SNARF: A Learning-Enhanced Range Filter

Kapil Vaidya
MIT
kapilv@mit.edu

Subarna Chatterjee
Harvard University
subarna@g.harvard.edu

Eric Knorr
Harvard University
eric_knorr@g.harvard.edu

Michael Mitzenmacher
Harvard University
michaelm@eecs.harvard.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

Tim Kraska
MIT
kraska@mit.edu

ABSTRACT

We present Sparse Numerical Array-Based Range Filters (SNARF), a learned range filter that efficiently supports range queries for numerical data. SNARF creates a model of the data distribution to map the keys into a bit array which is stored in a compressed form. The model along with the compressed bit array which constitutes SNARF are used to answer membership queries.

We evaluate SNARF on multiple synthetic and real-world datasets as a stand-alone filter and by integrating it into RocksDB. For range queries, SNARF provides up to 50x better false positive rate than state-of-the-art range filters, such as SuRF and Rosetta, with the same space usage. We also evaluate SNARF in RocksDB as a filter replacement for filtering requests before they access on-disk data structures. For RocksDB, SNARF can improve the execution time of the system up to 10x compared to SuRF and Rosetta for certain read-only workloads.

PVLDB Reference Format:
Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. SNARF: A Learning-Enhanced Range Filter . PVLDB, 15(8): 1632 - 1644, 2022.
doi:10.14778/3529337.3529347

1 INTRODUCTION

Filters are space efficient, but approximate, data structures that are used to answer membership queries on a set S. Filters allow significant improvements in the performance for an array of applications, including big data systems [44] and networking [4]. For example RocksDB [15], a Log-Structure-Merge Tree (LSM) [39] based key-value store, stores data onto disks in blocks (called SST's). However, because of the LSM structure, RocksDB often needs to load several blocks from disk into main memory to determine which block contains the data for a given search key. To avoid loading disk blocks that do not contain the search key, RocksDB creates a filter per block for all keys stored in the block.

Point filters, such as Bloom Filters, support point queries of the form: "Is x in the set S?". Range membership filters answer more general queries of the form "Is there a key in the set S in between values p and q?" [1, 34, 47]. Here we are focused on approximate filters that guarantee that there are no *false negatives*. This is an important property many applications/systems require. There may, however, be false positives. For point queries, if the filter returns *true* for a search key, the key *might or might not be* contained in the block, but if it returns *false* it is guaranteed that the key is not in the set/block; and this extends similarly to range queries. The probability of a false positive for a key not in the set is the false positive rate (FPR) of the filter; the FPR can be defined similarly for range queries.

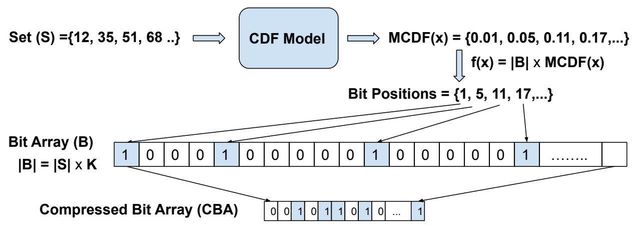
In RocksDB, filters are usually orders of magnitude smaller than the blocks and are cached in main memory. Before loading a disk block into main memory, the filters are checked if the key might be contained in the block. A filter with low false positive rate helps to significantly reduce the number of unnecessary I/O requests to disk blocks to find the key. The benefit a filter can provide depends on the trade-off between its false positive rate and the size of the filter; the smaller and the more precise, the better it is. Interestingly, the latency of a filter to process a query normally matters less as they tend to protect against very expensive operations (e.g., disk or other cold storage access) that are often orders of magnitude slower (see also Experiment 6.2.1).

Range Filters: Range queries are often used in social web applications [9], distributed key-value storage replication [43], statistics aggregation for time series workloads [27], and SQL table accesses [32]. For example, from a table of customer orders, one might ask the following SQL query to retrieve all the orders between two particular dates: SELECT * FROM Orders WHERE Order_Date BETWEEN "07-14-2014" AND "07-21-2014" . Past work has shown that range filters can significantly improve the performance of systems for synthetic and real-world workloads. For example, [34, 47] showed that workloads on RocksDB can benefit from range filters, whereas [1] showed the advantages of range filters for Hekaton, which is part of the MS SQL Server.

Existing Range Filter Designs: Past efforts to provide range filtering resulted in the current state-of-the-art filters Succinct Range Filter (SuRF) [47] and Rosetta [34]. SuRF utilizes a compact trie-like data structure which can filter arbitrary range queries, whereas Rosetta utilizes a different approach by using a Bloom filter (a point query filter) [3] for range queries along with the help of a hierarchy of prefix Bloom filters. Unfortunately, which of the two filters is better depends highly on the workload. For the same filter size, Rosetta has a lower false positive rate for very short range sizes because of its clever combination of Bloom and prefix filters, whereas SuRF has a lower FPR otherwise.

SNARF: In this paper, we introduce an entirely new approach to range filters, called Sparse Numerical Array-Based Range Filters

Proceedings of the VLDB Endowment, Vol. 15, No. 8 ISSN 2150-8097.
doi:10.14778/3529337.3529347



Set (S) ={12, 35, 51, 68 ..}
CDF Model
MCDF(x) = {0.01, 0.05, 0.11, 0.17,...}
f(x) = |B| x MCDF(x)
Bit Positions = {1, 5, 11, 17,...}
Bit Array (B)
|B| = |S| x K
1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 ....
Compressed Bit Array (CBA)
0 0 1 0 1 1 0 1 0 ... 1

Figure 1: SNARF Idea: Given a set of keys S, SNARF builds a model $MCDF(x)$ to estimate the empirical cdf of the keys, which it then uses to set corresponding bits in a large bit array B for all $x \in S$. This sparse bit array which encodes key information is then compressed. The model and the compressed bit array are the main parts of SNARF data structure.

(SNARF). SNARF is a learning enhanced range filter[1] that models the data distribution of the underlying key set S. SNARF then uses the model to encode partial information of the data in a sparse bit array. SNARF controls the false positive rate by changing the size of the bit array. The sparse bit array is then compressed to store it efficiently. SNARF answers range queries by using the model to extract the relevant information from the compressed bit array. Exploiting the data distribution and using effective compression schemes allow SNARF to encode the data set more effectively than previous schemes, leading to better space/false positive rate tradeoffs, while being competitive in terms of query latency.

SNARF Results: We evaluate SNARF on multiple synthetic and real-world numerical datasets against state-of-the-art range filters, such as SuRF and Rosetta, and also against point filters, such as Bloom filters [3] and Cuckoo filters [16]. We use a variety of query workloads, such as uniform, sampled from real-world, skewed (certain part of data is queried more often), and correlated (query endpoint is close to existing key) to test the effectiveness of the filters. For range queries on both real-world and synthetic datasets, SNARF is consistently able to provide up to a 50x better FPR than SuRF under the same space budget, and SNARF has up to 100x better FPR than Rosetta under same space budget. We do note, however, that performance depends on the dataset and query structure; for example, we have found that Rosetta is better than SNARF specifically in the case where the query workload has very short range queries and high correlation between queries and keys. Moreover, for point queries, SNARF can empirically provide FPRs that are better than Bloom filters and slightly better than Cuckoo filters under the same space budget across a diversity of query workloads.

Finally, we measured SNARF's impact on performance of an end-to-end system by integrating it with RocksDB. Here we found that SNARF can improve the workload execution time by up to 10x compared to SuRF and Rosetta for certain read only workloads.

In summary, we make the following **contributions**:

- We introduce SNARF, a novel range filter which combines models and compression schema (Section 2).
- We provide a heuristic theoretical analysis of SNARF that matches our empirical experiments well (Section 3).
- We discuss possible extensions of SNARF, including support for updates and support for approximate count queries (Section 4).
- We evaluate SNARF against state-of-the-art baselines and test the impact SNARF can have on a real system like RocksDB (Section 6).

[1]We acknowledge that the term "learned" range filter might be a misnomer as we use simplistic modelling of the data using linear splines. However, the name is in line with previous works [14, 17, 29, 30].

2 SNARF: A LEARNED FILTER

We first explain the idea behind SNARF (see Sec.2.1). Later, we describe the details of the model (see Sec.2.2) and the compressed bit arrays (see Sec.2.3.1).

2.1 SNARF Description

2.1.1 ***SNARF Construction:*** Given a set of keys $S = \{x_1, x_2, \ldots, x_n\}$, we want to build a filter that answers range queries on this set. SNARF maps the keys into a bit array B, which has $|B| = K \times n$ bits for a suitably large K[2], via a monotonic function f. Initially, all bits are 0, but bit position $f(x_i)$ is set to 1 for all $x_i \in S$. The exact mapping function f is $f(x) = \lfloor MCDF(x) \times nK \rfloor$, where MCDF is a monotonic estimate of the empirical CDF (eCDF) of the keys in S. Storing an entire sparse bit array directly is not space-efficient, so SNARF stores a compressed version of the bit array. The compressed bit array (CBA) encodes the locations of the one bits in the array. Fig.1 illustrates the idea of SNARF.

Alg.1 has the pseudo-code for SNARF construction. Given a set of keys S and scale factor K for the bit array, the construction algorithm outputs a model of the eCDF of the keys in S and the compressed bit array. The first step is to train a model to estimate the eCDF of the keys. In the next step, this model is used to generate the set of bit positions in the bit array that are set to one. The bit array is then compressed into the CBA.

2.1.2 ***SNARF Range Query:*** To answer a range query $[p, q]$, SNARF uses the model to get the bit positions $f(p)$ and $f(q)$ corresponding to the query endpoints. The data structure then returns true if a one bit is found in the range $[f(p), f(q)]$ of B and false otherwise. Alg.2 shows the pseudo-code for SNARF range query. Note that we want our CBA structure to efficiently support queries of the form: "Is there a one bit between bit positions a and b (inclusive)?".

Standard rank-select structures [21, 24, 40, 49] can provide compressed bit arrays with an efficient predecessor query which can be used to answer such queries. (One can check if the first one bit preceding b is before or after a.) Such a structure is naturally more efficient than decompressing the entire array and checking all bits between a and b. While rank-select structures could be used to speed up the computation of the a predecessor operation, we find they take more space than alternatives to do so. In our case, space is the primary resource we want to optimize for. This is because the latency of a filter to process a query normally matters less in RocksDB (see Sec.6.2.1). Also, with exponentially growing data, it is important to be able to filter more data with smaller filters. Thus, SNARF uses encoding schemes which provide near-optimal compression rather than fast query responses. We discuss simple techniques to optimize query response times in Sec.2.3.2.

[2]We use K to control the FPR of the structure which we discuss in detail later on

Algorithm 1 SNARF Construction:

```
Input $S$ - set of keys
Input $n$ - number of keys
Input $K$ - Scaling factor for the bit array size
Output $MCDF$ - Monotonic CDF estimate of keys
Output $CBA$ - Compressed bit array
Function $Train(S)$ - function that returns a model to estimate the cdf of keys in set S
Function $Encode(S)$ - function that encodes the numbers in the set S
1:  procedure CONSTRUCTION($S, K$)
2:  //Building the monotonic CDF model for set of keys
3:      $MCDF \leftarrow Train(S)$
4:
5:  //Get Bit positions that are set to one
6:      $BitPositionList \leftarrow \{\}$
7:      for $key$ in $S$ do
8:          $BitPositionList$.add( $\lfloor MCDF(key) \times nK \rfloor$)
9:
10: //Compress the Bit Positions that are set to one
11:     $CBA \leftarrow Encode(BitPositionList)$
12:
13:     return $MCDF, CBA$
```

Algorithm 2 SNARF Range Query

```
Input $n$ - number of keys
Input $K$ - Scaling factor for the bit array size
Input $MCDF$ - Monotonic CDF estimate of keys
Input $CBA$ - Compressed bit array
Input $p, q$ - the range query endpoints
Output $r$ - boolean answer of the range query
Function $CheckOneBit(a, b)$ - function that returns true if there is a 1 bit between bit locations $[a, b]$ else false.
1: procedure RANGEQUERY($MCDF, K, n, CBA, p, q$)
2: //Get the bit location of the query endpoints
3:     $LowerBitLoc \leftarrow \lfloor MCDF(p) \times Kn \rfloor$
4:     $UpperBitLoc \leftarrow \lfloor MCDF(q) \times Kn \rfloor$
5:
6: //Check if 1 bit exists in the range.
7:     $r \leftarrow CBA.CheckOneBit(LowerBitLoc, UpperBitLoc)$
8:     return $r$
```

2.1.3 *Essential properties of Mapping Function f.*

Monotonicity: The monotonicity of the mapping function, so that $p < q \implies f(p) \leq f(q)$, is an essential property that ensures no false negatives in SNARF. Monotonicity ensures that for any range query $[p, q]$ with $p < q$, any key from S between the query endpoints will be mapped to a position between the bit positions of these endpoints. That is, if $x_i \in [p, q]$, then $f(p) \leq f(x_i) \leq f(q)$; there is a bit set in the range $[f(p), f(q)]$. Note, however, that it is possible that $x_i \notin [p, q]$, but either $f(x_i) = f(p)$ or $f(x_i) = f(q)$, leading to false positives.

Uniform Mapping: SNARF aims for a uniform mapping into the bit array B for performance reasons; that is, we desire the bits set in the array to be as equally spaced as possible. Mapping the keys approximately uniformly allows the range filter to be robust to skewed query workloads (workloads where certain part of the range is queried more often) as we discuss in detail in Sec.3. The empirical cumulative distribution function of a (discrete) set S has the property that it maps the keys uniformly over the range $[0, 1]$. Hence, SNARF makes use of a monotonic CDF model of the set S to achieve a monotonic and approximately uniform mapping of the keys. The details of the model we utilize are presented in Sec.2.2.

2.2 Model Details

As discussed before, the model needed for SNARF must be monotonic and provide an estimate of the empirical CDF. Further, we want the space overhead added by the model to be small. Here, we present models for fixed size numerical values such as doubles, floats, and 32/64/128 bit signed/unsigned integers. Recently, a hierarchy of linear models have been used for indexing numerical keys [14, 17, 25, 29]. This ensemble of linear models is both small in size and provides fast evaluation for numerical values.[3] However, these models do not always guarantee monotonicity.

Inspired by them, we use linear spline models for CDF estimation. Given a set S, the idea is to use a small sample of keys from the input set and build linear models between consecutive keys in the sample to estimate the CDF as shown in Fig.2. This sample is stored in a sorted order, and we refer to it as the key array. The size of the sample determines how large the model is and the quality of the CDF estimation. Larger samples lead to better CDF approximation and larger models which increase the space used by SNARF.

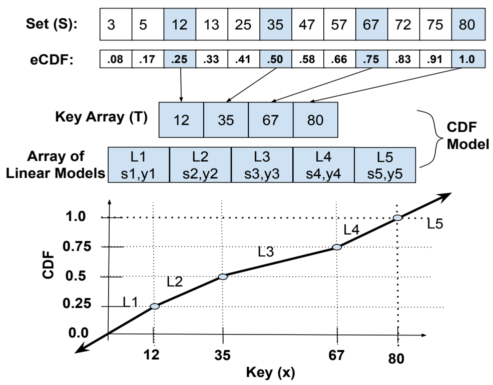

Set (S): 3 5 12 13 25 35 47 57 67 72 75 80
eCDF: .08 .17 .25 .33 .41 .50 .58 .66 .75 .83 .91 1.0
Key Array (T) 12 35 67 80
CDF Model
Array of Linear Models
L1 s1,y1
L2 s2,y2
L3 s3,y3
L4 s4,y4
L5 s5,y5
CDF
1.0
0.75
0.5
0.25
0.0
L1
L2
L3
L4
L5
12
35
67
80
Key (x)

Figure 2: SNARF Numerical Model

Querying the Model: The number of keys stored by the model is one less than the number of linear models. The first step is to binary search in the sorted array of keys ($MCDF.keys$). The number of keys in the array that are less than the query point x gives the index to the linear model parameters that are supposed to be used. We then use the corresponding line's slope and intercept to obtain the final estimated CDF value for the value. If the computed CDF is outside the range $[0, 1]$, we correct the value to 0 or 1 as appropriate.
Training the Model: During training, we sort the input set S and compute the empirical CDF. We then choose keys at regular intervals (every $(N/R)^{th}$ key for a suitable R) and these keys form endpoints for linear spline models. Between every pair of consecutive sample points, we compute the slope and y-intercept of the line segment connecting the two points.

The number of line segments we use in our model can be tuned to improve the tradeoff between the CDF estimate and overall model size. The more lines the better one can potentially approximate the CDF, but the more space used as well. A good value for number of line segments will depend on the dataset. We empirically found that using $|S|/1000$ line segments generally gives good CDF estimates along with small model size. The space overhead of model when using $|S|/1000$ line segments is approximately 0.2 bits per key.

[3]We experimented with monotonic cubic splines [18] but found them to be slightly worse than a series of linear models

2.3 Managing the Bit Array

We describe compression schemes for bit arrays and simple techniques to make range queries faster on the compressed bit array.

2.3.1 ***Compressing the Bit Array.*** The main idea for space efficient encoding of a sparse bit arrays is to simply encode the positions of the one bits. We discuss two such specific techniques.
Golomb Coding: Golomb coding is a form of lossless delta compression which is the optimal lossless compression scheme for a sparse bit array with uniformly randomly spread one bits [20].[4]

In general delta compression schemes, the values to be encoded are sorted and then the differences, or deltas, between consecutive values are stored efficiently. In Golomb coding, for each delta value X to encode, X is divided by a fixed constant M to obtain a quotient $\lfloor X/M \rfloor$ and a remainder $X\%M$. The remainder is stored in a fixed length binary format using $\log_2(M)$ bits, whereas the quotient, which is expected to be small, is encoded in unary. The choice of the fixed constant is important in determining the size of the compressed array. For uniformly randomly generated values, the average delta value is the optimal constant. In our case, the average delta value will be the bit array size nK divided by number of one bits, which is approximately n. We therefore use K as the constant for our Golomb coding. Fig.3 describes an example of Golomb encoding a sparse bit array.

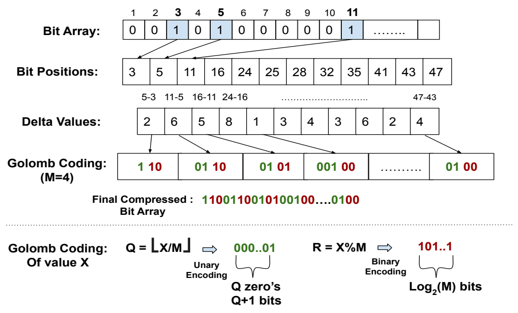


Figure 3: Golomb coding

In order to check if a bit is one in the range of bit positions $[a, b]$, one needs to decode the array from the start by adding the deltas one by one. This process continues until you either find a 1 after a and before b or you go past b. Decoding the array for each query can be slow; we discuss better approaches shortly.

Elias-Fano Encoding: Elias-Fano is a form of entropy encoding to represent a monotone non-decreasing sequence of N integers. The bit positions in our case form the non-decreasing sequence. In Elias-Fano encoding, the integers are first binary encoded using $\log_2(M)$ bits if $[0, M)$ is the universe range. This representation is split into two parts: an upper $\log_2(N)$ bits and the remaining lower $\log_2(M/N)$ bits. The lower bits are trivially stored by concatenating them and this uses $N \log_2(M/N)$ bits. The higher part is represented by a bit vector of $2N$ bits as follows. We first create a count of occurrences of upper bit values for all values between $[0, N-1]$. We then put this count vector in unary notation; that is, each count is represented in unary (a sequence of 1s) with 0 stop bit between values. This leads to $2N$ total bits, with one bit set to 1 for each of the N elements and one 0 bit for each possible values for the upper bits. Finally, the Elias-Fano representation is the concatenation of these two vectors. Fig.4 describes a Elias-Fano encoding for a set of integers. In our case, we will be encoding the bit positions so $M = nK$ and $N = n$. Thus, we will binary encode the $\log_2(K)$ lower bits and unary encode the upper $\log_2(n)$ bits for each bit position.

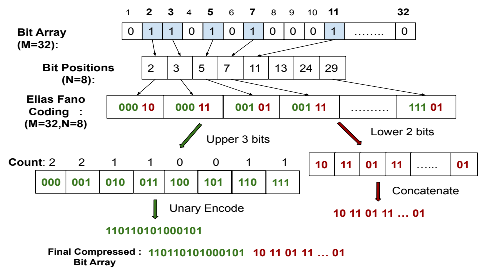


Figure 4: Elias Fano Encoding

While checking if a bit is one in the range $[a, b]$, one can decode the upper bit array from the start (similar to Golomb coding) but accessing the lower part is not always necessary. Any bit position with upper bit value less than $\lfloor \frac{a}{2^{M-N}} \rfloor$ will definitely be smaller than a. This is because the value of the lower part can be at max $2^{M-N} - 1$ and that is not enough for it to be greater than a. Thus, we only need to check the lower bits if the upper bits are relevant. This property greatly reduces the amount memory accessed during a range query compared to Golomb coding. On the other hand, Elias-Fano coding uses slightly more space ($\approx$ 0.4-0.5 bits per key) than Golomb coding. Thus, Elias-Fano coding has a faster query time compared to Golomb coding but with a slightly higher space overhead.

2.3.2 ***Making Compressed Bit Arrays Efficient:*** As noted earlier, simply decoding from the beginning is an expensive approach; in the worst case, we might need to decode the entire bit array. To avoid this, we split the bit array into equal sized segments and then compress them separately. If nK is the bit array size and n is the number of keys, we divide the bit array into $K\beta$ sized segments generating n/β segments. Now to perform a range query $[p, q]$ for S we only need to decode the corresponding segments that overlap the range $[f(p), f(q)]$ in the CBA. On an average each segment has around β one bits. While answering the range query $[p, q]$, one only needs to consider segments from segment number $\lfloor f(p)\beta/n \rfloor$ to $\lfloor f(q)\beta/n \rfloor$. The first value greater then $f(p)$ either exists in segment number $\lfloor f(p)\beta/n \rfloor$ or in the next non-empty segment. Generally, decoding segment number $\lfloor f(p)\beta/n \rfloor$ is sufficient as we find a number greater than p in it or the next segment.

Even though the uncompressed bit array size is the same, the compressed size of each segment differs. Hence, we need to store the starting point of each compressed segment. This creates a tradeoff between space used by SNARF and the range query response time provided by SNARF. Using more segments would lead to faster queries but larger metadata space overhead. Empirically, we found that $\beta \approx 50 - 100$ provides good range query response times and has negligible memory overhead (shown in Sec.6.1.6).

[4]We expect nearly uniform randomly place one bits in our case.

3 ANALYSIS

In the following section, we provide an analysis regarding the trade-off between the space used by SNARF and the corresponding false positive rate. We show that for point queries SNARF is competitive with Bloom filter variants. The results extend to queries over small ranges in the natural way. While this analysis is only for certain workloads, it provides understanding for why SNARF works well in many scenarios.

We start by showing that SNARF for uniformly distributed queries (point queries and small ranges) provides an FPR of approximately $1/K$ while using $2.4 + \log_2(K)$ bits per key.

Initial Assumptions: We assume all key values are in the range $[0, z]$ for some suitably large z with $z >> nK$.[5]

Notation: Our set S of n keys $S = x_1, x_2, .., x_n$. We use a model with t linear models and thus, we have one linear model per n/t points. Recall we use a bit array of size $n \times K$ and divide it into blocks of size $K\beta$ bits for faster queries; we assume also a per block metadata of c bits.

Analysis: Our goal is to show that for uniform workload SNARF provides a false positive rate of $1/K$ for point and small range queries, while using around $2.4 + \log_2(K)$ bits per key. For uniform point queries, we have z total queries out of which $z-n$ are negatives. We proceed by showing that SNARF only gives false positive for $(z - n)/K$ point queries.

We divide the key range into t segments of size $\Delta z_1, \Delta z_2, ...\Delta z_t$, where $\sum_{i=1}^{t} \Delta z_i = z$ and each segment has a separate linear model. Let the corresponding segment endpoints be $z_0, z_1, \ldots, z_t$. For each segment the following holds:

- The number of keys from S in the segment $[z_{i-1}, z_i)$ is n/t as we build separate linear model for every n/t keys.
- Over each segment $[z_{i-1}, z_i)$, we have a total of $z_i - z_{i-1}$ distinct possible point queries out of which $((z_i - z_{i-1}) - n/t)$ are negative queries.
- The keys of S in the segment $[z_{i-1}, z_i)$ are evenly spread over the range $[(i-1)(nK/t), i(nK/t))$ in the bit array.

An implication of these statements is that for a non-key in the range $[z_{i-1}, z_i)$, the probability of false positive is at most the number of 1 bits in the range, which is at most n/t, divided by the corresponding size of the range in the bit array, which is nK/t. It follows that the number of keys that give false positives is

$$\sum_{i=1}^{t} ((z_i - z_{i-1}) - n/t)) \times \frac{n/t}{nK/t} = \frac{1}{K} \sum_{i=1}^{t} ((z_i - z_{i-1}) - n/t)) \,.$$

But since $\sum_{i=1}^{t} (z_i - z_{i-1}) = z$ the summation collapses, giving the total number of false positives is $(z - n)/K$. Since, we have $z - n$ negatives in the range the false positive rate turns out to be $1/K$ for the uniform distribution. This shows that for uniform workload using a bit array that is K times larger than the number of keys yields a false positive rate of approximately $1/K$ for point queries.

Extending to small ranges: Here, we perform a similar analysis for uniform range queries of size R. The main idea is to show that the total number of false positive range queries is at most the total number of false positive point queries. We show this for a region and then aggregate across the entire domain.

Consider a region $[p, q]$ of the domain such that all points in the region map to a one bit and values just outside the region map to zero bits. That is, the bit at location $f(p - 1))$ is 0, and the bit at location $f(q + 1))$ is 0, but for all $x \in [p, q]$, the bit at $f(x)$ is 0. Let l be the number of keys in this region. The number of false positive point queries is $(q + 1 - (p + l))$. The total number of range queries of size R intersecting with the region would be $(q + 1 + R - p)$. Out of the these, the number of true positive range queries is at least $(l + R)$ as we show later. Thus, the false positive range queries end up being at most $(q + 1 + R - p) - (l + R) = (q + 1 - (l + p))$ which is exactly equal to the number of false positive point queries in the region. Now, we can simply sum up the queries in each such region to get the total number. Thus, we can conclude that total number of false positive range queries is at most the total number of false positive point queries.

We argue that the number of true positive range queries is at least $l + R$ in a region. Let $k_1, k_2, ...k_l$, be the keys in the region $[p, q]$ in sorted order For the smallest key k_1 in the region, we have $R + 1$ true positive ranges of size R as enumerated by set $\{(k_1 - R, k_1), (k_1 - R + 1, k_1 + 1), ...(k_1 - R + R, k_1 + R)\}$. Now, if we consider k_2, then we can add a unique true positive range query $(k_2, k_2 + R)$ to the set. Similarly, every subsequent addition of a key increases the size of the set by at least one. Earlier, we showed that the total number of false positive point queries is z/K. The number of negative range queries is at least $z - nR$. Thus, the false positive rate for range queries is at most

$$\frac{(z/K)}{z - nR} \approx \frac{1}{K}$$

Here the approximation holds for small ranges R, so that $nR << z$, yielding a false positive rate close to $1/K$.

Extending to skewed workloads: We assume there is a distribution with cdf $w(x)$ that generates a point query, such that over suitably small intervals $[z_{i-1}, z_i]$, the probability of querying any point in the range is approximately uniform. Each segment would independently have a false positive rate of approximately $1/K$, thus it follows that the false positive rate for point queries is:

$$FPR = \sum_{i=1}^{t} (w(z_i) - w(z_{i-1}))) \times \frac{n/t}{nK/t} = \frac{1}{K} \sum_{i=1}^{t} (w(z_i) - w(z_{i-1}))) \,.$$

The ratio in the summation is approximately $1/K$, giving an approximate false positive rate of $1/K$.

We indeed observe that the false positive rate is approximately $1/K$ for point queries as well as range queries over various query distributions for SNARF empirically for synthetically generated datasets and workloads, as we discuss in Sec. 6.1.1.

Model Size: The size of the model is dependent on the number of keys and linear models it stores. We assume the linear models utilize 2 double values and hence we use 128 bits per linear model. For uint64 integers, we need 64 bits to store each key in the key array. In our experiments, for example, we stored $n/1000$ models and thus, the space used by model is around $192n/1000$. This accounts to approximately 0.2 bits per key.

[5]If $z < nK$, then each value in the domain would likely map to a different bit position. If each value has a different bit position then false positive rate would be zero.

Compressed Bit array Size: Given that the bit array is Kn bits long, the compressed version of the bit array using Golomb and Elias Fano coding takes no more than $2n + n\log_2(K)$ bits in total[6]. This is because the unary code for both Golomb and Elias Fano coding takes no more than $2n$ bits and the binary representations take $\log_2 K$ bits per key . The space overhead due to dividing the compressed bit array into blocks of size βK bits is approximately nc/β, bits where c is the number of bits per block needed to store the metadata. In our experiments, c is around 20 bits and we fix β to be around 100. Thus, the space used by SNARF per key is around $(2 + \log_2(K) + c/\beta + 192n/1000) \approx (2.4 + \log_2(K))$ bits[7].

Recall that our heuristic analysis gives a false positive rate for point queries of $1/K$. This is close to the theoretical space lower bound of $\log_2(K)$ bits per key for Bloom filter variants [4]. Empirically we observed that SNARF gives a similar false positive rate for point queries as cuckoo filters with the same space usage on synthetic datasets and workloads as shown in Sec.6.1.1.

4 DISCUSSION

We discuss here various aspects of SNARF behavior, including performance on workloads with high key-query proximity, SNARF use for other queries, and handling updates.

4.1 Key-Query Correlation in Workloads

For the purpose of range filters, we say that a workload is correlated with the data, if the end point of a query is consistently close to some key. Assume a data set contains all multiples of 10 from 1 to 1000 (e.g., 10, 20, 30,...,1000). A correlated workload would be one which consistently ask for ranges close to these keys (e.g., 10.01-11.01, 28.99-29.99, etc.). When a query end point is consistently close to an actual key but the query does not include a key, it may yield a false positive in SNARF and SuRF. Meanwhile, Rosetta is relatively unaffected by correlated queries as it is uses Bloom filters which are robust towards correlation.

The fact that performance degrades for SuRF and SNARF for correlated queries in these ways is not surprising based on the lower bound result in [22]. The lower bound shows that a range filter that supports range queries up to size R and guarantees a false positive rate of FPR will take at least $\log_2(R) + \log_2(1/\text{FPR})$ bits per key. Hence, for a fixed memory budget, a range filter data structure cannot handle large ranges and a low false positive rate simultaneously without making further assumptions about the data set or workload.

Due to this FPR degradation in SNARF with correlation, Rosetta turns out to be the better filter for workloads with highly correlated and very short range queries. We demonstrate these behaviors empirically in Sec.6.1.1. In big data systems like RocksDB, data is stored in multiple blocks (called SST's in RocksDB) with each block having its own filter. Even if a query is correlated to a key in a certain block, SNARF is still useful for the rest of the blocks (as shown in Sec.6.2).

[6]Note this is the worst case space used. Golomb coding generally uses less space than $2n + n\log_2(K)$ bits.

[7]In practice, we do even better than $(2.0 + \log_2(K))$ bits per key

4.2 Handling Updates

A variety of systems like LSM-based key-value stores use immutable files and thus do not need filters that support updates. On the other hand, OLTP systems which are not based on log structured storage schemes would benefit from an updatable range filter. SNARF is able to naturally support updates owing to its design. To support updates, we keep the mapping function static and only modify the bit array. Because we divide SNARF into small blocks for query efficiency, incremental updates only affect the corresponding block without affecting other blocks.

Update Procedure: To perform an insertion/deletion of a key, we use the mapping function to get the bit location of that key. The bit location is used to identify the corresponding block. We simply add/remove the bit location from the block depending on whether it was an insert/delete. In our basic implementation, we allocate a new block and copy all the bit locations from the old block to the new one after adding/removing the bit location corresponding to the update. Updates to a block can be made faster by using the standard technique of over-allocating memory for that block.

Particularly with deletes, removing a bit location might lead to inconsistency as multiple keys might be mapping to the same bit location. A simple workaround for this is to store duplicates of the bit location. If d keys map to the same bit location, we store precisely $d - 1$ duplicates.[8] We show some experiments with our basic implementation of updatable SNARF in Sec.6.1.8.

Effect of Updates on SNARF's FPR: We modify the bit array but the mapping function($CDF(x) \times nK$) remains static during updates. Updates can lead n and $CDF(x)$ of the mapping function to diverge from the ideal values. We refer to updates that do not change the distribution of the data as in-distribution updates whereas the ones who do as out-of distribution updates.

In-distribution updates do not change the data distribution but may affect the number of keys. Let the final number of keys after updates be n'. After the updates, the ideal mapping function would have been $CDF(x) \times n'K$ to achieve an FPR of $1/K$, whereas we use $CDF(x) \times nK$. The FPR for SNARF thus ends up being $n'/(nK)$ instead of $1/K$. If the in-distribution updates are dominated by inserts, then the FPR becomes worse, and similarly with deletions it gets better.

Out-of-distribution updates may change the data distribution and the number of keys. For out-of distribution updates, predicting the FPR is more complex and it also depends on distribution of queries. We expect the combined effect of change in n and $CDF(x)$ to worsen the FPR of SNARF more than in-distribution updates.

The above discussion also applies to the case of append-only databases. In this setting, when a series of updates significantly reduces the FPR sufficiently, the model should be re-trained and a complete rebuild of the structure would be necessary.

5 RELATED WORK

Filter Data Structures: There is a long history of using compact filters to represent sets that are deemed too expensive to store and query explicitly, for reasons including memory limitations, speed, hardware amenability, and others. Indeed, there are now many variants of the canonical Bloom filter [3] that use various hashing

[8]Duplication adds small space and query latency overhead for small values of K and the impact is not significant for larger K's.

schemes to encode the key set (e.g., Cuckoo filters, Quotient filters, Xor filters, Ribbon filters [2, 13, 16, 23]). These filtering schemes are limited to testing a single key at a time. In some ways, our technique resembles compressed Bloom filters [37] and Golomb coded sets [41]. However, these structures do not handle range queries nor do they take advantage of the data distribution.

We note that theoretical results from [22] show that in the worst case, a data structure that can answer a range query of size up to R with a false positive rate of FPR needs to store $\Omega(\log_2(R) + \log_2(1/\text{FPR}))$ bits per key. Their lower bound suggests looking for structures that may not have worst case guarantees, which can obtain better performance in practical scenarios by focusing on the data and query distributions.

Learning Enhanced Data Structures and Algorithms: We utilize the incorporation of learned models into traditional structures and algorithms. This technique has been applied for indexing [11, 14, 17, 19, 25, 28] and sorting [30, 31]. However, while both like SNARF leverage a model of the eCDF, those structures cannot be used as range filters unless they store all keys, which would not make them space efficient (e.g., one should consider how a B-Tree could be used as a space efficient Range or Bloom filter, which is equally hard/impossible). Learning-enhanced approaches also have been proposed for Bloom filters design [35, 38, 45] but again they are not designed for range queries. Moreover, existing ml-enhanced bloom filters are actually based on classification models, not empirical CDFs.

LSM based Key Value Stores: An important application of filter structures are key-value store data systems [26] based on log-structured merge trees (LSM) [39]. Numerous workloads served by key-value stores (social media, networking, security) include heavy portions of both point and range queries. LSM-based key-value systems store data in multiple immutable files on a disk. Retrieving a particular item or set of items in a particular range leads to multiple expensive I/O's to look up the items in these immutable files. In many settings, the item may not be present in the files, leading to unnecessary I/O's that degrade total query response time. Modern LSM-based key-value systems have extended the basic LSM structure with in-memory filters to address this problem: if a query has no corresponding item, the filter most likely returns false and saves expensive I/O.

Adaptive Range Filter: The Adaptive Range Filter (ARF) [1] uses a binary trie to encode integer key spaces. ARF only stores a number of prefixes of the key set and range queries are then processed by searching the trie for any prefixes of the given range. If a leaf node results in a false-positive, then it is extended until it would no longer do so and, if needed, an old branch is pruned to maintain memory constraints. ARF is not a space efficient data structure for many workloads and in some cases 1300× bigger than SuRF while having a worst FPR (see [47]). Hence, we do not consider ARF further here.

SuRF: The Succinct Range Filter (SuRF) [47] utilizes a compact trie-like data structure which can filter arbitrary range queries. The trie is culled at certain prefix lengths. The basic version of SuRF stores minimum-length prefixes such that all keys can be uniquely represented and identified. Other SuRF variants store additional information such as hash bits of the keys (SuRF-Hash) or extra bits of the key suffixes (SuRF-Real). A weakness of SuRF is that for point queries, SuRF can provide up to 100x worse FPR compared to Bloom filter variants such as Cuckoo filters under the same space budget.

Rosetta: Rosetta utilizes a different approach that performs better for point queries, correlated workloads, and very short ranges. Rosetta essentially uses a Bloom filter for range queries along with the help of a hierarchy of prefix Bloom filters that form an implicit segment tree. Empirically, this design helps Rosetta achieve little to no degradation for point queries compared to Bloom filters. On the other hand, the FPR for Rosetta, while good for small ranges, becomes worse with increasing range query size. For large range queries, Rosetta provides almost no filtering.

LSM Range Queries: ElasticBF [33] proposes a method to adapt Bloom filters in LSMs to query workload. The idea is to use larger filters for hot regions which can be used with SNARF or any other range filter as well. BloomRF [42] is another proposed filter which uses the idea of implicit segment tree with hierarchy of filters similar to Rosetta. It also suffers from FPR degradation with range size like Rosetta. Orthogonal to our approach, REMIX [48] focuses on making range queries faster by creating an alternative path on top of an LSM tree that maintains range indexing info.

Compression Schemes: SNARF needs to compress a sparse bitmap of size nK with n one bits. Assuming a uniform random spread of the one bits, the asymptotic information theoretic lower bound for lossless compression of such a bit array would be $\log_2(K)$ bits per key ($\log_2(K) - O((\log nK)/n)$ bits per key to be precise). Golomb Coding and Elias Fano Coding are near optimal coding schemes as they use at most 2 bits per key over this lower bound ($2n+n\log_2(K)$). Other compression techniques such as WAH[12], CONCISE[8], and Roaring[5] are less space efficient for our particular task, though they can be somewhat faster, so if speed was a concern they could be substituted for our compression approach.

6 EXPERIMENTAL EVALUATION

We now demonstrate that SNARF can bring more than one order of magnitude improvement when compared to state-of-the-art filters. We evaluate SNARF both as a standalone filter as well as part of RocksDB. [9]

6.1 Standalone Analysis

Our experiments comparing SNARF against other baselines aim to support the following key claims:

- SNARF offers a better FPR-space tradeoff curve than other baselines on various synthetic and real world datasets/workloads.
- The FPR provided by SNARF is robust to increasing query range sizes as well as skew in query workload (certain part of data queried more often).
- SNARF performance drops with correlation (as discussed in Sec. 4.1) resulting in Rosetta being better for very short and highly correlated range queries.
- SNARF has a reasonable construction time and its query response time can be tuned as needed. SNARF with Elias Fano encoding has a faster query response time than with Golomb Coding at a slightly higher space cost.
- SNARF supports updates at reasonable throughput.

[9]For our experimental design, we follow the evaluation setup as done in SuRF and Rosetta as much as possible.

We now provide experiment details.

Baselines:We evaluate SNARF against three other baselines:

SuRF: We use the SuRF implementation from [7] with real suffixes as they provided the best performance. [10]

Rosetta: We use the original Rosetta implementation [34].

Cuckoo Filter: For our point queries, we compare against the Cuckoo Filter implementation from [6] in the semi-sorted setting as it achieved the best FPR-space tradeoff.

Datasets: For our experiments, we build a filter on 100 million keys chosen from the following datasets.[11] We use two synthetic datasets and three real world datasets from [36]:

Uniform Random: Keys are generated uniformly at random in the range $[0, 2^{50}]$.

Normal: Keys are generated from normal distribution ($N(\mu = 100, \sigma = 20)$) and are linearly scaled to range $[0, 2^{50}]$.

wiki: Keys represent the time an edit was made on Wikipedia.

osm: cell IDs from Open Street Map representing a location.

fb: unique Facebook user IDs [46].

Workload: We use 100 million queries for our experiments. The queries are of the type [*left*,*left*+*range_size*]. If *range_size*=0, then the query is a point query. We first generate the left endpoint(*left*) of the range query from a certain distribution and then the right endpoint of the query is calculated by adding the left endpoint and the *range_size.* The range query workloads use a range size of 256 while the mixed-query workloads use range sizes of 0, 16, 64 and 256 in equal proportion. We generate the left endpoint(*left*) of the queries in following manner:

Uniform Random: left endpoint chosen uniformly at random in the range $[0, 2^{50}]$.

Exponential: We use an exponential distribution($p(x) = \lambda e^{-\lambda x}; \lambda = 10$) which results in certain part of the data being queried more often. We then scale them to range $[0, 2^{50}]$.

Correlated: This distribution generates queries which are close to the keys. A key is chosen uniformly at random from the dataset and then left endpoint is chosen uniformly at random from [key, key+$2^{30*(1-corr\_degree)}$]. Higher *corr_degree* implies increased proximity between keys and queries, so that $corr\_degree = 1$ generates extremely correlated queries (left end point being $key + 1$) whereas $corr\_degree = 0$ generates queries independent of the key value.

Sampled Data: This is used to generate range queries for real world datasets (as previously done in SuRF). We first divide the dataset into two equally sized parts by choosing keys uniformly at random. A filter is built on one half of the dataset and the other half is used as the left endpoints for queries in the respective workload.
SNARF parameters: The CDF model uses $(N/1000)$ linear models unless stated otherwise. By default, we choose $\beta = 100$ and thus divide the bit array into $(N/100)$ equally sized segments. We use Golomb coding for SNARF unless specified.

[10]Note, SuRF has a limited range of operation as the implementation starts with minimum of 10 bits per key (0 bits as the suffix length).

[11]We evaluate on integer keys but would also work for floats. Floats are numerical keys, the current CDF model for SNARF works for them. We expect minimal change in the performance of SNARF for floating point values.

6.1.1 *FPR vs Space Tradeoff for Synthetic Dataset/Workloads:*

In Fig.5(A), each subfigure corresponds to a particular key and query distribution along with a particular query workload. Each subfigure shows the space used by the baselines in bits per key and the FPR achieved by them on the corresponding query workload. For point queries, SNARF achieves performance similar to Cuckoo filters for all cases. For range queries, SNARF consistently has a better Pareto curve than all other baselines. When using 16 bits per key, SNARF and SuRF provide false positive rates of 6.2×10^{-5} and 1.1×10^{-3}, respectively. Rosetta is competitive for point queries but its performance degrades as query range size increases.

Even with exponentially distributed data, SNARF maintains its performance as the CDF model can capture this skew in data distribution. As discussed in Sec.3, mapping the keys evenly across the bit array results in a robust false positive rate and consistent performance across different skewed query distributions.

6.1.2 *FPR vs Space Used Tradeoff for Real Dataset/Workloads:*

In Fig.5(B), each subfigure corresponds to a particular dataset along with a particular query workload. Each dataset is divided into two equal parts. One part forms the set of keys and the other half forms the left endpoint of the query. The right endpoint is decided by the range query size.

SNARF has a better Pareto curve than other baselines for all cases. SNARF is able to perform particularly well on real-world datasets due to certain patterns present in them. A common pattern we observed in our real-world datasets is that they have large empty contiguous ranges; for example, S={10,78,95,10045,10052,10089,30011,.....}, where the sorted keys suddenly jump by large amounts. While we do not have a clear global reason for such behavior, it is natural for settings such as when the set is a collection of numerical IDs; different ID subranges may be assigned by different entities. Both SNARF and SuRF effectively model large empty ranges in a way that is both succinct and avoids false positives.

In some cases, while keys may be from a large domain, they may be concentrated in a small range. For example, the keys may lie in the domain $[0, 2^{32})$ but all appear in the small range $[2^{10}, 2^{12}]$. The modelling step of SNARF automatically takes advantage of this type of pattern to benefit performance[12]. For most cases, SNARF is able to achieve a low FPR (below 10^{-4}) using less than 10 bits per key. For the *osm* dataset, SuRF also achieves a FPR below 10^{-4} but still uses more memory ($\approx 15-16$ bits per key). Similar to our previous experiments on synthetic data, Rosetta is competitive for point queries but its performance degrades as query range size increases.

6.1.3 *Correlated Workload:*

As discussed in Sec.4.1, the correlated workloads are when the query endpoint is close to a key, which is more likely to lead to a false positive in SNARF and SuRF. In Fig.6, we show the FPR vs key-query correlation degree tradeoff for various baselines for a fixed memory budget of 15 bits per key. Higher the key-query correlation degree closer the queries are to the keys. As expected, FPR of both SNARF and SuRF degrades with increasing correlation. Both SNARF and SuRF provide virtually no filtering for uniform dataset when workload is highly correlated. On the other

[12]This is similar to the case when $z < nK$ and all values are mapped to distinct bit positions leading to no false positives.

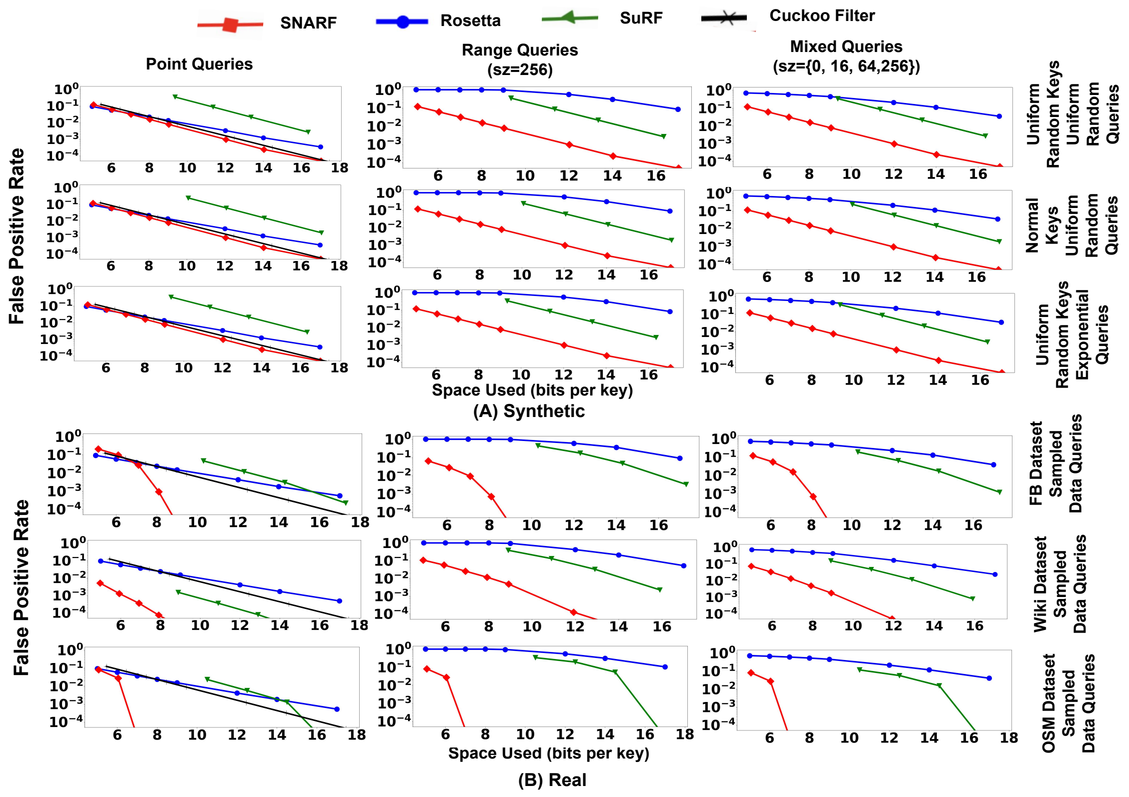


Figure 5: FPR vs Space Used(in bits per key) by various filters. Each subfigure shows the space-FPR tradeoff for a (A) synthetic (B) real dataset and workload distribution and for a particular range query type (point, range query of size 256 and mixed query workload of size 0,16,64,256).

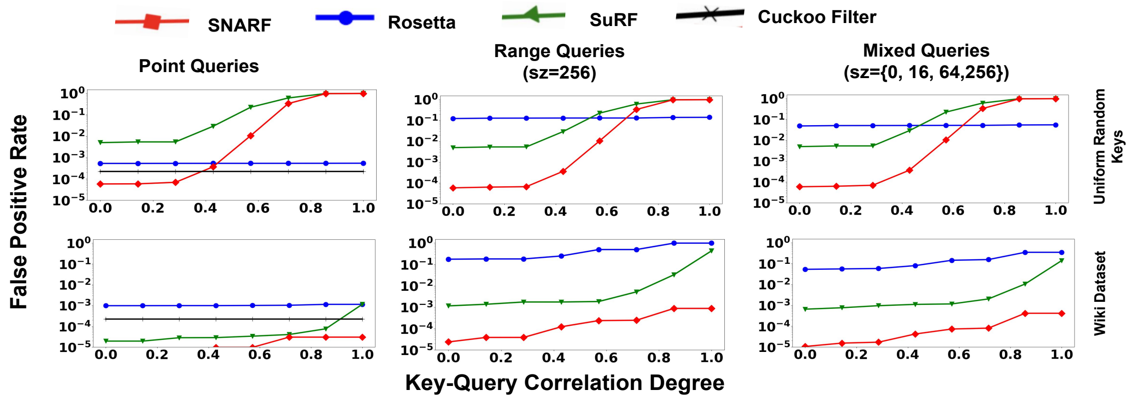


Figure 6: FPR vs Key-Query Correlation Degree of the queries on uniformly random/wiki keys. With increasing key-query correlation, SNARF and SuRF become worse and Rosetta turns out to be the better filter for very short and highly correlated range queries.

hand, Rosetta is unaffected by this correlation and performs the best for very short range queries and highly correlated workloads.

6.1.4 ***FPR vs Range Size:*** In Fig.7(A), we vary the range query size from 1 to 10^6 and report the FPR of various range filters under a memory budget of 15 bits per key. We use uniformly randomly distributed keys and workloads for this experiment. As discussed in Sec.3, the FPR of SNARF stays constant with the range query size. SuRF also maintains its FPR with increasing range query size but has a 17x worse FPR than SNARF. Rosetta becomes worse with increasing range size and provides almost no filtering for range sizes greater than 1000.

6.1.5 ***Filter Query Latency vs Space Used:*** In Fig.8, we show the query latency of various filters with increasing filter sizes for uniform random and FB datasets for mixed range queries. We skip other datasets/workloads as we observed similar trends for them. For this experiment, we fix the size of the dataset to 100 million

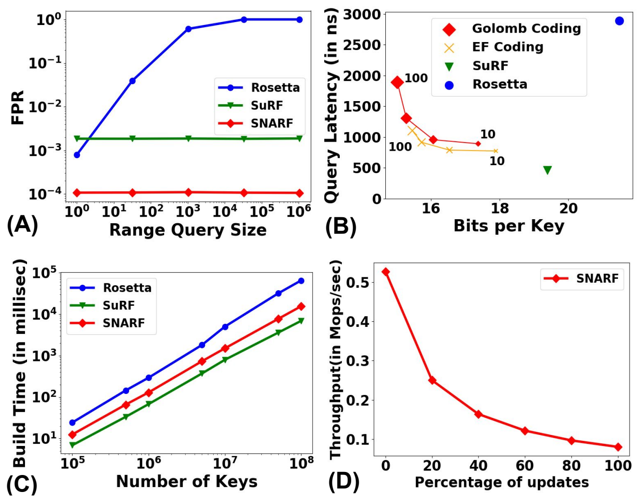


Figure 7: (A) FPR with increasing range query size for fixed space budget. (B) Filter Latency (in ns) against space used (bits per key) (C) Build Time(in millisecs) with increasing number of keys. (D) Filter Throughput as we vary the percentage of updates in the workload.

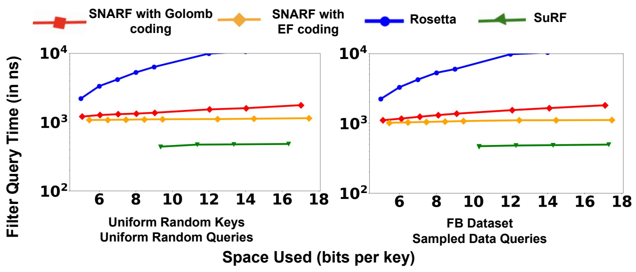


Figure 8: Filter Query Time (in ns) vs Space used by various filters across various datasets and workloads for mixed queries.

keys and vary the filter parameters that control its size. In both of the subfigures, both variants of SNARF are slower than SuRF but faster than Rosetta. SNARF with Elias Fano encoding is consistently faster than SNARF with Golomb coding. This is because Golomb coding (a form of delta coding) requires decoding of the first key and the following delta values to retrieve a key, which is not the case for Elias Fano coding. For SNARF, filter query time increases slightly with increasing filter size. This is because as the filter size increases, the model size remains constant but the encoded bit array size increases, so SNARF then has to parse more data to decode the bit array. The filter query latency increases drastically for Rosetta as its filter size increases, as larger internal Bloom filters mean Rosetta has to perform a greater number of random accesses.

6.1.6 ***Effect of Bit Array Division on Space and Query Latency:*** As discussed in Sec.2.3.2, for SNARF we can improve the query latency by reducing the segment size. Recall we use small segments of size βK in the bit array, and using smaller β can improve latency at the cost of extra space overhead. The overhead arises because when we have a larger number of segments in the bit array there is more associated metadata. Fig.7(B) shows the query latency and the space used by the various baselines to achieve a FPR of 2^{-13} on uniform random keys and uniform randomly generated mixed sized queries. We show multiple configurations for SNARF with β values 10, 20, 50, and 100 (increasing marker size representing larger β values). The results show that with decreasing β we get better query latency. Elias Fano coding is faster than Golomb coding for the same number of segments. By varying β, Golomb coding and EF coding with SNARF are able to achieve a query latency of 890 ns and 746 ns, respectively. SuRF is the fastest baseline with latency of 480ns, but uses around 19.4 bits per key.

6.1.7 ***Build Time:*** In Fig.7(C), we vary the number of the keys from 10^5 to 10^8 and report the build times of various range filters. We use a uniformly random distribution for the keys. The build times of all the filters grow linearly with the number of keys. The build time for learned range filters is around 5x faster than Rosetta and around 2x slower than SuRF. Depending on the application, filter construction might play a more or less important role. For example, for LSM trees, filter construction only plays a minor role as part of the merge phase as shown in Sec.6.2.3.

6.1.8 ***Updates:*** In Fig.7(D), we vary the percentage of updates(50% insertions and 50% deletions) in the query workload (the rest of the workload is range queries) and report the throughput. SuRF and Rosetta do not support both inserts and deletes, so we only analyze SNARF here. We use the SNARF variant with duplication in order to support deletes. We use a uniform random distribution for the keys. The workload contains 1 million operations overall and is also uniformly randomly distributed. Since, the updates do not change the distribution of the data, the FPR stays constant. On average an update takes around 12k ns whereas a range query takes around 1898 ns. The throughput of the filter decreases with increase in proportion of updates as updates are slower than range queries.

6.2 RocksDB Experiments

Our experiments on RocksDB integrated with SNARF aim to support the following key claims:

- RocksDB with SNARF offers better read performance than other baselines on various synthetic and real world datasets and workloads.
- SNARF's as well as other filters impact reduces as the proportion of empty range queries in the workload decreases. This leads to SNARF's performance improvement over other filters to reduce as well.
- In RocksDB with SNARF, read performance drops with correlation (as discussed in Sec. 4.1) resulting in Rosetta being better for very short (range size less than 16) and highly correlated range queries.
- SNARF adds little overhead to RocksDB
- SNARF improves end-to-end performance of RocksDB for a typical read-write workload.

Integration with RocksDB: We use a RocksDB integration and workload generation setup identical to that of Rosetta [34]. We utilized an API of filter functionalities such as populating, querying, serializing, and deserializing the filter to integrate SNARF. RocksDB stores its data in multiple immutable tables called SST (Sorted String Tables). A SNARF instance is created for each SST file. We store the filter on disk as a character array and the process of converting the

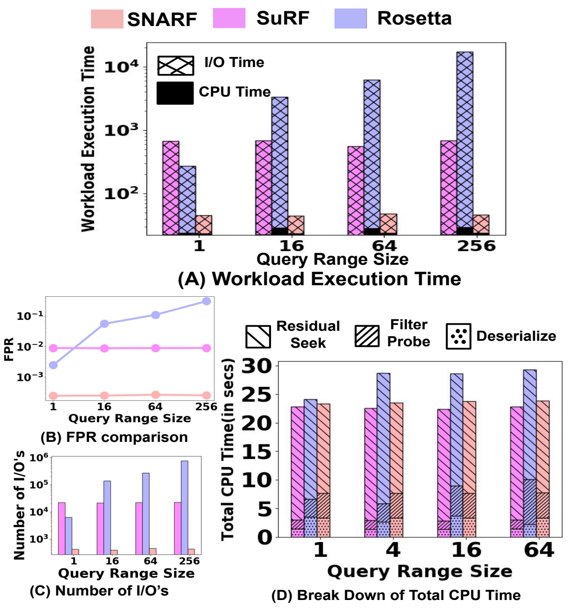


Figure 9: SNARF outperforms other baselines when fully integrated in RocksDB.

filter to char array is called serialization. In order to use the filter we need to read it to memory from the disk and deserialize it[13]. We enable the block cache and allow the caching of filters.[14]

Implementation Overview of a Range Query: For a range query $[p, q]$, RocksDB probes filter instances of all levels for existence of keys within this range. If all filter instances return negative, an empty result is returned. If one or more filters return positive, RocksDB seeks the lower end (p) incurring an I/O. When RocksDB get a valid pointer, it reads data until q is reached and incurs as much I/O's needed to reach q.

Setup and Workloads: We use 14 bits per key for all the filter baselines(as previously done in SuRF)[15]. We first populate RocksDB with 50 million 64-bit keys from a distribution and 512 byte values. Each experiment has a description of the workload. After population, we run the workload on this populated RocksDB instance. Total execution time of this workload is usually the metric of interest.

We use uniform random distribution for keys/workload generation by default and we have 100k queries in a workload as default. We used the same distributions of dataset and workload mentioned in Sec.6.1. The workloads are primarily read only to highlight the impact of filters, but we also have a few experiments with a mixture of reads and writes. Each workload is run with read queries of various range sizes (1, 16, 64, 256).

6.2.1 ***SNARF improves RocksDB Performance:*** For this experiment, we generate YCSB key-value workloads that are variations of Workload E, a majority range scan workload modeling a web application use case [10]. The quality of the filter is best judged with empty range queries as filters enhance performance by identifying empty queries for which an unnecessary seek can be avoided. Thus, we compose our workload with 100, 000 empty range queries.

Fig.9(A) shows the workload execution time of various baselines. The workload execution time consists of two parts, time spent by the CPU and time spent on I/O. We observe that I/O time dominates the CPU time. SNARF's workload execution time is consistently one order of magnitude less than the other baselines. SNARF has a better FPR than SuRF and Rosetta leading to fewer I/O's and hence lower workload execution time. As shown in Fig.9(B), SuRF has a FPR 40x worse than that of SNARF across all range sizes. Rosetta's FPR becomes worse with increasing range size. Worse FPR leads to more block I/O's as shown in Fig.9(C). In summary, Rosetta and SuRF have significantly high I/O time due to their worse FPR.

SNARF adds little CPU overhead In the previous experiment, we further break down the total CPU time for various baselines in Fig.9(D). The CPU time is further divided into deserialization time, filter probing time, and residual seek time. The residual seek time is the time taken for routine jobs performed by RocksDB iterators – looking for checksum mismatch and I/O errors; going forward and backward over the data, filters and fence pointers; and creating and managing database snapshots for each query. The filter probe time is time taken to probe the filters and deserialization time is the time taken for filter deserialization. The filter probe time accounts for at most 20% of the total CPU time for even the slowest filter (Rosetta). Residual seek time accounts for the dominant portion of the CPU time. Thus, a CPU intensive filter does not affect the performance of RocksDB much.

SNARF improves RocksDB Performance on real world datasets For this experiment, we populate RocksDB with 50 million keys from real world datasets and use sampled-data workload consisting of 100k empty range queries. Fig.10(A) shows the workload execution latency of this workload in RocksDB. SNARF exhibits a lower workload latency than other baselines for all three datasets. Same as previous experiment, this is due to the better FPR that SNARF delivers compared to other filters.

As range size increases, SNARF improves RocksDB Performance for correlated workloads As discussed in Sec.4.1, SNARF and SuRF become worse with increased correlation between queries and keys. For this experiment, we use a correlated workload consisting of 100k empty range queries. In Fig.10(B), we vary the key-query correlation degree of the queries and measure the workload execution latency. The execution time of SNARF and SuRF increases with correlation but not beyond a certain level. This is because even if a query is highly correlated to a key in a particular SST file, SNARF and SuRF are still useful for the rest of the SST files. Rosetta is the better filter for range size equal to one and a highly correlated workload otherwise SNARF is the better filter.

6.2.2 ***SNARF performance for mixed workload (empty and non-empty range queries):*** Here, we measure the workload execution latency on a mixed read-only workload of empty and non-empty range queries by varying the percentage of empty range queries from 10 to 100. As shown in Fig.10(C), the workload execution time of all filters decreases with an increase in the proportion of empty range queries. This is because filters are more effective

[13]To reduce the deserialization overhead we maintain a dictionary that has the deserialized bits for each filter instance and its corresponding SST similar to [34]

[14]*cache_index_and_filter_blocks*=true. We also ensure that the fence pointers and filter blocks have a higher priority than data blocks when block cache is used *cache_index_and_filter_blocks_with_high_priority*=true, *pin_l0_filter_and_index_blocks_in_cache*=true.

[15]14 bits per key allows reasonable performance with fpr below 10% for all filters

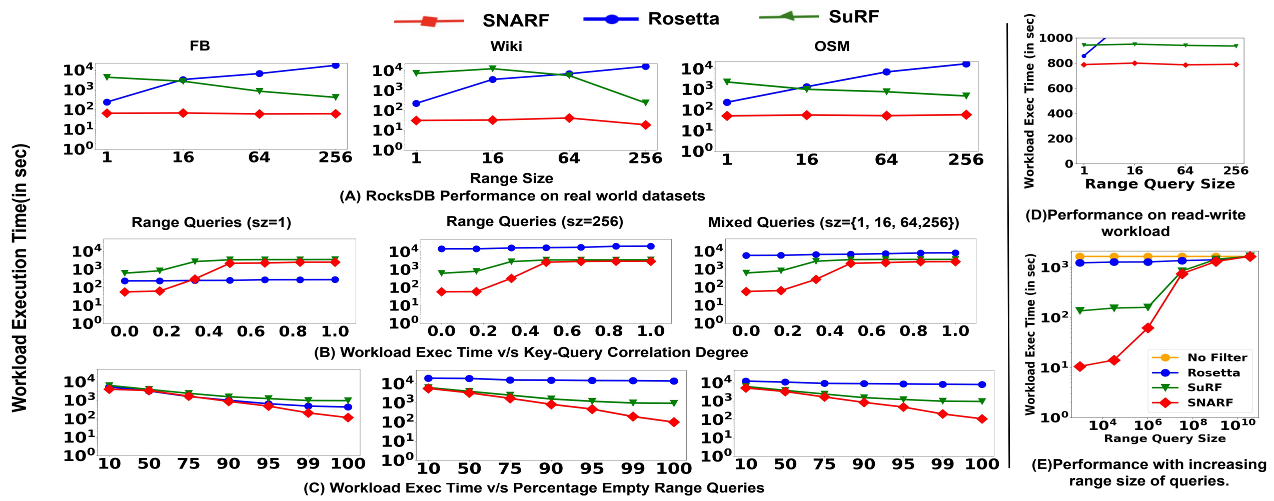


Figure 10: Workload Execution time in RocksDB for (A) real world datasets/workloads (B) correlated workloads and (C) varying percentage of empty queries (D) read-write workload (E) varying rnage query sizes

on empty queries than non empty queries. Notice, even with majority non-empty workload filters are still useful. This is because non-empty range queries will return a true positive for one SST but other SST's might still return a false positive leading to additional unnecessary scans. The decrease in execution time is faster for SNARF than other baselines because SNARF has better FPR than others and thus, is more effective in reducing unnecessary I/O's.

6.2.3 ***SNARF performance for read-write mixed workload:*** In order to simulate real working of RocksDB, we used a majority write workload (only 1 percent reads) with 10 million operations similar to YCSB workload A(majority updates). Read and writes are performed in an interleaved manner. We first start with a RocksDB instance that already has 50 million uniform randomly distributed keys . Reads and writes are generated using the uniform random distribution. Each write operation is a point write which inserts a unique key into the RocksDB instance with a corresponding randomly generated value. All the read queries are empty range queries and we evaluate 4 different workloads for read queries with 4 different range sizes: 1, 16, 64 and 256.

Writing keys to RocksDB leads to compaction and creation of new SST files. Creation of new SST files involves constructing the filter and thus filter construction time gets accounted for in the overall execution time. While performing reads, the query response time of the filter gets accounted for in the execution time. Thus, this experiment evaluates the end-to-end filter performance as it accounts for reduced I/Os due to filtering, filter query response time and filter construction time. In Fig.10(D), we show the workload execution time of the workload for various range sizes. Owing to its lower FPR, SNARF has a lower workload execution latency than SuRF and Rosetta. SNARF's slightly slower filter query time and construction time compared to SuRF is offset by gains produced in lower I/Os.

6.2.4 ***SNARF impact with increasing range query size:*** In Figure 10(E), we show the workload execution time as we increase query range size for uniformly randomly generated keys and workloads. The impact of filters decreases with increasing range size. For range sizes around $\approx 10^3 - 10^4$ most queries are empty; accordingly, filters have a large impact and here SNARF outperforms other filters by an order of magnitude. For range sizes around $\approx 10^7 - 10^8$ most queries are non-empty, touching a few SSTs, and filter have less impact. For range sizes around $\approx 10^9 - 10^{10}$ most queries touch most SSTs and filters have negligible impact.

7 CONCLUSION

We introduce Sparse Numerical Array-Based Range Filters (SNARF), a learning-enhanced range filter supporting both point and range queries for numerical data. We have shown that SNARF appears highly beneficial for point and range queries, both via an analysis and empirically across various synthetic and real world datasets. For future goals, we would like to extend this approach to other types of data, most notably strings. Finding better learning models or proving via a theoretical framework that linear spline model is a near-optimal model also remain open questions. Finally, making SNARF workload dependent is an interesting direction for future work.

ACKNOWLEDGMENTS

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL), and NSF IIS 1900933, CCF-2101140, CNS-2107078, and DMS-2023528. This research was also sponsored by the US Air Force Research Laboratory and the US Air Force Artificial Intelligence Accelerator under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Air Force or the US Government. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. This work was partially supported by a gift to the Center for Research on Computation and Society at Harvard University and the USA Department of Energy Project DE-SC0020200.

REFERENCES

[1] Karolina Alexiou, Donald Kossmann, and Paul Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. In *Proceedings of the VLDB Endowment, Vol. 6, No. 14*.

[2] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *Proc. VLDB Endow.* 5, 11 (2012), 1627–1637.

[3] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. (1970), 422–426.

[4] Andrei Z. Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Math.* 1, 4 (2003), 485–509.

[5] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with roaring bitmaps. *Software: practice and experience* 46, 5 (2016), 709–719.

[6] Efficient Lab CMU. 2020. https://github.com/efficient/cuckoofilter.

[7] Efficient Lab CMU. 2020. https://github.com/efficient/SuRF.

[8] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'n'composable integer set. *Inform. Process. Lett.* (2010), 644–650.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. 143–154.

[11] Andrew Crotty. 2021. Hist-Tree: Those Who Ignore It Are Doomed to Learn. In *CIDR*.

[12] François Deliège and Torben Bach Pedersen. 2010. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps *(EDBT '10)*. 228–239.

[13] Peter C Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *arXiv preprint arXiv:2103.02515* (2021).

[14] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and et al. 2020. ALEX: An Updatable Adaptive Learned Index. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).

[15] Facebook. 2015. http://myrocks.io/.

[16] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proc. CoNEXT*.

[17] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index. *Proceedings of the VLDB Endowment* 13 (2020).

[18] Frederick N Fritsch and Ralph E Carlson. 1980. Monotone piecewise cubic interpolation. *SIAM J. Numer. Anal.* 17, 2 (1980), 238–246.

[19] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure *(SIGMOD '19)*. 1189–1206.

[20] R. Gallager and D. van Voorhis. 1975. Optimal source codes for geometrically distributed integer alphabets (Corresp.). *IEEE Transactions on Information Theory* 21, 2 (1975), 228–230.

[21] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. 2005. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*. 27–38.

[22] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. 2014. Approximate Range Emptiness in Constant Time and Optimal Space. arXiv:1407.2907 [cs.DS]

[23] Thomas Mueller Graf and Daniel Lemire. 2020. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA)* 25 (2020), 1–16.

[24] Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. 2009. More Haste, Less Waste: Lowering the Redundancy in Fully Indexable Dictionaries. arXiv:0902.2648 [cs.DS]

[25] Ali Hadian and Thomas Heinis. 2021. Shift-Table: A Low-latency Learned Index for Range Queries using Model Correction. arXiv:2101.10457 [cs.DB]

[26] Stratos Idreos and Mark Callaghan. 2020. Key-Value Storage Engines. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*.

[27] Tamer Kahveci and Ambuj Singh. 2001. Variable length queries for time series data. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 273–282.

[28] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. arXiv:2004.14541 [cs.DB]

[29] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. arXiv:1712.01208 [cs.DB]

[30] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 1001–1016.

[31] Ani Kristo, Kapil Vaidya, and Tim Kraska. 2021. Defeating duplicates: A re-design of the LearnedSort algorithm. arXiv:2107.03290 [cs.DS]

[32] Cockroach Labs. 2015. https://github.com/cockroachdb/cockroach.

[33] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. Elasticbf: elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*. 739–752.

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

[35] Stephen Macke, Alex Beutel, Tim Kraska, Maheswaran Sathiamoorthy, Derek Zhiyuan Cheng, and EH Chi. 2018. Lifting the curse of multidimensional data with learned existence indexes. In *Workshop on ML for Systems at NeurIPS*.

[36] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. arXiv:2006.12804 [cs.DB]

[37] M. Mitzenmacher. 2002. Compressed Bloom filters. *IEEE/ACM Transactions on Networking* 10, 5 (2002), 604–612.

[38] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/0f49c89d1e7298bb9930789c8ed59d48-Paper.pdf

[39] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[40] Mihai Patrascu. 2008. Succincter. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. 305–313.

[41] Felix Putze, Peter Sanders, and Johannes Singler. 2007. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*. 108–121.

[42] Christian Riegger, Arthur Bernhardt, Bernhard Moessner, and Ilia Petrov. 2020. bloomRF: On Performing Range-Queries with Bloom-Filters based on Piecewise-Monotone Hash Functions and Dyadic Trace-Trees. arXiv:2012.15596 [cs.DB]

[43] Russell Sears, Mark Callaghan, and Eric Brewer. 2008. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment* 1, 1 (2008), 526–537.

[44] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2011. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials* 14, 1 (2011), 131–155.

[45] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. 2020. Partitioned Learned Bloom Filter. *CoRR* abs/2006.03176 (2020). arXiv:2006.03176 https://arxiv.org/abs/2006.03176

[46] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 36–53.

[47] 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 *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 323–336. https://doi.org/10.1145/3183713.3196931

[48] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *FAST*.

[49] Dong Zhou, David G Andersen, and Michael Kaminsky. 2013. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *International Symposium on Experimental Algorithms*. Springer, 151–163.