is a one to one correspondence between prefixes on level $\ell$ and DIs on level $\ell$, i.e., all lookup keys $y$ with an identical prefix on level $\ell$ form a DI on level $\ell$. Consider, for instance, a domain $D$ of non-negative integers represented by $d = 3$ bits. The prefixes of a key $y = 5$ (bin $0b101$) are $1 = 0b1$ on level 2, $2 = 0b10$ on level 1 and $5 = 0b101$ on level 0. The prefixes of $y=6=0b110$ are $0b1$ on level 2, $0b11$ on level 1 and $0b110$ on level 0.

![Figure 2: (a) bloomRF relies on PMHF and prefix hashing. (b) Range lookups traverse two prefix paths, probing automatically the area in between (shaded).](image-url)
The prefixes of $y = 7 = 0b111$ are $0b1$ on level 2, $0b11$ on level 1 and $0b111$ on level 0. Finally, the prefix $0b1$ on level 1 corresponds to the DI $I = [6, 7]$ on level 1. Indeed, exactly the keys 6 and 7 share the prefix $0b11$ on level 1.

3 BLOOM-RANGE-FILTER

Based on the above concepts we now introduce the main aspects of bloomRF such as prefix hashing and PMHF.

3.1 Prefix Hashing

In a BF a lookup key $y$ corresponds to code$(y)$ of bit-array positions. Thus, we check if $y$ is in $X$ by testing if the bits at code$(y)$ are set.

The core idea of bloomRF is to encode range information in the code$(y)$ itself. To this end, we introduce code$(y)_i$, as the prefix of code$(y)$ on layer $i$. We define code$(y)_i$, as an ordered sequence of the first $k - i$ hash-functions of code$(y)$:

$$
\text{code}(y)_i = \{ h_{k-i}(y), h_{k-i-1}(y), \ldots, h_i(y) \}.
$$

Thus the prefixes code$(y)_i$ are sub-sequences of bit-array positions of code$(y)$. As an example we refer to Fig. 3A, which will be explained in detail below. Here the code of key 42 and prefixes code$(42)_i$ for all layers $i \in \{3, 2, 1, 0\}$ are shown.

When performing a lookup, our goal is to check prefixes of lookup key $y$ by testing bits at prefixes of code$(y)$. The issue at hand is that there are $d + 1$ dyadic levels, but code$(y)$ comprises $k$ hash-functions, making it impossible to encode each level. Therefore, we choose to consider only certain levels $\ell_{k-1} \geq \ell_{k-2} \geq \ldots \geq \ell_0$. (Fig. 3A exemplifies equidistant levels.)

On this premise, we define prefix hashing as a key property of bloomRF. It mandates that for each layer $i \in \{k - 1, \ldots, 1, 0\}$ a prefix of $y$ on dyadic level $\ell_i$ corresponds to a prefix of code$(y)$ on layer $i$, i.e., arbitrary lookup keys $y, y' \in D$ satisfy

$$
y >> \ell_i = y' >> \ell_i \implies \text{code}(y)_i = \text{code}(y')_i.
$$

Prefix hashing allows using code$(y)$ to test if DIs on level $\ell_i$ include keys $x \in X$. Remember that such DIs are prefixes on level $\ell_i$. By prefix hashing such DIs correspond to prefixes code$(y)_i$, which are checked by testing if the bits at code$(y)$ are set. This way, bloomRF implicitly encodes range information in the code$(y)$.

Table 1: Most important symbols and abbreviations.

| Symbol | Definition |
|--------|------------|
| $D$    | domain of size $|D| = 2^d$ elements, e.g., $2^{16}$ for UInt16 |
| $x, X, n$ | $x \in X \subseteq D$ - keys in the filter, $|X| = n$ - number of keys |
| $y \in D$ | lookup keys |
| $\ell$ | level $\ell \in \{d, \ldots, m-1\}$ - defines prefixes/dyadic intervals of keys |
| $M, m$ | $M = \{0, 1, \ldots, m-1\}$ - bit-array positions, $|M| = m$ bits |
| $h_i : D \to M$ | $h_i$ - hash function, $i \in \{k-1, \ldots, 1, 0\}$ |
| code$(y)$ | a sequence of bit-array positions |
| layer $i$ | $i \in \{k-1, \ldots, 1, 0\}$ - defines prefixes of code$(y)$ |
| code$(y)_i$ | $i$th prefix of code$(y)$ on layer $i$ |
| $\Delta$ | distance between levels, $\ell_{i+1} - \ell_i$ |
| $k$ | $k = \lceil d / \Delta \rceil$ - number of hash functions $h_i$ |
| $n$ | number of keys |
| $\ell_0$ | $\ell_0 = \ell_{k-1}$ |

Arbitrary hash-functions can be used for prefix hashing: eq. (4) is satisfied, if hash-functions of code$(y)_i$ only operate on prefixes on level $\geq \ell_i$. Using (2) we achieve this by:

$$
\text{code}(y) = (h_{k-1}(y >> \ell_{k-1}), \ldots, h_1(y >> \ell_1), h_0(y >> \ell_0)).
$$

Finally, we have to determine the levels $\ell_i$ and the number of hash-functions $k$. A natural choice are equidistant levels. Thus we define a distance $\Delta$ between two adjacent levels and set $\ell_i = i \Delta$. Then the number of hash-functions is given by $\lceil (d+1) / \Delta \rceil$. Depending on the number of keys top layers saturate (Sect. 7). We omit such levels and therefore use $k = \lceil (d - \log_2 n) / \Delta \rceil$ hash-functions.

**Introductory example.** Consider a set $X = \{42, 1414, 50000\}$ (Fig. 3B) of $n = 3$ keys in a domain $D$ with $d = 16$ bits. We use 10 bits/key and $\Delta = 4$, yielding a bit-array with $m = 10|X| = 30$ bits and $k = \lceil (d - \log_2 n) / \Delta \rceil = 4$ hash-functions. For hashing we use multiplication with prime numbers $a_i$ and $b_i$, followed by mod $m$ to determine a position in $M$, i.e., $h_i(x) = (a_i + b_i \cdot (x >> \ell_i))$ mod $m$.

Figure 3A shows layers $i$, levels $\ell_i$ and values for the hash-functions. For example, key 42 has a code$(2, 3, 19, 19)$ of positions in the bit-array. Inserting all keys of $X$ leads to a bit-array where the bits 2, 3, 8, 10, 17, 18, 19, 20 and 21 are set to one and all others are zero (Fig. 3B). Since we choose $\Delta = 4$ bit shifts by levels $\ell_i = 4i$ can be displayed in hexadecimal representation. For example, for key 42 (hex 0x002A) the prefix on level $\ell_3 = 12$ is 0x0, on level $\ell_2 = 8$ is 0x00, on level $\ell_1 = 4$ is 0x002 and on level $\ell_0 = 0$ is 0x002A.

Remember that due to prefix hashing (eq. 4), a prefix of $y$ on a certain dyadic level $\ell_i$ corresponds to a code$(y)_i$, i.e. a prefix of code$(y)$ on layer $i$. Thus, keys 42 and 43 = 0x002B have the same prefix on level $\ell_1 = 4$, $42 >> \ell_1 = 0x002 = 43 >> \ell_1$ and code$(42)_1 = (2, 3, 19) = \text{code}(43)_1$ as required by eq.(4).
Recall also that prefix hashing allows us to use code(y) to test if DIs on level ℓ include keys x ∈ X by testing the positions of code(y), since these DIs are in fact prefixes of y on level ℓ corresponding to code(y). All y ∈ [32, 47] = \{0x0020, 0x002F\} have the same prefix 0x0020 on level ℓ and therefore the same prefix code(y)_1 = (2, 3, 19) on layer 1 (Fig. 3.A,B). Thus we check positions (2, 3, 19) to test, if a key x ∈ X is included in I. In this example, the answer is positive, which is true since indeed 42 ∈ I. All lookup keys in [48, 63] = \{0x0030, 0x003F\} have (2, 3, 26) as code prefix on layer 1 (e.g. 48). Checking positions (2, 3, 26) results negative, since 26 is set to zero, and here indeed X ∩ [48, 63] = ∅.

A DI I on level ℓ, ℓ₁ < ℓ < ℓ₄₊₁, can be decomposed in up to 2Δ⁻¹ intervals on level ℓ₁, thus I can be tested via these DIs on level ℓ₁. For example I = [42, 43] on level 1 can be checked by testing [42, 42] and [43, 43] on level 0. The corresponding prefixes code(y)₁ only differ in the hash-function on layer 0. While 42 and 43 are adjacent, the positions of the hash-functions h₀(42) = 19 and h₀(43) = 0 are not. Clearly, a hash-function on layer i does not preserve the order of the prefixes y >> ℓᵢ. We tackle this in Sect. 3.2.

Prefix hashing is hierarchical. DIs are arranged hierarchically by inclusion. Prefixes are DIs and follow the same hierarchy – eq. (2). The prefixes code(y) also inherit that hierarchy by (4), hence bloomRF uses hierarchical hashing. Thus, by testing key y ∈ D, all DIs on levels ℓᵢ including y are automatically tested.

For example, when testing key y = 43 with code(y) = (2, 3, 19, 0), the following prefixes are checked: prefix (2, 3, 19) = code(43)₁ corresponding to DI [32, 47], prefix (2, 3) = code(43)₂ corresponding to [0, 253] and prefix (2) = code(43)₃ corresponding to [0, 4095].

Space Efficiency. bloomRF has a near-optimal space efficiency (Sect. 6) since code(y) itself contains range information in terms of corresponding DIs. In particular, prefix hashing encodes the difference between any two consecutive prefixes of a key in a single position as a single bit. For example, the difference between prefixes 0x0020 on level 4 and 0x002A on level 0 of key 42 is encoded in a single bit.

3.2 Piecewise-Monotone Hash-Functions

Although prefix hashing results in near-optimal space consumption the order of prefixes y >> ℓᵢ is not preserved by hash-function hᵢ, increasing significantly the query time of intervals I on level ℓ, ℓ₁ < ℓ < ℓ₄₊₁. To this end, and as a second core idea, bloomRF introduces piecewise-monotone hash-functions (PMHF) that are locally order preserving and place corresponding bits side by side in the bit-array. This allows checking all bits of DIs of I on level ℓᵢ with hash-function hᵢ, in a single memory access, yielding better performance.

Noticeably, arbitrary hash-functions hᵢ can be easily extended to satisfy this property and remain compute-efficient: \( \left( hᵢ(x >> (Δ - 1)) \mod \frac{m}{2^Δ} \right) << (Δ - 1) + x \& (2^Δ - 1) \)

The new h must preserve the order of the least significant Δ − 1 bits of a prefix. Therefore, x is right-shifted by Δ − 1 bits, such that h only operates on the rest. The bit-array is accessed in words of size 2Δ⁻¹, therefore m must be a multiple of 2Δ⁻¹. In fact, the bit-array can be viewed as an array of m/2Δ⁻¹ words. The modulo operation determines a position in this word-array. Finally a left-shift by Δ − 1 bits, yields the position of the word in the bit-array. To keep the order the least significant Δ − 1 bits are added to the position. These bits are extracted with a bitwise AND (&) with the mask 2Δ − 1.

| keys | MH₀ | MH₁ | MH₂ | MH₃ |
|------|-----|-----|-----|-----|
| 42   | 16  | 24  | 10  | 2   |
| 1414 | 16  | 29  | 0   | 3   |
| 50000 | 28 | 27  | 29  | 8   |

Combining with prefix hashing we get

\[ MHᵢ(x) = \left( \left( hᵢ(x >> (Δ - 1)) \mod \frac{m}{2^Δ} \right) << (Δ - 1) \right) + (x >> ℓᵢ) \& (2^Δ - 1), \]

which we call piecewise-monotone hash-functions.

For example, consider again the set X = [42, 1414, 50000] (Fig. 4) for a domain with d = 16 bits. Again, we use Δ = 4 and k = 4 hash-functions. Here we set m = 32 since m must be a multiple of 2Δ⁻¹ = 8, thus we use approximately 10 bits per key. Again we use the hash-functions hᵢ(x) = aᵢ + bᵢx as in the previous example. Figure 4 shows the codes of keys y ∈ X using corresponding PMHF. Inserting all keys of X leads to a bit-array where the bits 0, 2, 8, 10, 16, 24, 27, 28, 29 and 30 are set to one and all others are zero.

To test the DI [42, 43] the codes (16, 24, 10, 2) and (16, 24, 10, 3) have to be checked. Both have the same prefix 16, 24 and 10 on levels 3 to 1 and the positions 2 and 3 on level 0 lie side by side. Thus on level 0 both can be tested with a single word access. The positions 2 and 3 on level 0 can be described by the bit-mask b = 0b00110000 and a word access on the first byte of the bit-array yields w = 0b10100000. The bits at 16, 24 and 10 are set b & w ≠ 0, thus a positive answer.

For interval [44, 47] all codes (16, 24, 10, 4), ..., (16, 24, 10, 7) have to be tested. They have the same prefix 16, 24 and 10 on levels 3 to 1 and positions 4 to 7 on level 0 lie side by side and can be tested with a single word access. Positions 4 to 7 on level 0 correspond to the bit-mask b = 0b00001111 and as above w = 0b10100000. The bits at 16, 24 and 10 are set, but b & w = 0, thus the negative answer.

Random Scatter. We now consider the scatter of PMHF as they should preserve local order, but also distribute words randomly over the bit-array. We compare bloomRF against the standard BF in RocksDB. For a fair comparison we use 2M keys and 10 bits/key, for which BF's have 10·ln2 = 6.93 hash functions, floored to 6 in RocksDB, as basic bloomRF with 64-bit words (Δ = 7) uses \( k = \lceil (d - \log₂n)/Δ \rceil = 6 \) PMHF. First, we investigate how well PMHF scatter words. To this end (Fig. 5.A), we measure how many times words (x-axis) of different layers are overlaid in a bit-array element for different data distributions. As the relative frequencies are mostly flat curves (the strong zipfian skew affects layers 2 and 3) we conclude that PMHF scatter randomly at word granularity for normal, zipfian and uniform data distributions. Second, we consider the scatter/overlying of bits within words, by looking holistically at the bit-array. To this end, we compare the length of 0-bit runs (Fig. 5.B), as well as the bit-distance between two consecutive 0-bit runs (Fig. 5.C), for both BF and bloomRF and zipfian, normal, uniform
data distributions. The 0-bit runs are a relevant metric as they indicate bit areas that have never been set. Thus, significant differences would indicate randomization issues. Clearly, both bit-arrays are in similar states. Intuition: bloomRF is not worse than BFs, with view of the scatter of words and their overlaying in bit array elements for common data distributions like zipfian, normal or uniform. PMHF randomize words sufficiently. These insights are substantiated by the relative point FPR of bloomRF vs BF in the evaluation (Fig. 10).

**Degenerate data distributions and PMHF.** There are rare cases of degenerate data distributions, where PMHF may become inefficient. The core observation is that certain bits of a key determine the bit position in a word of the bit-array, since PMHF are piecewise monotone. In basic bloomRF with distance among levels $\Delta = 7$, for example, if all bits 0-5, 7-12, 14-19, ..., $i\Delta$ to $(i+1)\Delta = 2$, ..., contain the value $\lambda \in 0, 1, \ldots, n = 2^{\Delta - 1} - 1$, then every PMHF sets bit $\lambda$ in its word. A data distribution that generates such keys with high probability can be defined by counting the number of appearances of these bits in a key $c_\lambda = \sum_{i=0}^{k-1} (x \gg i\Delta \& w) = \lambda$ \!+\! 1

![Figure 5](image_url)  
**Figure 5:** (a) Random scatter over bloomRF layers; (b), (c) Comparison of the bit-array scatter to a BF.

and finally normalizing $p_x = c_x / \sum y c_y$. bloomRF can handle such cases. We can employ slightly different hash functions, which permute the bits in the word. For instance, on each layer, we can apply the original PMHF on half of the keys, while the other half is tackled by a PMHF that writes the words in reverse order.

**Vertical PMHF and error-correction.** The hierarchical structure of PMHF allows a new interpretation of hashing in bloomRF, which can also be transferred to BFs. DIs on a level $\ell$ correspond to prefixes code$(y)$. With PMHF only the hash-function on layer $i$ operates on $y \gg \ell$, all others only on prefixes of $y >> \ell$. Thus one hash-function is primarily responsible for level $\ell$, namely $MH_i$. Therefore hashing in bloomRF is hierarchical with a separate PMHF for each layer (Fig. 6.C). The hash-functions on higher layers are used for error-correction. In the example (Fig. 4) the DI $I = \{416, 431\}$ as level $\ell = 4$ is represented by the prefix $(16, 25, 2)$. We have $I \cap X = \emptyset$, but hash-function $MH_0$, which is primarily responsible for layer 1, yields an error, since the bit at position 2 is set to 1. Hash-function $MH_2$ checks bit 25 of the bit-array, which is zero. Thus, we get a negative as the error of $MH_0$ is corrected by $MH_2$. BFs can be viewed in the same way: Keys are represented by one hash-function while the others are used for error-correction. Since hashing in BFs is planar and not hierarchical (Fig. 6.A,B), none of the hash-functions is preferred for representing keys or error-correction.

### 4 bloomRF OPERATIONS

We now provide a detailed description of the main operations in bloomRF such as insertion, point- or range-queries.

**Insertion and Point-lookup.** To insert a key $x \in X$ the code$(x)$ of bit-array positions is computed via piecewise-monotone hash-functions $MH_i(x)$, $i=k-1, \ldots, 1, 0$, and the corresponding bits in the bit-array are set to one. To test, if a lookup key $y \in D$ is in $X$, the code$(y)$ of bit-array positions is computed via PMHF $MH_i(y)$, $i=k-1, \ldots, 1, 0$, and bloomRF checks if all corresponding bits in the bit-array are set. For these operations bloomRF behaves like a regular BF, except that the hash-functions are replaced by PMHF.

**Range-lookup.** Range-queries in bloomRF are based on the decomposition of arbitrary lookup intervals $I$ in DIs. Hierarchical prefix hashing allows testing all these intervals together in one pass. Additional DIs covering $I$ are automatically checked. Next, we explain, which DIs are considered for an interval $I$. Upon that we elaborate on the algorithm that computes and tests all these DIs.
Decomposition in DIs. For an arbitrary interval \( I \) the DIs to be considered are defined in a two-path algorithm, one for the left and one for the right bound of \( I \). Starting from the top level, \( J_0 = D \) is a covering of \( I \). We proceed recursively. Suppose \( J_{l+1}, l < d \), is a covering of \( I \). Then we decompose \( J_{l+1} = K \cup K' \) in two DIs and set \( J_l \) as the one covering \( I \). If \( I \) is not covered by a single DI the path of covering intervals splits in two, a left \( J'_l = K \) and a right \( J''_l = K' \).

We describe only DIs considered for the right path as the right one is mirror-inverted. Suppose \( J'_{l+1}, l < d \), is a covering of the left bound of \( I \). We decompose \( J'_{l+1} = K \cup K' \) in two DIs. If \( K \cap I \neq \emptyset \), we know that \( K' \subseteq I \), thus \( J''_l = K \) belongs to the decomposition of \( I \) in DIs and \( J''_l = K' \) is a covering of the left bound of \( I \). Else, if \( K' \cap I \neq K' \), then \( K' \) covers the left bound of \( I \), thus we set \( J''_l = K' \). Otherwise the decomposition of the left side is complete and we set \( J''_l = K' \). As example we look at the considered DIs for \( I = [45, 60], d = 16 \) (Fig. 7), from level 16 to 5 \( I \) is covered by single DIs. On the top levels left and right path coincide. On level 4 the paths split with a covering of \( I \) by two DIs. On level 3 the first DI, \( J' \), of the decomposition of \( I \) is calculated. Finally, \( I = [45, 46] \cup [46, 47] \cup [48, 55] \cup [56, 59] \cup [60, 60] \).

Next, we map the above intervals onto the layers. A covering \( J'_{l+1} \), on level \( l \) is split into several DIs on the levels \( l \leq l < l + \Delta \). These can be represented by at most \( 2^\Delta \) DIs on level \( l \). Some of them are coverings, while others belong to the decomposition of \( I \). All DIs of the decomposition have to be tested. The covering should be as tight as possible, thus we take the intersection of the intervals \( J'_{l+1} \) which is \( J'_{l} \). Using PMHF \( 2^{3-1} \) DIs an level \( l \) lay side by side in the bit-array, thus all DIs to be tested can be checked with at most two word-accesses. The same applies to the right path, thus checks require at most four word-accesses per layer. In our example the decomposition of \( I = [45, 60] \) results in intervals \( [45, 47] = J'_0 \cup J'_1, [48, 55] = J''_0 \) and \( [56, 60] = J''_3 \cup J''_4 \) to be probed. Thereby, the coverings are automatically checked: \( J'_0 = J_{16} = [0, 65535] \) on level 16, \( J'_2 = J_{12} = [0, 4095] \) on level 12, \( J''_0 = J_8 = [0, 255] \) on level 8 and on level 4 a covering with \( J'_{l+1} = [32, 47] \) and \( J''_{l+1} = [48, 63] \).

**Detailed algorithm.** We now describe how bloomRF performs range queries for arbitrary intervals \( I = [l_{\text{key}}, r_{\text{key}}] \) (Algorithm 1). The main loop iterates over the layers, with \( i \) being the current layer, which ranges from the top \( i = k - 1 \) (Line 2) down to the bottom \( i = 0 \). On layer \( i \) several tests are performed using PMHF \( M_{HI} \). The algorithm checks coverings \( J_l \) and intervals of the decomposition of \( I \) (unions of intervals \( I_l \)). The variable \( checks \) (L. 3) contains the data for these tests: \( check.l_{\text{key}}, check.r_{\text{key}} \) and \( check.is.coving \) are the results of the checks over the checks of layers \( i \) (L. 6). For a covering \( I \), only a single bit must be tested. A true positive can only occur, if all keys \( x \) of \( I \) are covered. As an early stop condition no further layers have to be checked. To test an interval of the decomposition of \( I \) (L. 11), the algorithm has to test several bits. We compute a bit-mask \( b \) and since we use PMHF, all necessary bits are read in a single-word bit-array access. If a bitwise AND yields a value \( \neq 0 \), then the filter claims the existence of a key \( x \) in \( I \) and returns a positive answer. Otherwise, if all intervals of the decomposition get excluded, \( checks \) gets empty, yielding a negative answer.

### Algorithm 1: bloomRF Range-Lookup

```python
def RangeLookup(I_key, r_key):
    init_checks[I_key, r_key, i] =
    while checks ≠ Ø do
        new_checks ← Ø;
        foreach check ∈ checks do
            if check.is.coving then
                filter.bit_access(MH[check.l_key]) = 1 then
                    Expand check to layer i−1 and append to
                    new_checks;
                else
                    filter.bit_access(MH[check.r_key]) = 1 then
                        b ← filter.bit_access(MH[check.r_key]);
                        w ← filter.bit_access(MH[check.l_key]);
                        if b & w ≠ 0 then
                            return true;
                        end if
                        let i ← i−1;
                        checks ← new_checks;
            end if
        end foreach
    end while
    return false;
```

5 THEORETICAL MODEL

We now analyze space and time complexity of bloomRF, and begin with an FPR estimate for range-queries. As shown in Sect. 4, for an interval \( I \) several DIs are considered. There are several special cases, depending on the position of \( I \). All have in common that they (phase 1, Fig. 7) start with a sequence of \( J_l \) coverings by single DIs \( J_l \), \( j = k − 1, \ldots, k − i_l + 1 \), which then (phase 2) split up in \( J_{l+1} \) coverings by two DIs \( J'_{l+1} \cup J''_{l+1} \), \( j = k − i_l + 1, \ldots, k − i_l + i_{l+1} + 1 \). Since here all intervals are coverings only single bits have to be checked. Let \( p \) be the probability that a bit in the bit-array is set to zero. A false positive can only occur, if all DIs of phase 1 yield positive and all DIs of left side of phase 2 yield positive, while on the right side an arbitrary combination is possible, or vice versa.
We estimate the FPR $\epsilon$ by eq. (5).

$$\epsilon \leq (1 - p)^{i_1+i_2} \left( \sum_{i=0}^{i_2-1} \binom{i_2}{i} p^{i_1} (1 - p)^i \right) + (1 - p)^{i_2}$$

$$\leq 2(1 - p)^{i_1+i_2}$$

The DIs on level $i_1 = i \lambda$ have length $2^{i \lambda}$. Thus, an arbitrary interval $I$ of length $|I| \leq 2^{i \lambda}$ is covered by at most two DIs on level $i_1$ and therefore in phase $2$ at least layer $i$ is reached, i.e. $i_1+i_2 \geq k-i$. Thus $\epsilon \leq 2(1 - p)^{k-i}$ and therefore $\epsilon \leq 2(1 - p)^{k-\log_2(|I|)/\lambda}$.

It remains to estimate the probability $p$ that a bit in the bit-array is set to zero. For BFs the assumption of perfect random hash-functions leads to a probability of $1/m$ of bits being set and therefore the standard estimate [3] yields

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$ 

We model the influence of the data distribution on PMHF by introducing a constant $C$, such that $p = (1 - C/m)^{kn} \approx e^{-C kn/m}$. Our experiments (PMHF random scatter, Fig. 5) suggest that $C = 1$ for common distributions such as uniform, normal and zipfian. Summary: for range lookups with max. query range $R$, such that $|I| \leq R=2^{i \lambda}$, and common distributions bloomRF has an FPR of

$$\epsilon \leq 2 \left(1 - e^{-kn/m}\right)^{\log_2(R)/\lambda},$$

where $k = \lfloor (d - \log_2|I|)/\lambda \rfloor$ (Sect. 3.1).

For point-queries bloomRF behaves like a BF, except that $k$, the number of hash-functions, is not a free parameter. Thus for common distributions the point FPR is $\epsilon \approx (1 - e^{-kn/m})^k$.

For time complexity we consider the operations in Sect. 4. The insertion of keys and point lookups requires evaluating of $k$ hash-functions, thus both have constant time $O(k)$. Range-queries are handled by algorithm 1. There are two loops: The outer loop (Line 4) iterates over the layers and the inner loop (Line 6) over the checks on layer $i$. Since there are $k$ layers and at most 4 word-accesses per layer, range-queries require also constant time $O(k)$. Notably, the query-time is independent of the size of the query-interval $I$.

6 COMPARISON: SPACE/TIME COMPLEXITY

With the theoretical model in place, we now compare bloomRF’s space and time complexity to Rosetta’s model, and to the theoretical lower bounds for point[7] and range-queries[20].

Space complexity. We estimate the space $m$ needed by bloomRF to achieve a given FPR $\epsilon$ by solving eq. (6) for $m$.

[7] has shown that any structure which answers point-queries with FPR $\epsilon$, needs at least $m \geq n \log_2(1/\epsilon)$ space. [20] shows that any structure, answering range-queries of range-size $R$ with FPR $\epsilon$, necessitates at least $m \geq n \log_2(R^{1-O(\epsilon)} / \epsilon) = O(n)$ space. [20] gives a family of lower bounds with a free parameter, $\gamma > 1$:

$$m \geq n \log_2 \left( \frac{R^{1-\gamma \epsilon}}{\epsilon} \right) + n \log_2 \left( \frac{1 - 4mR}{2^\gamma} \right) \left( 1 - \frac{1}{2^\gamma} \right).$$

The lower bound is therefore the point-wise maximum of these bounds. We can determine $\gamma$ as a function of $\epsilon$ to achieve this maximum, leading to a single curve for the lower bound (Fig. 8).

Furthermore we compare with Rosetta [29], which has four variants for point-range filters and the variant (F) first-cut solution in analyzed in terms of space complexity. The first-cut solution uses a BF for each level of DIs, with FPR $\epsilon$ on the bottom level and $1/(2 - \epsilon)$ on all others. In [29] it is stated, that (F) achieves an FPR of $\epsilon$ for range-queries of intervals up to length $R$ using $m \geq \log_2(\epsilon) n \log_2(R/\epsilon)$. Figure 8 shows the estimates for bloomRF, lower bounds and Rosetta for point-queries (left) and range-queries (right), a domain of $2^{16}$ bit integers. For point-queries bloomRF and Rosetta are close, but bloomRF always uses a little bit more memory except at one FPR. The reason is, that for bloomRF the number of hash-functions is determined by the datatype’s domain size, $k = \lfloor (d - \log_2n)/\lambda \rfloor$, such that the for BFs known optimal choice of $k = \ln(2) \cdot m/n$ cannot be used. For range-queries the distance between Rosetta an the lower bound is given by a near-constant factor. bloomRF improves over Rosetta, especially with larger $R$, i.e., larger $\lambda$, and gets closer to the theoretical lower bound. The foundation for space savings in bloomRF is prefix hashing, yielding a near space-optimal PRF.

Time-Complexity. Range queries in bloomRF are answered in constant time $O(k)$, independent of the range size $R$.

Rosetta uses a BF for every dyadic level, but all levels except to lowest have larger FPRs, e.g. $1/(2 - \epsilon)$ in the first-cut solution (F). To improve FPR a process of doubling is applied. If a DI on level $\ell$ yields a positive result, the two DIs on the level below are tested. In the worst case, this may yield query-time linear in $R$. According to [29], (F) has avg. query-time $O(\log_2(R)/\theta^2)$ for intervals $\leq R$.

Two more variants also have log. avg. query-time $O(\log_2(R)/\theta^2)$ [29]. An optimized variant (O), where as in (F) a BF is used for each level, but the FPRs $\epsilon_i$ on the levels are adjusted to how often intervals are queried, and a variable-level variant (V) similar to (O), but using different weights, pushing more bits to lower levels, improving FPR of lower at cost of higher FPR of the middle and top levels. Finally a single-level variant (S) is suggested, where only a single level of Rosetta is used. Here range-queries are answered by testing every element of an interval, yielding linear time [29].

Space efficiency, FPR and Query-range size. Rosetta is designed for relatively small query ranges and KV-stores, and covers its design space well. Consider Rosetta (F) due to its logarithmic time complexity for longer ranges and its space requirement of $\log_2(\epsilon) \cdot n \log_2(R/\epsilon)$ bits/key to achieve an FPR $\epsilon$ for range-queries $R \approx 2^\ell$.

Figure 8: Comparison of bloomRF to Rosetta and the theoretical lower-bound [7, 20] for (a) point and (b) range lookups.
Variable Distance Between Levels. While large distance between levels typically employs a single PMHF per layer; (ii) mid-layers with small word sizes (e.g., 8-bit or smaller) are stored in a separate and sparser segment with replicating hash-functions besides the PMHFs to lower the error-rates; (iii) a mid-upper layer is stored exactly in an exclusive segment; (iv) the top layers are discarded as they saturate.

Extended Model. We now describe a generalized bloomRF model to evaluate the FPR given the above optimizations. According to the filter the DIs on each level $t$ can be classified as: (a) empty ($t_n$); or (b) non-empty and include a key ($t'_p$); or (c) non-empty and do not include a key ($t'_n$). Therefore, the FPR on level $t$ is $\text{fpr}_t = \text{fpr}_t' (\text{fpr}_t + t_n)$. The number of true positives $t_p$ on each level can be derived from the distribution of the keys. For example, assuming uniform distribution, the $n$ keys lie in approximately $n$ DIs on large enough levels. Hence, the estimate: $t_p = \min(n, 2^{d_1})$. The numbers $t_p$, $t_n$, and $t_n'$ are estimated by recursion on the levels $l_i = \sum_{j=0}^{i-1} \Lambda_j$ corresponding to layers $i$. We assume level $k_0$ is stored exactly, therefore $t_p = 0$ and $t_n = 2^{d_0} - t_p$, $t = d, d-1, \ldots, k_0$.

Suppose we have computed $t_p$, $t_n$, and $t_n'$ corresponding to layer $i$. For the layer below, i.e., layer $i-1$, we consider the levels $l_i = l_{i-1} + 1, l_{i-1} - 2, \ldots, l_{i-1}$. A DI on a level splits in two DIs on the underlying level. Therefore, each DI on level $l_i$ includes $2^{d_i-1}$ DIs on level $l_{i-1}$. If $l_i$ is false negative, all $2^{d_i-1}$ intervals are also true negatives. If $l_i$ is false or true positive, then some of the $2^{d_i-1}$ intervals can be false positive. Since $t_p$ are true positive the number of potentially false positive intervals on level $l_i$ is $2^{d_i-1} (t_p t_n' + t_p t_n') - t_p$. For these intervals the corresponding bits in segment $j_{i-1}$ of the bit-array will be probed. Let $p$ be the probability that such a bit is set to zero. Analogous to section 5 we use the estimate $p = (1 - C/m_{j_{i-1}})k^{-n}$, where $C$ models the influence of the data distribution and here $k' = \sum_{j_{i-1}=j_{i-1}}^r v$ is the number of hash functions of segment $j_{i-1}$. For common distributions such as uniform, normal and zipfian we can assume $C = 1$ (PMHF random scatter, Fig. 5).

For each potentially false positive DI on level $l_i$ one or more bits will eventually be probed, depending on the number of hash-functions $r_{i-1}$ and layer $i - 1$. Let $p'$ be the probability that such a probe yields true, then $t_p' = p p_{i-1}$ and $t_n = 2^{d_i-1} (t_n' + 1 - p') p_{i-1}$. The probability $p'$ can be computed by combinatorial formulas. For example for DIs on level $l_{i-1}$ single bits are checked for each hash-function. Hence, $p' = (1 - p')^{r_{i-1}}$. For DIs on level $l_{i-1} + 1$ two bits must be checked. For $r_{i-1} = 1$ we get $p' = 2p(1-p) + (1-p)^2$, for $r_{i-1} = 2$ we get $p' = 2p^2(1-p)^2 + 4p(1-p) + (1-p)^3$, etc.

We apply the FPR model to our example in Section 3. The size of the domain is $|D| = 2^{d_3} = 16384$, $d = 16$, and we store $n = 3$ keys. We assume $\Delta = 4$ and thus $k = [d - \log_2 n]/\Delta = 4$, or $\Delta = (4, 4, 4, 4)$. We also assume one hash function per layer and a single shared segment, which is the bit-array with $m_1 = 32$ bits. Level $l_0 = d$ is the interval $[0, 16384]$, which is set when the first key is inserted. Thus, we assume it is stored exactly (it is a single bit, which is actually unused). In our model we estimate $p = 0.683$, where the relative frequency of bits set to 0 is $22/32 \approx 0.688$. As estimate for the FPR on each level we get $\text{fpr} = (0, 0.95, 0.78, 0.53, 0.32, 0.27, \ldots, 0.04, 0.03, 0.02, 0.01)$. So for point-queries we expect an FPR of 0.01 (1%) and for the intervals $[0, 32767], [32768, 65535]$ an FPR of 0.95 (95%).
Tuning Advisor. Given standard parameters like the number of keys \( n \), the memory budget \( m \) and considering an (approx. max.) query range size \( R \), the tuning advisor computes and selects an appropriate bloomRF configuration, comprising the parameters: vector \( \Delta = (\Delta_k, \Delta_{k-1}, \ldots, \Delta_1) \in \mathbb{N}^k \), number of hash-functions \( r_i \) and the assigned memory segment \( j_i \) per layer, while using three segments \((m_1, m_2, m_3)\). Now we describe the procedure.

First, we determine the exact level by means of a heuristic: its size should be \( \leq 60\% \) of the memory budget \( m \). Thus, \( \ell_x = \min\{\ell | 2^\ell - 1 < 0.6m\} \). The advisor examines multiple exact level candidates. For the sake of simplicity, here we consider only: \( \ell_x \) and \( \ell_x + 1 \).

The position of the exact layer determines the vector \( \Delta \), the number of hash-functions and the assigned memory segments by the following heuristics: For the lower layers we use \( \Delta_j = 7 \), which leads to a word-size of 64 bit and is as large as possible. The mid layers are the transition region between lower layers and exact level. Starting from the lower layers we reduce \( \Delta_l \) to match the exact layer. As an example we consider \( n = 50 \cdot 10^6 \) keys with 14 bits/key in a domain with \( d = 64 \) bit. The lowest level with \( 2^{\ell - 1} < 0.6m \) is 36. For the bottom levels we start with \( \Delta_l = 7 \) and then reduce \( \Delta_l \) to match 36. This results in a vector \( \Delta = (2, 2, 4, 7, 7, 7, 7) \), which sums up to 36. We aim for as few replicated hash-functions as possible, therefore we use only one hash-function per layer, and only on the highest layer \( r \), e.g., \( r = (2, 1, 1, 1, 1, 1, 1) \). The heuristic applied here is: the closer we are to the exact layer, the higher the precision has to be, and therefore we employ smaller \( \Delta_l \) and use replicated hash-functions (but as few as possible). Finally memory segment \( m_1 = 2^{\ell - 1} \) is used for the exact, \( m_2 \) for the middle and \( m_3 \) for the bottom layers, e.g., \( j = (2, 2, 2, 3, 3, 3, 3) \).

Second, with all other parameters defined by the above heuristics, for a given exact level it remains to determine \( m_2 \), since \( m_1 + m_2 + m_3 = m \). The goal is to minimize the FPR for range-queries of size up to \( R \). Let \( \text{fprr}_m = \max_{l, x} \log_2(R) \), \( \text{fprr}_p \) be the maximum FPR of DIs used for ranges \( \leq R \). Since the largest FPR-rates result from mid-top levels (= large intervals), small intervals (= bottom levels) are underprioritized. Thus we also consider \( \text{fprr}_p \) = \( \text{fprr}_p \), i.e. point-query FPR. The advisor makes a trade-off between lowering the range-query FPR (\( \text{fprr}_m \)) and the point-query FPR (\( \text{fprr}_p \)), as decreasing \( \text{fprr}_m \) might imply higher \( \text{fprr}_p \). To this end, we define and minimize the weighted squared norm \( \text{fprr}_m^2 \) \( = \text{fprr}_m^2 + C^2 \text{fprr}_p^2 \). It always holds \( \text{fprr}_p \leq \text{fprr}_m \). As compensation we can increase \( C \) to weight point-queries stronger. We determine all parameters for our exact level candidates \( \ell_x \) and \( \ell_x - 1 \) and select the configuration with \( \min \text{fprr}_m \). Finally, we select the configuration with \( \min \text{fprr}_p \). The auto-tuning process is inexpensive, with computation times of \( \text{~8ms} \).

Figure 7?L shows an example. For \( n = 500M \) keys, 16 bits/key and query range \( |R| = 10^{10} \), the advisor examines \( \ell_x = 27 \) (red curve) and \( \ell_x = 28 \) (blue curve). The minimum \( \text{fprr}_m \) is marked on each curve and the blue one is chosen. Thus, we estimate an FPR of \(-0.5\% \) for point-queries and \(-5\% \) for dyadic ranges up to size \(|R| \).

8 DATATYPE SUPPORT
Variable-length strings. The string support in bloomRF resembles SuRF-Hash [49] and considers the first seven characters in the seven most-significant bytes. In addition, for point queries it computes a one-byte hash-code of the rest of the string, including the length, and places it in the least significant byte. This way bloomRF achieves a UINT64 representation of variable-length-strings.

Floating-Point Numbers. Floating-point numbers are represented with \( q \) bits for the mantissa \( \mu \), \( r \) bits for the exponent \( e \) and one bit for the sign \( s \). For a bit combination \( x \) the represented value is \( f(x) = s \cdot \mu \cdot 2^e \). The bit combinations \( x \) are ordered as binary numbers. Since floats are signed, this order is reversed for negative numbers and is therefore lost. To this end, we use a map \( \varphi \) with \( \varphi(x) = x + 2^{r+1}r \). For point-queries and ~values \( \varphi(x) = \mathbb{F} \) (bitwise inverse) otherwise, which is a monotone coding, i.e., \( \varphi(x) < \varphi(y) \Leftrightarrow f(x) < f(y) \). For all operations, we use \( \varphi(x) \) instead of \( x \). To insert \( x \) into bloomRF, we insert \( \varphi(x) \). For a point-query \( x \) we test \( \varphi(x) \).

Multi-Attribute bloomRF. The ability to filter on multiple attributes simultaneously is necessary for complex operations in interactive analytics, scientific packages, IoT and AI. bloomRF supports two-dimensional filtering with reduced precision. To this end we concatenate the attribute-values and insert them in both combinations. For instance, bloomRF \((A,B)\) will concatenate the values of \( A \) and \( B \), and insert them as tuples \( \langle A,B \rangle \) and \( \langle B,A \rangle \). The increased space-requirements are lowered by reducing the precision of \( A \) and \( B \), e.g., to a 32-bit integer. As a result bloomRF can answer queries such as \( A < 42 \) AND \( B > 4711 \), \( A = 42 \) AND \( B > 4711 \) or \( A = 42 \) AND \( B = 4711 \).

9 EXPERIMENTAL EVALUATION
Integration in RocksDB [30], bloomRF has been implemented in a standalone library and has been integrated in RocksDB v6.3.6 through a standard filter policy. The policy is extended to pass query-range information (lower/upper bounds) to the filter by means of slice structures. For persistence we implement our own self-serialization mechanism, placing it as regular full filter block in each compaction-disabled SST file of a block-based table format.

Baselines. Throughout the evaluation the following baselines are used: BFs, Prefix-BFs and fence pointers as well as state-of-the-art point-range filters such as SuRF [43, 44, 49] and Rosetta [29]. We perform two types of experiments. First, system-level experiments, where all baselines are compared in RocksDB v6.3.6 to stress the overall effects in a real system. Second, standalone experiments are performed to stress specific aspects in isolation.

Workloads. Throughout the evaluation we use a set of different workloads. Firstly, we employ a derivative of YCSB [8] Workload E, which is range-scan intensive. The dataset comprises 50M 64-bit integer keys, while the values are 512 bytes long. The data is uniformly distributed, while the workloads are of normal, uniform and zipfian distributions. We issue \( 10^9 \) queries of a single fixed range-size that is specified in the respective experiments. All point- and range-queries in this workload are empty (unless specified otherwise), which represents the worst-case. Depending on the workload, non-empty queries may perform better, e.g. due to bloomRF’s early stop conditions. In fact, in a perfect system a perfect filter would incur minimal I/O, and thus the worst-case may overstate their impact.

Rosetta and bloomRF rely on parameter tuning methods that compute the proper filter-configurations, for given space budgets, number of keys and range sizes. SuRF, however, requires a suffix-length parameter setting to tune itself to a space budget and trade
Experimental Setup. The experimental server is equipped with an Intel E5-1620 3.50GHz, 32GB DDR4, and runs Ubuntu 16.04.

Experiment 1: **bloomRF is general-purpose and can handle various query ranges, from large to small.** We begin by comparing bloomRF against SuRF and Rosetta in RocksDB under conditions favorable to all approaches. To this end, we employ a space budget of 22 bits/key, 50M uniformly distributed keys and vary the query range sizes and workload distributions (Fig. 9.A1, B1 and C1).

In terms of end-to-end probe latency, bloomRF outperforms all baselines, due to its two-path range-lookup and its CPU-efficient PMHF (Fig. 12.G). The sudden rise in bloomRF latency at $|R| = 10^{11}$ is due to approx. 1% non-empty ranges generated by the workload driver because of the large interval size. Overall, bloomRF also has the lowest FPR of all baselines. Rosetta is more accurate for very short ranges ($|R| \leq 8$) as they hit its precise lower BF. Due to the error-correcting effect of its PMHF bloomRF is more accurate than Rosetta for small ranges of $16 \leq |R| \leq 64$, which must probe larger area in its filters. The sudden fluctuations of Rosetta can be explained with the switch between different variants. The good FPR of bloomRF for large ranges (e.g., $10^7 \leq |R| \leq 10^{11}$) is due to the ability to probe more bits and the exact layer configurations. However, SuRF’s LOUDS-encoding excels, for very large ranges (e.g., $|R| = 10^{11}$), while bloomRF still achieves an acceptable FPR of 0.0454, as it probes larger areas of its mid-upper layers. Under the same settings, we investigate the point-query FPR (Fig. 9.A2, B2 and C2 shown as figure-in-figure in Fig. 9). Rosetta exhibits the lowest point-query FPR due to its accurate bottom filter-layer. bloomRF needs more space for its mid-upper layers yielding slightly higher FPR. SuRF has the highest FPR due to its tri-truncation. All PRF outperform Prefix-BFs and fence pointers (Fig. 9.D).

Insight: bloomRF can handle a broad set of query ranges and outperforms all baselines, under various workload distributions, addressing Problem 1 (Sect. 1).

**Experiment 2: bloomRF is efficient.** We continue our comparison, by varying the space budget in RocksDB (Fig. 10). We start from the 22 bits/key (favorable for all approaches and used in the previous experiment) and proceed to 10 bits/key, which is typical for standard BFs. As we go, small (Fig. 10.A-C), medium (Fig. 10.D-F) and large (Fig. 10.G-I) range queries are performed. We use 50M keys; data and workload are uniformly distributed.

bloomRF outperforms all baselines. It remains competitive to Rosetta for very small ranges and bigger space budgets ($\geq 18$ bits/key). bloomRF also outperforms SuRF, except for very long ranges ($|R| \geq 10^{11}$). For point-lookups in RocksDB (Fig. 10, on the right) bloomRF is more accurate than the RocksDB BF due to the random scatter and the error-correction. For point-queries and 2M keys, but in a standalone setting (Fig. 12.E1-E3) we compare all PRF, the Cuckoo-Filter [16, 17] and the BF from LevelDB [19]. We vary the fingerprint sizes provided by the Cuckoo-Filter [16] and aim for high occupancies (95%) to keep within the space budgets.

In terms of throughput bloomRF outperforms Rosetta 7% to 44% at 22 and 10 bits/key, respectively. We elaborate by providing a detailed breakdown of the probe-costs in RocksDB (Fig. 12.G). We use 22 bits/key, 50M keys (2.06M per SST/filter), $10^5$ queries, uniform workload/data distribution. bloomRF has the CPU- and total costs.

Insight: Considering the performance and FPR at smaller space budgets (Fig. 10, $\leq 18$ bits/key), we observe that bloomRF is efficient in terms of: (i) performance per bits/key; and (ii) FPR per bits/key.

**Experiment 3: bloomRF can handle skewed data distributions.** So far we only considered uniform data distributions. Now we relax this assumption and investigate the impact of normal and zipfian data distributions in a standalone setting (Fig. 11). We also vary the number of keys ($10^3..50$M), the space budget, the query range and the workload. The color of each point in Fig. 11 denotes the best filter, while the symbol stands for the FPR difference to the second best filter or to bloomRF, in case it is not the best.

We observe that bloomRF can handle skewed data distributions across various settings. For zipfian bloomRF is outperformed only in isolated cases. This is due to the underlying structure based on Bloom-techniques, where bits from bottom-mid layers can be accurately probed due to its vertical error-correction, while SuRF is truncating beyond a certain length. Rosetta (presumably its hash functions or its variable-level design) loses efficiency with $|R| \geq 16$.

Insight: Fig. 11 depicts a holistic comparison among the PRF, on relevant parts of the problem space. All three approaches bring significant advantages to the design space and augment each other. Due to its LOUDS-encoding, SuRF tends to be better for large ranges ($10^8..10^{11}$), at higher space budgets with $\geq 14$ bits/key and more keys. Rosetta tends to be better for very small query ranges with more than 16 bits/key. bloomRF is generally applicable to various...
memory budgets, different number of keys, and performs well for different data distributions and workloads (Problem 3, Sec. 1). Experiment 4: bloomRF is online and concurrent insertions have acceptable impact on its probe-performance at different insert/probe ratios. We now quantify the online behavior, by investigating the impact of concurrent insertions on query performance and address Problem 2 (Sec. 1). To this end, we insert 50M, not sorted or prepared, uniformly distributed keys with different (uniform) insert/lookup ratios (x-axis) in a standalone setting. In single-threaded settings (Fig. 12.A), the overall throughput increases with higher insert/lookup ratios. Hence, the impact of insertions is acceptable. A deeper analysis in multi-threaded settings (Fig. 12.B) with varying the number of concurrent lookup/insertion-threads shows that insertions have marginal impact on the lookup performance per thread. The overall insert-throughput increases with more threads, although the throughput per insert-thread decreases. This is not surprising as bloomRF is a parallel data structure.

Next, we investigate the filter-construction costs (Fig. 12.C) on the 50M, uniform dataset in RocksDB, where L0 comprises 25 SST files. We report the total creation and the serialization time (incl. tuning). bloomRF has the lowest creation time, due to its high insertion performance. SuRF has relatively high overhead due to space budget tuning and trie creation. Experiment 5: bloomRF can handle floats. Our floating-point numbers dataset [33], contains positive and negative numbers. We execute 1.8M range queries (standalone), of size 10⁻³. In absence of other baselines we only investigate bloomRF (Fig. 12.D). In absence of other baselines we only show that bloomRF achieves an avg. FPR of 0.18 for 10-22 bits/key and 4M lookups/s.

Experiment 6: bloomRF can serve as multi-attribute filter. We evaluate multi-attribute querying in bloomRF on a Sloan Digital Sky Survey DR16 [42] dataset and extract the ObjectID and the Run columns. Their values roughly follow a normal distribution. In a standalone setting, we compare a multi-attribute bloomRF(Run, ObjectID) probed with Run<300 AND ObjectID=Const against two separate filters bloomRF(Run) for Run<300 and bloomRF(ObjectID) for ObjectID=Const, combining the probe-results conjunctively. As shown in (Fig. 12.F) bloomRF(Run,ObjectID) yields better FPR than the combined FPR of the two separate filter-lookups bloomRF(Run) and bloomRF(ObjectID). This observation is surprising since the separate filters operate on 64-bit integers, while the multi-attribute bloomRF reduces precision and operates on 32-bit integers. The core intuition is that the FPR of bloomRF(Run,ObjectID) depends on Z/Y, where Y and Z are the number of data points satisfying ObjectID=Const and Run<300 AND ObjectID=Const respectively.

10 RELATED WORK
Bloom-Filters are well-known and with many variants [1, 5, 28, 46] covering different aspects: counting [4, 18, 41]; compressibility [31]; SIMD vectorization [25, 37]; partial deletes [40]; efficient hashing [15, 23]; and data locality and novel hardware [6, 14, 25, 27, 39]. Recently, there have been numerous novel proposals [11, 12, 21, 35, 47], all of which are point-filters with different properties. Pioneered by [24, 32], the concept of learned BFs, leads to interesting applications [22, 26, 48] and is a future direction for bloomRF. The Adaptive Range Filter (ARF) [2] is one of the first approaches to describe the use of a simple form of dyadic numbering scheme to compute the covering intervals of a point. ARF, however, relies

![Figure 10: bloomRF is efficient, with better performance for different space budgets and query ranges in RocksDB.](image-url)
on a binary tree as a data structure and a powerful set of (learning) optimizations. Like bloomRF, ARF relies on the concept of covering the whole domain of the datatype. SuRF [49] shows the full potential of trie-based filters (Fast Succinct Trie) with a powerful encoding scheme (LOUDS-Dense/Sparse). In bloomRF prefix hashing serves as an encoding scheme.

Rosetta [29], like bloomRF utilizes DIs and dyadic decomposition for point-range-filtering. The concept itself is applicable to a wider range of other applications such as stream processing and summarization [9], hot/cold data separation techniques [10] or persistent sketches [36]. The Segment Trees employed by [9, 29, 36] help encoding interval information and mapping range-queries into prefix-queries. bloomRF’s prefix hashing achieves near space-optimal and computationally efficient encoding interval. Another major difference to [9, 29, 36] is that bloomRF employs PMHF to preserve local order. They reduce the number of memory accesses when range querying and yields high range query performance.

11 CONCLUSIONS

We introduce bloomRF as a unified PRF that extends BFs with range-lookups. We propose novel prefix hashing to encode range information in the hash-code of the key, and novel PMHF for fast lookups and fewer memory accesses. We describe basic bloomRF that is simple and tuning-free, and propose optimizations for handling larger ranges. bloomRF has near-optimal space- and constant query-complexity and outperforms existing PRF by up to 4x.

Acknowledgments. We thank the anonymous reviewers for the useful comments and suggestions. We are deeply grateful to the authors of [29] and [49] for providing the source code.
Figure 12: (a, b) online behavior; (c) filter creation; (d) floats; (e) point-queries; (f) dual-attribute filter; (g) cost breakdown.
