New Wine in an Old Bottle: Data-Aware Hash Functions for Bloom Filters

Arindam Bhattacharya
CSE, IIT Delhi, India
arindam@cse.iitd.ac.in

Chathur Gudesa
EE, IIT Delhi, India
ee1180459@iitd.ac.in

Amitabha Bagchi
CSE, IIT Delhi, India
bagchi@cse.iitd.ac.in

Srikanta Bedathur
CSE, IIT Delhi, India
srikanta@cse.iitd.ac.in

ABSTRACT

In many applications of Bloom filters, it is possible to exploit the patterns present in the inserted and non-inserted keys to achieve more compression than the standard Bloom filter. A new class of Bloom filters called Learned Bloom filters use machine learning models to exploit these patterns in the data. In practice, these methods and their variants raise many questions: the choice of machine learning models, the training paradigm to achieve the desired results, the choice of thresholds, the number of partitions in case multiple partitions are used, and other such design decisions. In this paper, we present a simple partitioned Bloom filter that works as follows: we partition the Bloom filter into segments, each of which uses a simple projection-based hash function computed using the data. We also provide a theoretical analysis that provides a principled way to select the design parameters of our method: number of hash functions and number of bits per partition. We perform empirical evaluations of our methods on various real-world datasets spanning several applications. We show that it can achieve an improvement in false positive rates of up to two orders of magnitude over standard Bloom filters for the same memory usage, and upto 50% better compression (bytes used per key) for same FPR, and, consistently beats the existing variants of learned Bloom filters.

PVLDB Reference Format:
Arindam Bhattacharya, Chathur Gudesa, Amitabha Bagchi, and Srikanta Bedathur. New Wine in an Old Bottle: Data-Aware Hash Functions for Bloom Filters. PVLDB, 15(9): 1924 - 1936, 2022.
doi:10.14778/3538598.3538613

1 INTRODUCTION

In many applications of Bloom filters there is an opportunity to exploit the patterns in the inserted set of keys that distinguishes them from non-inserted keys, so that the performance of the filter can be improved. Kraska et al. [16] proposed learned Bloom filters which try to exploit this pattern using a machine learning model. Given a set of keys X to be inserted and a set of keys Y representing non-inserted keys, a learned Bloom filter uses a binary classifier trained on X and Y to answer set membership queries based on a fixed threshold. They additionally have a backup Bloom filter to ensure the structure returns no false negatives. These learned Bloom filters achieve more compression than standard Bloom filters since standard Bloom filters assume no information about patterns in the data. Several improvements over this method has been suggested since then, such as pre-filtering using a Bloom filter [25], adapting it for incremental workloads [4, 19], and partitioning the Bloom filter using the model scores [9, 27].

While learned Bloom filters have shown significant performance improvements over standard Bloom filters, they also have several drawbacks:

- The need of a separate machine learning classifier requires major implementation level changes to be made in order to replace standard Bloom filters with their learned counterparts in applications such as Apache Cassandra [7]. Apart from software changes, depending on the classifier employed, additional hardware such as GPUs/TPUs may be required in order to efficiently train the machine learning model.
- They require several design choices (in addition to Bloom filter configuration choices) such as the suitable machine learning model, thresholds and partitioning parameters, and so on, to be made before their deployment.
- More importantly, out of the box machine learning models are optimized for better generalization over unseen data rather than for the task of replacing index structures. As a result, the model size required to achieve requisite performance in a learned Bloom filter is often quite large. Consider the use of a simple neural network (NN) for MNIST dataset of handwritten digits each of which is represented as a 28×28 image. A reasonable NN would have 784 input neurons, and 10 output neurons –one for each digit class– and a two hidden layers of (say) 16 neurons each. This would result in nearly $13,000$ floating point values for the entire model. A random forest classifier, which has been shown to be a better fit for the task [27], also requires a relatively large model (empirically we found it be close to 92 KB – see Section 5).

In this paper, we propose *Projection Hash Bloom Filter (PHBF)*, a novel learning based method that works seamlessly with standard Bloom filters by simply replacing standard hash functions (such as Murmur3), with *efficient* and *data-aware* hash functions. The bit-vector of the Bloom filter is partitioned into equally sized regions and each region is associated with a hash function. Each hash function projects keys to a selected set of d-dimensional unit vectors chosen so as to separate the sets X and Y well. Note that our data-aware hash functions are designed keeping in mind the need for them to be computationally fast and lightweight in size. While this method fails to generalize for use in broader machine learning applications, it can distinguish the inserted keys from the non-keys effectively.

We theoretically show that in order to maintain a desired false positive rate (FPR) of the Bloom filter, our data-aware hash functions require memory that is proportional to the dimensionality of the data rather than the number of keys inserted into the filter (under

Proceedings of the VLDB Endowment, Vol. 15, No. 9 ISSN 2150-8097.
doi:10.14778/3538598.3538613

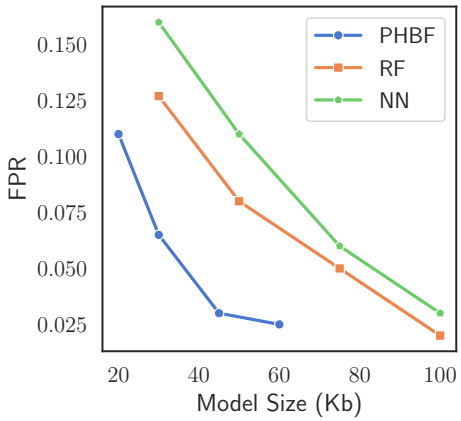

Figure 1: Performance of various models on Malicious URLs.

reasonable and natural assumptions regarding the separability of keys in X and Y – details in Section 4).

Finally, we experimentally demonstrate that for a given a fixed space usage (bytes used by the index per key) in the range less than 0.2, that is when the index 0.2 bytes of storage per key, PHBF outperform standard and learned variants of Bloom filters significantly in terms of FPR. This range of space usage covers many practical applications ([11, 21]). For lower required FPR, the learned variants start to gain advantage over our simple model, at the cost of higher space usage. Figure 1 shows the relative memory requirement of neural networks, random forests and our approach for detecting Malicious URLs.

It is worth noting that the only parameters that are needed in a PHBF deployment are the number of hash functions, bits used by the Bloom filter bit-vector and the number of random vectors to be sampled – just one more than those of standard Bloom filters. The choice of each of these parameters is guided by theoretical analysis.

1.1 Contributions

In this paper, we present a novel learned Bloom filter called Projection Hash Bloom Filter (PHBF), and make the following key contributions:

(1) We describe Projection Hash Bloom Filter (PHBF), a space-efficient partitioned Bloom filter which uses random projection based data-aware hash functions. Unlike other learned Bloom filters, the design parameters of PHBF are the number of hash functions, bits used per partition, and number of vectors to be sampled. Selection of each of these are guided by theoretical analysis. This makes our method an easy plug-in replacement for standard Bloom filters (Section 3).

(2) We provide a theoretical analysis bounding the false positive rate in terms of the properties of the sets X and Y. We analyze the bounds for three increasingly restrictive scenarios: (i) data with finite mean and variance, (ii) data with independent features where each feature is drawn from a bounded range, and finally, (iii) data drawn from Gaussian distribution. We show that, assuming reasonable separation, the memory requirement no longer scales with number of elements inserted but with the dimensionality of the keys. Unlike other learned Bloom filter approaches, this gives us an analytical basis to select the design parameters (Section 4).

(3) We perform extensive experiments to empirically compare PHBF with other learned Bloom filter variants on a range of real world datasets spanning a wide range of applications. We show that PHBF can provide up to two orders of magnitude better FPR for a given memory usage than the standard Bloom filter. It also provides similar space usage for a given FPR as compared to state of the art learned Bloom filter models while requiring an order of magnitude less time for construction and queries. Section 5).

2 RELATED WORK

A standard Bloom filter (SBF) is a memory-efficient data structure that can check for the existence of an element in a set. It provides a trade-off between memory efficiency and the false positive rate. For a Bloom filter designed to store n elements while achieving a target false positive rate (FPR) of ε, we need

$$m = -\frac{n \ln \varepsilon}{(\ln 2)^2}$$

bits. We note that the number of bits required for maintaining a certain FPR target increases linearly with the number of elements inserted. This makes SBF infeasible for very large sets.

There have been investigations into making standard Bloom filters more memory efficient, but at best, they provide a constant factor improvement while still scaling linearly with n. The method proposed in [6], which can be extended to standard Bloom filters, uses *d-left* hashing, a hashing technique which uses d hash tables. While this approach scales linearly with the number of inserted elements as well, it reduces the size of the Bloom filter by a factor of around 2.5 for a target false positive rate of the order of 10^{-4}. Compressed Bloom filters [24] achieve a significant compression over other Bloom filters, making them easier to transmit. However, they still require decompression in memory to use and therefore do not address the problem of memory requirement. Another approach proposed in [29] to make standard Bloom filters more efficient is to control the false positive rate by selecting hash functions that produce the least number of false positives. It employs a two-step process to first check which hash functions work best given the inserted and query sets, and then to use the selected hash functions to insert elements.

Chang et al. [8] propose a partitioned Bloom filter that works by partitioning the M sized bit array into k slices of size $m = M/k$ bits using k hash functions. Each hash function produces an index over m for its respective slice. Thus, each element is described by exactly k bits, ensuring that the distribution of false positives is uniform across all elements. The method we propose in this paper is a variant of Chang et al.'s method, where each of the k hash functions are *learned* from the dataset.

Distance sensitive hashes, such as locality sensitive hashing (LSH) [13] aim to map similar keys to the same hash value, thus making it more likely that a collision of keys implies similarity of keys based on some metric. These methods are designed to find approximate nearest neighbors and are not a suitable replacement for learned Bloom filters that learn the patterns specific to the inserted and queried keys.

None of the improvements mentioned so far attempt to find and exploit any pattern in the data. But if the keys inserted into the Bloom filter display some predictable pattern, new avenues for memory efficiency open up. One approach is to replace the Bloom filter with a machine learning model [16]. Other Bloom filter variants have been proposed [4, 9] based on this approach. The false positives for such learned Bloom filters (LBF) depend on the false positive rates of the models. The model may become complex and large depending on the target false positive rate requirement. A backup Bloom filter of a much smaller size is also required to ensure that LBF has no false negatives. A partitioned learned Bloom filter [27] proposes an alternate usage of the model, using it to partition the key space optimally. The idea is to formulate the problem into a two-part optimization problem: (i) how to best partition the scores from the model into a given number of regions, and (ii) how to choose thresholds for the regions to minimize the overall space consumption of Bloom filters. The authors use dynamic programming and dual formulation to solve these optimization problems respectively.

Another approach to exploit patterns in keys could be to use specialized hash functions that are learned from the data. Data dependent hashes utilize the distribution of the data to *learn* the hash function. These methods include unsupervised approaches based on simple optimization such as [14] and [28] that learn a descriptor for the data and are independent of the label of the data. Data dependent hashing can also be supervised, ranging from simple models to generate hash codes [20] to complex supervised deep models [18]. Using a supervised deep model for hashing was also suggested in [16]. Data dependent approaches consist of a training phase that adds to the construction time but are capable of learning patterns in the data, allowing it to reach significantly lower false positive rates for a given size. Our approach can be categorized as a data dependent approach learned Bloom filters, but unlike the machine learning based models, our random projection based methods produce a smaller model, thus achieving a better space usage in terms of bytes of memory used per key, which providing a principled approach to select design parameters.

3 PROPOSED METHOD

In this section we propose a method for constructing a partitioned Bloom filter with learned hashing called Projection Hash Bloom Filter (PHBF). We describe our method for the same setting as that of the other learned Bloom filter works [4, 16, 29]: we are given the set of *positive* keys that are to be inserted, and a set of *negative* keys sampled from a distribution generating negative query points. We also contrast this with a few closely related methods.

3.1 Projection Hash Bloom Filter

Projection Hash Bloom Filter (PHBF) is based on the idea that the keys can be projected on to a lower dimensional subspace and that these projections can be used to distinguish between the keys inserted into the structure from non-keys. Unlike other applications of random projections that require distance preservation [1, 13], PHBF works with fewer vectors, since preservation of distances is not a required criterion for the application.

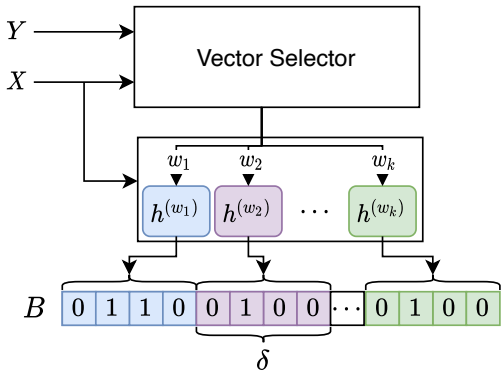

Figure 2: Architecture of PHBF. Given X and Y, Vector Selector selects k random vectors. These are used to compute the k hash functions, each populating their own partition of size δ in B.

PHBF works in two phases. Figure 2 shows the architecture of PHBF. The first phase is the vector selection phase. In this phase, we sample some vectors and use the sets X and Y to select $w_1, \ldots, w_k$, the k best vectors among them. We explain what this means mathematically in following paragraphs, but simply put, the selection phase can be seen as *training* phase, where we use X and Y as positive and negative training examples to train k weak δ-class classifiers, each returning a label from $\{1, \ldots, \delta\}$. In the second phase we populate the bit array. Figure 3 demonstrates the working of this phase. The set X is projected on to the vectors selected in the previous phase (w_1 and w_2 in the example shown in the figure), each of which acts as a hash function mapping the points to bins labelled $\{1, \ldots, \delta\}$. The bins in which the projected points fall are set to 1. Each of these hash functions then populate a disjoint segment of the bit array B, which avoids inter-hash collisions. During a query, the point is evaluated by each of the hash functions and is assigned a bin. If all of the assigned bins are set to 1, PHBF returns *true*, that is, the queried point has been inserted into the index.

This construction is different from distance sensitive Bloom filters (DSBF) proposed in [15], which are designed for the task of identifying approximate nearest neighbors. Since DSBF does not train using the data, it fails to capture pattern specific to the data. In Section 5, we demonstrate the unsuitability of this method for the task of set membership queries.

We now formally define PHBF. Assume that we are given a set of d dimensional keys $X \subseteq \mathbb{R}^d$ to be inserted into a Bloom filter, and a set of keys Y sampled from $\mathbb{R}^d \setminus X$. Then, given the size of bit array, m, and number of partitions, k, a PHBF consists of the following elements:

(1) a bit array $B[1, \ldots, m]$ of size m, partitioned in k segments of size $\delta = m/k$,

(2) projection vectors w_i, for $i \in \{1, \ldots, k\}$, chosen from the space of d dimensional unit vectors, $\mathbf{1}_d$, and,

(3) k hash functions $h^{(w_i)} : \mathbb{R}^d \to \mathcal{I}^{(i)}$ that maps $\mathbb{R}^d$ to $\mathcal{I}^{(i)} = [(i-1)\delta, i\delta - 1]$.

We now provide a detailed explanation of the two phases of construction of PHBF.

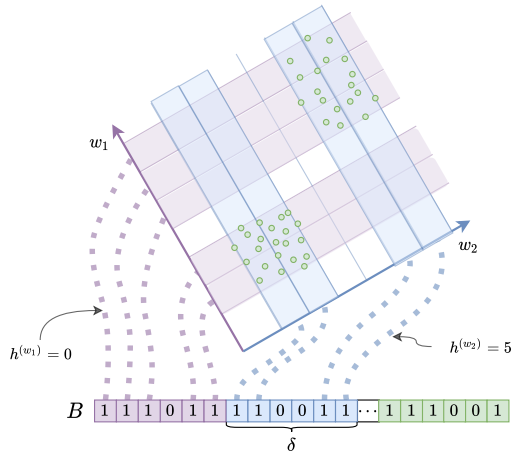

Figure 3: Hash computation and Bit Array population. The set X is represented by the green points. The figure shows 2 out of k random vectors. X is projected on to w_1 and w_2. The range of the projection is divided into δ bins, and the bins occupied by $x \in X$ is set as 1 in B.

Projection Vector Selection. Given a sampling factor s, we first sample $s \cdot k$ vectors $v_1, \ldots, v_{sk}$ uniformly at random from the space of d dimensional unit vectors, $\mathbf{1}_d$. Then we compute the hash values of the elements of X and Y for each of the sk hash functions. The method of computation will be explained in the next paragraph. Once we have the hash values, we compute the number of positions where there is at least one collision between the elements of X and Y. For each vector v_j, $j \in \{1, \ldots, sk\}$,

$$c_j = \sum_{i=1}^{m} \mathbf{1}\{E_{i,j}\}$$

where $E_{i,j} := \exists x \in X, y \in Y,\ s.t.\ h^{(v_j)}(x) = h^{(v_j)}(y) = i$, and m is the size of the bit array.

Then, the k projection vectors, $w_1, \ldots, w_k$, are selected as:

$$w_i = v_j : rank(c_j) = i,$$

for $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, sk\}$, where $rank(c_j)$ gives the position of c_j in the sequence $\{c_1, \ldots, c_{sk}\}$ sorted in ascending order. Algorithm 2 (SelectVectors) shows the details of vector selection phase.

Populating the bit array. Once we have selected the projection vectors, we shift and scale the projections to the range $[0, 1]$, and compute the hash values for all $x \in X$ as follows:

$$h^{(w_i)}(x) = \left\lfloor \frac{|\langle w, x \rangle|}{||x||} \cdot \delta \right\rfloor. \tag{1}$$

Then, for all $x \in X$ and $i \in \{1, \ldots, k\}$, we populate an empty bit array B of size m as:

$$B[(i-1)\delta + h^{(w_i)}(x)] = 1.$$

Once PHBF is constructed, it is queried as follows. Given a key $z \in \mathbb{R}^d$, we compute the k hash function $h^{(w_1)}, \ldots, h^{(w_k)}$. Then, PHBF returns *true* to the question *is* $z \in X$ only if $B[(i-1)\delta + h^{(w_i)}]$ is 1 for all $i \in \{1, \ldots, k\}$. Algorithm 3 shows the steps when querying PHBF.

Algorithm 1: Construction of PHBF

Data: X, Y
input : m, k, s
output: PHBF

```
Initialize B[0, ..., m − 1] to 0
w_1, ..., w_k ← SelectVectors(X, Y, s, m, k)
δ ← m/k
foreach x ∈ X do
    for i ← 1 to k do
        // Compute hash using Equation 1
        loc ← (i − 1)δ + h^(w_i)(x)
        B[loc] ← 1
PHBF ← (w_1, ..., w_k, B)
return PHBF
```

Algorithm 2: SelectVectors subroutine

Data: X, Y
input : m, k, s
output: $w_1, \ldots, w_k$

```
for j ← 1 to sk do
    v_j ~ Uniform(1_d)
    c_j ← 0
    for i ← 1 to m do
        if ∃x ∈ X, y ∈ Y, s.t. h^(v_j)(x) = h^(v_j)(y) = i then
            c_j = c_j + 1
for i ← 1 to k do
    w_i ← v_j : rank(c_j) = i
return w_1, ..., w_k
```

Algorithm 3: Query on PHBF

input : PHBF, z
output: *true* if likely that $z \in X$. Else *false*.

```
δ ← m/k
for i ← 1 to k do
    // Compute hash using Equation 1
    loc ← (i − 1)δ + h_X^(w_i)(x)
    if B[loc] = 0 then
        return false
return true
```

Time complexity. PHBF is extremely efficient. For d dimensional keys, the time to compute a single hash value is $\Theta(d)$, which is same as a standard uniform hash function such as Murmur3 on a byte string key of length d. The total construction phase takes $\Theta((s+1)nkd)$ time, compared to $\Theta(nkd)$ of SBF. The query time,

$\Theta(kd)$, is same as the query time of SBF. In addition to this, the disjoint regions and the independence of hash functions allow simple parallelization of both the construction and the query process. The process of discretization and parallelization, and the advantages gained from them are discussed in detail in [5].

Space complexity. The total memory consumption of PHBF is the size of the bit array B and the total size of the hash functions. Each hash function $h^{(w_i)}$ is defined by $w_i \in \mathbb{R}^d$, and takes $\Theta(d)$ space. The total space consumed by PHBF then is $\Theta(kd + m)$.

4 THEORETICAL BOUNDS ON FPR

In this section, we analyze the behavior of PHBF when the inserted set of points X and the queried set of points Y are drawn from particular distributions. We will denote these distributions as $\mathcal{D}^+$ and $\mathcal{D}^-$ respectively, since for FPR calculation we are assuming that the query point is from outside the distribution of the inserted points. We will assume we know the mean and variance of both these distributions, denoted μ_X and μ_Y respectively for $\mathcal{D}^+$ and $\mathcal{D}^-$. We will analyze the FPR for three different assumptions of what we know about these distributions.

- $\mathbf{A}_1$: We know the variances, σ_X^2 and σ_Y^2 of $\mathcal{D}^+$ and $\mathcal{D}^-$ and we know nothing else.
- $\mathbf{A}_2$: $\mathcal{D}^+$ and $\mathcal{D}^-$ are d-variate distributions such that for $X \sim \mathcal{D}^+ = x_1, \ldots, x_d$, $a_i^+ \le x_i \le b_i^+$ and $Y \sim \mathcal{D}^- = y_1, \ldots, y_d$, $a_i^- \le y_i \le b_i^-$ for $1 \le i \le d$ and, the dimensions are independent of each other.
- $\mathbf{A}_3$: $\mathcal{D}^+ = \mathcal{N}(\mu_X, \sigma_X^2)$ and $\mathcal{D}^- = \mathcal{N}(\mu_Y, \sigma_Y^2)$.

We show that if the distance between the centers of the distributions ℓ increases with the number of inserted elements n, we need to neither increase the size of the Bloom filter nor increase the number of selections in 2 (SelectVectors subroutine) in order to maintain some given FPR. This is unlike the traditional Bloom filter which requires the size m to scale linearly with the number of inserted elements to maintain a given FPR. Table 1 summarizes the notations used in this section.

Note that we assume that $\delta = 1$ throughout this section in order to simplify our analysis. Because we shift and scale our projections to the range $[0, 1]$, using $\delta = 1$ means that every point y which satisfies $\min_X h^{(w_i)}(x) \le h^{(w_i)}(y) \le \max_X h^{(w_i)}(x)$ is considered to be a positive by the partition corresponding to hash function $h^{(w_i)}$.

Theorem 4.1. *Let $X \subset \mathbb{R}^d$ be a set of n points that were drawn independently at random from a distribution $\mathcal{D}^+$ with mean μ_X and are inserted into an PHBF with parameter $\delta = 1$. Let Y be a set of query points drawn from $\mathcal{D}^-$ with mean μ_X such that $|\mu_X - \mu_Y| = \ell$. We can then say for $i \in \{1, 2, 3\}$ that if the distributions follow assumption $\mathbf{A}_i$ then we can achieve expected FPR ε using $k = -c_{i,1}\ln(\varepsilon) + c_{i,2}$ hash functions chosen by sampling $-c_{i,3}d\ln(\varepsilon) + c_{i,4}d$ vectors in our SelectVectors subroutine, where $c_{i,1}, c_{i,2}, c_{i,3}$ and $c_{i,4}$ are some constants, under the conditions that*

(1) $\mathbf{A}_1$: There exists a constant C_1 such that $\ell > C_1 d\sqrt{n}$.

(2) $\mathbf{A}_2$: There exists a constant C_2 such that $\ell > C_2 d \sum_{i=1}^{d}(b_i - a_i)^2 \ln(n)$.

(3) $\mathbf{A}_3$: There exists a constant C_3 such that $\ell > C_3 d \ln(n)$.

Table 1: Summary of notations used

Notation	Definition
X	Set of keys to be inserted, drawn from a distribution with mean μ_X
Y	Query points, drawn from a distribution with mean μ_Y
d	Dimensions of inserted and queried data
ℓ	Distance between μ_X and μ_Y
n	Size of X
m	Size of bit array
k	Number of hash functions
s	Sampling factor; we sample a total of sk vectors
ε	False positive rate
δ	Bits assigned per partition (m/k)
w	A sampled vector
$h^{(w_i)}$	Hash function which uses vector w_i for computing projections
θ	Angle between w and $\mu_X - \mu_Y$
$\phi(x)$	$\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$
$\Phi(x)$	$\int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$

In the case of $\mathbf{A}_2$ if we assume that the b_is and a_i are all equal and independent of d, the condition on ℓ becomes $\ell > C_4 d^2 \ln(n)$. Note that as we make stronger assumptions, the separation between the positive and negative keys that is required to achieve a target FPR ε when using $k = O(\ln(\frac{1}{\varepsilon}))$ hash functions decreases.

Before we proceed with our proof, we make the following two calculations. Calculations 4.2, 4.3, and, 4.4 bounds the expected FPR of a PHBF with parameters $k = 1$ and $\delta = 1$ for each of the three cases mentioned above assuming that the vector w used by the hash function forms an angle θ with the vector $\mu_X - \mu_Y$. Calculation 4.5 computes the probability that a randomly selected vector forms a small angle with the vector $\mu_X - \mu_Y$.

Calculation 4.2 (FPR bound for assumption $\mathbf{A}_1$). *The expected FPR ε of a PHBF with parameters $\delta = 1$ and $k = 1$ can be bounded as*

$$\varepsilon \le \frac{4nd(\sigma_X^2 + \sigma_Y^2)}{\ell^2 \cos^2\theta}$$

where θ is the angle between the vector $\mu_X - \mu_Y$ and the unit vector w used by the hash function.

Proof. Without loss of generality, we can shift our coordinates such that $\mu_X = \mathbf{0}$. After projecting onto the unit vector w associated with hash function $h^{(w)}$, the distance between the centers is $\ell' = ||\mu_Y|| \frac{\langle \mu_Y, w \rangle}{||\mu_Y||} = \ell \cos\theta$, where θ is the angle between the vectors μ_Y and w. Let $\hat{x}_m = \max_{x \in X} \langle x, w \rangle$ and let $\hat{y} = \langle y, w \rangle$.

Then, the FPR ε can be bounded as

$$\begin{aligned} \varepsilon &\le \int_{-\infty}^{\infty} \Pr\{\hat{x}_m = r\} \Pr\{\hat{y} \le r\}\, dr \qquad (2) \\ &\le \int_{-\infty}^{r_0} np_X(r) \Pr\{y \ge (\ell - r_0)\} + \int_{r_0}^{\infty} np_X(r) \Pr\{y \ge (\ell - r)\} \end{aligned}$$

For $r_0 = \frac{\ell}{2}$, and using Chebyshev's inequality to bound the RHS, we get:

$$\varepsilon \le \frac{4n(\sigma_X^2 + \sigma_Y^2)}{\ell^2 \cos^2\theta}$$

□

Calculation 4.3 (FPR bound for assumption $\mathbf{A}_2$). *The expected FPR ε of a PHBF with parameters $\delta = 1$ and $k = 1$ can be bounded as*

$$\varepsilon \leq \frac{4n\sqrt{\pi}}{2} e^{-\frac{\ell^2 \cos^2\theta}{cd}}$$

where θ is the angle between the vector $\mu_X - \mu_Y$ and the unit vector w used by the hash function, and $c \geq (b_i - a_i)^2$ for all $i \in [1, d]$ is some constant.

Proof. Then, the FPR ε can be bounded as

$$\varepsilon \leq \int_{-\infty}^{\infty} \Pr\{\hat{x}_m = r\} \Pr\{\hat{y} \leq r\}\, dr \tag{3}$$

Using Hoeffding's lemma to bound the RHS, we get:

$$\varepsilon \leq \frac{4n\sqrt{\pi}}{2} e^{-\frac{\ell^2 \cos^2\theta}{\sum_{i=1}^{d}(b_i - a_i)^2}}.$$

Let $(b_i - a_i)^2$ be upper bounded by the constant c. Replacing it in the equation gives us the result. □

Calculation 4.4 (FPR bound for assumption $\mathbf{A}_3$). *The expected FPR ε of a PHBF with parameters $\delta = 1$ and $k = 1$ can be bounded as*

$$\varepsilon \leq \frac{n}{\ell \cos\theta} \sqrt{\frac{\sigma_X^2 + \sigma_Y^2}{2\pi}} e^{-\frac{\ell^2 \cos^2\theta}{2(\sigma_X^2 + \sigma_Y^2)}}$$

where θ is the angle between the vector $\mu_X - \mu_Y$ and the unit vector w used by the hash function.

Proof. We denote $\Phi(x)$ as the Gaussian distribution function and $\phi(x)$ as the Gaussian density function.

Then, the FPR ε can be bounded as

$$\varepsilon \leq \int_{-\infty}^{\infty} \Pr\{\hat{x}_m = r\} \Pr\{\hat{y} \leq r\}\, dr \tag{4}$$

$$\leq \frac{n}{\ell \cos\theta} \sqrt{\frac{\sigma_X^2 + \sigma_Y^2}{2\pi}} e^{-\frac{\ell^2 \cos^2\theta}{2(\sigma_X^2 + \sigma_Y^2)}} \tag{5}$$

We used the result $\int_{-\infty}^{\infty} \phi(x)\Phi(a + bx)dx = \Phi\left(\frac{a}{\sqrt{1+b^2}}\right)$ and Mill's inequality which states that $1 - \Phi(x) \leq \frac{1}{x\sqrt{2\pi}} e^{-\frac{x^2}{2}}$ to get the above result. □

The result of the calculations above shows the usual increase in FPR that when n is increased can be offset by increasing the distance ℓ between the inserted and queried distributions.

Calculation 4.5. *Let $v \in \mathbb{R}^d$ be a given vector. Suppose we sample a set of s vectors $\{w_1, \ldots, w_s\}$ uniformly at random from $\mathbf{1}_d$. If θ_i is the angle formed between w_i and v, then*

(1) with a probability at least $1 - f_1(d, l)^s$, we will have

$$\max_i \{\cos(\theta_i)\} \geq \frac{1}{\sqrt{d\ell}}$$

, where $f_1(d, l) = \left(1 - \frac{1}{d}\left(1 - \frac{1}{d\ell}\right)^{\frac{d}{2}}\right)$, and

(2) with a probability at least $1 - f_2(d, l)^s$, we will have

$$\max_i \{\cos(\theta_i)\} \geq \sqrt{\frac{d}{\ell}}$$

, where $f_2(d, l) = \left(1 - \frac{1}{d}\left(1 - \frac{d}{\ell}\right)^{\frac{d}{2}}\right)$.

Proof. If w is a unit vector randomly sampled from $\mathbf{1}_d$, then the distribution of $w^{(1)}$, the first dimension of w, can be given by $f_{w^{(1)}}(w) = \frac{(1-x^2)^{\frac{d}{2}-1}}{B(\frac{d}{2}, \frac{1}{2})}$, where $B(p, q) = \int_0^1 t^{(p-1)}(1-t)^{(q-1)}$ is the Beta function. If we rotate our axis such that the vector $\mu_X - \mu_Y$ is aligned along the first dimension, then note that $w_1 = \cos\theta$, where θ is the angle between the vectors w and $\mu_X - \mu_Y$. Therefore,

(1) when $\max_i\{\cos(\theta_i)\}$,

$$\mathcal{I} = \Pr\left\{\cos\theta \geq \frac{1}{\sqrt{d\ell}}\right\} = 2\int_{\frac{1}{\sqrt{d\ell}}}^{1} \frac{(1-x^2)^{\frac{d}{2}-1}}{B(\frac{d}{2}, \frac{1}{2})} dx$$

We then use the fact that $B(\frac{d}{2}, \frac{1}{2}) \leq 2$ whenever $d \geq 2$ to get

$$\mathcal{I} \geq \int_{\frac{1}{\sqrt{d\ell}}}^{1} (1-x^2)^{\frac{d}{2}-1} dx \geq \int_{\frac{1}{\sqrt{d\ell}}}^{1} x(1-x^2)^{\frac{d}{2}-1} dx = \frac{1}{d}\left(1 - \frac{1}{d\ell}\right)^{\frac{d}{2}}.$$

If we sample s vectors, then the probability that

$$\max_i \{\cos(\theta_i)\} \geq \frac{1}{\sqrt{d\ell}}$$

is at least $1 - f_1(d, l)^s$.

(2) And when $\max_i\{\cos(\theta_i)\} \geq \sqrt{\frac{d}{\ell}}$,

$$\mathcal{I} \geq \int_{\sqrt{\frac{d}{\ell}}}^{1} x(1-x^2)^{\frac{d}{2}-1} dx = \frac{1}{d}\left(1 - \frac{d}{\ell}\right)^{\frac{d}{2}}.$$

If we sample s vectors, then the probability that

$$\max_i \{\cos(\theta_i)\} \geq \sqrt{\frac{d}{\ell}}$$

is at least $1 - f_2(d, l)^s$.

□

The result of calculation 4.5 shows that the probability that a random vector will form a small angle with $\mu_X - \mu_Y$ increases rapidly as s increases. Equipped with the results from the preceding calculations, we proceed with the proof our theorem as follows.

Proof of Theorem 4.1. Suppose we use k_0 hash functions for our Bloom filter. We may treat this PHBF as k_0 independent PHBFs with parameters $\delta = 1$ and $k = 1$ and sample s_0 vectors per hash function, where $s_0 = 2d\left(\ln(k_0) + \ln\left(\frac{2}{\varepsilon}\right)\right)$. We show the bound for each case below:

(1) $\mathbf{A}_1$. Using our results from calculations 4.2 and 4.5, we can say that with a probability at least $1 - f_2(d, l)^{s_0}$, the expected FPR of the ith filter is upper bounded as $\varepsilon^{(i)} \leq \frac{4n(\sigma_X^2 + \sigma_Y^2)}{\ell^2}$. Since each of the k_0 filters would behave independently of each other, the expected FPR of the overall PHBF is upper bounded as $\left(\frac{4nd(\sigma_X^2 + \sigma_Y^2)}{\ell}\right)^{k_0} = \frac{\varepsilon}{2}$ with a probability at least $(1 - f_2(d, l)^{s_0})^{k_0} \geq 1 - k_0 e^{\frac{s_0}{2d}} = 1 - \frac{\varepsilon}{2}$.

Also, with a probability at most $\frac{\varepsilon}{2}$, the expected FPR of the PHBF is trivially upper bounded by 1. It follows from this that the expected FPR is at most $(1-\frac{\varepsilon}{2})\frac{\varepsilon}{2}+\frac{\varepsilon}{2} \leq \varepsilon$.
We have therefore shown that if we use k_0 hash functions and select s_0 vectors in our SelectVectors subroutine, we can achieve an expected FPR of ε.
We can now proceed to show the bounds on k_0 and s_0. We may write $k_0 \leq -\frac{\ln\frac{\varepsilon}{2}}{\ln l - \ln nd - (\sigma_X^2+\sigma_Y^2)}$. Since there exists a constant C such that $\ell > Cd\sqrt{n}$, we may further write $k_0 \leq -c_1\ln(\varepsilon)+c_2$, for some constants c_1 and c_2.
Similarly, we can bound $s = 2d\left(\ln(k)+\ln\left(\frac{2}{\varepsilon}\right)\right) \leq -c_3 d\ln(\varepsilon)+c_4 d$ for some constants c_3 and c_4.

(2) $\mathbf{A}_2$. Using our results from calculations 4.3 and 4.5, we can say that with a probability at least $1-f_1(d,l)^{s_0}$, the expected FPR of the ith filter is upper bounded as $\varepsilon \leq \frac{4n\sqrt{\pi}}{2}e^{-\frac{\ell^2\cos^2\theta}{cd}}$.
Since each of the k_0 filters would behave independently of each other, the expected FPR of the overall PHBF is upper bounded as $\left(\frac{4n\sqrt{\pi}}{2}e^{-\frac{\ell^2\cos^2\theta}{cd}}\right)^{k_0} = \frac{\varepsilon}{2}$ with a probability at least

$$\left(1-f_1(d,l)^{s_0}\right)^{k_0} \geq 1-k_0 e^{\frac{s_0}{2d}} = 1-\frac{\varepsilon}{2}.$$

Also, with a probability at most $\frac{\varepsilon}{2}$, the expected FPR of the PHBF is trivially upper bounded by 1. It follows from this that the expected FPR is at most $(1-\frac{\varepsilon}{2})\frac{\varepsilon}{2}+\frac{\varepsilon}{2} \leq \varepsilon$.
We have therefore shown that if we use k_0 hash functions and select s_0 vectors in our SelectVectors subroutine, we can achieve an expected FPR of ε.
We can now proceed to show the bounds on k_0 and s_0. If we treat σ_X and σ_Y as constants, we may write $k_0 \leq -\frac{2c_1 d\ln(\frac{\varepsilon}{2})}{\ell - dc_2(\ln(n)+c_3)}$. Since there exists a constant C such that $\ell > Cd\ln(n)$, we may further write $k_0 \leq -c_1\frac{d\ln(\frac{\varepsilon}{2})}{\ell} \leq -c_1\ln(\varepsilon)+c_2$, for some constants c_1 and c_2.
Similarly, we can bound $s = 2d\left(\ln(k)+\ln\left(\frac{2}{\varepsilon}\right)\right) \leq -c_3 d\ln(\varepsilon)+c_4 d$ for some constants c_3 and c_4.

(3) $\mathbf{A}_3$. Using our results from calculations 4.4 and 4.5, we can say that with a probability at least $1-f_1(d,l)^{s_0}$, the expected FPR of the ith filter is upper bounded as $\varepsilon^{(i)} \leq n\sqrt{\frac{d}{2\pi\ell}\left(\sigma_X^2+\sigma_Y^2\right)}e^{-\frac{\ell}{2d(\sigma_X^2+\sigma_Y^2)}}$.
Since each of the k_0 filters would behave independently of each other, the expected FPR of the overall PHBF is upper bounded as $\left(n\sqrt{\frac{d}{2\pi\ell}\left(\sigma_X^2+\sigma_Y^2\right)}e^{-\frac{\ell}{2d(\sigma_X^2+\sigma_Y^2)}}\right)^{k_0} = \frac{\varepsilon}{2}$ with a probability at least

$$\left(1-f_1(d,l)^{s_0}\right)^{k_0} \geq 1-k_0 e^{\frac{s_0}{2d}} = 1-\frac{\varepsilon}{2}.$$

Then, proceeding as above, we get the result.

□

Theorem 4.1 tells us that the number of hash functions k, and thus the number of bits m since $k = m$ for $\delta = 1$, required to maintain an expected FPR of ε remains constant even as n increases, given that the separation between the distributions increases as well. This is unlike the traditional Bloom filter whose size must increase linearly with n in order to maintain a constant FPR. Furthermore, the number of required vector selections s also does not have a dependence on n.

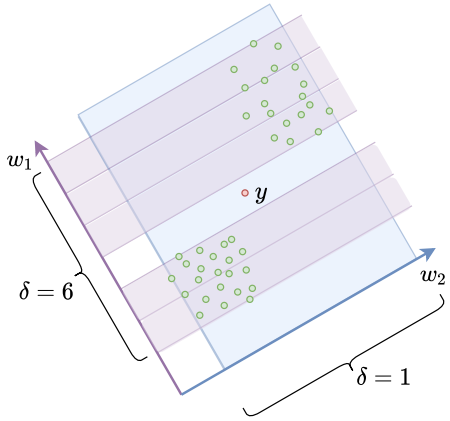

Figure 4: *Role of δ in discriminating $y \in Y$ from the points from X:* The hash function $h^{(w_2)}$ cannot separate y with $\delta = 1$, and generates a false positive. The hash function $h^{(w_1)}$ with $\delta = 6$ can successfully separate y from X.

Here, we note that this analysis is different from the analysis of FPR performed in [27]. The purpose of our analysis is to express the false positive rate in terms of the design parameters of PHBF and thus guide the selection of design parameters. To do this, we need to quantify the classifier's performance, which requires making assumptions of varying degrees about the data. To demonstrate these, we perform three analyses. By making no assumptions on the data, we get looser bounds on the FPR. By assuming that the features are independent and are bounded, we get a tighter bound. Finally, by assuming that the data comes from a particular distribution, we achieve even tighter bounds. On the other hand, the goal of the analysis performed in [27] is to compare it with other learned Bloom filters using similar models and therefore do not need to quantify the classifier's performance. They make the assumption that the model learned the distribution of scores of non-keys perfectly, which leads to the implicit assumption that the data distribution is simple enough for the model to learn or that enough training data is provided for the model.

4.1 Practical Considerations

As mentioned, this analysis makes the simplifying assumption of $\delta = 1$. This is a worst-case analysis that gives us an upper bound on the number of hash functions. To see this, we consider the situation in Figure 4. The negative key y cannot be discriminated by using $\delta = 1$, but can be separated by using $\delta = 6$. Section 5.3 talks about the choice of δ for real world datasets.

The analysis in this section is intended to provide a general guideline for selection of design parameters for PHBF. A typical workflow to incorporate PHBF is

(1) Given X and Y, determine the empirical means μ_X and μ_Y and the variances σ_X and σ_Y.

(2) Compute the ε using Equation 2. Select k such that ε^k gives the desired target FPR.

(3) Select k vectors by sampling sk random vectors, where $s = \Theta(d \ln d)$ using Algorithm 2.

(4) Select an appropriate δ and create a Bloom filter with bit-vector of size $m = \delta k$, and use the k vectors to compute the hash functions to populate the Bloom filter according to Algorithm 1. The choice of $\delta = 32$ is determined empirically to work best across all datasets. See Section 5.3 for more details on the choice of δ.

We demonstrate these steps for designing PHBF using the example of Facebook Check-in data.

(1) The means and variances of the positive and negative points of the dataset are $\mu_X = 0$, $\mu_Y = 8.64$, $\sigma_X = 1$, $\sigma_Y = 2.56$.

(2) The FPR by evaluating Equation 2 numerically yields $\varepsilon \approx 5 \times 10^{-1}$. For a target FPR of 10^{-3}, we get $k = 6$.

(3) We calculate $s = 5 \ln 5 \approx 8$.

(4) With this configuration, and setting $\delta = 32$, we get $m = 8 * 32 = 256$.

Experimentally, this configuration gives us an FPR of 7.5×10^{-4}. This means that we over provisioned the memory slightly, showing that the theory gives us the upper bound.

Section 5.6 further shows that these bounds are pessimistic and provides an upper bound for FPR. Section 5.6 verifies the theory empirically.

5 EXPERIMENTS

In this section we present the empirical comparison of the false positive rates and memory utilization of PHBF with the baseline methods on various datasets, and analyze the sensitivity of PHBF to its design parameters. We also perform a series of experiments on synthetic data generated from Gaussian distribution to confirm the theoretical results from Section 4.

5.1 Datasets

We evaluate PHBF on various well known benchmark datasets, whose details are summarized in Table 2. The datasets span a wide spectrum of applications including detection of network attacks (Kitsune [23]), identifying malicious files and URLs (Malicious URLs [22], EMBER [2]), particle detection (HIGGS [3]), detecting check-in location (Facebook Check-in [10]) and image identification (MNIST [17]).

Data preparation. We preprocessed the data to have two labels: 1 if it is inserted into the index structure, and 0 if it is not inserted. For binary datasets, we retain the labels. For example, for EMBER dataset, we insert the malicious files (labelled 1). For MNIST, which is a non-binary dataset, for each run we randomly select a class from 0, 1 and 4 as the positive class (labelled 1), and the rest as negative class (labelled 0). The result is reported on the average of five runs.

5.2 Baselines

We compared PHBF against the following methods.

Standard Bloom filter (BF). Acts as a baseline for the methods. We use Murmur3 hash, which is widely used in Bloom filters [7]. Murmur3 serializes the key to a byte stream and process it by chunking into fixed sizes. Larger keys mean more chunks to process.

Table 2: Statistics of the datasets used for comparison. Set X is inserted and set Y is queried on.

Dataset	Size of X	Size of Y	Dimensions
Kitsune	642516	121620	116
EMBER	400000	400000	2381
Higgs	56540	54323	28
Facebook Check-in	50000	50000	5
Malicious URLs	16273	2709	79
MNIST	6000	6000	784

Distance Sensitive Bloom Filter (DSBF). This is an implementation of Distance Sensitive Bloom filter using random projection based LSH [8]. We use this as a baseline because the approach seems similar to us at the surface, but as we demonstrate here, it is not suitable for the task.

Hash Adaptive Bloom Filter (HABF). Hash adaptive Bloom filters [29] uses the positive and negative keys differently that other learned Bloom filters. A data structure called HashExpressor selects the appropriate hash functions that causes the least number of collisions among the keys. They do not use a machine learning model.

Bloom Filter with Learned Hash Model (LHBF). This is an implementation of learned Bloom filters using classifier as the hash function [16]. We tried SVM, neural networks and random forests as the model and chose random forest as it performed best across all datasets.

Learned Bloom Filter (LBF). We use the implementation of LBF using sandwiching [25], which is shown to outperform the original LBF [16] consistently. We use a random forest classifier as the model. The variation in model sizes for the experiments were achieved by varying the number of trees used by the model.

Adaptive Learned Bloom Filter (Ada-BF). Adaptive Bloom filter [9] generalizes the LBF by partitioning the range of the classifier scores and assigning different Bloom filters to different ranges. This allows them to use smaller Bloom filters in regions of high confidence and larger ones where there are significant overlaps between the keys.

Partitioned Learned Bloom Filter (PLBF). Partitioned Learned Bloom filter [27] uses the idea of partitioning the space like Ada-BF. PLBF formulates the model utilization as optimization problem and selects the optimal thresholds and false positive rates for each partition.

5.3 Implementation Details

We implemented PHBF using Python 3.9 with `numpy` library. We used `dask` library for the parallelized execution of both vector selection algorithm and population of the Bloom filter partitions. All experiments were conducted on a server with Intel Xeon Gold 6248 at 2.50GHz, with 400GB of RAM.

The random vector selection was parallelized to utilize 8 cores, which gave the optimal speedup. The computation of projections and computing overlaps was vectorized using `numpy`. The hash

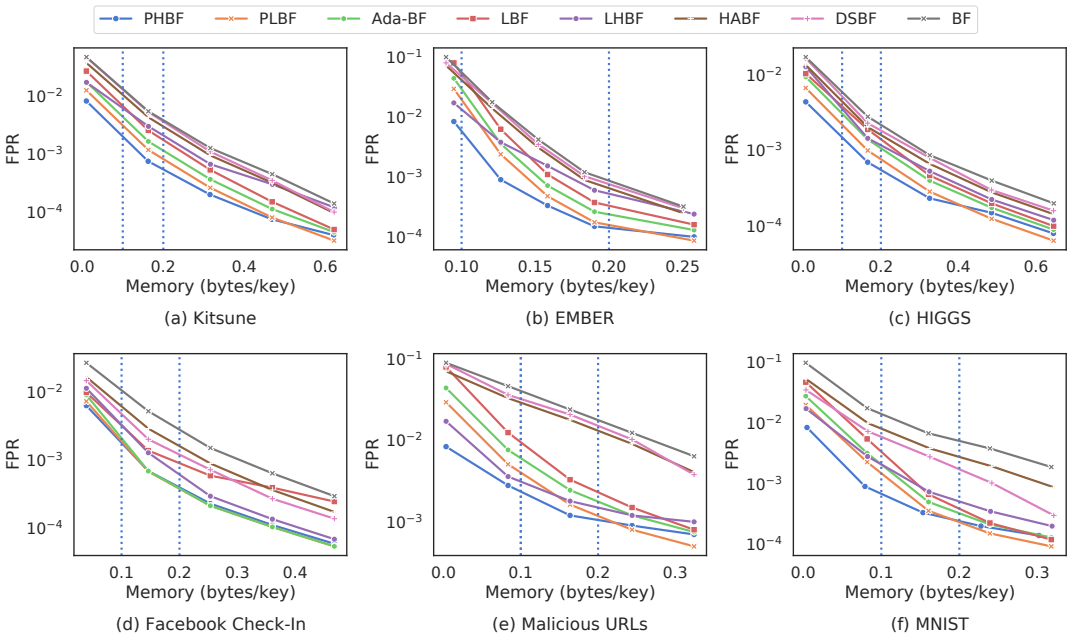

Figure 5: Comparison of FPR. The horizontal axis represents the memory usage as bytes used per key of the data. The two vertical lines indicate space usage of 0.1 and 0.2 bytes per key.

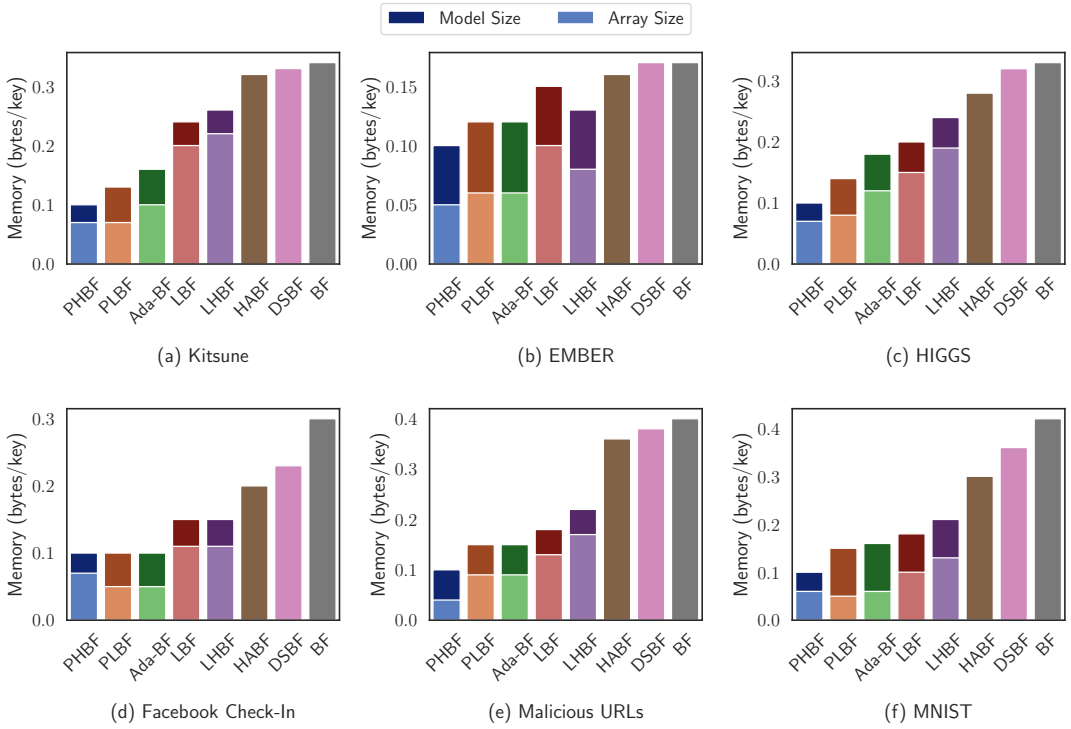

Figure 6: Comparison of bit array size and model size. The light (bottom) bar shows the bit array size and the dark (top) bar shows the model size. Note that some baselines do not use a model, and the size is entirely due to the bit array.

computation for both construction and query is parallelized using 8 cores as well.

The parameter δ was selected using grid search. The conceptual effect of δ was discussed in Section 4. In practice, we found that the value of $\delta = 32$ produces the best result across all datasets.

5.4 Results and Discussion

Figure 5 shows the comparison of PHBF in terms of memory utilization and corresponding FPR with the benchmark methods for all six datasets. PHBF outperforms all the other methods by achieving lower FPR for a given fixed space usage when the desired space usage is less than 0.2 bytes per key. The gain in performance is more pronounced in relatively higher FPR range (10^{-2}–10^{-3}). For some datasets like MNIST and Facebook Check-In, PHBF shows up to 2 orders between FPR score compared the standard Bloom filter using similar memory.

For some datasets, like Kitsune and Facebook Check-In and malicious URLs, the gap in FPR for a given memory usage is large between the learned methods and the standard Bloom filter. This shows that the pattern in the data can be exploited by these methods. In some datasets, such as EMBER, the gap between BF and LBF is quite small while PHBF outperforms LBF significantly. This shows that PHBF can distinguish between the inserted and non-inserted keys for these datasets using less memory, while the other models cannot achieve that.

We notice that PHBF performs significantly better than the baselines when operating with low memory (that is, achieving a higher compression). When achieving a space usage of 0.1 bytes per key, PHBF beats all the other baselines. At a space usage of 0.2 bytes per key, PHBF has superior performance on 5 out of 6 datasets. When more memory is used, we notice that PLBF and Ada-BF start to catch up, or even outperform PHBF. This trend holds for LBF as well. This is because given enough space, the random forest learns the data with high accuracy and thus achieve low FPR. The gain, of course, comes at the cost of high memory usage. This point confirms our assertion that PHBF is the ideal method when the application demands low memory usage.

We also note that the random projection based LSH is not a suitable hash function to exploit the complex patterns in these datasets. It only provides significant benefits over standard BF when the data is easily distinguishable, such as Facebook Check-In and MNIST. PHBF takes advantage of the multiple partitions to get independent projections, and, discretization which allows it to handle datasets with multiple modes, to consistently outperform DSBF using LSH. Similarly, while HABF outperforms standard Bloom filters, it cannot compete with methods that use the data to learn a model, rather than selecting proper hash functions.

Finally, we notice that for HIGGS, the performance of all the methods are very close. This shows that the dataset does not have enough information for the learned methods to exploit the pattern.

Figure 6 shows the memory usage of model and bit array separately. The values are selected at the FPR where PHBF achieves a space usage of 0.1 bytes per key. PHBF uses a small fraction of the total memory to store the model, thus allowing larger bit arrays. Partitioned based methods, on the other hand, uses a larger fraction of the memory to store the model. For the methods that do not use a model, the entire size is due to the bit array.

5.5 Comparison of Construction, Training, and Query Times

Figure 7 compares the construction, training and query times of PHBF with the baselines. We notice that the construction of PHBF is much faster than other learned methods, although, as expected, it is slower than the methods that do not have a *learning phase*. The training overhead for learned methods is shown in Figure 7. The training phase of PHBF, that is, the vector selection phase, is an order of magnitude faster than the machine learning models used in learned Bloom filters. HABF is faster in selection of hash functions using HashExpresssor [29], but it is not designed to learn the specific patterns of the data but rather to choose the desirable hash functions from a given set. More importantly, with parallelization, the query time per 1000 queries of PHBF is comparable to the standard Bloom filters, and order of magnitude faster than the methods using learned models. In particular, note that the query time of standard Bloom filters suffer when the dimensionality (key size) increases because this causes the Murmur3 hash function to slow down. This is because Murmur3 has to serialize the large key and break it into larger number of chunks.

5.6 Confirmation of Theoretical Results

In this section we validate our theoretical results empirically. To test the theory, we generate three datasets:

Synthetic 1 This dataset is generated by sampling points from bivariate β distribution. The reason for this choice is to make the two dimensions dependent on each other. The positive points were centered at $\mathbf{0}$ and the negatives we shifted to vary ℓ.

Synthetic 2 To generate this dataset, we sampled each dimension independently for uniform distribution with the bound of $\left[0, \frac{1}{\sqrt{d}}\right]$. The negative points are generated by shifting each of the intervals, as well as changing the bounds to vary ℓ.

Synthetic 3 This dataset uses points generated from Gaussian distribution. Positive points were generated from $\mathcal{N}(0, 1)$, and the mean and variance of negative distributions were adjusted to vary ℓ.

We construct the PHBF for each of these datasets and use the theorem corresponding to the assumptions $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$ described in Section 4.

Figure 8 shows the relationship between the number of hash functions k and the FPR for each of the three datasets and corresponding assumption. We notice that with each stronger assumption, the theoretical bound becomes tighter, while still being pessimistic. Comparison of FPR with the separation ℓ in Figure 9 displays similar trend. The theoretical value remains as upper bound and the gap between theoretical and empirical results decreases as the assumptions get stronger.

5.7 Sensitivity to Hyperparameters

We now analyze the sensitivity of the performance of PHBF with changes in various parameters.

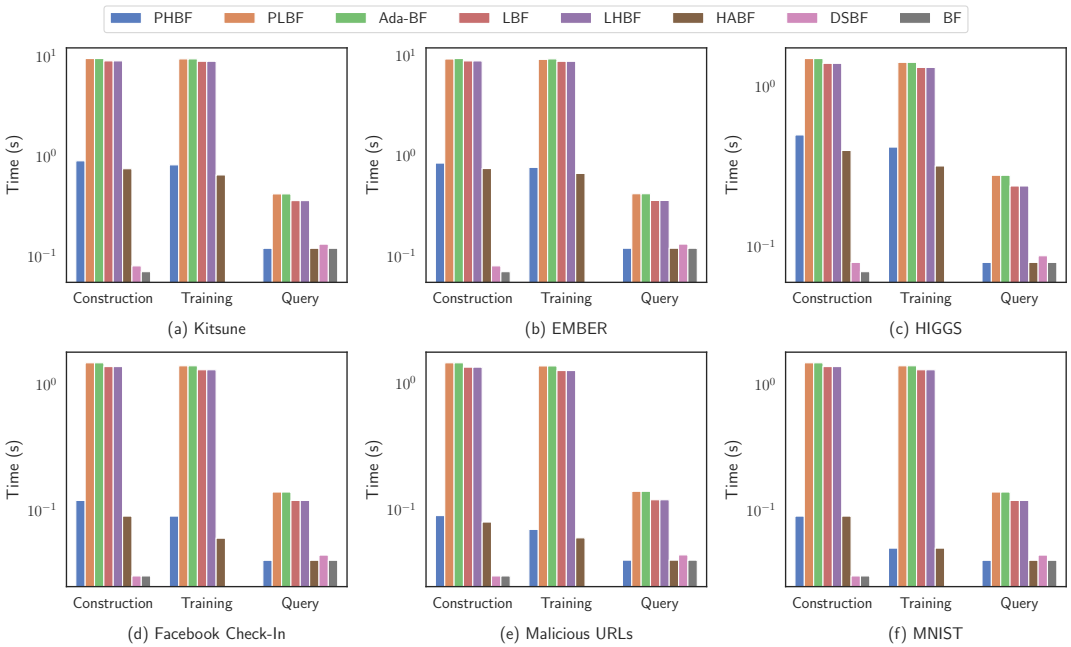

Figure 7: Comparison of construction time and time per 1000 queries. The set of first 5 methods utilizes the sets X and Y to *train* a model, leading to higher construction time compared to the last two, which do not use them.

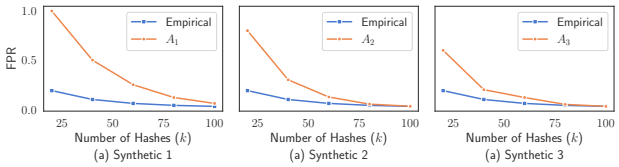

Figure 8: Comparison of empirical and theoretical relation between number of hash functions (k) and FPR.

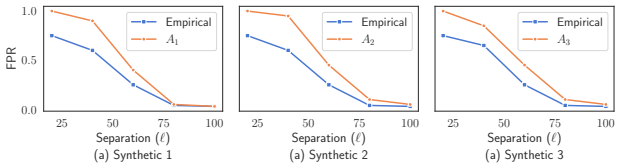

Figure 9: Comparison of empirical and theoretical relation between the separation of positive and negative keys (ℓ) and FPR.

Sensitivity to sizes and number of hash functions. Figure 10 shows the variation in false positive rate with changing the size of the bit array, m and the number of bits per partition, δ for EMBER and MNIST. Other datasets follow a similar trend. The small values of δ doesn't allow enough discriminative power to each Bloom filter. But for a fixed bit array size, increasing δ beyond a certain value leaves room for fewer hash functions. We notice that $\delta = 32$ produces the optimal results across datasets. For a given size of the bit array, the false positive rate of PHBF doesn't change significantly when the

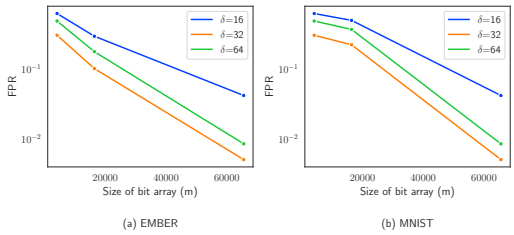

Figure 10: Sensitivity of FPR to the changes in the size of the bit array (m) and number of bits per partition (δ).

size of the bit array is small. This is because PHBF gets to work with few hash functions. As we increase the size of the bit array, increasing the size of bits per partition provides a larger gain in FPR, because the number of hash functions are large. For some datasets, like EMBER and HIGGS, the FPR changes by an order of magnitude by increasing δ from 16 to 32. This shows the clusters in data demanded a range greater than 16 for the hash functions. When increasing to 64, the lack of hash functions more than offsets the gain from greater range. For some datasets, such as MNIST and Facebook Check-In, the changes in δ doesn't affect the FPR as much. This means that the data is clustered close together and can be discriminated using hash functions of smaller range.

Sensitivity to training size. Figure 11 shows the relation between FPR and the fraction of training samples used. We see that the training time scales linearly with increase in training size. The decrease in FPR is not linear. For Kitsune and Ember, which are relatively large dataset, initially the FPR drops linearly, but the gain

in FPR diminishes with more training points. For relative smaller dataset, MNIST, all the data points are crucial and the FPR decreases almost linearly with increase in training samples.

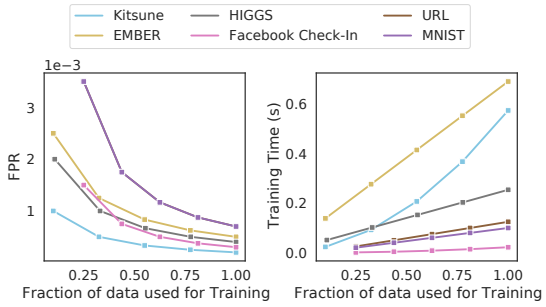

Figure 11: Sensitivity of FPR and training time with varying training data size.

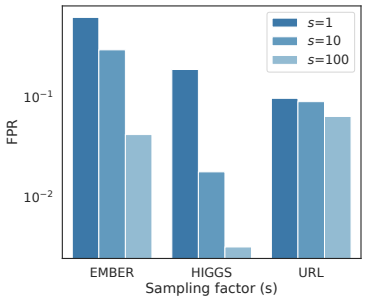

Figure 12: Sensitivity of FPR for varying sampling factor s.

Sensitivity to sampling factor. The sampling factor determines the quality of the random vectors used for the hash functions. As Figure 12 shows, the false positive rate improves as we sample more vectors. The gain, however, depends on the distribution of the data. Some datasets, such as EMBER and HIGGS gain significant advantage with sampling more vectors. Some datasets like URL doesn't show major improvements. This shows that these datasets has low effective dimensionality, beyond which additional vectors don't contribute much to distinguish the data.

5.8 Impact of Projection Hash Bloom Filter on end-to-end application

We have demonstrated the performance advantage of PHBF over other methods by comparing the false positive rates and memory usage. In this section, we consider the impact this can have on a real world application.

We demonstrate this using the example of a database system. Databases are known to use complex keys which can be used as feature vectors for machine learning. One such example is RAMBO [12] that uses sequences as keys. The cost of false positives for a database system is a false disk access when the index structure mistakenly says that a key exists in a disk.

Let us consider the scenario where the database system stores keys from Kitsune network attack dataset. We also assume that the system is designed for space usage of 0.1 bytes per key. For this configuration, from Figure 5, we see that PHBF provides a false positive rate of 0.0048, and PLBF, the next best structure provides an FPR of 0.0072. We also see from Figure 7 that PHBF takes 0.2ms and PLBF takes 0.6ms to perform a query. Additionally, a disk access for a standard 600MB/s SATA-III disk using the standard LRU caching takes 43ms per read operation [26]. This is ignoring the seek time, which can vary depending on the load. With this setup, the expected access time per key of PLBF is: (1 - 0.0072) * 0.6 + 0.0072 * (0.6 + 43) = 0.9096 ms. In contrast, PHBF takes (1 - 0.0048) * 0.2 + 0.0048 * (0.2 + 43) = 0.4064 ms, which is almost twice as fast as PLBF.

6 CONCLUSIONS

In this paper we presented a novel way to partition the Bloom filter where each partition uses a single data dependent hash function. Each hash function is derived by sampling vectors uniformly at random from $\mathbf{1}_d$ and projecting the given keys onto it. During the learning phase, PHBF selects a set of k vectors that best serve the purpose of distinguishing inserted and non-inserted keys. Then each of the hash functions populates the corresponding partition of the bit array of size m. During a query, the k hash functions are evaluated, and the corresponding bins are checked. We presented a theoretical analysis which provides a bound on false positive rate and guides the selection of design parameters.

We demonstrate the superior performance of PHBF over state of the art learned Bloom filter methods on a variety of benchmark datasets spanning a wide range of applications. We show that when the data is distinguishable, PHBF achieves 2 orders of magnitude better performance in terms of FPR for a given fixed space usage compared to the standard Bloom filter. Even when the data distribution is less favorable, PHBF can extract information better than the models used in LBF for a given memory cost.

ACKNOWLEDGMENTS

We thank Manish Borthakur and Aditi Jain for their assistance in the preliminary analysis that led to the results developed in this paper. We also thank all the anonymous reviewers their comments and suggestions that helped greatly in improving the manuscript. Srikanta Bedathur was partially supported by DS Chair of AI.

REFERENCES

[1] Nir Ailon and Bernard Chazelle. 2006. Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform. In *SoTC*.
[2] Hyrum S Anderson and Phil Roth. 2018. Ember: an open dataset for training static PE malware machine learning models. *arXiv preprint arXiv:1804.04637* (2018).
[3] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. 2014. Searching for exotic particles in high-energy physics with deep learning. *Nat. Commun.* 5, 1 (2014), 1–9.
[4] Arindam Bhattacharya, Srikanta Bedathur, and Amitabha Bagchi. 2020. Adaptive Learned Bloom Filters under Incremental Workloads. In *CoDS-COMAD*.
[5] Arindam Bhattacharya, Sumanth Varambally, Amitabha Bagchi, and Srikanta Bedathur. 2021. Fast One-class Classification using Class Boundary-preserving Random Projections. In *KDD*.
[6] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An Improved Construction for Counting Bloom Filters. In *ESA*.
[7] Apache Cassandra. 2014. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra* 13 (2014).

[8] Francis Chang, Wu-chang Feng, and Kang Li. 2004. Approximate caches for packet classification. In *IEEE INFOCOM 2004*, Vol. 4. IEEE.

[9] Zhenwei Dai and Anshumali Shrivastava. 2020. Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier with Application to Real-Time Information Filtering on the Web. In *NeurIPS*.

[10] Facebook 2016. Facebook V Check In Dataset. https://www.kaggle.com/c/facebook-v-predicting-check-ins/data.

[11] Shahabeddin Geravand and Mahmood Ahmadi. 2013. Bloom filter applications in network security: A state-of-the-art survey. *Comput. Netw.* (2013).

[12] Gaurav Gupta, Minghao Yan, Benjamin Coleman, Bryce Kille, RA Leo Elworth, Tharun Medini, Todd Treangen, and Anshumali Shrivastava. 2021. Fast processing and querying of 170tb of genomics data via a repeated and merged bloom filter (rambo). In *Proceedings of the 2021 International Conference on Management of Data*.

[13] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*.

[14] Zhongming Jin, Yao Hu, Yue Lin, et al. 2013. Complementary projection hashing. In *ICCV*.

[15] Adam Kirsch and Michael Mitzenmacher. 2006. Distance-sensitive bloom filters. In *ALENEX*.

[16] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*.

[17] Yann LeCun. 1998. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/

[18] Haomiao Liu, Ruiping Wang, Shiguang Shan, and Xilin Chen. 2016. Deep supervised hashing for fast image retrieval. In *CVPR*.

[19] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable Learned Bloom Filters for Data Streams. *Proc. VLDB Endow.* 13 (2020), 2355–2367.

[20] Wei Liu, Jun Wang, et al. 2012. Supervised hashing with kernels. In *CVPR*. IEEE.

[21] Michael J Lyons and David Brooks. 2009. The design of a Bloom filter hardware accelerator for ultra low power systems. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*.

[22] Mohammad Saiful Islam Mamun et al. 2016. Detecting malicious URLs using lexical analysis. In *NSS*. IEEE.

[23] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. 2018. Kitsune: an ensemble of autoencoders for online network intrusion detection. In *Proceedings of the 2018 Network and Distributed System Security Symposium*.

[24] Michael Mitzenmacher. 2002. Compressed Bloom Filters. *IEEE ACM Trans. Networking* 10, 5 (2002).

[25] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *NeurIPS*.

[26] A Skendžić, B Kovačić, and Edvard Tijan. 2016. Effectiveness analysis of using Solid State Disk technology. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE.

[27] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. 2021. Partitioned Learned Bloom Filter. In *ICLR*.

[28] Yair Weiss, Antonio Torralba, Robert Fergus, et al. 2008. Spectral hashing. In *NIPS*, Vol. 1.

[29] Rongbiao Xie, Meng Li, Zheyu Miao, Rong Gu, He Huang, Haipeng Dai, and Guihai Chen. 2021. Hash Adaptive Bloom Filter. In *ICDE*.