Stable Learned Bloom Filters for Data Streams

Qiyu Liu[§], Libin Zheng[§,*], Yanyan Shen[†], and Lei Chen[§]
[§]The Hong Kong University of Science and Technology [†]Shanghai Jiao Tong University
{qliuau, lzhengab, leichen}@cse.ust.hk, yanyanshen14@gmail.com

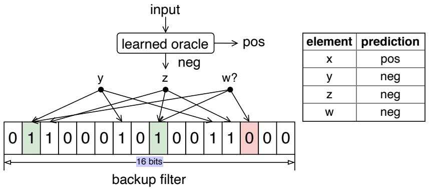
ABSTRACT

Bloom filter and its variants are elegant space-efficient probabilistic data structures for approximate set membership queries. It has been recently shown that the space cost of Bloom filters can be significantly reduced via a combination with pre-trained machine learning models, named Learned Bloom filters (LBF). LBF eases the space requirement of a Bloom filter by undertaking part of the queries using a classifier. However, current LBF structures generally target a static member set. Their performances would inevitably decay when there is a member update on the set, while this update requirement is not uncommon for real-world data streaming applications such as duplicate item detection, malicious URL checking, and web caching. To adapt LBF to data streams, we propose the Stable Learned Bloom Filters (SLBF) which addresses the performance decay issue on intensive insertion workloads by combining classifier with updatable backup filters. Specifically, we propose two SLBF structures, Single SLBF (s-SLBF) and Grouping SLBF (g-SLBF). The theoretical analysis on these two structures shows that the expected false positive rate (FPR) of SLBF is asymptotically a constant over the insertion of new members. Extensive experiments on real-world datasets show that SLBF introduces a similar level of false negative rate (FNR) but yields a better FPR/storage trade-off compared with the state-of-the-art (non-learned) Bloom filters optimized on data streams.

PVLDB Reference Format:
Qiyu Liu, Libin Zheng, Yanyan Shen and Lei Chen. Stable Learned Bloom Filters for Data Streams. *PVLDB*, 13(11): 2355-2367, 2020.
DOI: https://doi.org/10.14778/3407790.3407830

1. INTRODUCTION

Bloom filters (BF) [7] are simple, space-efficient probabilistic data structures designed for answering membership queries, that is, testing whether a queried element x is in a given set $\mathcal{S}$. Due to its great importance, optimizations and variants of Bloom filter as well as its applications, especially in database and networking, have attracted much research efforts over the past decades [11, 16, 17, 19, 18, 23, 15, 13]. Although Bloom filters have been well explored and evaluated in both academia and industry, a recent proposal, *"The Case for Learned Index Structures"* [25], suggests that machine learning models like neural networks can be combined with traditional index structures, including B-tree, hash tables and Bloom filters, to further improve the space utilization and query efficiency.



element	prediction
x	pos
y	neg
z	neg
w	neg

Figure 1: Illustration of the learned Bloom filter built on an element set $\mathcal{S} = \{x, y, z\}$ with a query element w.

In their seminal work, Kraska et al. [25] claim that the membership queries can be regarded as an instance of classification problem on a dataset $\{(x_i, y_i = 1) | x_i \in \mathcal{S}\} \cup \{(x_i, y_i = 0) | x \in \mathcal{N}\}$ where $\mathcal{S}$ and $\mathcal{N}$ are the sets of members and non-members. They first use a classifier which classifies elements into members/non-members. Then, to remove possible false negatives, a small backup Bloom filter is built on the set $\mathcal{S}_N = \{x | x \in \mathcal{S}, x \text{ is predicted as non-member}\}$ to distinguish the true members from the predicted non-members. Such a data structure, called Learned Bloom Filter (LBF), is illustrated in Figure 1. For query processing, a non-member decision is made i.f.f. both the classifier and the backup filter determine a queried element as a non-member.

Example 1. As shown in Figure 1, let us consider a toy element set $\mathcal{S} = \{x, y, z\}$ for constructing an LBF. x is correctly predicted by the classifier as a member of $\mathcal{S}$, so it needs no further treatment. In contrast, for y and z, which are wrongly classified, we insert them into a standard Bloom filter [7] with an array of 16 bits and 3 hash functions. When the constructed LBF is queried with an element w, supposing that the classifier determines w as a non-member, w would be further tested over the backup filter, which finally yields a non-member decision, i.e., $w \notin \mathcal{S}$.

Similar to standard BF, LBF has one-sided error (i.e., only false positives). Compared with non-learned filters, the advantage of LBF is that, on a static element set, it requires much smaller storage but retains competitive query efficiency and error rate. The reason is that the space cost of LBF comes from storing both the classifier and the backup

[*] Libin Zheng and Yanyan Shen are corresponding authors.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407830

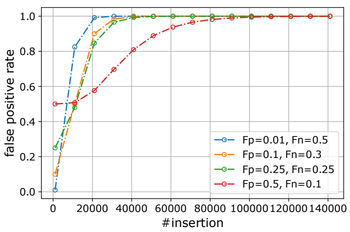


Figure 2: Simulation result of the FPR decay effect for LBF. The backup filter is initialized with #expected elements=1,000, #bits per element=9.85, #hash functions=6, and FPR is calculated by varying the number of insertions and trying 4 different combinations of F_p (FP rate) and F_n (FN rate) of the classifier.

filter. However, $\mathcal{S}_N$, the set used to build the backup filter, is relatively small, and storing a well-trained classifier usually requires much less space [25].

However, similar to the standard BF, LBF is designed for *static* element set $\mathcal{S}$ whose total cardinality is known in advance. As a result, when there are new elements outside $\mathcal{S}$ being inserted to the filter, the false positive rate (FPR) would inevitably grow due to the limited space of backup filter, which is determined upon its construction. Compared with standard BF, such performance decay effect caused by insertions is more severe for LBFs, as the backup filter is usually small, and the insertion capacity can be easily reached.

Example 2. (FPR Decay) Figure 2 illustrates such an effect by reporting the FPR versus the number of insertions according to the theoretical analysis for LBF in [30], under the assumption that newly inserted elements are sampled from the same distribution as $\mathcal{S}$. It shows that FPR tends to reach 100% after 80K new insertions under all the four settings. More specifically, though a smaller F_n (false negative rate of the classifier) can slow down the increase of FPR, it introduces a higher initial FPR. Intuitively, this is because F_p and F_n usually contradict with each other, and F_p is the lower bound of FPR (discussed in Section 2.2).

Though space-efficient, LBF is only applicable for a predefined element set and query-only workloads, which limits its real-world applications and motivates us to design new learned filters usable on insertion-intensive workloads. However, building such an insertion-aware LBF is non-trivial. Existing works addressing this dynamic insertion issue all target standard BF [19, 6, 13, 18]. They cannot be directly applied to the context of LBF due to the existence of an extra classifier. The major challenges lie in four aspects.

- First, ML models are usually non-deterministic. Therefore, we need to devise a new mathematical model for analyzing the performance of learned filters over data streams, just like what is done for the LBF on static sets in [30].
- Second, different from standard BF, since we wish the FPR to be controlled when the filter is applied to an unbounded element stream with limited storage, it would inevitably introduce false negatives. Consequently, we need to carefully quantify the false negative rate and achieve a proper trade-off among FPR, FNR, and storage.
- Third, when handling a static element set, overfitting of the classifier is usually good since it improves both F_p and F_n, which is contrary to the case of streaming data as we hope that the classifier generalizes well to future elements.
- Fourth, parameter setting becomes harder under the context of dynamic insertions. For the original learned Bloom filter on a static element set, the parameters of the backup filter, e.g., number of hash functions, are optimized based on the size of element set, classifier performance and a user-provided FPR threshold, which will change as new elements inserted to the filter.

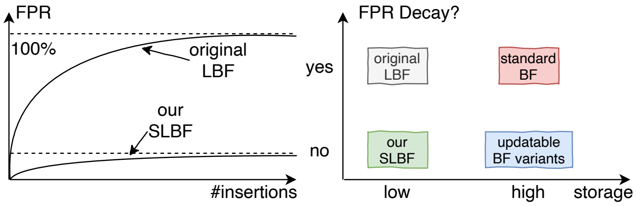


Figure 3: Left: FPR v.s. #insertions under the same storage cost. Right: sketch of performance feature under the same expected FPR upper bound.

To handle the dynamic insertion of new elements, in this paper, we design a new insertion-aware LBF structure called Stable LBF (SLBF). SLBF is expected to have the following features: 1) the performance decay effect is under control, i.e., FPR has a non-trivial upper bound even for a large number of insertions; 2) the total storage cost is limited and less than that of using a standard filter at the same error level; and 3) the membership query is as efficient as a standard filter, i.e., in $O(1)$ time. Figure 3 outlines the major performance merits of our SLBF. When applied to dynamic insertions, SLBF has low storage cost and survives from the performance decay issue. Many real-world applications, like duplication detection [29], IP traffic monitoring [26], and search engine refinement [22], can benefit from using our SLBF. The applications of SLBF as well as a detailed discussion about the proper choice among using BF, LBF or our SLBF on different scenarios are put in Appendix A and Appendix B.

To the best of knowledge, this is the first work that considers optimizing LBF over dynamic insertion workloads. We summarize the major technical contributions as follows.

- We introduce two new learned Bloom filters, Simple Stable Learned Bloom Filter (s-SLBF) and Grouping Stable Learned Bloom Filter (g-SLBF), to achieve the three objectives mentioned above.
- We perform detailed analysis on the performance of our proposed data structures over dynamic insertion workloads, regarding which we explain its parameter setting and classifier selection.
- We conduct extensive experimental studies on the real data, which show that g-SLBF can effectively reduce up to 97% storage cost.

The rest of this paper is organized as follows. Section 2 introduces some preliminaries and formulates the problem of devising an insertion-aware Bloom filter with a performance guarantee. We present our s-SLBF and g-SLBF and analyze their performance over data streams in Section 3. Section 4 discusses the parameter settings for our SLBF. We report the experimental results in Section 5. Finally, we review previous works in Section 6 and conclude in Section 7.

2. PRELIMINARIES

This section overviews the structures and analytical results of standard Bloom filers and learned Bloom filters. Then, we discuss the property of stability for Bloom filters on unbounded data streams. For quick reference, all the notations used hereafter are summarized in Table 1.

Table 1: Notations and descriptions.

Notation	Description
$\mathcal{S}$	the element set of n distinct members
$h_1, \cdots, h_k$	k independent hash functions
$B[1 \cdots m]$	the array of m bits used in BF
$f : x \to [0, 1]$	a trained classifier
τ	the decision threshold of classifier f
$SBF[1 \cdots m]$	the array of m counters used in stable BF
g	the number of total groups
$\tau_1 < \cdots < \tau_{g-1}$	a partition to interval $[0, 1]$
$SBF_1, \cdots, SBF_g$	a collection of g stable BFs
Max_j	the max value of each counter of SBF_j
m_j	the number of counters of SBF_j
K_j	the number of hash functions of SBF_j
P_j	the number of decrements of SBF_j
F_p	the false positive prob. of classifier f
F_n	the false negative prob. of classifier f
$x_1, \cdots, x_N$	a sequence of N elements to be inserted

2.1 Standard Bloom Filter

Given a set $\mathcal{S}$ of n member elements, a standard Bloom filter [7] represents $\mathcal{S}$ using a bit array B of size m and k independent hash functions $h_1, \cdots, h_k$ which uniformly map the elements to the range $1 \sim m$ (inclusive). The bit array is initialized to all 0's in the beginning. Then, for each $x \in \mathcal{S}$, the bits located at $h_0(x), \cdots, h_k(x)$ are set to 1. For a membership testing of element y, a *positive* answer is given if all k bits pointed by $h_0(y), \cdots, h_k(y)$ are 1, otherwise a negative answer is returned. Such a construction and query mechanism ensure there is no false negatives but possibly false positives. After inserting all n elements of $\mathcal{S}$ into the bit array B, the probability that an arbitrary bit of B is still 0 can be calculated as:

$$\Pr\left(B[i] = 0\right) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}. \qquad (1)$$

We denote $p_0 = \Pr\left(B[i] = 0\right)$. For any non-member element $y \notin \mathcal{S}$, the false positive rate (FPR) is then given by,

$$\begin{aligned} \mathrm{FPR_{BF}} &= \Pr(B[h_1(x)] = 1 \wedge \cdots \wedge B[h_k(x)] = 1) \\ &= (1 - p_0)^k \approx \left(1 - e^{-kn/m}\right)^k. \end{aligned} \qquad (2)$$

In practice, n is known in advance as the expected size of element set, and m is the size of the bit array. Given n and m, the optimal number of hash functions is $k^{opt} = \frac{m}{n} \ln 2$ by setting the derivative of Eq. (2) to zero, and the corresponding optimal FPR is $0.5^{\frac{m}{n} \ln 2} \approx 0.6185^{\frac{m}{n}}$. To make it more general, the FPR of standard Bloom filter as well as its variants like Cuckoo filter [16] on a static element set can be modeled as α^t where $\alpha \in (0, 1)$ is a constant and $t = \frac{m}{n}$ stands for the number of bits used to encode each element.

2.2 Learned Bloom Filter

The formal definition to the learned Bloom filter (LBF), which is first introduced by Kraska et al. [25] and further refined by Mitzenmacher [30], is given as follows.

Definition 1. (Learned Bloom Filer) Given an element set $\mathcal{S}$, an LBF can be represented as a triple (f, τ, BF) where $f : x \in \mathcal{S} \to [0, 1]$ is a pre-trained classifier (learned oracle), τ is the decision threshold of f where $f(x) \geq \tau$ implies $x \in \mathcal{S}$, and BF is the backup standard Bloom filter built on the set of all elements in $\mathcal{S}$ that are wrongly predicted as non-members, i.e., $\{x | f(x) < \tau, x \in \mathcal{S}\}$.

As shown in Figure 1, to process a membership query of element y, we trust the positive prediction but challenge the negative outputs from the classifier, and a negative answer is made i.f.f. both classifier and backup filter determine y is a non-member. Such a construction of LBF ensures one-sided error (i.e., no false negatives), and for any non-member element $y \notin \mathcal{S}$, the FPR is given by:

$$\mathrm{FPR_{LBF}} = \Pr(f(y) \geq \tau) + \Pr(f(y) < \tau) \cdot \alpha^{m/|\mathcal{S}_N|}, \qquad (3)$$

where m is the number of bits allocated to the backup filter BF, $\mathcal{S}_N = \{x | f(x) < \tau, x \in \mathcal{S}\}$, and α is a constant that depends on the implementation of the backup filter.

In Eq. (3), $\Pr(f(y) \geq \tau)$ can be interpreted as the false positive probability of the classifier, which is essentially a *random variable* and depends on how y is picked, i.e., the query distribution. In statistics literature, $\Pr(f(y) \geq \tau)$ can be estimated by using a probe dataset which is assumed to be sampled from the same distribution of the dataset used to train classifier f. Given an LBF (f, τ, BF), supposing the classifier f and the backup filter BF use ζ and m bits respectively, combining Eq. (2) and Eq. (3), an LBF is better than a standard BF consuming the same space (i.e., $\zeta + m$ bits) if the following inequality holds,

$$F_p + (1 - F_p) \cdot \alpha^{b/F_n} < \alpha^{\zeta/n + b}, \qquad (4)$$

where F_p is the false positive probability of the classifier, $F_n = |\mathcal{S}_N|/n$, $b = m/n$, and α is a constant (Section 2.1). Note that, the left- and right-hand sides of Eq. (4) stand for the FPR of LBF and BF with the same storage and the optimal number of hash functions.

2.3 BF Stability on Data Streams

The construction of either standard BF or LBF relies on knowing the whole picture of the element set $\mathcal{S}$. To start our discussion on BF stability over dynamically growing element sets, we first define the membership testing on data streams.

Definition 2. (Membership Query on Data Stream) Consider an unbounded stream of elements $x_1, \cdots, x_n$ where n can be infinite, for a queried element y, the membership query of y returns true i.f.f. $y \in \{x_1, \cdots, x_n\}$, i.e., y has been seen before timestamp n.

As we have stated earlier, any Bloom filter using limited space cannot achieve bounded one-sided error on unbounded data streams. Intuitively, this can be explained using the model FPR $= \alpha^{m/n}$ (Section 2.1), according to which we have $\lim_{n \to \infty}$ FPR $= 1$. To achieve a non-trivial FPR bound using limited storage, Deng and Rafiei [13] first introduce the concept of Stable Bloom Filter (SBF) with an idea of clearing random bits when inserting an element, in order to make rooms for future elements.

An SBF represents a dynamically growing set using an array of m counters $SBF[1, \cdots, m]$, instead of bit array used by standard BF, and each counter is allocated with d bits (i.e., $SBF[i]$ is between 0 and $Max = 2^d - 1$). To insert an element x, P counters are first randomly selected and decremented by 1 if they are non-zero. Then, similar to a standard BF, K independent hash values $h_1(x), \cdots, h_K(x)$ are calculated and counters $SBF[h_1(x)], \cdots, SBF[h_K(x)]$ are set to Max. For a membership query of an element y, "yes" would be returned if none of $SBF[h_1(y)], \cdots, SBF[h_K(y)]$ is 0, otherwise "no" would be returned. The insertion algorithm and membership query processing using SBF are shown in Figure 4a and Figure 4b.

When applying SBF to a data stream $x_1, \cdots, x_n$, a key observation is that, with the counter decrement behavior,

```
Function $insert(SBF, x)$
    for $p = 1, \cdots, P$ do
        $idx \leftarrow Rand(1, m)$
        if $SBF[idx] > 0$ then
            $SBF[idx] \leftarrow SBF[idx] - 1$
        end
    end
    for $k = 1, \cdots, K$ do
        $SBF[h_k(x)] \leftarrow Max$
    end
end
```

(a) Insertion algorithm of SBF.

```
Function $query(SBF, y)$
    for $k = 1, \cdots, K$ do
        if $SBF[h_k(y)] = 0$ then
            return false
        end
    end
    return true
end
```

(b) Query processing using SBF.

```
Input: an s-SLBF $(f, \tau, SBF)$ and a
       sequence of $N$ elements to be
       inserted $x_1, \cdots, x_N$
for $i = 1, \cdots N$ do
    if $f(x_i) > \tau$ then
        continue
    else
        $insert(SBF, x_i)$
    end
end
```

(c) Insertion algorithm of s-SLBF.

```
Input: an s-SLBF $(f, \tau, SBF)$ and a
       query element $y$
$conf \leftarrow f(y)$
if $conf > \tau$ then
    return true
else
    return SBF
end
```

(d) Query processing using s-SLBF.

```
Input: a classifier $f$, a filter array
       $SBF_1, \cdots, SBF_g$, and a
       sequence of $N$ elements
       $x_1, \cdots, x_N$
for $i = 1, \cdots, N$ do
    $j \leftarrow$ the interval $[\tau_{j-1}, \tau_j]$ which
      $f(x_i)$ belongs to
    $insert(SBF_j, x_i)$
end
```

(e) Insertion algorithm of g-SLBF.

```
Input: a classifier $f$, a filter array
       $SBF_1, \cdots, SBF_g$, and a
       query element $y$
$j \leftarrow$ the interval $[\tau_{j-1}, \tau_j]$ which
  $f(x_i)$ belongs to
return $query(SBF_j, y)$
```

(f) Query processing using g-SLBF.

Figure 4: Insertion and membership query processing algorithms of SBF, s-SLBF and g-SLBF.

the fraction of '0' counters in the array tends to be a constant as the insertion number $n \to \infty$. According to Theorem 2 of [13], given an SBF with m counters, denoting $p_0^{(n)}$ as the fraction of counters with value 0 after inserting n elements, the limit of $p_0^{(n)}$ is,

$$\lim_{n\to\infty} p_0^{(n)} = \left(\frac{1}{1+\frac{1}{P(1/K-1/m)}}\right)^{Max}. \tag{5}$$

In addition, $p_0^{(n)} - p_0^{(n-1)} \approx \frac{K}{m}\left(1-\frac{K}{m}\right)^n$, which indicates an exponential convergence.

The zero fraction $p_0^{(n)}$ can be interpreted as the probability that an arbitrary counter $SBF[i]$ is 0 after inserting n elements from a data stream. Thus, for any query element y not in the data stream, the false positive rate[1] generated by SBF is given by,

$$\lim_{n\to\infty} \mathrm{FPR}_{\mathrm{SBF}} = \lim_{n\to\infty}\left(1-p_0^{(n)}\right)^K \overset{(m\gg K)}{\approx} \left(1-\left(\frac{1}{1+K/P}\right)^{Max}\right)^K. \tag{6}$$

This property of reaching a non-trivial FPR after a large number of insertions, instead of decaying to 1, is called "stable" for Bloom filters on data streams.

However, with the counter decrement operations, SBF achieves stable at the cost of a non-zero number of false negatives, which means an already inserted element x_i may be wrongly determined as a non-member by SBF. According to [13], the false negative rate (FNR) for SBF relates to not only the filter's parameters but also how the query elements distribute, i.e., the query distribution. Please refer to Section 3.3 for a detailed discussion on FNR over data streams.

2.4 Problem Statement

We have overviewed structures and analytical results of standard BF, LBF, and SBF. On static element sets, compared with standard BF, which has been used for decades, LBF shows its advantage of reducing the memory cost [25]. This inspires us to devise a new learned Bloom filter structure for approximate membership queries over a *dynamic element set* (as shown in Definition 2).

Specifically, we consider two operations for such a data structure: `insert` which adds an element x to the filter, and `query` which returns the membership testing result of an element y using the filter. When applied to an element stream, such a learned filter is expected to 1) achieve the stable property (i.e., the FPR reaches a non-trivial value as $n \to \infty$); and 2) consume less storage at a competitive FPR/FNR level compared with a non-learned filter optimized on data streams (e.g., the SBF).

3. STABLE LEARNED BLOOM FILTER

In this section, we present two data structures (Section 3.1 and Section 3.2) as well as theoretical analysis (Section 3.3) to address the approximate membership testing problem for streaming data under the context of learned indexes.

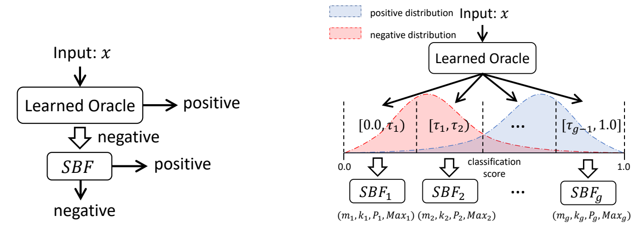
3.1 Single SLBF

To make the original LBF framework *stable* after an unbounded number of insertions, an intuitive idea is to replace the backup filter in LBF, which is a standard BF, with a stable Bloom filter (Section 2.3). Such structure, as illustrated in Figure 5a, is referred to as single stable Learned Bloom filter (s-SLBF) where *single* means there is a single backup filter in such framework.

Definition 3. (s-SLBF) An s-SLBF can be represented as a triple (f, τ, SBF) where f is a pre-trained learned oracle (i.e., classifier), τ is the corresponding decision threshold and SBF is a backup stable Bloom filter.

To insert a new element x (as shown in Figure 4c), the membership confidence $f(x)$ is first calculated and compared

[1] When referred to the false positive rate over data streams, we simply use FPR, instead of the limit of FPR when $n \to \infty$, unless discussing the convergence rate of FPR.



(a) Illustration of s-SLBF.

(b) Illustration of g-SLBF.

Figure 5: Motivation and overview of our stable learned Bloom filter structures.

with the threshold τ. If $f(x) \geq \tau$, which means the model determines x is already predicted as a member, the insertion process will directly terminate; otherwise, x is inserted to SBF. To query an element y (as shown in Figure 4d), a positive answer is returned if $f(y) \geq \tau$ or $f(y) < \tau$ but SBF determines y as positive.

Though very similar to the original LBF, the way how the classifier in s-SLBF (i.e., f and τ) is obtained is intrinsically different. Recall that the classifier f used in the original LBF is trained over a binary dataset $\{(x_i, y_i = 1)|x_i \in \mathcal{S}\} \cup \{(x_i, y_i = 0)|x_i \in \mathcal{N}\}$ where $\mathcal{S}$ is the element set used to build the filter, which is static, and $\mathcal{N}$ is the set of synthetic negative samples. In contrast, for s-SLBF, since it is built before fed with the data stream, instead of the exact element set (i.e., $\mathcal{S}$), the prior knowledge would be accessible to train the classifier. This yields the fundamental difference of applicable scenarios between LBF and SLBF. Please refer to Appendix A for a detailed discussion.

It is obvious that s-SLBF is *stable* after a substantial number of insertions, if we assume the streaming elements follow the distribution used in training the classifier. Suppose the parameters used in SBF are m, K, P, Max, based on Eq. (3) and Eq. (6), the expected FPR of s-SLBF is given by,

$$\mathbf{E}[\text{FPR}] = F_p + (1 - F_p) \cdot \left(1 - \left(\frac{1}{1 + K/P}\right)^{Max}\right)^K \tag{7}$$

where $F_p = \Pr_{y \sim \mathcal{D}_N}(f(y) \geq \tau)$ and $\mathcal{D}_N$ is the distribution of non-members.

If the classifier performs well, at the same expected FPR level (at stable), s-SLBF is supposed to save more space compared with a pure SBF. We explain this advantage using the following example.

According to Eq. (3), for SBF's, the FPR at stable, is insensitive to the number of counters m since $m \gg K$. Without loss of generality, we assume $F_p = 0.01$ and $F_n = 0.5$ for the classifier of s-SLBF. We further pick SBF parameters P, K and Max such that $(1 - (1/(1 + K/P))^{Max})^K \approx 0.1$. Under such settings, according to Eq. (7), the FPR bound at stable of s-SLBF is $0.01 + 0.99 * 0.01 \approx 0.1$, which is at the same level compared with a pure SBF using identical parameters. Since the total storage cost of an SBF is $m \cdot \lfloor \log_2(Max) + 1 \rfloor$ where Max is fixed, the only factor influencing total storage lies on the number of used counters m. According to Eq. (5) and Eq. (6), for SBF's, increasing or decreasing m does not influence the FPR at stable (converged); however, m influences how fast FPR converges to its stable point. Thus, to make it fair, we compare the difference in storage cost for s-SLBF and SBF under similar stable FPR and convergence rate. Suppose the numbers of counters used by SBF and s-SLBF are m and m', respectively. Let the two filters have the identical convergence rate, and then we have the following equation,

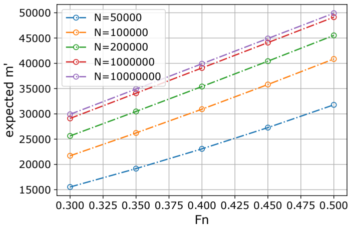


Figure 6: Numerical simulation result of Eq. (8) by varying the number of insertions N and the false negative probability of the classifier (i.e., F_n).

$$\frac{K}{m}\left(1 - \frac{K}{m}\right)^N = \frac{K}{m'}\left(1 - \frac{K}{m'}\right)^{N \cdot F_n}. \tag{8}$$

By setting $m = 10^6$, $K = 6$, $N = 10^8$, and $F_n = 0.5$, we can solve the numerical solution as $m' \approx 4.7 \times 10^4$, which implies a 53% reduction in terms of total storage. Note that we ignore the space cost caused by the classifier in s-SLBF since it is usually much smaller than the counter array.

To further understand the relationship between m' and F_n regarding Eq. (8), we vary F_n from 0.3 to 0.5 and N from 5×10^4 to 10^7, and show the value of m' as a function of F_n in Figure 6. The gap between neighboring lines decreases as N increases, which demonstrates that the filter approaches the stable point when the number of insertions N becomes substantially large. We can observe an approximately linear relation between m' and F_n, which is reasonable since F_n determines how many elements in the insertion stream "escape" from the classifier and are added to the backup SBF. A higher F_n of the classifier implies more elements need to be inserted to the backup SBF and thus more counters are required to retain a similar convergence rate. Note that, the above simulation fixes other parameters like F_p to simplify the analysis and provide a general insight into the advantage of learned Bloom filters.

3.2 Grouping SLBF

As a straight extension, the s-SLBF has been shown to be stable using potentially lower storage than an SBF as we expect. However, it also inherits a major drawback from the original LBF framework, i.e., trusting all the positive predictions made by the classifier. This over reliance makes s-SLBF as well as the original LBF fragile when the classifier is not trustworthy. Such effect can be explained using Eq. (7) where F_p, the false positive probability of the classifier, is the lower bound of the overall FPR.

Besides the over reliance issue of the learned oracle, the single backup filter design also omits useful information provided by the learned oracle. Figure 5a illustrates how the classifier works in s-SLBF, from which we can find that both the false positives and the false negatives of the classifier come from the setup of a hard decision boundary. That is, negative (positive) elements falling on the left side are all categorized as positive (negative). This hard decision rule does not distinguish the confidence levels of elements falling on the same side, which is illustrated as follows.

Example 3. Suppose the decision threshold is set to $\tau = 0.7$, there is no difference for elements with prediction scores 0.69 and 0.01, respectively, both of which would be treated in the same way, i.e., feeding to the same backup SBF. Similarly, for elements with scores 0.71 and 0.99, both of which would be directly judged as members without any further action on the backup filter.

From the analysis above, we realize the major flaw of s-SLBF as well as the original LBF is the single backup filter nature. To further improve s-SLBF, we introduce the second data structure called grouping stable Learned Bloom filter (g-SLBF) by breaking the classification score (in range $[0, 1]$) into several intervals and allocating independent sub-filters for each interval.

Definition 4. (g-SLBF) A Grouping SLBF (g-SLBF) consists of a classifier f, and g heterogeneous SBF's (also known as "sub-filters") $SBF_1, \cdots, SBF_g$ where the j-th SBF SBF_j is described by a tuple of parameters (m_j, K_j, P_j, Max_j). A partition of the interval $[0, 1]$ leads to g sub-intervals, i.e., $[\tau_0 = 0, \tau_1], [\tau_1, \tau 2], \cdots, [\tau_{g-1}, \tau_g = 1]$, which are used to map an element x to the SBF's regarding their prediction values $f(x)$. More specifically, a new element x is inserted to SBF_j if $f(x) \in (\tau_{j-1}, \tau_j]$ (as shown in Figure 4e). To test the membership of element y, we directly ask the sub-filter SBF_j mapped from $f(y)$ (as shown in Figure 4f).

As illustrated in Figure 5b, the basic idea of g-SLBF is to partition elements in the insertion stream into several subgroups based on the membership confidence given by the classifier. Intuitively, for those elements to be inserted with low membership confidence (i.e., locating at the left side of the confidence distribution as shown in Figure 5b), since the classifier has a relatively high F_p in this range, we can adjust the corresponding sub-filter SBF_j to compensate the loss of FPR by setting K, P and Max appropriately. Besides, since F_n is low in this range, which means there would not be too many elements to be inserted to SBF_j, fewer counters are needed to be allocated to achieve the desired FPR at stable in a satisfactory convergence speed. On the other hand, for elements with high confidence (i.e., locating at the right side in Figure 5b), the FPR requirement of the SBF can be loosed since elements in this range already have a high prior possibility of being a member, and similarly, since the classifier might wrongly determine many member elements as non-member (i.e., high value of F_n), more counters are required to let SBF_j converge to its stable point.

It is noteworthy that s-SLBF can be regarded as a special case of the g-SLBF by setting $g = 2$, i.e., only one decision threshold τ, and letting the sub-filter in range $[\tau, 1]$ always give positive answers. Compared with s-SLBF, where the positive predictions from classifier are fully trusted, g-SLBF ($g > 2$) is more conservative towards the classifier output since all the membership decisions are jointly made by the classifiers and the backup filter. Such property makes the g-SLBF more robust against the quality of the classifier (e.g., the incoming element stream does not strictly follow the distribution as that of the training data). Note that, the superiority in robustness of g-SLBF will be demonstrated both analytically and experimentally in the following sections.

3.3 Analytical Results

In this section, we analyze the FPR, FNR and convergence behavior of our two SLBF structures. Note that, we focus on g-SLBF since s-SLBF is a special case of g-SLBF, whose theoretical results naturally apply to s-SLBF. We first give some preliminary notations in the following.

For the j-th classification score interval $[\tau_{j-1}, \tau_j]$ in g-SLBF, we define two probabilities p_j and q_j as follows,

$$\begin{aligned} p_j &= \Pr_{x \in \mathcal{D}_N}(f(x) \in [\tau_{j-1}, \tau_{j-1}]), \\ q_j &= \Pr_{x \in \mathcal{D}_P}(f(x) \in [\tau_{j-1}, \tau_{j-1}]), \end{aligned} \tag{9}$$

where $\mathcal{D}_N$ and $\mathcal{D}_P$ are distributions of non-members and members. The pair (p_j, q_j) depicts the false positive and false negative behaviors of the classifier in the range $[\tau_{j-1}, \tau_j]$. Note that, it is generally hard to know the exact values of p_j and q_j as $\mathcal{D}_N$ and $\mathcal{D}_P$ are unknown. However, as what would be shown in the parameter setting (Section 4), we use the test datasets to estimate p_j and q_j.

In the analysis presented hereafter, we adopt the following two assumptions about the classifier and data, which are not hard to understand and have been adopted by existing learned Bloom filter works [12, 25, 30].

Assumption 1. The members and the non-members of the filter follow the distributions $\mathcal{D}_P$ and $\mathcal{D}_N$, respectively. Thus, the FPR of an SLBF is the expected false negative rate over the non-member distribution $\mathcal{D}_N$.

Assumption 2. For $j = 1, \cdots, g$, it holds that $p_1 \geq p_2 \geq \cdots \geq p_g$ and $q_1 \leq q_2 \leq \cdots \leq q_g$.

3.3.1 False Positive Rate and Stability

Suppose a sequence of n elements $x_1, \cdots, x_n$, where $x_i \sim \mathcal{D}_P$, has been inserted to a g-SLBF. Then, for a new query with an element drawn from the non-member distribution $\mathcal{D}_N$, the expected FPR of this g-SLBF (at stable) is

$$\mathbf{E}[\text{FPR}] = \sum_{j=1}^{g} p_j \cdot \underbrace{\left(1 - \left(\frac{1}{1 + K_j/P_j}\right)^{Max_j}\right)^{K_j}}_{\text{denoted by } \alpha_j}. \tag{10}$$

Suppose there are g SBF's with α_j's satisfying $\alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_g$. Then considering the intervals depicted in Assumption 2, Lemma 1 describes how to allocate the SBF's to these intervals to minimize $\mathbf{E}$[FPR], which also validates our discussion in Section 3.1.

LEMMA 1. *Allocating the filter with α_j to the interval $[\tau_{j-1}, \tau_j]$ for $j = 1, \cdots, g$ minimizes $\mathbf{E}[FPR]$.*

PROOF. Following Assumption 2, $p_1 \geq \cdots \geq p_g$ and $\alpha_1 \geq \cdots \geq \alpha_g$, then according to the Rearrangement inequality [20], for any other permutation $\alpha_{\sigma(1)}, \alpha_{\sigma(2)} \cdots \alpha_{\sigma(g)}$,

$$\sum_{j=1}^{g} p_j \cdot \alpha_{\sigma(j)} \geq \sum_{j=1}^{g} p_j \cdot \alpha_j = \mathbf{E}[FPR], \tag{11}$$

which proves this lemma. □

Based on Lemma 1, we then prove an upper bound of the expected FRP of g-SLBF (at stable), which is free of p_j.

THEOREM 1 (FPR UPPER BOUND). *An upper-bound of the expected FPR of g-SLBF at the stable state is the arithmetic mean of FPR of the g sub-filters $SBF_1, \cdots, SBF_g$, i.e., $\mathbf{E}[FPR] \leq \frac{1}{g}\sum_{j=1}^{g} \alpha_j$.*

PROOF. Since $p_1 \geq p_2 \geq \cdots \geq p_g$ and $\alpha_1 \geq \alpha_2 \geq \cdots \geq \alpha_g$, according to the Chebyshev's sum inequality [20], it always holds that,

$$\begin{aligned} \mathbf{E}[\text{FPR}] &= \sum_{j=1}^{g} p_j \cdot \alpha_j \leq g \cdot \left(\frac{1}{g}\sum_{j=1}^{g} p_j\right) \cdot \left(\frac{1}{g}\sum_{j=1}^{g} \alpha_j\right) \\ &= 1 \cdot \frac{1}{g}\sum_{j=1}^{g} \alpha_j = \frac{1}{g}\sum_{j=1}^{g} \alpha_j. \end{aligned} \tag{12}$$

Thus we complete the proof. □

Recall that in Section 3.2, we argue that g-SLBF is more robust against the classifier quality than the s-SLBF. As we discussed earlier, the s-SLBF (as well as the original LBF) adopts a single backup filter structure, which makes the classifier's FPR directly upper bounds the overall FPR as a consequence. However, according to Theorem 1, the FPR of g-SLBF is bounded by the arithmetic mean of sub-filters' FPR, which is independent of the classifier quality and specific distribution assumption. Note that, this inequality holds under the assumption that $p_1 \geq p_2 \geq \cdots \geq p_g$, which generally holds if the dataset is "learnable" (see our validation of this assumption in Section 5.3). We also conduct experimental study to validate the robustness claim by adding distortions to the distribution of element streams. The results show that the speed of FPR getting deteriorated of g-SLBF is much slower than that of s-SLBF w.r.t. distribution distortion. Please refer to Appendix E for more details.

3.3.2 Convergence Rate

The following theorem describes the convergence rate of g-SLBF, i.e., how fast the filter approaches its stable FPR.

THEOREM 2 (G-SLBF CONVERGENCE). *g-SLBF converges to its stable point, which is shown in Eq.* (10)*, at a speed of* $O(exp(-C \cdot n))$ *where* n *is the number of total insertions and* $C = \min_j \frac{q_j m_j}{K_j}$ *for* $j = 1, \cdots, g$.

PROOF. As introduced in Section 2.3, the rate of convergence for sub-filter SBF_j is,

$$\begin{aligned}\frac{K_j}{m_j}\left(1-\frac{K_j}{m_j}\right)^{q_j \cdot n} &= \frac{K_j}{m_j}\left(1-\frac{K_j}{m_j}\right)^{\frac{K_j}{m_j}\cdot\frac{m_j q_j n}{K_j}} \\ &\approx O\left(exp\left(-\frac{q_j}{k_j}n\right)\right).\end{aligned} \quad (13)$$

Apparently, the g-SLBF approaches its stable point if and only if $SBF_1, \cdots, SBF_g$ are all stable. Thus, the overall convergence rate is that of the slowest sub-filter, i.e., $O(exp(-n \cdot \min_j \frac{q_j m_j}{K_j}))$. □

3.3.3 False Negative Rate

A false negative occurs when a negative answer is given to a query of a member, i.e., an element which has been inserted before. Similar to SBF, our g-SLBF allows a number of false negatives to achieve a bounded FPR (stable) using limited storage on unbounded data streams. To quantify the influence of false negatives for our data structure, we first review the FNR results of SBF, which is adopted by our g-SLBF as sub-filters.

Different from the FPR which is determined by only filter parameters, the FNR of SBF also depends on the characteristic of input data stream and query workloads. Given an element x_i in a data stream, let δ_i be the number of timestamps between the most recent insertion and query of the element x_i, which is referred to as the "gap" of x_i. According to [13], for an inserted element x_i, the false negative probability for element x_i is,

$$\Pr(FN_i) = 1 - \prod_{j=1}^{K}(1 - \Pr(SBF[h_j(x_i)] = 0|\delta_i)), \quad (14)$$

where $\Pr(SBF[h_j(x_i)] = 0|\delta_i)$ is the probability that counter $SBF[h_j(x_i)]$ becomes 0 after δ_i times new insertions. Note that, if $\delta_i < Max$, then $\Pr(SBF[h_j(x_i)] = 0|\delta_i)$ is always 0 since it is impossible to decrease the counter to 0.

For our g-SLBF, which adopts a sequences of independent SBF's as sub-filters, supposing an element x_i which has been inserted to j-th sub-filter SBF_j, according to Eq. (14), its false negative probability is

$$\Pr(FN_i|x_i \in SBF_j) = 1 - (1 - p_N(\delta_i, k_{ij}))^{K_j}, \quad (15)$$

where $p_N(\cdot)$ is the probability that one of the K_j counters (x_i is mapped to) has been decremented to 0 after δ_i insertion operations to the filter (note that δ_i is the gap of x_i). $p_N(\cdot)$ is a function of δ_i and k_{ij} which is the probability of a mapped counter to be set to Max_j. Note that, k_{ij} is a *random variable* which depends on the occurrence frequency of each element in the insertion stream. That is to say, $p_N(\cdot)$ is different for each x_i, which makes it rather difficult to precisely compute FNR as we do not have prior knowledge towards such insertion frequencies. On the other hand, the simulation results shown in [13] reveal that such frequency features make a little impact on the overall FNR result provided that the data stream is large enough ($n \to \infty$). Thus, in our work, without loss of generality, we derive the expected FNR of the g-SLBF by assuming that an element appears only once in the insertion data stream, i.e., there is no duplicate insertion.

Under the above assumption, k_{ij} are in the same form for each x_i, which is $k_{ij} = k_j = \frac{1}{n_j}(1+\sum_{l=1}^{n_j-1} I_l)$ where n_j is the number of elements already inserted to the filter SBF_j and I_l is an Bernoulli distributed random variable with $\Pr(I_l = 1) = K_j/m_j$. Consequently, the overall expected FNR of g-SLBF can be deducted as

$$\begin{aligned}\mathbf{E}[\text{FNR}] &= \sum_{j=1}^{g} \Pr(FN|x \in SBF_j) \cdot \Pr(x \in SBF_j) \\ &= \sum_{j=1}^{g}\left(1 - (1 - p_N(\tilde{\delta}, k_j))^{K_j}\right) \cdot q_j,\end{aligned} \quad (16)$$

where $\tilde{\delta}$ is the average gap of the data stream. The concrete evaluation of $p_N(\cdot)$ based on $\tilde{\delta}$ and k_j can be found in Appendix C. Once p_N and the filter parameters are determined, we can estimate FNR using the equation above.

In summary, as a side effect, false negatives are inevitable for our g-SLBF to obtain stability over an unbounded number of insertions, which is similar to SBF [13]. Unlike FPR, determining FNR of g-SLBF relies on the prior knowledge of both insertion element stream as well as query element stream (to calculate the gap value δ_i for each inserted element x_i). To tackle the false negative issue, in the following section, we devise a parameter setting strategy with the objective of minimizing FNR while bounding FPR by a user-given threshold. Detailed evaluation results presented in Section 5 demonstrate that our g-SLBF has a similar FNR compared with SBF but achieves a better FPR/storage trade-off, i.e., in the same FNR and FPR level, our proposed learned filter is supposed to save more storage.

3.3.4 Time Complexity

Both the insertion operation and the membership query processing using g-SLBF take $O(1)$ time. Supposing the model prediction takes time $O(M)$, the time complexities of insertion and query processing are $O(M + \max_j(P_j + K_j))$ and $O(M + \max_j(K_j))$, respectively. Since all M, P_j, K_j are user-specified constants, we conclude that the insertion and membership testing using g-SLBF take constant time.

4. PARAMETER SETTING

In this section, we discuss how to properly set the parameters of g-SLBF according to the analytical results in Section 3.3. Again, without loss of generality, we focus on g-SLBF, and the parameter setting strategies can be naturally extended to s-SLBF.

Overview. The parameters include the number of groups g, the partition values $\tau_1, \cdots, \tau_{g-1}$, and the specification for each sub-filter, i.e., (m_j, P_j, K_j, Max_j) for SBF_j. The users are enabled to provide their desired g, upper bound of expected FPR ϵ, and storage budget B (i.e., number of bits). Then, we set up the aforementioned parameters by minimizing the expected FNR (Eq. (16)) while bounding the expected FPR (Eq. (10)) within ϵ and the storage cost within B, similar to the setting of SBF [13].

Setting of $\tau_1, \cdots, \tau_{g-1}$. Given the total number of groups g, we uniformly partition the interval $[0, 1]$, leading to $[\frac{j-1}{g}, \frac{j}{g}]$ as $[\tau_{j-1}, \tau_j]$. The intervals are equally important in terms of our analysis in Section 3.3, so we simply adopt an uniform partition. For each specified decision interval, two test datasets are used to estimate its p_j and q_j. Specifically, given the member and non-member sample sets $\widetilde{\mathcal{S}_P}$ and $\widetilde{\mathcal{S}_N}$, p_j and q_j can be estimated as

$$\begin{aligned} \widehat{p_j} &= |\{x | x \in \widetilde{\mathcal{S}_N}, f(x) \in [\tau_{j-1}, \tau_j]\}| \ / \ |\widetilde{\mathcal{S}_N}|, \\ \widehat{q_j} &= |\{x | x \in \widetilde{\mathcal{S}_P}, f(x) \in [\tau_{j-1}, \tau_j]\}| \ / \ |\widetilde{\mathcal{S}_P}|. \end{aligned} \tag{17}$$

Recall that we need to bound the overall expected FPR within ϵ. According to Lemma 1, by assuming $p_j \cdot \alpha_j = C$ where C is a constant, which implies an inverse proportional relationship between p_j and α_j, an upper bound of FPR for each sub-filter SBF_j, denoted by α_j^{obj}, is then derived as

$$\alpha_j^{obj} = \frac{\frac{1}{\widehat{p_j}} \cdot \epsilon}{\sum_{l=1}^{g} \frac{1}{\widehat{p_l}}}. \tag{18}$$

Setting of K_j and Max_j. We then determine the values of K_j and Max_j by minimizing the expected FNR (Eq. (16)). According to the the observation in [13], the optimal or near optimal value of K_j is determined mainly by Max_j and the FPR bound α_j^{obj}, and insensitive to m_j and the input data stream. The optimal Max_j relates to the average gap value of the data stream. Besides, Max_j should not be too large, as a large Max_j will lead to a significantly large P_j (counter decrements) during query and thereby low query efficiency. Thus, we search K_j and Max_j in the space $K_j = 1, \cdots, 10$ and $Max_j \in \{1, 3, 7\}$ (corresponding to #bits per counter$\in \{1, 2, 3\}$). We enumerate all the possible combinations of (K_j, Max_j) to pick the optimal pair such that the estimated FNR of SBF_j (using Eq. (15) with a presumed average gap value) is minimized.

Setting of P_j. With K_j, Max_j and α_j^{obj}, P_j can be solved w.r.t. Eq. (6) as follows,

$$P_j = \frac{K_j}{1 - \left(1 - \left(\alpha_j^{obj}\right)^{1/K_j}\right)^{1/Max_j}}. \tag{19}$$

Setting of m_j. Finally, to set the number of counters for BSF_j m_j w.r.t. the total bit budget B, as we have discussed in Section 5b, we require all the sub-filters are required to converge at a similar speed. Thus, according to Theorem 2, we have $\frac{q_j m_j}{K_j} = W$ where W is a constant. To bound the total number of bits usage, m_j can be determined by

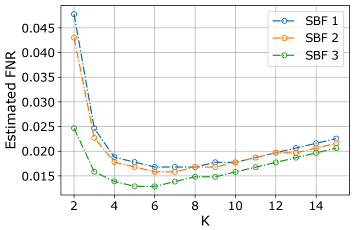


Figure 7: Simulated FNR for each sub-filter in Example 4 with $\delta = 100$, $m = 10^5$ and $Max = 1$.

$$m_j = \frac{\frac{K_j}{\widehat{q_j}} \cdot B}{\sum_{l=1}^{g} \frac{K_l}{\widehat{q_l}} \cdot \lfloor \log_2(Max_l) + 1 \rfloor}. \tag{20}$$

Example 4. (Parameter Setting) Suppose that $g = 3$, $B = 16{,}384$, $\epsilon = 0.01$, and we are given a classifier which have $\widehat{p_1} = 0.485, \widehat{p_2} = 0.390, \widehat{p_3} = 0.125$ and $\widehat{q_1} = 0.090, \widehat{q_2} = 0.347, \widehat{q_3} = 0.563$. The FPR upper bound can then be calculated using Eq. (18) as $\alpha_1^{obj} = 0.0016, \alpha_2^{obj} = 0.0020, \alpha_3^{obj} = 0.0063$. To find the optimal K_j and Max_j, we presume the gap value as 100 and compute the values of FNR over different pairs of K_j & Max_j, which is shown in Figure 7. Note that, we only show the case of $Max = 1$ to due to the limited space. Figure 7 suggests the optimal setting of (K_j, Max_j) for $j = 1, 2, 3$ w.r.t. FNR is $(6, 1)$, $(6, 1)$ and $(5, 1)$, respectively. Then, according to Eq. (19), all P_j's are set to 12, and according to Eq. (20), the number of bits allocated to each sub-filter are 11,764, 3,054, and 1,566, respectively.

5. EXPERIMENTAL STUDY

In this section, we report the implementation details and experimental results on datasets from real-world applications. All the experiments were conducted on a Ubuntu laptop with Intel(R) Core(TM) i7-8550U CPU @ 1.99GHz and 16GB memory, and all the methods are implemented in C and compiled using GCC with -O3 optimization.

5.1 Baselines and Implementation Details

To show the effectiveness of our proposed data structures, we implement and compare five filters, including the standard Bloom filter (BF), the stable Bloom filter (SBF), the original learned Bloom filter (LBF), the simple SLBF (s-SLBF) and the grouping SLBF (g-SLBF).

BF and LBF. BF and LBF are the baselines in this experiment, to show how non-stable filters behave over streaming data. The BF implementation follows the most standard space-optimal Bloom filter scheme [9] where the number of hash functions is always set to the optimum. For LBF, we use relatively simple models like gradient boosting trees for the efficiency concern in the scenario of streaming data, instead of the deep learning models used in [25] (details would be discussed later).

SBF, s-SLBF and g-SLBF. These three filters achieve stability in the scenario of streaming data. The parameters of SBF and s-SLBF & g-SLBF are set according to [13] and our discussion in Section 4, respectively.

Hash function Implementation. All filters require the computation of K hash values. We adopt xxHash [1], which is an extremely fast non-cryptographic hashing scheme with high quality. In addition, instead of exactly computing K independent hash values, we use the speedup suggested in [23] where only two independent hash values h_a, h_b are calculated and the j-th hash value is given by $h_a + j * h_b$ for $j = 1, \cdots, K$.

Classifier Implementation. In the first paper on learned indexes [25], deep learning models, i.e., neural networks, are suggested to construct learned data structures. Specifically, they suggest a Recurrent Neural Network (RNN) for learned Bloom filters. Though deep models perform better on many tasks, considering the real-time requirement for membership query processing over data streams, commonly used deep learning platforms, like Tensorflow and PyTorch, are too heavy to be deployed. Tough the inference efficiency issue can be alleviated by using GPU, a new bottleneck might be migrating data between CPU and GPU. Since this is not a paper introducing new machine learning schemes, in pursuit of efficiency, we test and compare three lightweight models: logistic regression, support vector machine and gradient boosting tree (GBT) based on Catboost [3]. We found that GBT classifiers perform well enough on all our three real-world tasks considering classification quality (e.g., AUC), storage cost, and inference efficiency (details will be discussed later). Then, we adopt GBT as the classifier for both LBF and SLBF in our experiments.

5.2 Datasets, Parameters and Metrics

To demonstrate the effectiveness and efficiency of our stable learned Bloom filters, we test all five filters on three real-world datasets: Amazon, Attack and Higgs. We briefly introduce each dataset as follows, and the statistics of these datasets are summarized in Table 2.

Task 1: Amazon [2]. This dataset consists of resource access records from Amazon employees collected from 2010 to 2011 in which employees are allowed or denied access to resources over time. Each record contains a unique ID and 10 features which are used to build the classifiers.

Task 2: Attack [5]. This is a dataset of web attack traces. A total of 23 features are extracted including packet source/destination and traffic statistics.

Task 3: Higgs [4]. The Higgs data is a scientific dataset which asks for classifying whether a signal process produces Higgs bosons or not. There are in total 28 kinematic features obtained through particle detectors.

Table 2: Statistics of datasets.

Name	#Samples	#Positives	#Negatives
Amazon	91,690	86,382	5,308
Attack	2,278,689	923,216	1,355,473
Higgs	11,000,000	5,829,597	5,170,403

For each dataset, a small portion, specifically 20%, is sampled to obtain a pre-trained classifier and to estimate some parameters like p_j, q_j (Eq. (9)), and the remaining 80% data are used to generate the insertion and query streams. We train the classifiers for each task using the gradient boosting tree model, and the model information is shown in Table 3.

Insertion Workloads. All positive samples in dataset are regarded as members, and negative samples are regarded as non-members. For a dataset, the corresponding insertion workload is the sequence of positive samples. Note that, when inserted elements into non-learned filters like BF and SBF, we only insert the unique identifier to the filter, and the features associated with the element are discarded. Similarly, for learned filters like LBF, s-SLBF and g-SLBF, features are used to calculate the membership confidence score using the classifier, and only identifiers are inserted to the filter if necessary.

Table 3: Classifier information.

Dataset	Training Time	AUC	Inference Throughput	Storage
Amazon	0.27 s	0.87	9 Mops/s	173 KBits
Attack	3.49 s	0.91	11 Mops/s	328 KBits
Higgs	44.8 s	0.82	11 Mops/s	215 KBits

Query Workloads. We need the query workloads to evaluate the performance of all filters. According to the analysis in Section 3, FPR and FNR are measured for the members and non-members, respectively. Besides, FNR is also affected by the time gap between a member element being inserted to the filter and being queried. Thus, for each dataset, given a gap value δ, for each element x_i inserted to the filter, we will query it after δ insertions of other elements to measure the FNR. After all elements have been inserted (from the positive sample set of each task), we will query the filter using the negative sample set to measure the FPR. The empirical FNR and FPR under the query workloads are then calculated as $\text{EFNR} = \frac{\#\text{false negatives}}{\#\text{positive samples}}$ and $\text{EFPR} = \frac{\#\text{false positives}}{\#\text{negative samples}}$. Note that, in the results reported in this section, we fix δ as 2,000, which is a reasonable value for real-world applications. However, we also report results by varying δ, the results demonstrate a clear increasing tendency of FNR as δ increases for both SBF and our SLBF. This is reasonable since a higher δ increases the likelihood of the a counter in the backup filter being decreased to 0, which leads to false negatives. Please refer to Appendix D for more information.

Control variables. Three parameters, the desired FPR upper bound ϵ, the total storage budget (#bits) B, and the number of groups g for g-SLBF, are varied to evaluate the robustness of the filters. Table 4 summarizes the parameter settings for each dataset where the underlined values are regarded as default values.

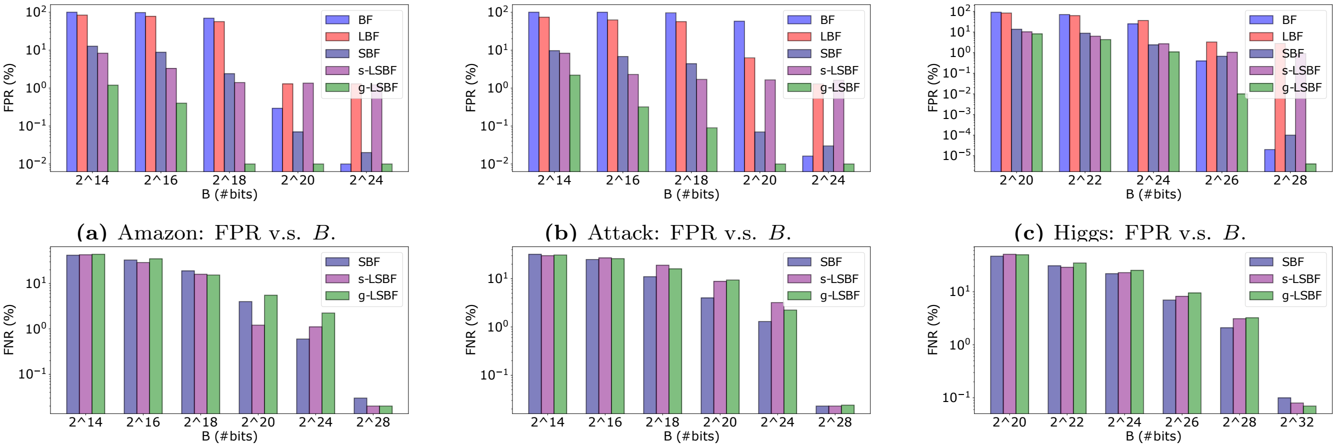
Table 4: Parameter setting.

Parameter	Values
ϵ	0.5%, 1%, $\underline{5\%}$, 10%, 20%
g	2, 4, $\underline{6}$, 8, 10
B	Amazon&Attack: $2^{14}, 2^{16}, \underline{2^{18}}, 2^{20}, 2^{24}$ Higgs: $2^{20}, 2^{22}, \underline{2^{24}}, 2^{26}, 2^{28}$

5.3 Experimental Results

Validation of Assumptions. In Section 3.3, to analyze the performance of our SLBF, we make the assumption that $p_1 \geq p_2 \geq \cdots \geq p_g$ and $q_1 \leq q_2 \leq \cdots \leq q_g$ for a well trained classifier. To validate such assumption, we use the positive set and negative set for datasets Attack and Higgs to calculate the corresponding classification scores and draw the histograms as shown in Figure 10, from which we can verify our assumption. Besides, the score histogram can also be used to guide the setting of g since we can keep on partitioning the interval until such monotonic relationship does not hold.

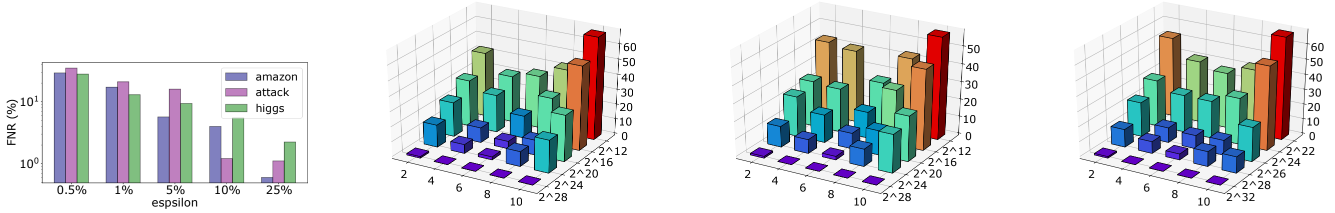
Validation of Stability. To verify the stability of the proposed filters, we test g-SLBF using the Amazon dataset by setting $g = 6$, $\epsilon = 10\%$ and varying B in the range $2^{10} \cdots 2^{18}$. Specifically, we measure the empirical FPR using the negative sample set after every 4,000 new insertions and plot the results in Figure 11. We can find that EFPR grows



(a) Amazon: FPR v.s. B. **(b)** Attack: FPR v.s. B. **(c)** Higgs: FPR v.s. B.

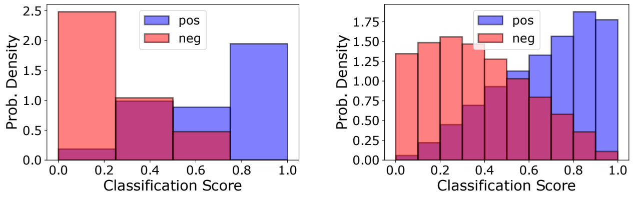
(d) Amazon: FNR v.s. B. **(e)** Attack: FNR v.s. B. **(f)** Higgs: FNR v.s. B.

Figure 8: Overall evaluation results on FPR and FNR w.r.t. the number of allocated bits for BF, LBF, SBF, s-SLBF and g-SLBF. Note that BF and LBF have no false negatives and all the ratios are in log-scale.



(a) FNR v.s. ϵ. **(b)** Amazon: FNR v.s. (g, B). **(c)** Attack: FNR v.s. (g, B). **(d)** Higgs: FNR v.s. (g, B).

Figure 9: Experimental results of g-SLBF under various parameter settings.



(a) Dataset Attack. **(b)** Dataset Higgs.

Figure 10: Histograms of classification score where blue bars and red bars refers to the positive (member) set and negative (non-member) set.

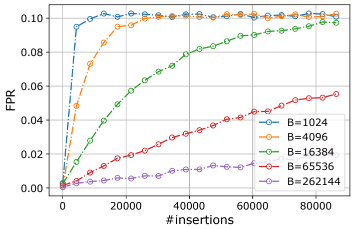


Figure 11: Experiment result of stability verification for g-SLBF on Amazon dataset with $g = 6$ and $\epsilon = 10\%$.

as new elements are inserted to the filter but approaches a stable value of 0.1. In addition, a smaller B leads to faster convergence of the filter. The stability results for g-SLBF on other datasets and those for s-SLBF are similar to Figure 11, and are omitted due to the limited space.

Overall Comparison. This experiment tests all five filters on three real datasets. By fixing g and ϵ to their default values, the empirical FPR and FNR (calculated using the query workloads) versus the bit budget B are shown in Figure 8. Note that, B refers to the bits allocated to the (backup) filters, which is also the actual storage cost for BF and SBF. For LBF, g-SLBF, and s-SLBF, their actual storage cost is B plus the model (classifier) size. However, as shown in Table 3, we can find that the model size is constant and relatively small. More specifically, consider the minimum B such that BF and LBF yields significant FPR in Figure 8a–8c, e.g., $B \approx 2^{20}$ for Amazon, $B \approx 2^{22}$ for Attack and $B \approx 2^{25}$ for Higgs, which means that the FPR of either BF or LBF is too high and unbearable when B is then less than this value. Even in this case, the cost of the classifier accounts for only 0.5% $\sim$ 16% of B. Thus, for all of the five filters, we can simply treat the bit budget B as their storage cost.

We first compare the FPR of the five filters regarding different B's, which is shown in Figure 8a–8c. The FPR for either BF or LBF is very high (near 100%) when B is very limited but will decrease as B increases, which means they need to be assigned a considerably large B to achieve a nice FPR. In contrast, the FPR of SBF, s-SLBF, and g-SLBF are all bounded within 5%, as required by our default setting for learned filters, i.e., the default FPR upper bound. We can observe that our g-SLBF always outperforms the other filters regarding FPR with the same storage cost. Its advantage is much more evident if we compare the storage cost under the same FPR. For example, on the Attack dataset, the FPR for g-SLBF when $B = 2^{18}$ and the FPR for SBF when $B = 2^{24}$ are approximately 0.1%. However, their storage cost ratio is $\frac{2^{18}\text{Bit}+173\text{KBits}}{2^{24}\text{Bit}} \times 100\% \approx 2.6\%$, which means that the g-SLBF requires a memory which is only 2.6% of that for SBF. For the FNR behaviors, which are shown in Figure 8d–8f, SBF, s-SLBF and g-SLBF perform similarly as B increases. The FNR performance of our learned SBF seems not significantly better than SBF. The reason is that, a good classifier can boost FPR by providing a good prediction score; however, it cannot help too much for avoiding false negatives since the negative (non-member) decisions are made only by the backup filters.

In addition, we find that the FPR of g-SLBF can be continually decreased by allocating more bits, which does not hold for s-SLBF because the classifier's FPR is the lower bound of the overall FPR for s-SLBF. This validates our argument that g-SLBF is more conservative to the classifier outputs and thus is more robust.

Varying ϵ. In this experiment, we study the effect of the FPR upper bound, ϵ, on g-SLBF. We plot the FNR of g-SLBF w.r.t. ϵ in Figure 9a, which clearly demonstrates an inverse relationship between ϵ and FNR, i.e., a higher desired FPR will decrease the number of false negatives. This is reasonable since according to Eq. (6), a higher ϵ implies a lower P and higher Max, which makes the counter harder to be decremented to 0 and thus decreases the FNR.

Varying g and B. We also look into the joint effect of the number of groups g and the total storage budget B on the performance of g-SLBF. The results are shown in Figure 9b–9d. We find that with a fixed storage budget B, increasing g leads to a V-shape curve. The reason is that, at first a more fine-grained partition can help the filter smartly insert elements to its most suitable group. But since the total storage is limited, a rather small space for each sub-filter will make counters easier to be decremented to 0, which leads to the increase of false negatives. We can see that compared with g, B has a more direct effect since more counters mean higher tolerance to the false negatives. Thus, as a suggestion of using g-SLBF, before increasing g, one should make sure the total storage is not too small, otherwise increasing B is better to reduce false negatives.

Time Efficiency. According to Table 3, our BGT based classifier can make around 10 million predictions per second. In our experiments on all three datasets, both s-SLBF and g-SLBF can perform 7 million insertions and 9 million queries per second. Note that, considering only the time efficiency, standard BF or SBF can outperform our SLBF due to their simple structures, but they are under the same order of magnitude, and SLBF is already efficient enough.

Summary of Results. In summary, we evaluate five Bloom filters on three real-world data stream applications. The results show that standard BF and LBF are not suitable for data stream tasks since their performance decays as the increasing insertions. Compared with SBF, the non-learned Bloom filter optimized for data streams, our learned index solution performs better. Specifically, under a similar error rate level, our g-SLBF can save up to 97% total storage compared to using SBF.

6. RELATED WORKS

Bloom Filters. Bloom filters are first introduced in [7] to approximately answer set membership queries. Due to the space efficiency and constant query processing time, Bloom filters are widely applied in database and networking applications, e.g., semi-join processing [8, 31], duplicate item detection [13, 15], and web cache [17]. To meet real world application requirements, different Bloom filter variants are developed, including the counting Bloom filter [17], spectral Bloom filter [11], Cuckoo filter [16], Bloomier filter [10], stable Bloom filter [13] and dynamic Bloom filter [18, 19]. Readers can find more BF variants as well as their corresponding applications in a survey paper [9].

Specifically, both SBF and dynamic BF are developed to handle the FPR decay issue over a large number of insertions (i.e., dynamically growing sets). However, different from the SBF, dynamic BF controls the FPR increase by dynamically allocating a new standard BF (using an extendable bits array as the physical storage) when the previous filter is "full" (i.e., the insertion number has reached a bound derived by the designated FPR). Compared with fixed storage of SBF as well as our LSBF, the incremental manner of dynamic BF makes its storage cost linear to the total insertions, which is not suitable for streaming applications.

Learned Indexes. In the seminal paper of Kraska et al. [25], the "learned index" was first introduced to refer to the new paradigm of index design by using a combination of machine learning models (to give quick answers like search key lookup and membership guess) and traditional data structures (to handle the corner cases that ML models cannot correctly process). Three types of learned indexes were introduced in [25], including the learned B-tree (range index), learned hash table (point index), and learned Bloom filer (set representation). The major attribute of learned indexes is their space efficiency, which comes from an observation that storing a trained model is usually less costly.

As a promising research direction, optimizations and extensions to the original learned indexes have been studied recently. Nathan et al. [32] extended the learned B-tree (1-D index) to the multi-dimensional case. Ding et al. [14] designed an updatable learned B-tree, in contrast to the fully static manner of the original version in [25]. As for learned Bloom filter, Mitzenmacher [30] first modeled the LBF mathematically and proposed a sandwiched learned filter structure where an extra initial filter is added to further optimize FPR. Rae et al. [33] proposed a meta-learning approach to obtain a better model used in learned filters. A recent preprint [12] suggested using varying numbers of hash functions considering the membership confidence values, which can be regarded as a simplified version of our g-SLBF as shown in Section 3.2. However, all existing works on learned Bloom filters, including [12], require knowing the (expected) size of element set, which can result in their poor performance when such knowledge is unavailable (e.g., data streams). To the best of our knowledge, this is the first work that has considered the streaming data insertions for learned Bloom filters.

ML in Data Management. Besides learned indexes, the emergence of ML techniques provides a more in-depth insight into the data we have and motivates a new paradigm to traditional data management problems like query cardinality/cost estimation [36, 34, 28], query optimization [27, 35], and self-tuning DBMS [24, 21]. Different from learned indexes where models are used to save storage, ML techniques, especially deep learning techniques, are widely applied to aforementioned problems due to their powerful capability of complex correlation modeling and decision making.

7. CONCLUSION

In this paper, we propose the Stable Learned Bloom filter to solve the approximate membership testing problem on data streams. SLBF extends the existing learned Bloom filter framework but conducts specific optimizations for data stream applications. The experimental studies demonstrate the effectiveness of our SLBF considering space efficiency. Nevertheless, there are still some optimization opportunities, e.g., how to update an out-of-date classifier w.r.t. new data to achieve satisfying FPR and FNR. We would like to leave them as our future work.

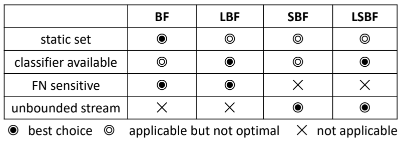
Acknowledgments

Lei Chen's work is partially supported by the Hong Kong RGC GRF Project 16209519, CRF Project C6030-18G, C10 31-18G, C5026-18G, AOE Project AoE/E-603/18, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, Didi-HKUST joint research lab project, and Wechat and Webank Research Grant.

APPENDIX

A. FILTER SELECTION CRITERIA

Though the learned Bloom filter framework has demonstrated its potential on further compression of storage, learned filters are not silver bullets for *any* scenario that requires approximate set membership query processing.



	BF	LBF	SBF	LSBF
static set	◉	◎	◎	◎
classifier available	◎	◉	◎	◉
FN sensitive	◉	◉	×	×
unbounded stream	×	×	◉	◉

◉ best choice ◎ applicable but not optimal × not applicable

Figure 12: Choice among Bloom filter variants.

We categorize the requirements to BFs from real-world applications into 4 aspects as illustrated in Figure 12. First, if the element set is static (knowing the whole set or the expected set size), standard BF is usually the best choice considering FPR/storage trade-off and implementation complexity. Second, the learned filters are powerful when a reasonable membership classifier (learned oracle) can be easily obtained. This can be explained using the inequality in Eq. (4) where LBF is better than BF when the F_p and F_n of the classifier satisfy certain conditions. This means that LBF is less appropriate in some applications where the membership prediction is extremely hard (e.g., using UUID to represent an element). Third, for applications relying on the one-sided error property (e.g., distributed join processing), SBF as well as our LSBF are not applicable due to the inevitable false negative issue. Finally, if the application cannot foresee a specific number of the element set and the total available storage to store a filter is limited, SBF and our LSBF are better choices to control the insertion-sensitive FPR with a sacrifice of possible FNR.

B. APPLICATION SCENARIOS

Consider an online malicious URL checking service which maintains a dynamically updated database of malicious links. A standard solution will be dynamically inserting the identifier of a URL to an updatable Bloom filter, i.e., the stable Bloom filter which uses limited space with a tolerable FNR. An alternative is to utilize the features like hostnames, primary domain and path tokens to build a classifier [37] to give a prior guess on the URL's, based on which deriving backup filters for making final decisions. Building a learned Bloom filer using the features to detect part of the malicious URL's can effectively reduce the storage cost since storing a classifier is usually much cheaper. Such storage reduction is especially meaningful since we can deploy such service to mobile devices like mobile phones and routers with limited RAM (hundreds of MBs to several GBs).

C. FNR CALCULATION

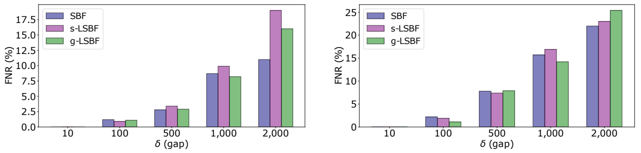
To enable the simulation of FNR of g-SLBF, we extend the model used in [13] to give the detailed calculation of $p_N(\delta_i, k_{ij})$ in Eq. (15). Denote C_i is one of the K_j counters mapped to element x_i, A_l as the event C_i is not set in recent l insertions, then

$$\begin{aligned} p_N(\delta_i, k_{ij}) =& \sum_{l=Max_j}^{\delta_i - 1} \Pr(C_i = 0|A_l) \Pr(A_l) + \\ & \Pr(C_i = 0|A_{\delta_i}) \Pr(A_{\delta_i}) \\ \Pr(C_i = 0|A_l) =& \sum_{h=Max_j}^{l} \binom{l}{h} \left(\frac{P_j}{m_j}\right)^h \left(1 - \frac{P_j}{m_j}\right)^{l-h} \qquad (21) \\ \Pr(A_l) =& k_{ij}(1 - k_{ij})^l \\ \Pr(A_{n_j}) =& (1 - k_{ij})^{\delta_i}. \end{aligned}$$

Note that, under our parameter setting strategy, once the FPR upper bound ϵ is provided, the parameter P_j can be determined by only K_j, Max_j and ϵ according to Eq. (19). Thus, $p_N(\cdot)$ can be regarded as a function of K_j, Max_j and gap δ_i. In practice, the gap value δ_i for x_i is presumed as some average gap $\tilde{\delta}$ based on the knowledge towards the insertion and query workloads.

D. EXPERIMENTS ON GAP

As we discussed in Section 3.3, the FNR of both SBF and our LSBF depends on the average gap value (δ) of the data stream, i.e., the number of timestamps between an element being inserted and queried. To investigate the effect of δ, we vary δ in range [10, 2000]. The results (Figure 13) show a clear positive correlation between FNR and δ for SBF, s-SLBF and g-SLBF. Specifically, for small value of δ (i.e., 10), the FNR is nearly 0 since $\Pr(SBF[h_j(x_i)] = 0|\delta)) \to 0$ when $\delta \to 0$ (see Eq. (14)). Such observation indicates that the false negative issue can be further alleviated if the incoming data streaming has a small gap.

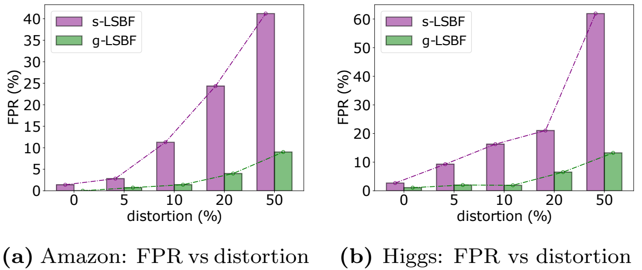


(a) Attack: FNR v.s. δ. **(b)** Higgs: FNR v.s. δ.

Figure 13: Results of varing δ (gap) on Attack and Higgs.

E. EXPERIMENTS ON ROBUSTNESS

We add experiments where the incoming data does not follow the training data distribution to show the robustness of our proposed data structures. To simulate such effect, for the stream of elements to be inserted to the filter, we randomly flip a fraction of labels (i.e., positive to negative and negative to positive). Since the training data of the classifier has not been disturbed, the label flipping fraction (from 5% to 50%) can be regarded as a measurement of the deviation between the incoming data distribution and training data distribution. The evaluation results are shown in Figure 14.



(a) Amazon: FPR vs distortion **(b)** Higgs: FPR vs distortion

Figure 14: Results of FPR vs different levels of distribution distortion on Amazon and Higgs.

8. REFERENCES

[1] xxhash. `http://cyan4973.github.io/xxHash/`. [Online; accessed 28-Feb-2020].

[2] Amazon.com - Employee Access Challenge. `https://www.kaggle.com/c/amazon-employee-access-challenge/data`, 2020. [Online; accessed 2-July-2020].

[3] CatBoost - open-source gradient boosting library. `https://catboost.ai/`, 2020. [Online; accessed 2-July-2020].

[4] UCI Machine Learning Repository: HIGGS Data Set. `https://archive.ics.uci.edu/ml/datasets/HIGGS`, 2020. [Online; accessed 2-July-2020].

[5] UCI Machine Learning Repository: Kitsune Network Attack Dataset Data Set. `https://archive.ics.uci.edu/ml/datasets/Kitsune+Network+Attack+Dataset`, 2020. [Online; accessed 2-July-2020].

[6] P. S. Almeida, C. Baquero, N. M. Preguiça, and D. Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.

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

[8] K. Bratbergsengen. Hashing methods and relational algebra operations. In *VLDB*, pages 323–333. Morgan Kaufmann, 1984.

[9] A. Z. Broder and M. Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[10] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA*, pages 30–39. SIAM, 2004.

[11] S. Cohen and Y. Matias. Spectral bloom filters. In *SIGMOD Conference*, pages 241–252. ACM, 2003.

[12] Z. Dai and A. Shrivastava. Adaptive learned bloom filter (Ada-BF): Efficient utilization of the classifier. *CoRR*, abs/1910.09131, 2019.

[13] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *SIGMOD Conference*, pages 25–36. ACM, 2006.

[14] J. Ding, U. F. Minhas, H. Zhang, Y. Li, C. Wang, B. Chandramouli, J. Gehrke, D. Kossmann, and D. B. Lomet. ALEX: an updatable adaptive learned index. *CoRR*, abs/1905.08898, 2019.

[15] S. Dutta, A. Narang, and S. K. Bera. Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams. *PVLDB*, 6(8):589–600, 2013.

[16] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT*, pages 75–88. ACM, 2014.

[17] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[18] D. Guo, J. Wu, H. Chen, and X. Luo. Theory and network applications of dynamic bloom filters. In *INFOCOM*. IEEE, 2006.

[19] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. The dynamic bloom filters. *IEEE Trans. Knowl. Data Eng.*, 22(1):120–133, 2010.

[20] G. H. Hardy, J. E. Littlewood, G. Pólya, D. Littlewood, G. Pólya, et al. *Inequalities*. Cambridge university press, 1952.

[21] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*. www.cidrdb.org, 2019.

[22] N. Jain, M. Dahlin, and R. Tewari. Using bloom filters to refine web search results. In *WebDB*, pages 25–30, 2005.

[23] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.

[24] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. In *CIDR*. www.cidrdb.org, 2019.

[25] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD Conference*, pages 489–504. ACM, 2018.

[26] A. Kumar, J. Xu, and J. Wang. Space-code bloom filter for efficient per-flow traffic measurement. *IEEE J. Sel. Areas Commun.*, 24(12):2327–2339, 2006.

[27] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[28] R. C. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *PVLDB*, 12(11):1733–1746, 2019.

[29] A. Metwally, D. Agrawal, and A. El Abbadi. Duplicate detection in click streams. In *WWW*, pages 12–21. ACM, 2005.

[30] M. Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In *NeurIPS*, pages 462–471, 2018.

[31] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Software Eng.*, 16(5):558–560, 1990.

[32] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. *CoRR*, abs/1912.01668, 2019.

[33] J. W. Rae, S. Bartunov, and T. P. Lillicrap. Meta-learning neural bloom filters. In *ICML*, volume 97, pages 5271–5280. PMLR, 2019.

[34] J. Sun and G. Li. An end-to-end learning-based cost estimator. *PVLDB*, 13(3):307–319, 2019.

[35] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3):210–222, 2018.

[36] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, P. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *PVLDB*, 13(3):279–292, 2019.

[37] P. Zhao and S. C. H. Hoi. Cost-sensitive online active learning with application to malicious URL detection. In *KDD*, pages 919–927. ACM, 2013.