
# Grafite: Taming Adversarial Queries with Optimal Range Filters

MARCO COSTA, University of Pisa, Italy
PAOLO FERRAGINA, University of Pisa, Italy
GIORGIO VINCIGUERRA*, University of Pisa, Italy

Range filters allow checking whether a query range intersects a given set of keys with a chance of returning a false positive answer, thus generalising the functionality of Bloom filters from point to range queries. Existing practical range filters have addressed this problem heuristically, resulting in high false positive rates and query times when dealing with adversarial inputs, such as in the common scenario where queries are correlated with the keys.

We introduce Grafite, a novel range filter that solves these issues with a simple design and clear theoretical guarantees that hold regardless of the input data and query distribution: given a fixed space budget of $B$ bits per key, the query time is $O(1)$, and the false positive probability is upper bounded by $\ell/2^{B-2}$, where $\ell$ is the query range size. Our experimental evaluation shows that Grafite is the only range filter to date to achieve robust and predictable false positive rates across all combinations of datasets, query workloads, and range sizes, while providing faster queries and construction times, and dominating all competitors in the case of correlated queries.

As a further contribution, we introduce a very simple heuristic range filter whose performance on uncorrelated queries is very close to or better than the one achieved by the best heuristic range filters proposed in the literature so far.

CCS Concepts: • **Theory of computation** → **Bloom filters and hashing**; • **Information systems** → **Unidimensional range search**.

Additional Key Words and Phrases: range filter, Bloom filter, range search, data structure

**ACM Reference Format:**
Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2024. Grafite: Taming Adversarial Queries with Optimal Range Filters. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 3 (February 2024), 23 pages. https://doi.org/10.1145/3639258

## 1 INTRODUCTION

Filters are data structures that allow checking whether a query key belongs to a given set of keys, with a chance of returning a false positive answer in exchange for a small space occupancy, i.e. much smaller than the storage of the full set.

Due to their compactness and the guarantee of not returning false negatives, filters are often kept in main memory and used to prevent unnecessary and costly accesses and searches in the set. For example, they can avoid unnecessary network communications if a remote server does not contain the sought resource, or they can avoid unnecessary disk reads when the set is stored on disk. In fact, since their introduction by Bloom [5] in 1970s, filters have been successfully used

*Corresponding author (giorgio.vinciguerra@unipi.it)

Authors' addresses: Marco Costa, University of Pisa, Italy, m.costa22@studenti.unipi.it; Paolo Ferragina, University of Pisa, Italy, paolo.ferragina@unipi.it; Giorgio Vinciguerra, University of Pisa, Italy, giorgio.vinciguerra@unipi.it.


ACM 2836-6573/2024/2-ART3
https://doi.org/10.1145/3639258

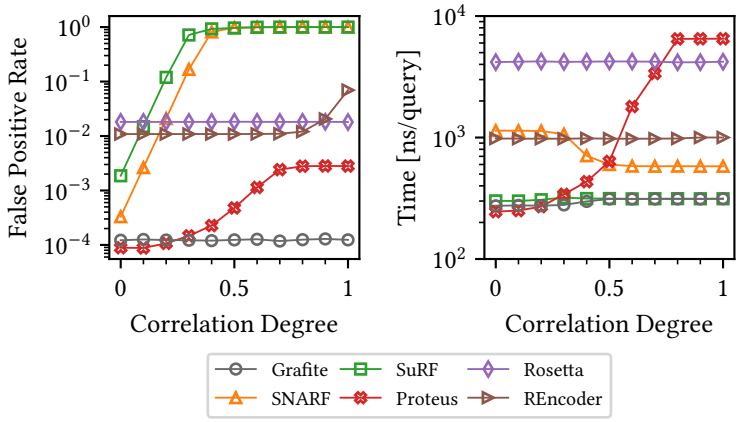


Fig. 1. Grafite is the only range filter to date that is both effective (low false positive rate) and efficient (low query time) as the endpoints of the query range get closer to the data.

in networking [6], distributed systems [35], databases [10], bioinformatics tools [7], and search engines [17], to mention just a few applications.

While the vast majority of filters are capable of answering approximate membership (point) queries [11, 13, 15, 19, 24, 32, 34], a new line of research, started a decade ago [1], focused on their generalisation to *range queries*, which occur frequently in big data systems such as key-value stores [18, 21, 25, 36, 40]. In this case, the filtering problem can be formally stated as follows.

Problem 1 (Approximate range emptiness). *Given a set $S$ of $n$ keys drawn from an integer universe $[u] = \{0, \ldots, u-1\}$, build a space-efficient data structure, called* range filter*, that answers* range emptiness queries *of the form $[a, b] \cap S \neq \emptyset$? for any $a \leq b$ in $[u]$. The range filter is allowed to return "not empty" when actually $[a, b] \cap S = \emptyset$ (i.e. a false-positive error) with probability at most $\varepsilon$.*

From a theoretical point of view, this problem was solved optimally by Goswami et al. [18], which first proved a space lower bound of $\Omega(\log \frac{L}{\varepsilon}) - O(1)$ bits per key, where $L$ is an upper bound on the query range size, and then they gave a data structure for the $w$-bit word RAM model matching this space up to a lower order additive term and offering constant-time queries when $w = \Omega(\log \frac{nL}{\varepsilon})$.

From a practical point of view, the literature offers a vast choice of range filters, such as ARF [1], SuRF [40], Rosetta [25], SNARF [36], Proteus [21], bloomRF [27], and REncoder [38]. These solutions, reviewed in Section 2, adopt totally different approaches to range filtering, thus offering a large number of trade-offs among space, *empirical* probability of a false positive error (henceforth, false positive rate), query time, and construction time. This notwithstanding, there is still one fundamental challenge that the literature has not yet been able to address:

**No practical solution is robust enough to efficiently handle all input data and query distributions.**

Existing practical range filters, indeed, adopt heuristic designs that sacrifice performance guarantees to improve upon some specific inputs. In fact, these range filters hardly guarantee a bounded false positive probability $\varepsilon$ for a given amount of space, thus, strictly speaking, they do not solve the approximate range emptiness problem unless some specific (and strong) assumptions on the kind of query workload and input data distribution are met.

As a consequence of this, there exist *adversarial distributions* that can drive the false positive rate arbitrarily close to 1, thus making the filter useless, if not dangerous for the big data systems

making use of it (e.g. because of increased disk or network activity that the filter was actually deployed to prevent). The importance of this issue has also been stressed by Knorr et al. [21], who after presenting a formal framework of existing range filters, conclude that *"no current design can handle [adversarial workloads] practically, suggesting the need for further expansion of the range filter design space."*

Notably, the vast majority of range filters suffer from the so-called *correlation* between keys and queries, that is, they provide little or no filtering at all when an endpoint of the query range is close to one of the keys in the input set, which is quite disappointing given the commonness of such a workload in applications that care about the local properties of data (such as in time series applications where we need to check if some events occurred in a time frame) [25], or given that malicious users can artificially issue these queries with just the knowledge of (a subset of) the keys. To demonstrate this issue, we show in Figure 1 how existing range filters quickly reach high false positive rates as the endpoints of the query range get closer to the input keys, denoted as "correlation degree" on the horizontal axis (and detailed in our experimental section). This holds true for SuRF, SNARF, REncoder, and Proteus, the latter even being auto-tuned on (i.e. overfitted to) the query workload. The only exception is Rosetta which has a constant false positive rate but a query time that is up to orders of magnitude higher compared to the other filters.
Apart from the lack of robustness, there is another challenge:

**Current range filters are complex to evaluate and deploy because of their complicated design.**

As stated above, existing range filters adopt complex design choices aimed at increasing their efficacy on some specific inputs. For instance, SuRF [40] encodes a trie with input keys truncated at their distinguishing prefix (thus providing better filtering when there is no correlation), while SNARF [36] maps each input key to a 1-bit in a bitvector via a model learned from the data (thus providing better filtering when there are no outliers or poisoned data [22]).

These designs, coupled with the lack of guaranteed bounds on the false positive rate, hinder our understanding of how the range filter will behave once deployed to production unless future data and queries will follow exactly the same distribution of the test data on which the *empirical* false positive rate was originally observed. The ability to auto-tune on a sample of queries and input keys, as in Proteus [21], only partially eases the hard job of integrating a range filter into a real system, as there is still the necessity to keep a proper set of sample queries (thus also allocating further space) and to detect when the filter needs to be rebuilt because of workload shifts (thus introducing additional delays and requiring to keep the input data in memories close to where the range filter is built).

Instead, we aspire to a practical range filter that, similarly to Bloom filters, works robustly out of the box regardless of the input data and future queries, while hiding the complexities of its design and exposing just simple knobs such as the false positive probability $\varepsilon$ or a space budget.

**Our contributions.**

- We introduce *Grafite*, a novel practical range filter that solves the lack of robustness and the high complexity of current solutions. Unlike all the practical range filters to date, Grafite offers clear guarantees that hold regardless of the input data and query distributions: given a fixed space budget of $B$ bits per key, the query time is $O(1)$, and the false positive probability is upper bounded by $\min\{1, \ell/2^{B-2}\}$, where $\ell$ is the query range size. Perhaps surprisingly, this is achieved via a simple design that maps the input keys into a smaller universe via a properly designed hash function [18], stores the resulting hash codes space-efficiently [14, 16], and checks hash codes for inclusion in a range via an efficient query algorithm.

- We provide a comprehensive related work section and propose the first theoretical comparison of the space-time performance of range filters, showing the superiority of Grafite over prior solutions.
- We perform the largest experimental comparison among range filters, both in terms of dataset size and in the number of tested solutions, which shows that *all* the existing filters provide little to no filtering or a high query time in the case of correlated query workloads. Instead, Grafite is the only range filter to date to achieve a robust and predictable false positive rate across all combinations of datasets, query workloads, and range sizes, while also providing faster queries and construction times, and dominating all competitors in the case of correlated query workloads.
- For datasets and uncorrelated query workloads previously tested in the literature, we show that there exists a very simple heuristic filter design — that we name *Bucketing* — that essentially matches the filtering effectiveness of all the existing heuristic range filters, which are however significantly more complex and incur in higher query and construction times. This demonstrates that, if we give up on robustness guarantees, the approximate range emptiness problem can sometimes be addressed with a very simple solution.

**Paper outline.** Section 2 discusses existing range filters. Section 3 introduces Grafite. Section 4 introduces Bucketing. Section 5 compares the space-time bounds of Grafite with those of existing range filters. Section 6 experiments with Grafite, Bucketing and existing range filters. Section 7 concludes the paper and suggests some open problems.

## 2 RELATED WORK

Consider an upper bound $L$ on the query range size $b - a + 1$. We can provide a trivial solution to the approximate range emptiness problem by using point filters which, given a false positive probability of $\gamma$, can be implemented in $n \log \frac{1}{\gamma} + O(n)$ bits of space and $O(1)$ query time [32, 34]. Indeed, by building a point filter on the input set $S$ with false positive probability $\gamma = \varepsilon/L$, we can check the existence of any element of $[a, b]$ in $S$ by executing at most $L$ point queries. This solution takes $n \log \frac{L}{\varepsilon} + O(n)$ bits of space, $O(L)$ query time, and the false positive probability is at most $\varepsilon$ by union bound.

The question is now how far is this trivial solution from being optimal. The answer was given by Goswami et al. [18], which proved the following lower bound.

Theorem 2.1 ([18]). *Any data structure solving approximate range emptiness queries of fixed length $L \leq u/(5n)$ on $n$ keys drawn from an integer universe $[u] = \{0, \ldots, u-1\}$ with a false positive probability of $\varepsilon$ must use at least $n \log \left(\frac{L^{1-O(\varepsilon)}}{\varepsilon}\right) - O(n)$ bits of space.*

This is a disappointing result because it states that, for a sufficiently small $\varepsilon$, at least $\log \frac{L}{\varepsilon}$ bits per key are needed. Hence, the larger is $L$ and/or the smaller is $\varepsilon$, the larger is the space required by any range filter. Note that we can restrict $L \leq u\varepsilon/n$, since otherwise it is more convenient to store the input keys in space close to $\log \frac{u}{n}$ bits per key (e.g. with an Elias-Fano encoding [14, 16]) thus solving the problem without false positives (i.e. $\varepsilon = 0$).

Furthermore, Theorem 2.1 implies that we cannot improve the space occupancy of the trivial solution stated above, but it challenges us to find a solution that matches its same space bound whilst improving the unattractive $O(L)$ query time. In this respect, Goswami et al. [18] also introduce a data structure that solves the range emptiness problem in $n \log \frac{L}{\varepsilon} + o(n \log \frac{L}{\varepsilon}) + O(n)$ bits of space, while offering $O((\log \frac{nL}{\varepsilon})/w)$ query time in the $w$-bit word RAM model, thus achieving $O(1)$ query time when $w = \Omega(\log \frac{nL}{\varepsilon})$. The overall approach is mainly theoretic in nature and thus very complicated to implement. Nevertheless, the idea in [18] to reduce the original universe $U$ into a smaller universe $h(U)$ via a proper hash function $h$ is effective, and it will be used in Grafite too.

We now turn our attention to practical range filters.

**Prefix Bloom Filter.** A Prefix Bloom Filter hashes key prefixes of a predetermined bit-length $l$ within a Bloom filter [12, 26]. Since each prefix encodes a range of the universe of size $2^{\log u - l}$, the filter can answer a range emptiness query by probing each range (i.e. configuration of $l$ bits) that overlaps with the query range, and returning "empty" if all the probes return false, "not empty" otherwise. We do not further consider Prefix Bloom Filters because they are generalised by Rosetta [25] and Proteus [21], which are described below and used in our experimental comparison.

**ARF.** The Adaptive Range Filter (ARF) [1] is based on a compactly-encoded binary tree whose leaves represent ranges of the universe and are associated with a flag indicating whether there is at least one key in that range. Internal nodes allow navigating to the leaf containing the left endpoint $a$ of the query range $[a, b]$, and the leaves to its right are inspected until either one of them has a true flag, thus the answer is "not empty", or the leaf covering $b$ is reached and has a false flag, thus the answer is "empty". ARF adapts to the data and query distribution by learning from false positive queries and adjusting its shape accordingly. As reported in [40], ARF can be up to 1300× larger than SuRF, described next, while also exhibiting a higher false positive rate. Thus, we do not further consider ARF.

**SuRF.** The Succinct Range Filter (SuRF) [40–42] is built upon a compactly-encoded trie, called Fast Succinct Trie, that stores, for each key $s \in S$, the shortest prefix $p_s$ of $s$ such that $s$ can be uniquely identified among all the strings in $S$, followed by a number $m$ of *suffix bits* following that key prefix. For improving the filter performance on just point queries, these $m$ bits can also be set to a hash of the key. A range emptiness query on $[a, b]$ is answered by looking in the truncated trie for the smallest key $k$ (which can include its suffix bits if the search reaches a leaf) such that $k$ is lexicographically $\geq a$. The result of the query is given by the result of the lexicographic comparison $k \leq b$. We use SuRF in our experimental comparison.

**Rosetta.** The Robust Space-Time Optimized Range Filter (Rosetta) [25] consists of $\log u$ Bloom filters organised in levels. For every key, each of its prefixes of length $k = 1, \ldots, \log u$ is inserted into the Bloom filter at level $k$. A range emptiness query is answered by decomposing the query range into dyadic intervals, which are used to probe the corresponding Bloom filters. If all the probes return a negative answer, the query range is empty. Otherwise, each range that returned a positive answer is recursively decomposed at the next level. In practice, Rosetta tunes itself to minimise the false positive rate under a given space budget by allocating more bits to the Bloom filters that are probed more frequently, based on a sample of the query workload. We use Rosetta in our experimental comparison.

**SNARF.** The Sparse Numerical Array-Based Range Filter (SNARF) [36] uses a bit array $B$ of $Kn$ bits, where $K$ is a suitably large parameter that impact on the false positive rate, and a function $f(x) = \lfloor \text{MCDF}(x) \cdot Kn \rfloor$, where MCDF is a monotonic estimate of the CDF of the keys in $S$, i.e. $\lfloor \text{MCDF}(x) \cdot n \rfloor$ is an estimate of the rank of $x$ in $S$. The function $f$ is built by taking one key every $t$ keys in the sorted $S$, and using these key samples as endpoints of linear splines. The bit array $B$, initially empty, is filled with $B[f(x)] = 1$ for each $x \in S$, and then compressed. A range emptiness query $[a, b]$ returns "not empty" if and only if there is at least a 1-bit in the range $B[f(a), f(b)]$. We use SNARF in our experimental comparison.

**Proteus.** Proteus [21] combines the trie-based prefix filtering of the Fast Succinct Trie with the filtering of the Prefix Bloom Filter. Differently from SuRF, Proteus does not encode in the trie a unique prefix for every key but rather all unique key prefixes of a fixed length $l_1$, and it implements a single (prefix) Bloom filter for all key prefixes of length $l_2 > l_1$. If the range emptiness query is not

resolved after descending the trie up to the prefix length $l_1$, i.e. if there are matching leaves so that we cannot yet return "not empty", then the Prefix Bloom Filter is probed for each length-$l_2$ prefix extending the length-$l_1$ prefix of each matching leaf, returning "empty" if all these probes return false, "not empty" otherwise. The values of $l_1$ and $l_2$ are determined by an algorithm that minimises the false positive rate given the input keys, a sample query workload, and a space budget. We use Proteus in our experimental comparison.

**bloomRF.**The Bloom Range Filter (bloomRF) [27] hashes a key into a hash code composed of positions that are used to set bits to 1 in a bit array $B$. The hash code is such that equal key prefixes have equal hash code prefixes (thus encoding range information in the hash code), and its position components preserve the order of prefixes (thus improving data locality). A query range is decomposed into dyadic intervals whose emptiness is determined by checking in $B$ the appropriate bits computed via the hash code above. We could not experiment with bloomRF because its implementation is not yet open source,[1] but we comment on it in the theoretical comparison of Section 5.

**REncoder.** The Range Encoder (REncoder) [38] too consists of a bit array $B$, initially empty. It splits each input key into a 4-bit suffix $s$ and the remaining prefix $p$. The suffix $s$ is conceptually represented by a leaf in a complete binary tree with 16 leaves, whose nodes in the path from that leaf to the root are marked with a 1, and the remaining nodes are marked with a `0`. Intuitively, nodes represent ranges of the universe and the bit marks record the presence of keys in a range. The bit marks are then concatenated to form a 32-bit value, which is written into $B[h_i(p), h_i(p) + 31]$ via an OR operation, where $h_i$ is a hash function, for $i = 1, \ldots, k$. The process is then repeated on the prefix $p$ of the key, and it stops when the whole key has been processed. A query range is decomposed into dyadic intervals whose emptiness is determined via traversals of binary trees, which are recovered from $B$ via AND operations. We use REncoder in our experimental comparison.

We conclude this section by mentioning that the problem of supporting efficient in-place insertions has only been touched upon in the literature. Indeed, current range filters are difficult to update efficiently due to their use of static compactly-encoded tries (SuRF and Proteus), or learned functions and compressed bitvectors (SNARF). Some other range filters like Prefix Bloom Filters, Rosetta, bloomRF and REncoder, instead, could be easier to update with new keys due to their design (loosely) based on Bloom filters, but the impact of insertions on the false positive rate has not yet been explored. Since in this paper we do not deal with these issues, we leave it as an open problem [11].

## 3 GRAFITE: AN OPTIMAL RANGE FILTER

We now introduce Grafite, which eventually solves the lack of robustness in state-of-the-art range filters. We start from the idea of Goswami et al. [18] to solve the approximate range emptiness problem through hashing, and we take this idea into a simpler, practical and yet more succinct solution that is closer to the lower bound of Theorem 2.1.

**Hashing input keys.** Recall we are given a set $S$ of $n$ keys in a universe $[u] = \{0, \ldots, u-1\}$, a false positive probability $\varepsilon$, and an upper bound $L$ on the query range size.

Set $r = nL/\varepsilon$, and let $q\colon [u/r] \to [r]$ be a hash function taken from a *pairwise-independent family* $Q$, i.e. a set of hash functions $Q = \{q\colon [u/r] \to [r]\}$ such that, for any pair of distinct keys $(x_1, x_2) \in [u/r]^2$ and any pair of (not necessarily distinct) hash codes $(y_1, y_2) \in [r]^2$, we have

$$\Pr_{q \in Q}\left[q(x_1) = y_1 \wedge q(x_2) = y_2\right] = \frac{1}{r^2}.$$

[1]Personal communication with the authors.

The simplest technique for constructing such a hash function [39] is to select a large prime number $p > r$ and two random numbers $c_1, c_2 < p$ such that $c_1 \neq 0$, and then define the hash function as $q(x) = ((c_1 x + c_2) \bmod p) \bmod r$.

In addition to $q$, we define a hash function $h$ that preserves the *locality* of the hashed items and has a small collision probability [18]:

$$h(x) = (q(\lfloor x/r \rfloor) + x) \bmod r. \tag{1}$$

We use $h$ to transform the set $S = \{x_1, \ldots, x_n\}$ of input keys from the original universe $[u]$ to the set $h(S) = \{h(x_1), \ldots, h(x_n)\}$ of hash codes in the reduced universe $[r]$, and then we store $h(S)$ by means of a compact *non-approximate* range emptiness data structure.

We will describe the data structure in a moment. For now, we notice that because of the hash function $h$, a range emptiness query $[a, b] \cap S \neq \emptyset$ ? can be answered by verifying the existence of a value $h(x) \in h(S)$ such that

$$\begin{cases} h(a) \leq h(x) \leq h(b) & \text{if } h(a) \leq h(b), \\ h(x) \leq h(b) \vee h(x) \geq h(a) & \text{otherwise.} \end{cases} \tag{2}$$

The first case is straightforward, while the second one occurs if there is an *overlap* of the hashed endpoints (i.e. $h(a) > h(b)$) as a consequence of the modulo and the reduced universe. If a value $h(x)$ which satisfies (2) is found, we answer "not empty". Otherwise, we answer "empty".

It should be clear that there can be no false negatives. For the false positives, there is the following result (which is a straightforward generalisation of a result in [18]).

Lemma 3.1 ([18]). *The approach based on the hash function* (1) *and the conditions* (2) *guarantees a false positive probability of at most $\varepsilon$ for query ranges of size $L$, and at most $\ell\varepsilon/L$ for ranges of size $\ell \leq L$.*

Proof. A false positive occurs when no key in $S$ is in the query range $I$ but there is a hash collision between a key $x \in S$ and a point $y \in I$. From [18, Lemma 3.1], such a collision happens with probability $\Pr[h(x) = h(y)] \leq 1/r$. The false positive probability is then given by a union bound over all possible collisions between keys in $S$, which are $n$, and points in $I$, which are $\ell \leq L$, thus it is

$$\sum_{x \in S} \sum_{y \in I} \Pr\left[h(x) = h(y)\right] \leq \sum_{x \in S} \sum_{y \in I} \frac{1}{r} = \frac{n\ell}{r} = \frac{n\ell}{\frac{nL}{\varepsilon}} = \frac{\ell\varepsilon}{L} \leq \varepsilon.$$

□

**Storing hash codes succinctly.** Having defined how approximate range emptiness can be achieved through hashing, the following step is to store the hash codes $h(S)$. Goswami et al. [18] store the hash codes together with a sophisticated prefix search data structure from [4] to check hash codes for inclusion in a query range. Our study proposes a much simpler data structure that builds on the classic Elias-Fano integer code [14, 16] together with an efficient procedure to check hash codes for inclusion in a range, explained below. As we will show, we will obtain a practical range filter, which actually has an even better space than the solution of [18].

Let $z_1, \ldots, z_n$ be the deduplicated sorted set of hash codes in $h(S)$.[2] We split the length-$\lceil \log r \rceil$ binary representation of each $z_i$ into a low part $z_i^{lo}$ consisting of the $l = \lfloor \log \frac{r}{n} \rfloor = \lfloor \log \frac{L}{\varepsilon} \rfloor$ least significant bits of $z_i$, and a high part $z_i^{hi}$ consisting of the remaining $\lceil \log r \rceil - l$ most significant bits of $z_i$. The low parts are concatenated into a vector $V[1, n]$ of $l$-bit cells, which thus takes $nl = n\lfloor \log \frac{L}{\varepsilon} \rfloor$ bits overall. The high parts are encoded in a bitvector $H[1, z_n^{hi} + n + 1]$ where the

[2]Due to collisions there can be fewer than $n$ distinct hash codes, but we prefer using $n$ in our description and bounds for simplicity. The space without the $k$ duplicates is $k \log \frac{L}{\varepsilon}$ bits lower, where the expected value of $k$ can be shown to be $\varepsilon(n-1)/(2L)$ with classic hash tables analyses under the simple uniform hashing assumption [9].

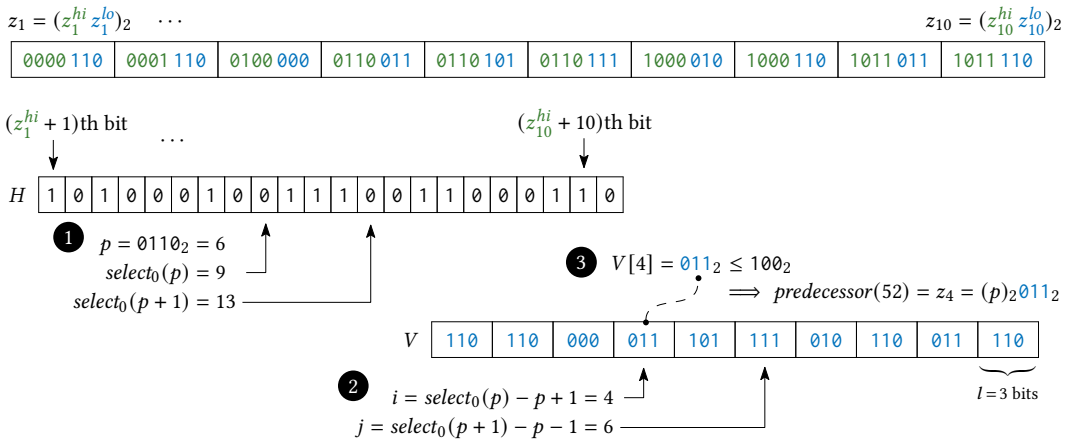


Fig. 2. An example of Grafite storing the compressed hash codes 6, 14, 32, 51, 53, 55, 66, 70, 91, 94 (see Example 3.2), and some steps needed for answering a range emptiness query (see Example 3.3).

positions $z_i^{hi} + i$ are set to 1, and the remaining positions are set to 0. This completes the succinct encoding of $h(S)$, whose bit-size can be shown to be upper bounded by $n \log \frac{L}{\varepsilon} + 2n$.

*Example 3.2.* Suppose we are given the set $S = \{9, 48, 50, 191, 226, 269, 335, 446, 487, 511\}$ with $n = 10$ keys, and the values $L = 4$, $\varepsilon = 0.4$, thus giving $r = nL/\varepsilon = 100$. Say we use the hash function $h(x) = (q(\lfloor x/r \rfloor) + x) \bmod r$, where we choose $q(x) = ((c_1 x + c_2) \bmod p) \bmod r$ with parameters $p = 2^{31} - 1$, $c_1 = 10$, and $c_2 = 5$, thus giving the set of hash codes $h(S) = \{14, 53, 55, 6, 51, 94, 70, 91, 32, 66\}$. Figure 2 shows the binary representation of the integers $z_1 = 6, \ldots, 94 = z_n$ corresponding to the sorted set $h(S)$ at the top (where the low $l = \lfloor \log \frac{L}{\varepsilon} \rfloor = 3$ bits and the remaining 4 bits are highlighted with different colours) and their Elias-Fano encoding via the vectors $H$ and $V$ at the bottom.

**Searching hash codes efficiently.** We now describe how to search within the hash codes $h(S)$ so that both conditions in (2) can be checked efficiently.

For the first branch in (2), we need to augment the Elias-Fano encoding with an operation that checks for the existence of an $h(x)$ such that $h(a) \le h(x) \le h(b)$. To this end, we use the well-known $predecessor(y)$ operation, which given $y \in [r]$, returns the largest element $z_k$ smaller than or equal to $y$ [29]. We first compute $z_k = predecessor(h(b))$ and then check if $z_k \ge h(a)$. If this is the case, then there is at least a hash code $z_k = h(x)$ in the range $[h(a), h(b)]$, where $x \in S$. Thus the first branch is satisfied, and the answer to the approximate range emptiness query is "not empty". If not, i.e. $z_k < h(a) \le h(b)$, then the first branch is not satisfied and the answer is "empty".

The $predecessor(y)$ operation is implemented by first identifying the range of hash codes $z_i, \ldots, z_j$ that share the same high part $y^{hi}$ of $y$, and then by binary searching in the subarray $V[i, j] = [z_i^{lo}, \ldots, z_j^{lo}]$ for the predecessor of $y^{lo}$. Since $V[i, j]$ might contain at most $2^l$ configurations of $l$-bit integers, the binary search runs in $O(\log 2^l) = O(\log \frac{L}{\varepsilon})$ time. Let us detail these steps for completeness [28, 30, 31, 37]. For identifying the range $[i, j]$, we need the $select_b(k)$ operation, which returns the position of the $k$th $b$-bit in $H$, for $b \in \{0, 1\}$ (we define $select_b(0) = 0$). This operation can be implemented in $O(1)$ time using $o(|H|) = o(n)$ bits [8, 20]. Then, by definition of $H$, the hash codes $z_i, \ldots, z_j$ with the same high part $p = y^{hi}$ form a contiguous sequence of 1-bits in the subarray $H[p+i, p+j]$ followed by a 0 in $H[p+j+1]$. Thus, the subarray $H[p+i, p+j+1] = 1^{j-i+1}0$ corresponds to the subarray $H[select_0(p) + 1, select_0(p+1)]$, and hence the range $[i, j]$ is given by setting the

**Algorithm 1:** Construction of Grafite.

**function** Construct($S[1,n]$, $L$, $\varepsilon$)
  $G.r \leftarrow nL/\varepsilon$
  $G.h \leftarrow$ The hash function (1)
  $V \leftarrow$ An empty array of size $n$
  **for** $i = 1$ **to** $n$ **do**
    $V[i] \leftarrow G.h(S[i])$
  Sort($V$)
  $G.ef \leftarrow$ BuildEliasFano($V$)
  **return** $G$

**Algorithm 2:** Range emptiness query in Grafite.

**function** RangeEmptinessQuery($G$, $a$, $b$)
  $h_a \leftarrow G.h(a)$
  $h_b \leftarrow G.h(b)$
  **if** $h_a > h_b$ **then**
    **return** "Not empty" **iff** $G.ef.first \leq h_b \vee G.ef.last \geq h_a$
  **return** "Not empty" **iff** $G.ef.predecessor(h_b) \geq h_a$

endpoints of these subarrays equal, thus yielding $i = select_0(p) - p + 1$ and $j = select_0(p+1) - p - 1$. Finally, in the corner case that the binary search in $V[i,j]$ finds that the predecessor of $y$ is $z_{i-1}$, we recover this element via a random access operation: the high part of $z_{i-1}$ is retrieved as $z^{hi}_{i-1} = select_1(i-1) - (i-1)$, and the low part is readily available as $z^{lo}_{i-1} = V[i-1]$.

For the second branch in (2), it is enough to random access the smallest and largest value in $h(S)$, i.e. $z_1$ and $z_n$, and do the comparison $z_1 \leq h(b) \vee z_n \geq h(a)$.

Algorithms 1 and 2 contain the pseudocode for the construction and query algorithms on Grafite, respectively. For the construction, we notice that BuildEliasFano runs in linear time, while Sort takes the time to sort $n$ integers of length $\lceil \log r \rceil$, for which there exist very efficient sequential and parallel algorithms [3].

*Example 3.3.* Let us solve the range emptiness query for the range $[44, 47]$ on the Grafite instance of Example 3.2 and Figure 2. First, we compute $h(a) = h(44) = 49$ and $h(b) = h(47) = 52$. Then, since $h(a) \leq h(b)$, we need to compute $z_k = predecessor(h(b))$ and do the check $z_k \geq h(a)$. As depicted in Figure 2, we find such a $z_k$ by: ❶ taking the high part $p = 0110_2 = 6$ of $h(b) = 52$ and computing $select_0(p) = 9$ and $select_0(p+1) = 13$; ❷ computing $i = select_0(p) - p + 1 = 4$ and $j = select_0(p+1) - p - 1 = 6$; ❸ doing a binary search in $V[i,j]$ for the predecessor of the low part $00_2 = 4$ of $h(b)$, which yields $V[4] = 011_2$. This reveals that $predecessor(h(b)) = z_4 = 51$, and since $z_4 \geq h(a) = 49$, we return "not empty", which is a false positive error since $[44, 47] \cap S = \emptyset$.

Combining the above description of Grafite with Lemma 3.1, and recalling the restriction on the range size $L \leq u\varepsilon/n$ (see Theorem 2.1 and the discussion below it), we can state the following result.

Theorem 3.4. *Given a set of $n$ keys from a universe $[u]$, $\varepsilon \in (0,1)$, and $L \in [1, u\varepsilon/n]$, Grafite takes $n \log \frac{L}{\varepsilon} + 2n + o(n)$ bits of space and answers approximate range emptiness queries in time $O(\log \frac{L}{\varepsilon})$ with a false positive probability of at most $\varepsilon$ for query ranges of size $L$, and at most $\ell\varepsilon/L$ for query ranges of size $\ell \leq L$.*

Since $n \log \frac{L}{\varepsilon} + 2n + o(n) = n \log \frac{L}{\varepsilon} + O(n)$, Grafite matches the space lower bound of the approximate range emptiness problem (Theorem 2.1), thus it is space-optimal, while offering constant time queries whenever $L/\varepsilon = O(1)$.

Notice that Grafite has several important features. First, the false positive probability is bounded *regardless* of the input set $S$ and query workload, thus solving the first challenge faced by known practical range filters mentioned in Section 1. Second, the query time is independent of $n$ (and $u$), thus making Grafite efficient even for large sets of input keys. Third, Grafite does not require any sophisticated tuning procedure, but it can be used out of the box by just specifying $\varepsilon$ and $L$.

We stress that, after $L$ has been set, Grafite can answer on both query ranges of size $\ell$ smaller than $L$ (with a smaller chance of false positives than $\varepsilon$) and larger than $L$ (with a higher chance of false positives than $\varepsilon$), because the presence of $L$ in Theorem 3.4 is technical and serves to make the false positive probability $\leq \varepsilon$. As a matter of fact, since the space usage is $\log \frac{L}{\varepsilon} + 2$ bits per key,[3] we can build Grafite by just setting the space budget to a constant $B$, hence $\varepsilon = L/2^{B-2}$, and we can answer range emptiness queries with a false positive probability of at most

$$\frac{\ell\varepsilon}{L} = \frac{\ell L/2^{B-2}}{L} = \frac{\ell}{2^{B-2}},$$

in time

$$O\left(\log \frac{L}{\varepsilon}\right) = O\left(\log \frac{L}{L/2^{B-2}}\right) = O\left(\log 2^{B-2}\right) = O(B),$$

thus proving the following result (which solves the second challenge faced by known practical range filters mentioned in Section 1).

Corollary 3.5. *Given a set of $n$ keys and a budget of $B = O(1)$ bits per key, Grafite answers approximate range emptiness queries in $O(1)$ time with a false positive probability of at most $\min\{1, \ell/2^{B-2}\}$, where $\ell$ is the query range size.*

Observe that, a similar derivation of Corollary 3.5 with the data structure of Goswami et al. [18] would lead to a false positive probability higher (actually, strictly higher, due to the lower-order terms we omit) than $\ell/2^{B-3}$, which is worse than the one achieved by Grafite (see also Section 5).

Finally, we mention that instead of returning a boolean answer, Grafite can return an approximate count of the keys that intersect the given query range without any change in its space or query time complexity, thus potentially being a practical and efficient solution for this interesting problem too [2]. It suffices to return the difference between the *ranks* at the hashed endpoints of the query range (possibly adjusting the result with the expected number of collisions in the range, as per Footnote 2), where the rank of a hashed element can be found easily during the *predecessor* operation on the Elias-Fano sequence.

## 4 BUCKETING: A HEURISTIC RANGE FILTER

We now introduce a very simple heuristic range filter named Bucketing. Bucketing has the same weakness of known heuristic range filters, namely, it provides little or no filtering on correlated query workloads, thus its purpose is not to compete with Grafite, which instead provides robust and consistent filtering effectiveness regardless of the input set and query workload. Rather, Bucketing will serve us to show experimentally that, on certain inputs experimented in the literature, one does not need to resort to the sophisticated heuristic filter designs proposed in the literature, because simpler solutions can experimentally match or improve their filtering effectiveness while being more efficient to query and construct.

[3]The $o(1)$ term we omit here can be just 0.035 bits per key in practice [23].

Given a set $S = \{x_1, \ldots, x_n\}$ of $n$ keys in the universe $[u]$, and given an integer $s \geq 1$, we split the universe into $u/s$ buckets of size $s$. Then, we create a bitvector $C$ of size $u/s$ that indicates with 1 in position $i$ if there exists at least a key $x \in S$ that falls in the $i$th bucket. That is, $C$ is initially empty, and we set $C[x/s] = 1$ for each $x \in S$ (we omit floors for simplicity).

Let $t$ be the number of 1-bits in $C$, which depends on the distribution of the input data. Clearly, $t$ cannot be more than the size of $C$ or than the number of elements in $S$, thus $t \leq \min\{u/s, n\}$. By compressing $C$ with the Elias-Fano encoding, the total space occupancy is $t(\log \frac{u}{ts} + 2)$ bits. The construction can actually be done without creating $C$ by just considering the deduplicated list of the 1-bit positions $x_1/s, \ldots, x_n/s$.

The parameter $s$ allows us to trade the space with the coarseness of such a lossy encoding of $S$. Indeed, when $s = 1$, we are losslessly encoding the input set (i.e. $t = n$) and the space is $n(\log \frac{u}{n} + 2)$ bits, whereas if $s = u$ then a single bucket exists for the whole set (i.e. $t = 1$) and its single entry in $C$ is 1, thus the space is 0.

Similarly to Grafite (Section 3), we augment the Elias-Fano encoding of $C$ with select data structures that occupy $o(t)$ bits and allow us to compute the *predecessor* operation in $O(\log \frac{u}{ts})$ time. Then, given a query range $[a, b]$, if $\mathit{predecessor}(b/s) \geq a/s$ is true, then $C[a/s, b/s]$ contains at least a 1-bit and we answer "not empty". Otherwise, we answer "empty".

It goes without saying that false negatives are not possible and that a false positive happens when $[a, b] \cap S = \emptyset$ but there is a key $k \in S$ such that $k < a$ and $k$ falls into bucket number $a/s$, or symmetrically if $k > b$ and $k$ falls into bucket number $b/s$. Similarly to other heuristic filters, a bound on the false positive rate that holds regardless of the input data and query distributions cannot be proved. Moreover, we expect this approach to provide no filtering as the correlation increases, due to endpoints of the query range falling in non-empty buckets.

## 5 THEORETICAL COMPARISON

We now compare the space-time bounds of Grafite (Theorem 3.4) with those of the state-of-the-art range filters discussed in Section 2. We distinguish two kinds of range filters, the ones that provide a *bounded* false positive probability $\varepsilon$ thus solving the approximate range emptiness problem formulated in Section 1, and the *heuristic* ones, which do not provide any guarantee unless some assumptions on the input data and query distribution are met. Therefore, the space complexity of these latter range filters cannot and will not be compared with that of Grafite (unless under said assumptions).

Table 1 summarises known and new bounds. Some complex time bounds are simplified with the $\Omega$-notation, which still allows comparing them with Grafite. All time bounds do not include the $O((\log u)/w)$ time to process the two endpoints of the query range, which is typically neglected because any solution has to read them.

**Goswami et al.'s solution.** The data structure proposed for the $w$-bit word RAM model by Goswami et al. takes $n \log \frac{L}{\varepsilon} + O(n) + o(n \log \frac{L}{\varepsilon})$ bits, where the $O(n)$ term hides $3n + o(n)$ bits [18, §3.2]. Hence, Grafite is better in space than this data structure by an additive term $n + o(n \log \frac{L}{\varepsilon})$, thus it is closer to the space lower bound of approximate range emptiness data structures (Theorem 2.1).

On the other hand, the query time of Grafite is $O(\log \frac{L}{\varepsilon})$ while the query time of Goswami et al.'s data structure is $O((\log \frac{nL}{\varepsilon})/w)$. The former is higher than the latter when $n = O((L/\varepsilon)^{w-1})$. In the case $L/\varepsilon = O(1)$, then queries in both data structures take $O(1)$ time because it is usually assumed $w = \Omega(\log n)$.

**Rosetta.**Rosetta allows tuning the false positive probability of its per-level Bloom filters. We use the tuning from [25, §3.1] that achieves approximately $1.44 \cdot n \log \frac{L}{\varepsilon}$ bits of space by setting the

Table 1. Summary of known and new results achieved by range filters. Recall that $n$ is the number of input keys from a size-$u$ universe, $L$ is an upper bound on the query range size, and $\varepsilon$ is the false positive probability.

| | | Space in bits | Query time | Practical impl. |
|---|---|---|---|---|
| Heuristic | SuRF [40] | $(10+m)n + 10z + o(n+z)^*$ | $O(\log u)$ | ✓ |
| | SNARF [36] | $n \log K + 2.4n^\dagger$ | $\Omega(\log n)$ | ✓ |
| | Proteus [21] | ? | ? | ✓ |
| | bloomRF [27] | ? | $O(\log \frac{u}{n})$ | ✓ |
| | Bucketing (this paper) | $t \log \frac{u}{ts} + 2t + o(t)^\ddagger$ | $O(\log \frac{u}{ts})$ | ✓ |
| FPR-bounded | Theor. baseline (Sect. 2) | $n \log \frac{L}{\varepsilon} + O(n)$ | $O(L)$ | |
| | Goswami et al. [18] | $n \log \frac{L}{\varepsilon} + 3n + o(n \log \frac{L}{\varepsilon})$ | $O((\log \frac{nL}{\varepsilon})/w)$ | |
| | Rosetta [25] | $1.44 \cdot n \log \frac{L}{\varepsilon}$ | $\Omega((\log L)\log(2-\varepsilon))^\S$ | ✓ |
| | Grafite (this paper) | $n \log \frac{L}{\varepsilon} + 2n + o(n)$ | $O(\log \frac{L}{\varepsilon})$ | ✓ |
| | **Lower bound** (Thm. 2.1) | $n \log \left( \frac{L^{1-O(\varepsilon)}}{\varepsilon} \right) - O(n)$ | | |

* Using the space-efficient LOUDS-Sparse encoding, where $z$ is the number of internal nodes, and $m$ is the given number of suffix bits.

† $K \geq 1$ is a suitably large parameter of SNARF.

‡ $s$ is a positive integer parameter (higher values encode the input set more coarsely), and $t \in [1, \min\{n, u/s\}]$ depends on the input data.

§ Expected time. Worst-case time is $O(L \log \frac{1}{\varepsilon})$.

probability of false positives to $\varepsilon$ for the last-level Bloom filter and to $1/(2-\varepsilon)$ for each other upper-level Bloom filter. The space of Grafite is better, since $1.44 \cdot \log \frac{L}{\varepsilon} < \log \frac{L}{\varepsilon} + 2$ if and only if $L < 23.36\varepsilon$.

For the query time, the worst-case number of Bloom filter probes done by Rosetta is $O(L)$ and the expected number is $\Omega(\log L)$, as per the analysis in [25, §3.2]. The probe time of the last-level Bloom filter is $\Theta(\log \frac{1}{\varepsilon})$, which is higher than the $\Theta(\log(2-\varepsilon))$ probe time of each other upper-level Bloom filter. So the worst-case query time of Rosetta is $O(L \log \frac{1}{\varepsilon})$, which is worse than the query time of Grafite, and the expected query time of Rosetta is $\Omega((\log L)\log(2-\varepsilon))$, which is equivalent to the query time of Grafite if $\varepsilon$ is a constant.

**SuRF.** Let $z$ be the number of internal nodes in the Fast Succinct Trie at the core of SuRF, and recall it stores one leaf and $m$ suffix bits for each of the $n$ input keys. The trie uses the LOUDS-Dense encoding for the upper levels and LOUDS-Sparse for the lower levels. Following [40, §2.5], we assume the more space-efficient LOUDS-Sparse encoding is used, in which each node takes 10 bits. Considering the $o(z+n)$ bits for the rank/select data structures, the total space sums up to $nm + 10(n+z) + o(n+z) = (10+m)n + 10z + o(n+z)$ bits. From this analysis (confirmed by experiments), we infer that SuRF needs at least 10 bits per key, which can be restrictive in applications with a low space budget.

The query time of SuRF is given by the time to traverse the trie and then compare the suffix bits. Thus, for a trie of height $h$, the time is $O(h)$ if the suffix bits fit into a machine word (thus they can be accessed in constant time), and if each branching step takes constant time, e.g. because the trie has a constant bounded fan-out. Since the input keys are of length $O(\log u)$, the query time is $O(h) = O(\log u)$. Even for a fairly large $L \leq u\varepsilon/n$ (cf. Theorem 3.4), Grafite is faster than SuRF because it has query time $O(\log \frac{L}{\varepsilon}) = O(\log \frac{u}{n})$.

**SNARF.** SNARF was shown to take approximately $n \log K + 2.4n$ bits, where $K$ is a suitably large parameter impacting on the false positive rate [36, §5]. For the query time, the paper does not give a precise analysis, but we notice that it requires performing a binary search on the sample of $n/t$ keys (where $t$ is a constant) to identify the correct spline model, followed by decoding the compressed bit array. Due to the binary search, SNARF takes time $\Omega(\log \frac{n}{t}) = \Omega(\log n)$, which is already at least asymptotically the query time of Grafite. Indeed, Grafite binary searches on a range with $\min\{n, L/\varepsilon\}$ keys.

Under the assumption of uniform keys, uniform query workload, and $u \gg nK$, SNARF was shown to have a false positive probability of $(u/K)/(u - nL)$ for query ranges of size $L$ [36, §3]. This is approximated to $1/K$ under the additional assumption of $nL \ll u$. In such a restricted setting, SNARF takes $\log L - 0.4$ bits per key less than Grafite with $\varepsilon$ set to $1/K$. On the flip side, SNARF suffers a high false positive rate in correlated workloads (see Section 6.2).

**Proteus.** The Proteus paper [21] does not provide a closed formula for the space and the query time taken by this data structure. Indeed, Proteus tunes its configuration parameters $(l_1, l_2)$ via an algorithm whose inputs are the keys, a query workload, and a space budget (cf. [21, Alg. 1]). This makes it difficult to provide a satisfactory space bound other than for the extreme configurations that turn it into either a full Fast Succinct Trie, or a full Prefix Bloom Filter.

For the query time, we could not derive a satisfactory analysis either, but we observe that Proteus uses a Fast Succinct Trie on prefixes of uniform depth $l_1$ and a Prefix Bloom Filter for prefixes of length $l_2 > l_1$, thus it might require a trie traversal plus several queries to the Prefix Bloom Filter (our experiments will show that Proteus is much slower than Grafite).

**bloomRF.** The authors of bloomRF build a model of the false positive rate given a space budget in [27, §5–6]. We prefer to not report this model here due to its complicated design and assumptions, but we content ourselves to notice that it is influenced by the input data distribution (cf. the constant $C$ in [27]), thus making bloomRF a heuristic solution.

The query time of bloomRF is given by the time to compute $k = \lceil \log \frac{u}{n}/\Delta \rceil$ hash functions, where $\Delta \geq 1$ is a parameter of the data structure [27, §6]. Thus bloomRF has query time $O(\log \frac{u}{n})$, which is no better than Grafite for the same considerations we make above for SuRF.

**REncoder.** The authors of REncoder show in [38, §4] that, under some assumptions, a false positive probability of $\varepsilon$ can be obtained using $O(n(k + \log \frac{1}{\varepsilon}))$ bits of space, where $k$ is the number of hash functions used in REncoder. This result is hard to compare with Grafite due to the lack of $L$ in the space bound (which seems to conflict with the lower bound of Theorem 2.1), due to the big-$O$, and due to the use of $k$, which also impacts on the query time (no precise indications on how to set $k$ are given). In any case, the analysis in [38, §4.C] suggests that REncoder too is affected by correlated workloads, which is confirmed by our experiments of Section 6.2.

## 6 EXPERIMENTS

We now perform the largest experimental comparison among range filters, both in terms of dataset size and in the number of tested solutions, and we show that:

(1) The vast majority of existing range filters provide no filtering or much degraded filtering and query performance in the case of correlated query workloads. Instead, Grafite is among the (few) robust range filters, and it offers the overall best false positive rate (FPR) and query time already starting from mildly correlated query workloads.
(2) On uncorrelated query workloads, Bucketing offers, simultaneously, a filtering effectiveness that is very close to or better than the one achieved by the best-performing heuristic range filters, 5–13× faster queries, and 5–24× faster construction than them.

(3) Among robust range filters, Grafite is the best choice because it offers, simultaneously, the best FPR by up to 5 orders of magnitude, 9–92× faster queries, and 4–10× faster construction.

Then, we conclude this experimental section by summarising our findings in terms of recommendations on which range filter to adopt for an application (Section 6.7).

### 6.1 Experimental Setup

All the experiments are run on a machine equipped with a 1.80 GHz Intel Xeon E5-2650Lv3 CPU and 64 GB of RAM. The code of Grafite and the competitors is in C++ and is compiled with gcc-11. Our source code is available at https://github.com/marcocosta97/grafite.

**Competitors.** As motivated in Section 2, we compare Grafite and Bucketing with the following state-of-the-art range filters: SuRF [40], Rosetta [25], SNARF [36],[4] Proteus [21] and REncoder [38], including its variants REncoderSE and REncoderSS. This makes our study the largest one in terms of number of considered competitors.

Rosetta, Proteus and REncoderSE are auto-tuned on a sample of the queries with the procedures designed by the respective authors. For SuRF, we use real suffixes when testing against range queries and hashed key suffixes when testing against point queries, as suggested by [40].

**Datasets.** We use synthetic and real-world datasets used in previous range filters evaluations [21, 25, 36, 38, 40]:

- Uniform: 200M keys chosen uniformly at random from $[0, 2^{64})$.
- Books: 200M keys representing Amazon book sale popularity.
- Osm: 200M coordinates of locations from Open Street Map.

By using up to the entire dataset to build the range filters, we double the scale of the previously largest evaluation [36].

We build each range filter with space budgets ranging between $\approx 8$ and 28 bits per key, which covers a large spectrum of trade-offs [25, 38].[5] We ensure that the space of a range filter does not exceed an explicit encoding of the input keys, namely $\log \frac{u}{n} + 2$ bits per key via an Elias-Fano encoding, since this approach would solve the problem without false positives as discussed after Theorem 2.1.

**Query workloads.** Following the literature [21, 25, 36, 38], we execute 10M range emptiness queries of the form $[x, x + L - 1]$ in a single thread. We distinguish between batches of *point queries* in which $L = 2^0$, *small range queries* in which $L = 2^5$, and *large range queries* in which $L = 2^{10}$.

For the *synthetic* dataset (Uniform), the left endpoint $x$ is chosen according to the following strategies:

- Uncorrelated: $x$ is chosen uniformly at random from $[0, 2^{64})$.
- Correlated: a key $k$ is chosen uniformly at random from the dataset, and then $x$ is chosen uniformly at random from $[k, k + 2^{30(1-D)}]$, where $D$ is the *correlation degree* that ranges from 0 (uncorrelated) to 1 (correlated) [36]. If not explicitly varied, we set $D = 0.8$.

For the *real* datasets, the left endpoint $x$ is a key extracted (and removed) from the dataset. Notice that this query workload may be a mix of correlated and uncorrelated query ranges, depending on the distribution of the original input keys, from which $x$ is extracted.

[4]During the experiments we found that SNARF returns some false negatives, which is contrary to the definition of (range) filter. The authors believe this is due to computation overflows in the learned model, recommending the use of SNARF on significantly smaller input numbers (personal communication, Kapil Vaidya, 17 April 2023). We still experiment with SNARF, with the hope the false negatives do not affect its performance.

[5]SuRF takes no less than 10 bits per key, as per our analysis in Section 5, and some configurations are not shown due to crashes. Some configurations of Proteus results in the same design thus giving overlapped points in our figures.

In all the above strategies, we enforce the generation of *empty* queries by discarding the query ranges that intersect the dataset. This way, we evaluate the false positive rate (FPR) as the ratio between the number of "not empty" answers and the size of the batch. In a separate experiment, we also test the query time of range filters on *non-empty* queries. Note that the query time does not include the time to access a slow resource, such as a disk or a network drive, where the dataset might be stored. This time can vary greatly depending on the FPR and the hardware, or it might even be absent if the application requires no further check of a "not empty" answer (i.e. checking whether it is a true positive or not).

**Other datasets and query workloads.** We ought to report that we have also experimented with (i) a dataset generated from a normal distribution (mean of $2^{63}$, standard deviation of $2^{64} \times 0.1$, which allows covering the universe and generating large range queries) in combination with the Uncorrelated and Correlated query workloads, and (ii) the Uniform dataset in combination with a normal query workload. In all these cases, consistently with previously published evaluations [21, 36], we found no interesting change in the relative performance of range filters compared to using Uniform only, so we do not show them.

We have also experimented with the Fb dataset used in [21, 36, 38] but we found it to be too simple to be included in our evaluation because the mean value of the keys is $\approx 2^{38}$, and if we exclude the last 21 keys (that are larger than $2^{38}$), then an Elias-Fano encoding of the dataset would provide no false positives in just $\log \frac{2^{38}}{200 \cdot 10^6} + 2 \approx 12$ bits per key. Indeed, we report that Grafite, due to its optimal design, provides an FPR of 0 on Fb when given a budget of only 12 bits per key, while the other range filters may still give false positives (as shown also in the papers above).

### 6.2 Robustness of Range Filters

Our first experiment aims to differentiate robust range filters from heuristic ones, thereby emphasising the necessity of treating them separately due to their distinct guarantees. We consider the Uniform dataset and the Correlated query workload where the correlation degree $D$ is varied from 0 to 1, using a space budget for the range filters fixed to 20 bits per key.

The results in Figure 3 show that the FPR of Grafite and Rosetta is not affected by correlation, so we classify them as robust range filters. Grafite offers a better FPR than Rosetta by up to two orders of magnitude. The FPR of REncoder is affected by correlation, but this effect diminishes for larger range sizes. Grafite offers a better FPR than REncoder by up to four orders of magnitude.

The FPR of Proteus too suffers from increased correlation, but it does not reach 1. In the case of slightly correlated (i.e. $D < 0.5$) large range queries, Proteus shows a smaller FPR than Grafite, while Grafite has a better FPR in all the other cases. We stress that Proteus is auto-tuned on the input keys and the query workload, so it has an advantage due to overfitting. In applications where the workload shifts, it might not retain this advantage.

The FPR of SuRF, SNARF and Bucketing approaches 1 for correlation degrees beyond 0.4, thus failing to provide any kind of filtering (the drop of FPR of SuRF in point queries is expected because it ends up comparing hashed key suffixes). The same holds for REncoderSS and REncoderSE for correlation degrees beyond 0.7 (the latter in the case of large range queries).

For what concerns the query time, Grafite is the fastest effective range filter across the various query range sizes and correlation degrees (Bucketing is the fastest range filter, but it is not always effective, as commented above). The query time of Proteus, Rosetta and REncoder increases for increasingly large query ranges, up to about 3 orders of magnitude more with respect to Grafite. The query time of Proteus, REncoderSE and REncoderSS is affected by the correlation degree, which is another reason to classify them as non-robust.

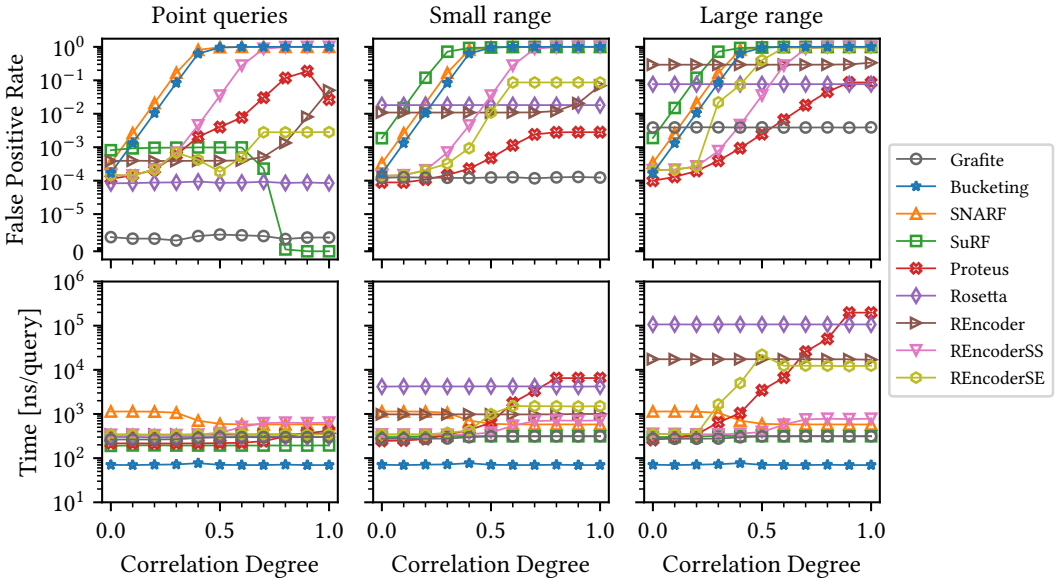


Fig. 3. The majority of range filters provide no filtering (Bucketing, SNARF, SuRF, REncoderSS) or much degraded filtering and query performance (Proteus, REncoderSE) as the key-query correlation increases. An adversary could exploit this weakness to make an attack on the availability of the system employing these heuristic range filters. Instead, Grafite and Rosetta are robust range filters, while REncoder is robust for large range queries. Grafite offers significantly better query time and FPR than Rosetta and REncoder.

In summary, with the exception of Grafite, Rosetta and, to a lesser extent, REncoder, the vast majority of range filters provide *no* filtering (SNARF, SuRF, REncoderSS) or *much degraded* filtering and query performance (Proteus, REncoderSE) in the case of correlated query workloads. This is a significant concern given the importance of these workloads in applications that care about the local properties of data [25] or given that malicious users could exploit this weakness to increase the network or disk accesses the range filters are deployed to prevent, thus posing a *risk* on the availability of a data system. Instead, Grafite is the overall best range filter in terms of FPR and query time already starting from mildly correlated query workloads, independently of the query range size.

Given the large number of competitors and the widely different guarantees they provide, our next experiments will focus separately on *heuristic range filters*, namely SNARF, SuRF, Proteus, REncoderSS, and REncoderSE, and on *robust range filters*, namely Grafite, Rosetta, and REncoder.

### 6.3 Evaluation of Heuristic Range Filters

Figure 4 shows the results of our experiments with heuristic range filters. Each column of the plot corresponds to a query range size (point, small and large), and each row corresponds to a dataset (the first two rows are the Correlated and Uncorrelated query workloads we experiment on Uniform data, and the other two rows correspond to the Books and Osm datasets). At the right of each row, we show a table with the query time of each range filter, averaged over the various space configurations and query range sizes, and next to each query time we show its ratio with respect to the fastest range filter.

In Correlated, in line with the experiment of Section 6.2, heuristic range filters provide no filtering (SNARF, REncoderSS, and SuRF, whose performance on point queries is commented in Section 6.2) or little filtering (Proteus and REncoderSE). These last two filters are actually advantaged

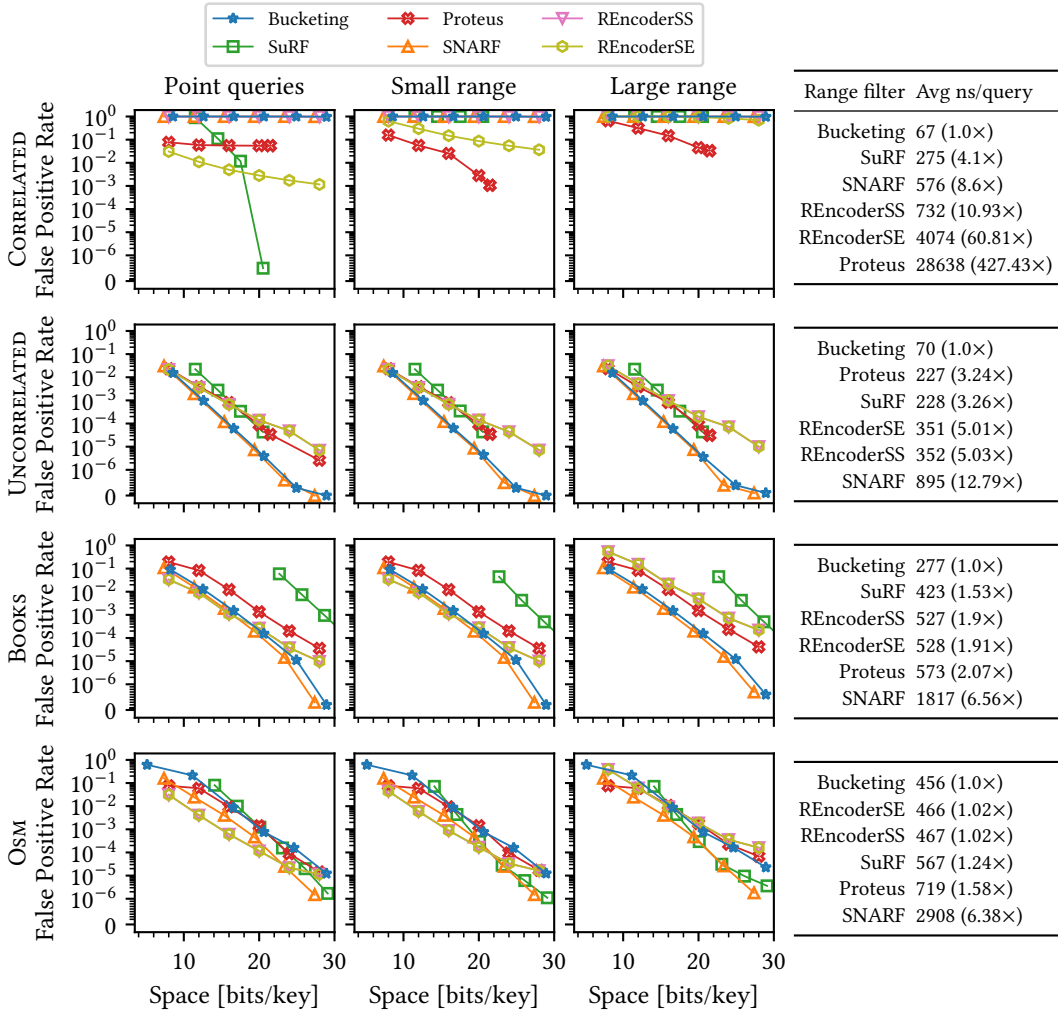


Fig. 4. Comparison among heuristic range filters. In the first row, only Proteus and REncoderSE provide some range query filtering (albeit unsatisfactorily, as discussed in Section 6.2) because they are auto-tuned on the correlated query workload. In the other rows, a simple solution like Bucketing provides very close or better FPR, and much better query time than all the other heuristic range filters. We remark that, unlike the other range filters, SNARF suffers from false negatives (see Footnote 4).

by being auto-tuned on the query workload, which might not be realistic in some applications due to rapidly-changing workloads or due to the additional space needed by query logs (which we did not account in their space usage).

For the other datasets, we notice that the filtering effectiveness of Bucketing essentially matches (on Uncorrelated and Books) or is very close (Osm) to the one of the best-performing heuristic range filter that is typically either SNARF (which, however, suffers from false negatives, see Footnote 4) or REncoderSE/SS, while simultaneously providing up to 13× faster queries than SNARF and up to 5× faster queries than REncoderSE/SS. Moreover, Bucketing provides the best construction times, as we will show in Section 6.6.

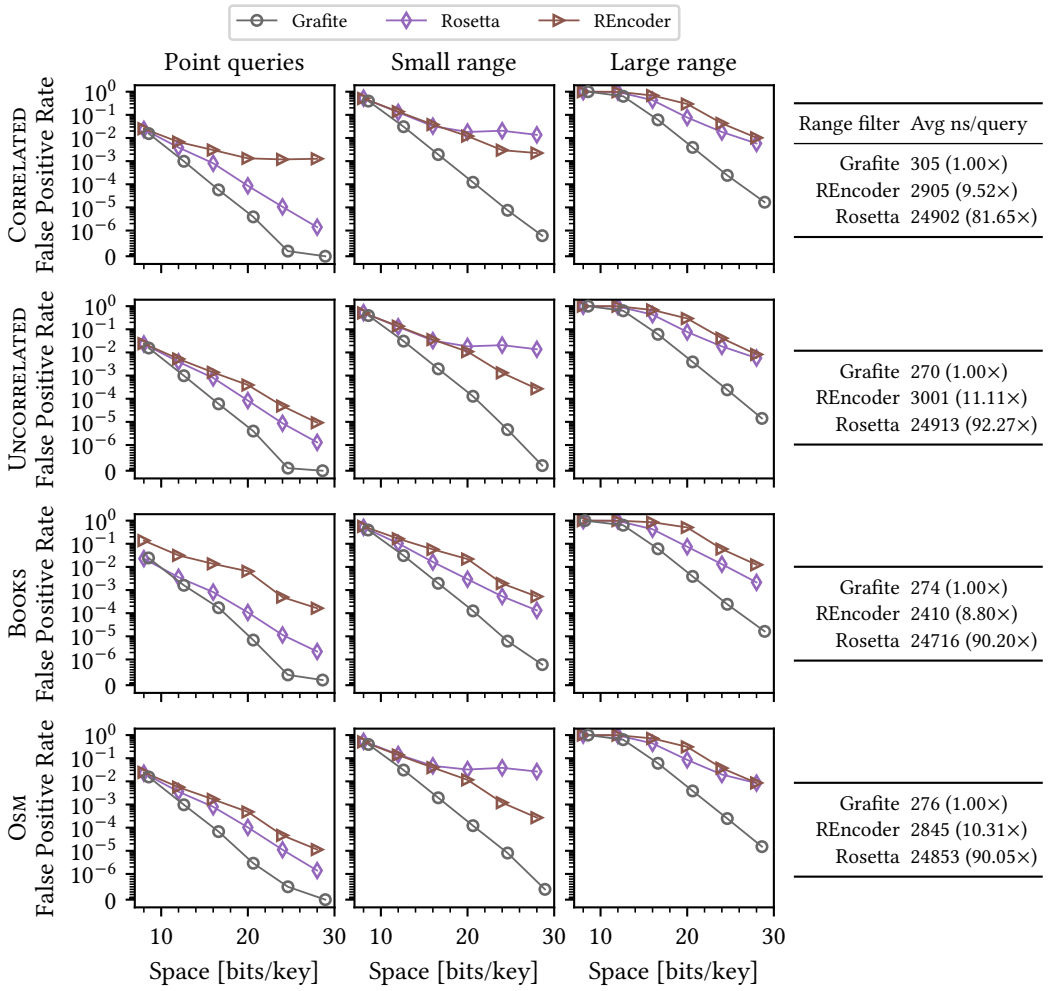


Fig. 5. Grafite dominates all other robust range filters by providing up to 5 orders of magnitude better FPR and up to 92× faster queries. These substantial improvements, coupled with its performance guarantees (Corollary 3.5), make Grafite the range filter of choice in applications handling a variety of data distributions and query workloads, even adversarial ones.

### 6.4 Evaluation of Robust Range Filters

We now experiment with robust range filters, namely Grafite, Rosetta, and REncoder (note this last one is slightly less robust in the case of small range queries, as discussed in Section 6.2).

As Figure 5 shows, in all datasets and query range sizes, Grafite dominates Rosetta and REncoder both in terms of FPR and query time. In particular, in terms of FPR, Grafite is up to 4 orders of magnitude more effective than REncoder, and up to 5 orders of magnitude more effective than Rosetta. In terms of query time, Grafite is 9.5–11.1× faster than REncoder, and 81.7–92.3× faster than Rosetta. Besides, we observe that Grafite has the most predictable FPR across all combinations of datasets, query workloads, and range sizes.

This consistent and substantial improvement of the state of the art corroborates the theoretical advantage of Grafite over prior solutions (Section 5), and demonstrates its potential to become the

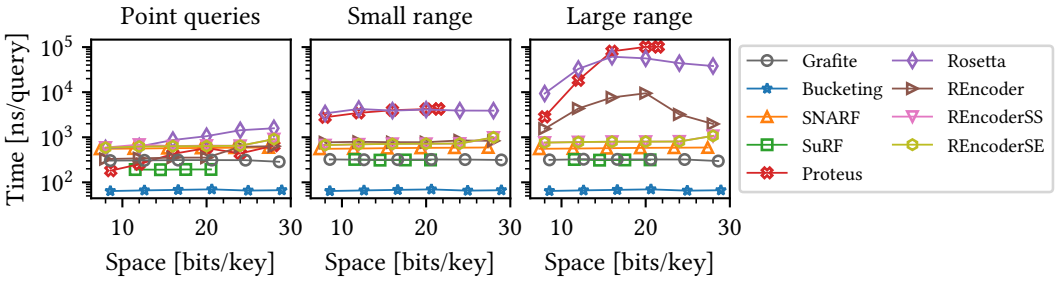


Fig. 6. The query time of range filters can vary a lot also in the case of non-empty queries, as shown in these plots with a logarithmic time axis. Grafite and Bucketing provide the best query times among robust and heuristic range filters, respectively.

range filter of choice in applications handling a variety of data distributions and query workloads (even adversarial ones).

### 6.5 Performance on Non-Empty Queries

We now experiment with queries that intersect the input dataset to show their impact on the query time. We use the Uniform dataset and create a query range $[x, x + L - 1]$ by first picking a key $k$ randomly from the dataset, and then picking the left endpoint $x$ randomly in $[k - L + 1, k]$.

Figure 6 shows the results: among heuristic range filters, Bucketing provides up to 3 orders of magnitude faster queries than the others; among robust range filters, Grafite provides the fastest queries, up to 1 order of magnitude faster than REncoder and up to 2 orders of magnitude faster than Rosetta.

A remark is necessary at this point. Even though filters are typically used in applications to prevent unnecessary (due to empty queries) network or disk accesses, they also increase CPU usage (regardless of the actual emptiness of the queried range). In some cases, high CPU usage might not compensate for the reduction in access frequency to a slow resource, thus making the choice of a query-efficient range filter preferable, even if it has a higher FPR. For example, Rosetta and Proteus in Figure 6 take up to 61.2 and 101.5 microseconds per query, respectively, which is comparable to the access latency of an SSD. In other cases, the opposite might be true, i.e. the cost of accessing a slow resource might be too high to be able to afford a range filter with a lower FPR but better CPU usage; thus the choice of which range filter to use ultimately depends on the specific application.

### 6.6 Construction Efficiency

Figure 7 shows the construction time of the various range filters as the number of keys increases from $10^5$ to $10^8$. We use the Uniform dataset (other datasets do not change our conclusions) and average the construction time over the different space budgets. We do not show REncoderSE and SS because their construction time is identical to that of REncoder. For both Rosetta and Proteus, the plot shows with a light colour the impact of the tuning process, which was evaluated with an Uncorrelated query workload of $n/10$ small range queries.

Among heuristic range filters, Bucketing is the fastest to construct, from 1.8 to 30.2× faster than the others. In particular, compared to its closest competitors in terms of FPR (see Section 6.3), Bucketing is 8.6–23.9× faster to construct than REncoderSE and 4.9–6.5× faster than SNARF.

Among robust range filters, Grafite is the fastest to construct, 6.7–10.3× faster than Rosetta and 3.8–7.9× faster than REncoder.

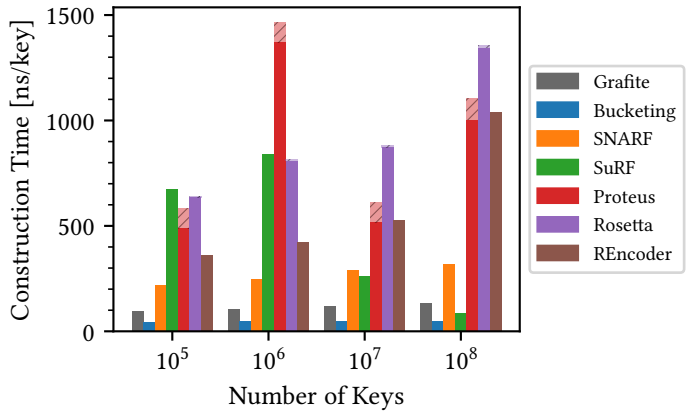


Fig. 7. Grafite has the best construction time among robust range filters (Rosetta and REncoder). Bucketing has the best construction time among heuristic range filters.

Furthermore, the construction time of both Grafite and Bucketing is linear for increasing values of $n$ (note the plot shows the construction time per key), which makes them highly scalable.

Actually, Grafite could achieve an even better construction time by using other sorting algorithms [3] in place of the `spreadsort` algorithm we are using, or by enabling multi-threaded sorting (not shown in the plots for fairness with the single-threaded competitors). For example, by using the parallel `block_indirect_sort` algorithm from the Boost library, the construction time for 200M keys with just 2 threads is reduced from 28.0 to 18.8 s (a 1.5× speedup), with 4 threads to 15.8 s (1.8× speedup), with 8 threads to 14.0 s (2.0× speedup)

### 6.7 Discussion and Recommendations

We now summarise our experimental findings by providing some guidance on which range filter to adopt for an application.

First and foremost, one should determine whether guarantees on the filtering effectiveness and query performance are needed regardless of the input data and future queries. If the answer is affirmative, then range filters providing a bounded FPR for a given space budget (and vice versa) are the choice, and among them Grafite is the best option.

For example, if future queries are correlated (i.e. close) to the input keys, the existing heuristic range filters provide little to no filtering, thus impacting the overall performance of the system (and possibly cloud costs) due to the network or disk accesses the filters are deployed to prevent. Correlated queries are common in practice [25], and malicious users can artificially issue them with just the knowledge of (a subset of) the keys. In these cases, Grafite is again the best option since it is unaffected by correlated queries.

If the application has no or infrequent correlated queries, and the query distribution does not change after the range filter is evaluated on a query sample and deployed, we recommend considering also Bucketing, Proteus, REncoderSS (and possibly SNARF, but refer to Footnote 4), which could provide better filtering effectiveness than Grafite. For example, Proteus can auto-tune itself and obtain a good FPR (see the rightmost plot in Figure 3). REncoderSS can offer a good FPR without any auto-tuning in some cases with small range queries (see last two rows of Figure 4). Bucketing always offered the best query and construction times in our experiments, and very good FPR in many cases (see Figure 4). Other than the FPR, query and construction times, deciding which range filter to adopt in a real application should consider factors like the cost of a false positive

(e.g., in terms of latency or cloud costs) and the frequency of queries (which impact on the CPU usage). Thus the best choice ultimately depends on the peculiarities of the application.

## 7 CONCLUSION

We introduced Grafite, a range filter that solves the lack of robustness in current practical solutions by providing strong theoretical guarantees on the false positive probability, optimal space usage, and very efficient and effective performance across many datasets, query workloads, and range sizes. We also introduced Bucketing, which simplifies the design of existing heuristic range filters while empirically providing very close or better filtering effectiveness, and much faster query and construction times, thus possibly resulting in a simple substitute for them.

For future work, we mention a more in-depth study of the Bucketing approach, which could be made workload-aware (e.g. by creating larger buckets for key ranges that are queried less frequently), or combined with Grafite. It is also worth engineering and experimenting with an extension of Grafite to string keys, for example by treating strings as integers and choosing $r$ as a power of two, say $r = 2^k$ for some $k > 0$, so that the hash function (1) can be efficiently implemented via bitwise and arithmetic operations as $h(x) = (q(x \gg k) + x) \,\&\, (r - 1)$, where $q$ could be chosen to be a practical hash function for strings like xxHash. Another open problem is to support the insertion of new keys in Grafite and Bucketing, for which dynamic Elias-Fano representations could help [33]. Finally, we mention again that Grafite can easily return an approximate count of the keys that intersect the given query range without any change in its space or query time complexity, thus potentially being a practical and efficient solution for this other interesting problem too [2].

## ACKNOWLEDGMENTS

We thank Mayank Goswami and Rasmus Pagh for clarifications on [18], and Subarna Chatterjee for clarifications on [25].

This work was supported by the European Union – Horizon 2020 Program under the scheme "INFRAIA-01-2018-2019 – Integrating Activities for Advanced Communities", Grant Agreement n. 871042, "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics" (http://www.sobigdata.eu), by the NextGenerationEU – National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) – Project: "SoBigData.it - Strengthening the Italian RI for Social Mining and Big Data Analytics" – Prot. IR0000013 – Avviso n. 3264 del 28/12/2021, by the spoke "FutureHPC & BigData" of the ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing funded by European Union – NextGenerationEU – PNRR, by the Italian Ministry of University and Research "Progetti di Rilevante Interesse Nazionale" project: "Multicriteria data structures and algorithms" (grant n. 2017WR7SHH).

## REFERENCES

[1] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013), 1714–1725. https://doi.org/10.14778/2556549.2556556

[2] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. 2001. Optimal static range reporting in one dimension. In *Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC)*. 476–482. https://doi.org/10.1145/380752.380842

[3] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2022. Engineering In-place (Shared-memory) Sorting Algorithms. *ACM Trans. Parallel Comput.* 9, 1 (2022), 2:1–2:62. https://doi.org/10.1145/3505286

[4] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2010. Fast Prefix Search in Little Space, with Applications. In *Proc.18th Annual European Symposium on Algorithms (ESA)*. 427–438. https://doi.org/10.1007/978-3-642-15775-2_37

[5] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[6] Andrei Broder and Michael Mitzenmacher. 2004. Network Applications of Bloom Filters: A Survey. *Internet mathematics* 1, 4 (2004), 485–509.

[7] Rayan Chikhi, Jan Holub, and Paul Medvedev. 2022. Data Structures to Represent a Set of $k$-long DNA Sequences. *ACM Comput. Surv.* 54, 1 (2022), 17:1–17:22. https://doi.org/10.1145/3445967

[8] David Richard Clark. 1996. *Compact Pat Trees.* Ph. D. Dissertation. University of Waterloo, Canada.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms* (4 ed.). The MIT Press.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Trans. Database Syst.* 43, 4 (2018), 16:1–16:48. https://doi.org/10.1145/3276980

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

[12] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. 2003. Longest Prefix Matching Using Bloom Filters. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM).* 201–212.

[13] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. 2022. Fast Succinct Retrieval and Approximate Membership Using Ribbon. In *Proc. 20th International Symposium on Experimental Algorithms (SEA)*, Vol. 233. 4:1–4:20. https://doi.org/10.4230/LIPICS.SEA.2022.4

[14] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM* 21, 2 (1974), 246–260.

[15] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proc. 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT).* 75–88. https://doi.org/10.1145/2674005.2674994

[16] Robert Mario Fano. 1971. *On the Number of Bits Required to Implement an Associative Memory.* Massachusetts Institute of Technology, Project MAC.

[17] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. 2017. BitFunnel: Revisiting Signatures for Search. In *Proc. 40th International ACM Conference on Research and Development in Information Retrieval (SIGIR).* 605–614. https://doi.org/10.1145/3077136.3080789

[18] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. 2014. Approximate Range Emptiness in Constant Time and Optimal Space. In *Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* 769–775.

[19] Thomas Mueller Graf and Daniel Lemire. 2020. Xor Filters. *ACM J. Exp. Algorithmics* 25 (2020), 1–16. https://doi.org/10.1145/3376122

[20] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS).* 549–554. https://doi.org/10.1109/SFCS.1989.63533

[21] Eric R. Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *Proc. 48th ACM International Conference on Management of Data (SIGMOD).* 1670–1684. Code available at https://github.com/Eric-R-Knorr/Proteus.

[22] Evgenios M. Kornaropoulos, Silei Ren, and Roberto Tamassia. 2022. The Price of Tailoring the Index to Your Data: Poisoning Attacks on Learned Index Structures. In *Proc. 48th International Conference on Management of Data (SIGMOD).* 1331–1344. https://doi.org/10.1145/3514221.3517867

[23] Florian Kurpicz. 2022. Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors. In *Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE).* 257–272. https://doi.org/10.1007/978-3-031-20643-6_19

[24] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. 2019. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *IEEE Commun. Surv. Tutorials* 21, 2 (2019), 1912–1949. https://doi.org/10.1109/COMST.2018.2889329

[25] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proc. 46th ACM International Conference on Management of Data (SIGMOD).* 2071–2086.

[26] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving facebook's social graph. *PVLDB* 13, 12 (2020), 3217–3230.

[27] Bernhard Mößner, Christian Riegger, Arthur Bernhardt, and Ilia Petrov. 2023. bloomRF: On Performing Range-Queries in Bloom-Filters with Piecewise-Monotone Hash Functions and Prefix Hashing. In *Proc. 26th International Conference on Extending Database Technology (EDBT).* 131–143. https://doi.org/10.48786/edbt.2023.11

[28] Gonzalo Navarro. 2016. *Compact data structures: a practical approach.* Cambridge University Press.

[29] Gonzalo Navarro and Javiel Rojas-Ledesma. 2020. Predecessor Search. *ACM Comput. Surv.* 53, 5, Article 105 (2020), 35 pages. https://doi.org/10.1145/3409371

[30] Daisuke Okanohara and Kunihiko Sadakane. 2007. Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX).* 60–70. https://doi.org/10.1137/1.9781611972870.6

[31] Giuseppe Ottaviano and Rossano Venturini. 2014. Partitioned Elias-Fano indexes. In *Proc. 37th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*. 273–282. https://doi.org/10.1145/2600428.2609615

[32] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. 2005. An Optimal Bloom Filter Replacement. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 823–829.

[33] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Dynamic Elias-Fano Representation. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 30:1–30:14. https://doi.org/10.4230/LIPIcs.CPM.2017.30

[34] Ely Porat. 2009. An Optimal Bloom Filter Replacement Based on Matrix Solving. In *Proc. 4th International Computer Science Symposium in Russia on Computer Science - Theory and Applications (CSR)*. 263–273. https://doi.org/10.1007/978-3-642-03351-3_25

[35] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2012. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Commun. Surv. Tutorials* 14, 1 (2012), 131–155. https://doi.org/10.1109/SURV.2011.031611.00024

[36] Kapil Vaidya, Subarna Chatterjee, Eric R. Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: A Learning-Enhanced Range Filter. *PVLDB* 15, 8 (2022), 1632–1644. https://doi.org/10.14778/3529337.3529347 Code available at https://github.com/kapilvaidya24/SNARF.

[37] Sebastiano Vigna. 2013. Quasi-Succinct Indices. In *Proc. 6th ACM International Conference on Web Search and Data Mining (WSDM)*. 83–92. https://doi.org/10.1145/2433396.2433409

[38] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. REncoder: A Space-Time Efficient Range Filter with Local Encoder. In *Proc. 39th IEEE International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/ICDE55515.2023.00158

[39] Mark N. Wegman and J. Lawrence Carter. 1981. New Hash Functions and Their Use in Authentication and Set Equality. *J. Comput. System Sci.* 22, 3 (June 1981), 265–279. https://doi.org/10.1016/0022-0000(81)90033-7

[40] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proc. 44th ACM International Conference on Management of Data (SIGMOD)*. 323–336. Code available at https://github.com/efficient/SuRF.

[41] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct Range Filters. *ACM Trans. Database Syst.* 45, 2 (2020), 5:1–5:31. https://doi.org/10.1145/3375660

[42] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2021. Succinct range filters. *Commun. ACM* 64, 4 (2021), 166–173. https://doi.org/10.1145/3450262

Received July 2023; accepted November 2023