Deletable Learned Bloom Filter

1 Abstract

The Bloom Filter is a space-efficient probabilistic data structure widely used for approximate membership queries. More recently, the Learned Bloom Filter (LBF) has been introduced to improve performance by leveraging models trained on the data. However, both traditional Bloom Filters and LBFs lack native support for element deletion - a critical limitation for applications such as spam filtering and IP routing.

This study addresses this gap by extending the LBF and Sandwiched Learned Bloom Filter (SLBF) to support deletions while maintaining low false positive rates (FPR). We propose two primary designs: (1) an SLBF with two Counting Bloom Filters (CBFs) that enables deletion by replacing Standard Bloom Filters (SBFs) with CBFs, and (2) an LBF with 3 SBFs that guarantees deletion by explicitly tracking removed elements, introducing a controlled false negative rate (FNR) to optimize for FPR.

We derive analytical expressions for FPR, deletability, and FNR for each design and validate our approach through simulation-based evaluation. Results show that the SLBF with 2 CBFs achieves the lowest FPR, while the LBF with 3 SBFs provides perfect deletability at the cost of slightly increased FNR. These findings underscore a fundamental trade-off among FPR, FNR, and deletability, and offer new strategies for enabling deletion in LBF variants.

2 Introduction

A Bloom Filter is a space-efficient probabilistic data structure used to test whether an element is a member of a set. It provides a compact representation and fast approximate membership queries, making it ideal for scenarios where minimizing memory usage is important and a small false positive rate is acceptable. The Standard Bloom Filter (SBF) [1] is the original design proposed by Bloom. In a SBF, for a set S of size n, k hash functions are used to map each element $x \in S$ to k positions in an m-bit array, setting each of those positions to 1. To perform a membership query, the input is hashed using the same k functions, and the filter returns true if all the corresponding bits are set to 1.

2.1 Motivation

Despite its widespread adoption in domains like IP lookup [2] and CDNs [3], the SBF suffers from a key limitation: it lacks support for deletion.

Deletion is a critical feature in numerous real-world scenarios. For example, Bloom filters are often used to maintain blacklists, such as those for spam email addresses [4]. These lists must be regularly updated to reflect changes in threat intelligence, necessitating the ability to remove outdated entries. In networking, Bloom filters support Longest Prefix Matching for IP routing [2], where prefix sets evolve dynamically. Without deletion, outdated prefixes remain in the filter, undermining routing accuracy.

More broadly, deletable filters are valuable for thinning out stale elements, reducing false positives, and reclaiming space for new insertions. All of these are essential for maintaining performance in dynamic systems.

2.2 Counting Bloom Filter

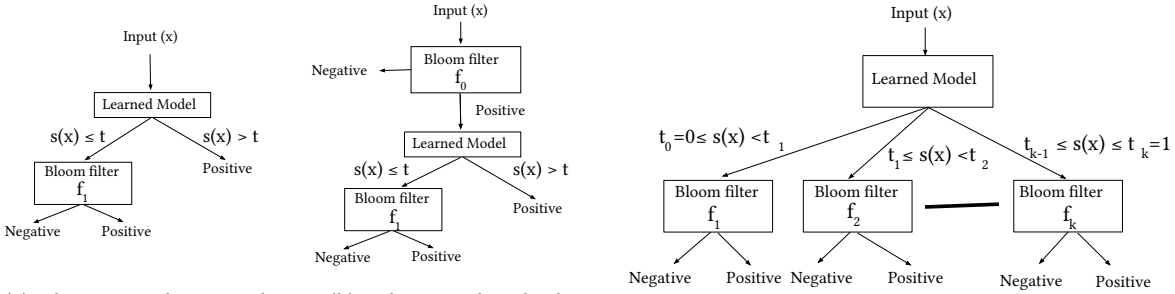
The Counting Bloom Filter (CBF) [5] extends the SBF to support deletions by replacing each bit in the bit array with a counter. It modifies the core operations of SBF as follows:

- **Insertion**: Instead of setting bits to 1, it increments the corresponding counters by 1.
- **Deletion**: Deletion is performed by decrementing the same counters by 1.
- **Query**: To check for membership, it verifies whether all corresponding counters are at least 1. If so, it returns true; otherwise, false.

This counter-based approach enables efficient and safe element removal while preserving the benefits and operations of SBF.

2.3 Learned Bloom Filter

In recent years, Kraska et al. [6] introduced the Learned Bloom Filter (LBF), a Bloom Filter that incorporates machine learning to aid the traditional Bloom filters. The core idea is to use a learned model as a pre-filter that assigns each query x a score $s(x)$, interpreted as the probability that $x \in S$. If the score is high, the LBF returns true directly; otherwise, the query is passed to a backup SBF to guarantee zero false negatives. This architecture is illustrated in Figure 1a. By leveraging patterns in data distributions, the LBF can significantly reduce the FPR while maintaining the same memory footprint as a traditional Bloom filter.



(a) The original Learned Bloom Filter.

(b) The Sandwiched Learned Bloom Filter.

(c) The Partitioned Learned Bloom Filter.

Figure 1: Different variants of Learned Bloom Filter. Adapted from Figure 1 of [7].

Mitzenmacher [8] introduces the Sandwiched Learned Bloom Filter (SLBF) to further reduce the FPR by placing a SBF before the learned model, as shown in Figure 1b. This initial SBF allows only positive queries to reach the learned model while immediately rejecting queries that are not in the set. This design is a strict generalization of the LBF as when the initial filter is empty, the SLBF behaves exactly like an LBF.

The Partitioned Learned Bloom Filter (PLBF) [7] enhances LBF performance by partitioning the score range $s(x)$ into multiple regions based on optimized thresholds and FPRs. A separate SBF is constructed for each region. This approach optimizes memory usage by allocating more bits to score regions with higher densities of both keys and non-keys, thereby improving overall FPR.

2.4 Deletable Learned Bloom Filter

Zeng et al. [9] investigate how to extend PLBF with deletion support and propose a Two-layer Partitioned and Deletable Deep Bloom Filter (PDDBF). Their approach replaces the SBFs in the original PLBF with CBFs and introduces a hash table to handle elements with scores in the range $[t_d, 1)$, where t_d is a threshold chosen to split the score space. Inputs with $0 \leq s(x) < t_d$ are processed using the standard PLBF mechanism, where deletion is supported via the CBFs. For inputs where $t_d \leq s(x) < 1$, the hash table stores keys that have been deleted. This hybrid design is illustrated in Figure 2.

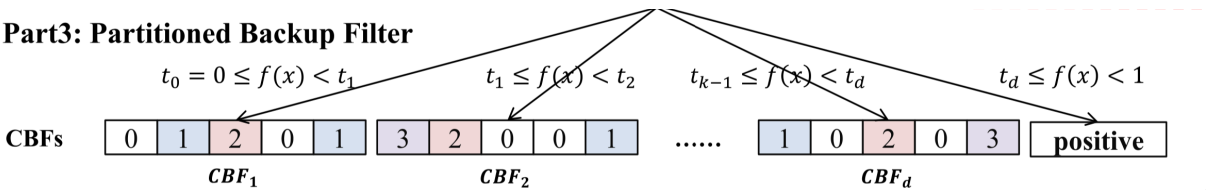


Figure 2: The PDDBF. Adapted from Figure 3 of [9]. Details on data partitioning and model training from the original work are omitted as they are not relevant to our context. The **positive** component refers to the hash table used for storing deleted keys in the range $t_d \leq s(x) < 1$.

However, they do not detail how the threshold t is chosen and simply rephrases the original PLBF optimization problem. Also, they failed to effectively address why the hash table is needed and why not use CBF all the time.

2.5 Our Focus and Approach

In this project, we focus on extending the LBF and the SLBF to support deletions. We select LBF and SLBF due to the limited research on enabling deletion in these variants. Although the PLBF can achieve lower false positive rates with the same memory budget, its construction is significantly more computationally expensive. Even with recent optimizations (e.g., [10]), PLBF construction still requires quasi-linear time relative to dataset size. In contrast, LBF and SLBF offer faster and more practical alternatives, particularly for large datasets. Furthermore, we do not focus exclusively on SLBF because, as shown in Section 5.2 of [8], constructing an initial filter is not always optimal. This depends on parameters such as bits per item and the false positive and false negative rates of the learned model.

We investigate multiple approaches for enabling deletion using CBF, and evaluate their FPR and deletability. Furthermore, we examine how introducing a controlled and provably bounded false negative rate (FNR) can offer an FPR tradeoff, and discuss scenarios where such a tradeoff may be acceptable or even desirable in practical applications.

3 Technical details

3.1 Definition

Following [8], we define several commonly used notations in this section. Let K denote the set of keys encoded by the Bloom Filter, and $m = |K|$. Let U denote the set of all possible queries. For each Bloom Filter variant, we assume support for the following operations:

- $\mathbf{Insert}(f, x)$: Inserts element x into Bloom Filter f, using f's insertion method.
- $\mathbf{Query}(f, x) \to \{0, 1\}$: Queries element x using Bloom Filter f, following f's query method. Returns 1 if $x \in f$, and 0 otherwise.

For Bloom Filter variants that support deletion, the following operation is also assumed:

- $\mathbf{Delete}(f, x)$: Deletes element x from Bloom Filter f, following f's deletion method.

We denote the learned model associated with LBF variants as f_L. Its empirical false negative rate is defined as

$$F_n = \frac{|\{x \mid \forall x \in k, f_L(x) \leq t\}|}{m}$$

We denote the false positive rate as $\mathrm{FPR}(f)$, where f refers to either a Bloom Filter or a learned model. For a deletable Bloom Filter f, we denote its deletability as $D(f)$, defined as the probability that $\mathbf{Query}(f, x) = 0$ after performing $\mathbf{Delete}(f, x)$, where $x \in K$.

3.2 SLBF with 2 CBFs

We start by exploring how to enhance SLBF. Borrowed the idea from adding deletion to PLBF, we start by replacing SBF in SLBF with CBF. Figure 3 illustrates the new design.

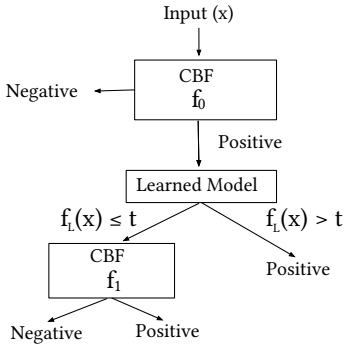


Figure 3: The SLBF with SBF replaced by CBF.

Definition 3.1 **SLBF with 2 CBFs**

The SLBF f with 2 CBFs supports three operations, defined as follows

- $\mathbf{Insert}(f, x)$: Similar to the SLBF's insertion, but using CBF's $\mathbf{Insert}$ to replace SBF's $\mathbf{Insert}$.
- $\mathbf{Query}(f, x) \to \{0, 1\}$: Similar to the SLBF's insertion, but using CBF's $\mathbf{Query}$ to replace SBF's $\mathbf{Query}$. So, the return is 1 only when

$$\mathbf{Query}(f_0, x) \wedge (f_L(x) > t \vee \mathbf{Query}(f_1, x))$$

- $\mathbf{Delete}(f, x)$: Run $\mathbf{Delete}(f_0, x)$. If $f_L(x) \leq t$, additionally run $\mathbf{Delete}(f_1, x)$.

We now analyze the FPR and deletability of this design. Let the total memory budget be mb bits, with f_0 allocated $b_0 m$ bits and f_1 allocated $b_1 m$ bits, such that $b_0 + b_1 = b$.

FPR. For the FPR, we know that for $x \notin K$, the probability that $\mathbf{Query}(f, x) = 1$ is

$$\mathrm{FPR}(f) = \mathrm{FPR}(f_0)(\mathrm{FPR}(f_L) + (1 - \mathrm{FPR}(f_L))\ \mathrm{FPR}(f_1)) \tag{1}$$

Remark **FPR of CBF**

Before expanding Equation 1, we need to know the FPR of CBF. For a CBF with total memory z, c-bits for each cell, and k hash functions that encodes y elements, its FPR is

$$\left(1 - \left(1 - \frac{1}{z/c}\right)^{ky}\right)^k \approx \left(1 - e^{\frac{-ky}{z/c}}\right)^k$$

Optimize this with $k = \frac{z}{cy} \ln 2$, we get

$$\left(1 - e^{\frac{-ky}{z/c}}\right)^k = \alpha^{\frac{z}{cy}} \quad \text{where} \quad \alpha = \left(\frac{1}{2}\right)^{\ln 2}$$

As CBF f_0 encodes m elements and CBF f_1 only encodes $F_n m$ elements, we can expand Equation 1 as

$$\alpha^{\frac{b_0}{c}} \left(\mathrm{FPR}(f_L) + (1 - \mathrm{FPR}(f_L))\alpha^{\frac{b_1}{cF_n}}\right) \tag{2}$$

Similar to [8], as $\alpha, \mathrm{FPR}(f_L), F_n, b$ are constants, we can optimize for b_0. The derivative of Equation 2 with respect to b_0 is:

$$\frac{\ln a}{c} \alpha^{\frac{b_0}{c}} \left[\mathrm{FPR}(f_L) + (1 - \mathrm{FPR}(f_L))\alpha^{\frac{b-b_0}{cF_n}} - \frac{1}{F_n}(1 - \mathrm{FPR}(f_L))\alpha^{\frac{b-b_0}{cF_n}}\right]$$

This equals 0 when:

$$\mathrm{FPR}(f_L) = (1 - \mathrm{FPR}(f_L))\left(\frac{1}{F_n} - 1\right)\alpha^{\frac{b-b_0}{cF_n}}$$

This means the optimal b_1^* is:

$$b_1^* = F_n c \log_a \left(\frac{\mathrm{FPR}(f_L)}{(1 - \mathrm{FPR}(f_L))\left(\frac{1}{F_n} - 1\right)} \right)$$

Similar to Mitzenmacher's result for SLBF with SBF [8], the optimal b_1^* is independent of b and the total memory budget. In other words, the optimal number of bits for the backup filter is a fixed number of bits. Also, when $c = 1$, our result matches Mitzenmacher's result, where the optimal $b_{1_{\mathrm{SBF}}}^*$ for SBF is

$$b_{1_{\mathrm{SBF}}}^* = F_n \log_a \left(\frac{\mathrm{FPR}(f_L)}{(1 - \mathrm{FPR}(f_L))\left(\frac{1}{F_n} - 1\right)} \right)$$

The increase of the optimal b_1^* by a factor of c is expected, as the SBF result derived by [8] can be interpreted to mean that the optimal backup filter f_1 requires $b_{1_{\mathrm{SBF}}}^*$ bits (i.e., counters of 1 bit each). Since the number of bits required for f_1 is independent of the total memory budget, the optimal b_1^* becomes $c \times b_{1_{\mathrm{SBF}}}^*$ when each counter must be c bits.

As a result, when b is large enough (i.e., $b > b_1^*$), the $\mathrm{FPR}(f)$ can be written as:

$$\mathrm{FPR}(f) = \alpha^{\frac{b_0^*}{c}} \frac{\mathrm{FPR}(f_L)}{1 - F_n} \quad \text{where} \quad b_0^* = b - b_1^*$$

Deletability. For deletability, we want to know for a random element $x \in K$, after deletion, the probability that $\mathbf{Query}(f, x) = 0$.

> **Remark** **Deletability of CBF**
>
> Consider a CBF f_c with total memory z, c-bits for each cell, and k hash functions that encodes y elements. For an x encoded in CBF, the probability that $\mathbf{Query}(f_c, x) = 0$ after deleting it is
>
> $$1 - \left(1 - \left(1 - \frac{1}{z/c}\right)^{k(y-1)}\right)^k \approx 1 - \left(1 - e^{-\frac{k(y-1)}{z/c}}\right)^k$$
>
> where
>
> $$\left(1 - \left(1 - \frac{1}{z/c}\right)^{k(y-1)}\right)^k \tag{3}$$
>
> denotes the probability that all the k cells used by x are all incremented due to the insertion of other keys. As a result, we can compute the probability that at least one of the k cells used by x remains not incremented by other insertions as 1 minus the Equation 3. This is precisely the probability that $\mathbf{Query}(f_c, x) = 0$, since after x is deleted from the CBF, any such cell will return to zero, thereby causing $\mathbf{Query}(f_c, x)$ to return 0.

According to the query definition, deletion is considered successful under the following conditions:

- x is successfully removed from f_0; or

- x is not successfully removed from f_0, but $f_L(x) \leq t$ and x is successfully removed from f_1.

Therefore, the probability that a random $x \in K$ is successfully deleted is:

$$D(f) = D(f_0) + (1 - D(f_0)) \cdot F_n \cdot D(f_1)$$

As CBF f_0 encodes m elements and CBF f_1 only encodes $F_n m$ elements, using the deletability formula and $k = \frac{z}{cy} \ln 2$ (optimized with respect to $\mathrm{FPR}(f_c)$), we can derive:

$$D(f_0) = 1 - \left(1 - \left(\frac{1}{2}\right)^{\frac{m-1}{m}}\right)^{\frac{b_0}{c} \ln 2}$$

$$D(f_1) = 1 - \left(1 - \left(\frac{1}{2}\right)^{\frac{F_n m - 1}{F_n m}}\right)^{\frac{b_1}{c F_n} \ln 2}$$

3.3 SLBF with 3 CBFs

After examining the two-CBF configurations, we now consider the case involving three CBFs in the SLBF design, as illustrated in Figure 4. In this design, each CBF has memory budget $b_0 m, b_1 m, b_2 m$ respectively and $b_0 + b_1 + b_2 = b$. We tried to compute partial derivative in terms of any two of them. However, the resulting transcendental equations prevent further optimization.

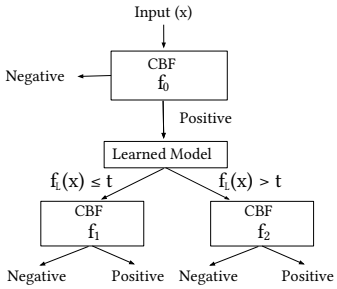


Figure 4: The SLBF with 3 CBFs.

3.4 LBF with 2 CBFs

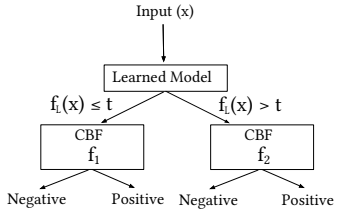


Figure 5: The LBF with 2 CBFs.

Since Figure 4 is difficult to optimize, we consider a simplified variant illustrated in Figure 5. This design incorporates two Bloom Filters: one for elements satisfying $f_L(x) \leq t$ and another for those where $f_L(x) > t$. While this structure resembles the PLBF with two regions, it is important to note that we have not performed the optimization necessary to derive the optimal threshold and false positive rates. As such, the design remains easy to construct.

Definition 3.2 **LBF with 2 CBFs**

The LBF f with 2 CBFs supports three operations, defined as follows

- $\mathbf{Insert}(f, x)$: Run $\mathbf{Insert}(f_1, x)$ when $f_L(x) \leq t$ else $\mathbf{Insert}(f_2, x)$.
- $\mathbf{Query}(f, x) \to \{0, 1\}$: Run $\mathbf{Query}(f_1, x)$ when $f_L(x) \leq t$ else $\mathbf{Query}(f_2, x)$. So, the return is 1 only when

$$(f_L(x) \leq t \wedge \mathbf{Query}(f_1, x)) \vee (f_L(x) > t \wedge \mathbf{Query}(f_2, x))$$

- $\mathbf{Delete}(f, x)$: Run $\mathbf{Delete}(f_1, x)$ if $f_L(x) \leq t$ else $\mathbf{Delete}(f_2, x)$.

This configuration can be interpreted in two ways:

- As an alternative design for scenarios where the memory budget b is too small to allocate any space for f_0 in SLBF, making its inclusion infeasible; or
- As a modified version of the LBF, augmented with an additional CBF for the region where $f_L(x) > t$. Here, we choose this version.

For FPR and deletability analysis, similar to before, suppose the total memory budget for these filters is mb number of bits, and f_1 has bits $b_1 m$ and f_2 has bits $b_2 m$ where $b_1 + b_2 = b$.

FPR. By the definition of the query, for $x \notin K$, the probability that $\mathbf{Query}(f, x) = 1$ is:

$$\mathrm{FPR}(f) = \mathrm{FPR}(f_L)\ \mathrm{FPR}(f_2) + (1 - \mathrm{FPR}(f_L))\ \mathrm{FPR}(f_1)$$

Since f_1 encodes $F_n m$ elements and f_2 encodes $(1 - F_n)m$ elements, the above equation can be rewritten as:

$$\mathrm{FPR}(f_L)\alpha^{\frac{b_2}{c(1-F_n)}} + (1 - \mathrm{FPR}(f_L))\alpha^{\frac{b_1}{cF_n}}$$

Compute the derivative:

$$\frac{\ln a}{c}\left[-\frac{\mathrm{FPR}(f_L)}{1 - F_n}\alpha^{\frac{b-b_1}{c(1-F_n)}} + \frac{1 - \mathrm{FPR}(f_L)}{F_n}\alpha^{\frac{b_1}{cF_n}}\right]$$

This equals 0 when:

$$\frac{\mathrm{FPR}(f_L)}{1 - F_n}\alpha^{\frac{b-b_1}{c(1-F_n)}} = \frac{1 - \mathrm{FPR}(f_L)}{F_n}\alpha^{\frac{b_1}{cF_n}}$$

Taking the logarithm of both sides and solving for b_1^*, we obtain:

$$b_1^* = bF_n - \frac{cF_n(1 - F_n)}{\ln a}\ln\left(\frac{(1 - F_p)(1 - F_n)}{F_n F_p}\right)$$

Consider an example where $F_n = 0.5$, $\mathrm{FPR}(f_L) = 0.01$, $\alpha = 0.6185$, $c = 2$, and $b = 20$. Then, the optimal b_1^* and b_2^* is

$$b_1^* = 14.782 \quad \text{and} \quad b_2^* = 5.218$$

Thus, the FPR is

$$\mathrm{FPR}(f) \approx 0.01 * 0.0815 + 0.99 * 0.000823 = 0.00163$$

As the SLBF is also possible to construct in this case, we can calculate its FPR:

$$b_1^* = 4.782 \quad \text{and} \quad b_0^* = 15.217 \quad \text{and FPR is} \quad 0.000517$$

Since this structure is not particularly competitive when SLBF is feasible, and its effectiveness heavily depends on $\mathrm{FPR}(f_L)$ and other parameters, we do not consider it a strong alternative. Rather, we propose it as a fallback option when SLBF construction is infeasible. Consequently, we do not derive the corresponding deletability formula for this design and evaluate it in the evaluation section. However, it should be easy to derive using the same principles outlined in Section 3.2.

3.5 LBF with 3 SBFs

Lastly, we explore an important trade-off between the FPR and the false negative rate (FNR). Until now, all previously discussed designs maintain the invariant that FNR is zero. However, we argue that this constraint is not always necessary, particularly if allowing a non-zero FNR enables guaranteed deletions in LBF.

For example, consider a Bloom Filter used to store a list of spam email addresses. In this context, a false negative, where a spam email bypasses the filter and reaches the user's inbox, may be acceptable. Moreover, consider the scenario where a user marks an email as spam using the Bloom Filter. It is undesirable if that email can still appear in the inbox due to the Bloom Filter's FPR. In such cases, guaranteed deletion (i.e., deletability equal to 1) is beneficial.

Therefore, we explored what we can do to make guaranteed deletion and tradeoff some FNR with FPR, as illustrated in Figure 6.

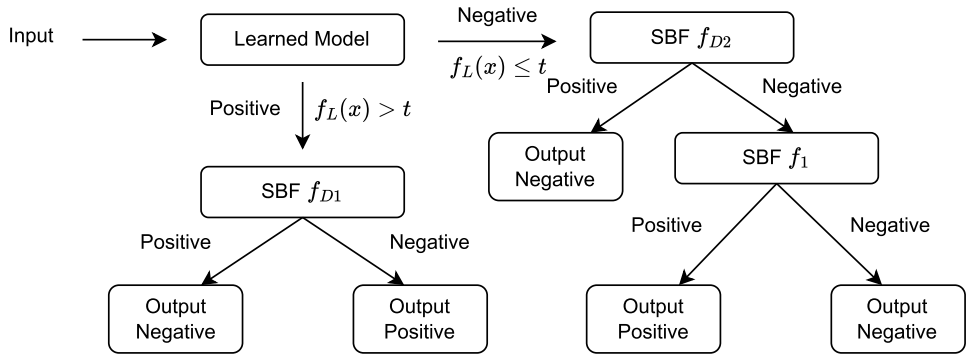


Figure 6: The SLBF with SBF replaced by CBF.

Definition 3.3 **LBF with 3 SBFs**

The LBF f with 3 SBFs supports three operations, defined as follows

- $\mathbf{Insert}(f, x)$: Run $\mathbf{Insert}(f_1, x)$ when $f_L(x) \leq t$.
- $\mathbf{Query}(f, x) \to \{0, 1\}$: the query method returns 1 iff

$$\left(f_L(x) \leq t \wedge \neg\, \mathbf{Query}\left(f_{D_2}, x\right) \wedge \mathbf{Query}(f_1, x)\right) \vee \left(f_L(x) > t \wedge \neg\, \mathbf{Query}\left(f_{D_1}, x\right)\right)$$

- $\mathbf{Delete}(f, x)$: Run $\mathbf{Insert}\left(f_{D_2}, x\right)$ if $f_L(x) \leq t$ else $\mathbf{Insert}\left(f_{D_1}, x\right)$.

Intuitively, this design extends the LBF by augmenting it with two additional SBFs: one for managing deleted elements x such that $f_L(x) > t$, and the other for those where $f_L(x) \leq t$. In essence, these two SBFs store the keys deleted from the LBF, enabling approximate membership queries for the deleted items.

For FPR, deletability, and FNR analysis, assume the total memory budget for the filters is mb bits. Let f_1, f_{D_1}, and f_{D_2} be allocated $b_1 m$, $b_2 m$, and $b_3 m$ bits, respectively, where $b_1 + b_2 + b_3 = b$. Additionally, assume that λm random elements are subject to deletion.

FPR. By the definition of query, for $x \notin K$, where x was not previously deleted, the probability that $\mathbf{Query}(f, x) = 1$ is:

$$\mathrm{FPR}(f) = \mathrm{FPR}(f_L)\Big(1 - \mathrm{FPR}\Big(f_{D_1}\Big)\Big) + (1 - \mathrm{FPR}(f_L))\Big(1 - \mathrm{FPR}\Big(f_{D_2}\Big)\Big)(\mathrm{FPR}(f_1))$$

In other words, x is misclassified only under the following conditions:

- The learned model misclassifies x, and the SBF f_{D_1} fails to recognize it as deleted (i.e., f_{D_1} does not report x as deleted, thereby allowing the learned model's misclassification to propagate); or
- The learned model correctly classifies x, and f_{D_2} also does not indicate deletion, but the backup SBF f_1 misclassifies x.

Deletability. If an element $x \in K$ was previously inserted and is later deleted, then by design it will appear in either f_{D_1} or f_{D_2}. Since the SBF has no false negatives, the probability that $\mathbf{Query}(f, x) = 1$ is zero. This implies that deletion is guaranteed and the deletability is 1. Moreover, as elements are deleted, the size of the complement of the set K, namely, $U - K$, increases. This implies that the FPR decreases as the number of deletions increases, because the number of false positive samples remains unchanged (due to the guaranteed deletion), as illustrated by the following formula:

$$\mathrm{FPR}(f) = \frac{|\{x \mid x \in U - K, \mathbf{Query}(f, x) = 1\}|}{|U - K|}$$

FNR. Guaranteed deletion comes at a cost. While the additional SBFs f_{D_1} and f_{D_2} enable guaranteed deletions, they also introduce false negatives. For an element $x \in K$, the probability that it is misclassified as negative is given by:

$$\mathrm{FNR}(f) = (1 - F_n) \cdot \mathrm{FPR}\Big(f_{D_1}\Big) + F_n \cdot \mathrm{FPR}\Big(f_{D_2}\Big)$$

This expression captures the FNR because positive items may only be misclassified as negatives under the following conditions:

- The learned model classifies x correctly, but f_{D_1} incorrectly identifies it as deleted; or
- The learned model misclassifies x, and f_{D_2} also incorrectly identifies it as deleted. In this case, if f_{D_2} were correct, the query would proceed to the backup SBF f_1, which, since $x \in K$, would return 1.

Given the assumption that λm random elements in K will be deleted, we note that f_{D_1} encodes $\lambda m(1 - F_n)$ elements, since $(1 - F_n)$ fraction of K is correctly classified as positive by the learned model. Similarly, f_{D_2} encodes $\lambda m F_n$ elements, corresponding to the portion of K misclassified as negative.

Substituting these quantities into the earlier expression, the FNR can be rewritten as:

$$\mathrm{FNR}(f) = (1 - F_n) \cdot \alpha^{\frac{b_2}{\lambda(1-F_n)}} + F_n \cdot \alpha^{\frac{b_3}{\lambda F_n}}$$

This formulation implies that the FNR of f can be explicitly controlled through the FPR of f_{D_1} and f_{D_2}.

4 Evaluation

We evaluate the SLBF with 2 CBFs (Definition 3.1) and LBF with 3 SBFs (Definition 3.3) in this section.

Experiment Setup. We adopt a simulation-based approach to conduct our experiments, avoiding the complexity of training a neural network from scratch. Experimental results are drawn from Kraska et al. [6], based on the Google Transparency Report dataset. The parameters used in our simulations are summarized in Table 1.

Parameter	Explanation	Value
n	Number of data to encode	1.7M
m_L	Memory budget for the learned model	0.0259MB
$\text{FPR}(f_L)$	False positive rate of the learned model	0.5% and 0.1% (two configurations)
F_n	Empirical false negative rate of the learned model	55% and 76% (two configurations)
c	Number of bits for each counter	4 bits
λ	Percentage of elements to be deleted	10%

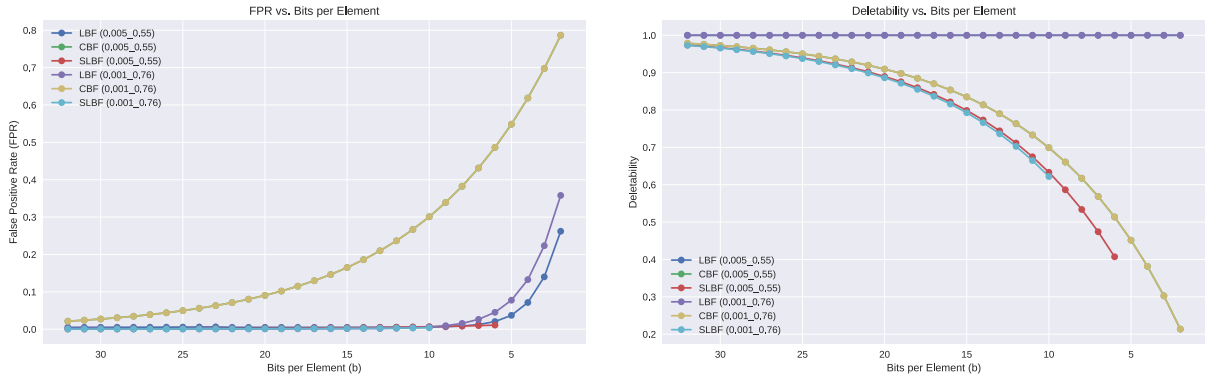
Table 1: Experimental result from [6].

We compare our two designs with each other and against a baseline single CBF design. For the LBF with 3 SBFs, we employ the SLSQP optimization method from the `scipy` library to allocate bits across the filters. We tried two objectives to minimize: 1) the sum of FPR and FNR and 2) the overall FPR.

Experiment Metrics. To evaluate the performance of our designs, we consider the following metrics:
- FPR: The false positive rate of our design.
- Deletability: The probability that an element can be deleted.
- FNR: The false negative rate of our design, specific to LBF with 3 SBFs.

We will examine how these metrics vary with respect to b, the number of bits per element. To control b, we will vary the total memory budget m for our design.

4.1 Results



(a) FPR of different designs vs. bits per element.

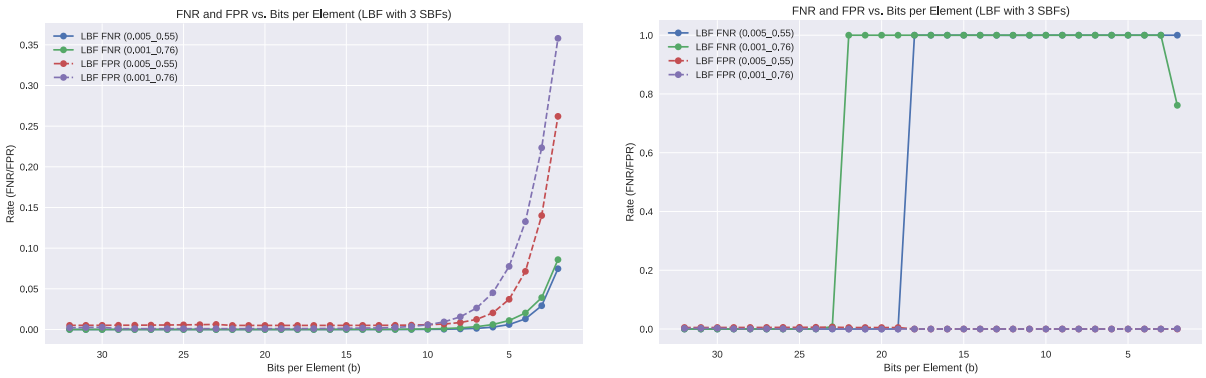
(b) Deletability of different designs vs. bits per element.

Figure 7: FPR and Deletability of different designs vs. bits per element. In the legend, the numbers in parentheses represent the $\text{FPR}(f_L)$ and F_n, respectively. The LBF with 3 SBFs is optimized with respect to the sum of FNR and FPR. The absence of some data points for SLBF reflects that, for certain b, SLBF construction is not feasible.

Figure 7a illustrates how the FPR varies as a function of b. **First**, it is evident that incorporating a learned model enhances Bloom Filter performance. This is demonstrated by the substantially higher FPR of the CBF-only model compared to the other designs, a result consistent with the findings of Kraska et al. in [6]. **Second**, the FPR remains relatively stable across different configurations of $\text{FPR}(f_L)$ and empirical F_n. For each design, the curves corresponding to the two configurations nearly overlap, indicating the robustness of our designs with respect to variations in learned model performance. **Third**, the SLBF with 2 CBFs and the LBF with 3 SBFs exhibit comparable FPR in the range $b = 32$ to $b = 9$, likely due to effective optimization. The SLBF with 2 CBFs performs slightly better in the lower range from $b = 9$ to $b = 6$.

Figure 7b illustrates the deletability performance across different designs. As expected, the LBF with 3 SBFs achieves the highest deletability, consistently maintaining a value of 1 due to its support for guaranteed deletion. Following this, the CBF-only design demonstrates relatively high deletability,

while the SLBF with 2 CBFs shows the lowest. The high deletability of the CBF-only design can be attributed to its use of the entire memory budget to construct a single filter. In contrast, the SLBF with 2 CBFs allocates its memory across 2 filters to optimize for FPR. In this configuration, only the initial filter is responsible for handling deletions across all samples, while the backup filter contributes to deletability only for specific elements x such that $f_L(x) \leq t$. In summary, this comparison underscores a key design trade-off: optimizing for FPR can come at the cost of reduced deletability.



(a) FNR of LBF with 3 SBFs versus bits per element. The LBF with 3 SBFs is optimized with respect to the *sum of FNR and FPR.*

(b) FNR of LBF with 3 SBFs versus bits per element. The LBF with 3 SBFs is optimized with respect to the *FPR only.*

Figure 8: FNR of LBF with 3 SBFs versus bits per element.

Figure 8 highlights the extent to which optimizing for FPR can significantly compromise the performance of FNR. As shown in Figure 8b, when the LBF with 3 SBFs is optimized solely for FPR, it can achieve extremely low false positive rates for all b. However, this comes at a significant cost, as the filter becomes nearly unusable from $b = 19$ onward due to an FNR approaching 1. A closer examination of the bit allocation reveals that under FPR-focused optimization, the configuration effectively degenerates to a standard LBF, where the filters f_{D_1} and f_{D_2} receive minimal memory. This occurs because the optimizer likely determines that the most effective way to achieve optimal FPR, given a limited memory budget, is to allocate very little memory to f_{D_1} and f_{D_2}. By doing so, these filters exhibit extremely high FPR, which in turn leads to an extremely low FPR for the outer LBF.

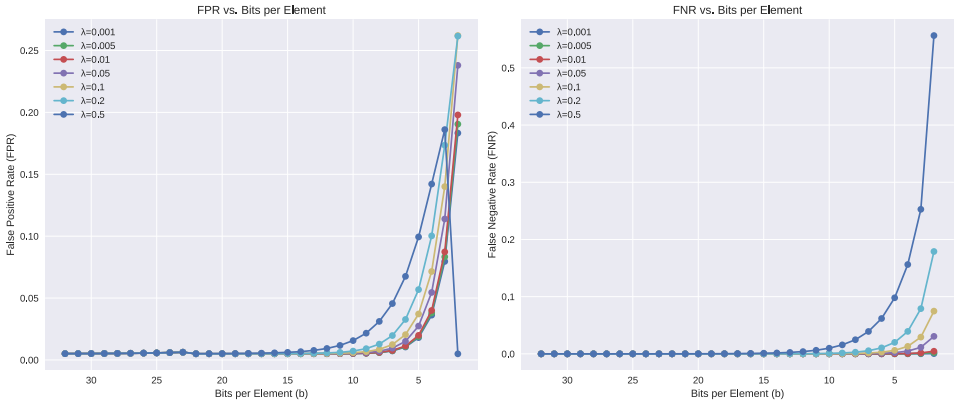


Figure 9: FPR and FNR of LBF with 3 SBFs versus bits per element for different λ. The LBF with 3 SBFs is optimized with respect to the sum of FPR and FNR.

We also perform a sensitivity analysis to examine how varying λ impacts the FPR and FNR of the LBF with 3 SBFs. The results, shown in Figure 9, reveal that for both metrics, higher values of λ cause performance degradation to start at higher b values. This is expected, as a larger λ requires more memory for f_{D_1} and f_{D_2}. As a result, its FPR and FNR begin to increase earlier as the available memory

budget decreases. Interestingly, for $\lambda = 0.5$, there is a sudden drop in FNR at $b = 1$. This is likely due to the limited memory budget at this point, prompting the optimizer to prioritize minimizing FPR over FNR. Specifically, it allocates a minimal number of bits to f_{D_2} to induce a high FPR for this filter, which, in turn, results in a low FPR for $f_L(x) \leq t$. The remaining bits are shifted to f_{D_1} to partially offset the FNR penalty.

Overall, when memory is abundant, both the SLBF with 2 CBFs and the LBF with 3 SBFs are viable, offering low FPR and, in the case of the LBF, low FNR. The LBF may be preferred when deletion support is critical. In contrast, when memory is limited, but still sufficient to construct the SLBF with 2 CBFs, and deletion requirements are minimal, the SLBF offers a strong balance of performance and deletability.

5 Related Work.

Adaptive Bloom Filter (Ada-BF). Similar to PLBF, Ada-BF [11] adopts the idea of partitioning classifier score ranges to reduce the FPR. However, instead of assigning a separate SBF to each score range, Ada-BF uses a single SBF and varies the number of hash functions per range. This design achieves a lower FPR than LBF and SLBF while maintaining the same memory footprint. Our proposed enhancements can also be applied to Ada-BF by replacing its SBF with a single CBF (like Definition 3.1) or incorporating a SBF (like Definition 3.3).

Deletable Bloom Filter (DIBF). To reduce the memory overhead introduced by CBF, DIBF introduces an additional bitmap to the SBF that marks whether a range of bits has a collision. When deleting an element, DIBF attempts to find one of the element's hashed positions that has no collision; if none exist, the deletion fails and the element remains in the filter, contributing to the FPR. While DIBF does not account for the presence of a learned model, which makes it less suitable for our setting, it can still be considered a memory-efficient alternative to CBF in our design.

Elastic Bloom Filter (EBF). Similar to DIBF, EBF [12] introduces an auxiliary structure to track collisions within regions. However, instead of using a single bit to indicate whether a bit range has a collision, EBF assigns a bucket to each bit in the SBF and stores a fingerprint of each input in the bucket associated with the bit it maps to. This additional structure enables efficient deletion: removing a fingerprint from a bucket and clearing the corresponding bit in the SBF if the bucket becomes empty (similar to the collision-free case in DIBF). EBF also supports dynamic expansion, which is triggered when inserting into a full bucket or when the number of set bits in the SBF exceeds a predefined threshold. However, similar to DIBF, EBF's deletion mechanism is not directly applicable to our setting. Moreover, we do not consider it a suitable alternative to CBF due to the additional memory overhead introduced by its bucket structure.

Adversary Resilient Bloom Filter. [13] analyzes the LBF under an adversarial model where the adversary can control a fraction of the queries and even access the internal representation of the LBF, with the goal of maximizing false positives. This security aspect of LBF, if ignored, can introduce vulnerabilities. As noted by [14], adversaries can potentially exploit SBF by crafting malicious queries to trigger denial-of-service attacks.

To address such threats, [13] proposes the Downtown Bodega Filter, a new construction based on a modified SLBF, which offers provable security in the adversarial model assuming the existence of a pseudo-random permutation. Their construction resembles our design with 2 CBFs (Definition 3.2). In theory, our construction can be adapted to this threat model by incorporating two additional secret keys for keyed pseudo-random permutation. This enables it to defend against query-only adversaries.

However, for our alternate design with 3 SBFs (Definition 3.3), the presence of false negatives makes its security guarantees more subtle. Developing a formal security model that accounts for these complexities is an open direction for future work.

Machine Unlearning. Machine unlearning tackles the challenge of selectively removing specific data points from trained machine learning models. *Exact unlearning methods*, such as retraining from scratch on the remaining data, offer complete data removal but are often computationally infeasible for large datasets. *Approximate unlearning methods* aim to address this by modifying the model through gradient-based techniques. For instance, Gradient Ascent (GA) [15] increases the loss for specific samples, effectively suppressing the model's ability to recall them. However, GA may unintentionally degrade performance on data that should be retained. Gradient Difference [15] mitigates this by combining GA and gradient descent, helping preserve performance on the retained data. Despite their promise, these methods generally lack formal forgetting guarantees and instead rely on empirical validation [16].

In our context, these unlearning techniques offer a potential avenue to directly modify the learned model for cases where it produces false positives. This could reduce or even eliminate the need for an additional Bloom Filter or lower the space required for such filters. However, the lack of formal guarantees and the computational cost of unlearning must be carefully considered before adopting this approach.

6 Discussion

6.1 Limitations

Memory Usage. The memory efficiency of all designs, except the final one, is suboptimal due to the reliance on CBFs. This directly affects the FPR of the resulting designs. For example, achieving the same FPR requires more memory in the SLBF with 2 CBFs (Definition 3.1) compared to the SLBF with 2 SBFs. This is primarily because the a^{b_0} term in the SBF case becomes $a^{\frac{b_0}{c}}$ in the CBF case, reflecting the increased memory demands of CBFs. As discussed in related work, replacing CBFs with DIBFs is a promising alternative. However, this comes at the cost of making some elements non-deletable, which may affect the overall FPR. Future work could explore how DIBFs influence both FPR and deletability.

Reinsertion. Reinsertion is as valuable as deletion in dynamic settings, where items may be deleted and later reinserted. However, in the LBF with 3 SBFs (Definition 3.3), reinsertion of deleted items is not supported. This limitation arises because the SBFs used to track deleted items do not themselves support deletions. Future work could investigate strategies to enable reinsertion while maintaining minimal memory overhead.

Limited Deletability. None of the proposed designs offers perfect support for deletion. In the SLBF with 2 CBFs (Definition 3.1), deletability depends on both the allocated memory and the empirical false negative rate of the learned model. Even in the LBF design with 3 SBFs (Definition 3.3), while it supports guaranteed deletion, this benefit comes at a cost: the FNR increases as a result. Consequently, to keep the FNR within acceptable bounds, the number of deletions performed in practice should be purposefully limited. Nonetheless, such limitations are inherent to Bloom Filter due to its compact and probabilistic nature, which intentionally trade exactness for space efficiency. That said, future work could investigate whether better trade-offs exist to enhance deletability without compromising performance or memory efficiency.

Evaluation. The evaluation of our designs could be further strengthened by incorporating real-world datasets and exploring a broader range of parameter configurations. This would help demonstrate the robustness and effectiveness of our designs across diverse application scenarios.

6.2 Future Work

SLBF with 2 CBFs vs. LBF with 2 CBFs. In Section 3.4, we examined how to optimize LBF with 2 CBFs in scenarios where constructing an SLBF is not feasible due to limited memory. However, we did not analytically determine whether the LBF with 2 CBFs can outperform the SLBF with 2 CBFs when both are constructible. Future work could derive the precise conditions under which the LBF variant is preferable and assess their practicality. Such analysis could also generalize to comparisons between LBF with 2 SBFs and the standard SLBF with 2 SBFs. This direction may benefit from adopting the strategy used in Section 4 and 5.2 of [8].

More Design Options. Given the expansive design space, we have not explored all possible design strategies. Future research could investigate approaches that use both CBF and SBF concurrently and experiment with different filter positions. Additionally, we have not yet addressed the general case of SLBF with 3 CBFs (Section 3.3). Future work may focus on solving this case, potentially eliminating the need for manual exploration of various configurations.

Balance Tradeoffs. For our final design LBF with 3 SBFs (Definition 3.3), we evaluated its FPR, deletability, and FNR. However, one aspect not addressed analytically is the trade-off between FPR and FNR under a fixed total memory budget and dataset size. This analysis is particularly challenging due to the inherently conflicting objectives of minimizing both FPR and FNR, and the need to determine an optimal memory allocation strategy across the 3 filters. Similarly, while the SLBF with 2 CBFs (Definition 3.1) achieves the best FPR, it exhibits the worst deletability. Balancing such trade-offs across different design choices remains an open question for future work.

Generalizability to PLBF. Although our design focuses on SLBF and LBF, some aspects are applicable to PLBF. For example, CBFs could be used to replace all SBFs in PLBF and compared against PDDBF [9]. Additionally, one could consider adding an SBF for each CBF in the PLBF, or incorporate SBF into the objective and constraints of PLBF's optimization problem to balance the resulting FPR and FNR.

7 Conclusion

This work presents two new designs to enable deletion in Learned Bloom Filters (LBFs) and Sandwiched Learned Bloom Filters (SLBFs): SLBF with 2 Counting Bloom Filters (CBFs) and LBF with 3 Standard Bloom Filters (SBFs). These designs offer different trade-offs: SLBF with 2 CBFs achieves the lowest false positive rate (FPR), while LBF with 3 SBFs ensures guaranteed deletability at the cost of slightly increased false negative rate (FNR).

Our evaluation confirms these trade-offs and highlights that no single design dominates across all metrics. Future work could explore reinsertion mechanisms, more memory-efficient structures, and analytical conditions under which different designs outperform each other. Multi-objective optimization may also help balance FPR and FNR under fixed memory budgets.

The code and data used in this study are included in the attached zip file, but not publicly released.

8 AI Usage Declaration

AI is used to improve the grammar and flow of report writing, as well as to double-check the quality and correctness of derivations.

Bibliography

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970, doi: 10.1145/362686.362692.

[2] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and*

Protocols for Computer Communications, in SIGCOMM '03. Karlsruhe, Germany: Association for Computing Machinery, 2003, pp. 201–212. doi: 10.1145/863955.863979.

[3] B. M. Maggs and R. K. Sitaraman, "Algorithmic Nuggets in Content Delivery," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 3, pp. 52–66, Jul. 2015, doi: 10.1145/2805789.2805800.

[4] J. Yan and P. L. Cho, "Enhancing Collaborative Spam Detection with Bloom Filters," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006, pp. 414–428. doi: 10.1109/ACSAC.2006.26.

[5] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area Web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000, doi: 10.1109/90.851975.

[6] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The Case for Learned Index Structures," in *Proceedings of the 2018 International Conference on Management of Data*, in SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 489–504. doi: 10.1145/3183713.3196909.

[7] K. Vaidya, E. Knorr, T. Kraska, and M. Mitzenmacher, "Partitioned Learned Bloom Filter." [Online]. Available: https://arxiv.org/abs/2006.03176

[8] M. Mitzenmacher, "A model for learned bloom filters, and optimizing by sandwiching," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, in NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 462–471.

[9] M. Zeng *et al.*, "Two-layer partitioned and deletable deep bloom filter for large-scale membership query," *Information Systems*, vol. 119, p. 102267, 2023, doi: https://doi.org/10.1016/j.is.2023.102267.

[10] A. Sato and Y. Matsui, "Fast partitioned learned bloom filter," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, in NIPS '23. New Orleans, LA, USA: Curran Associates Inc., 2023.

[11] Z. Dai and A. Shrivastava, "Adaptive learned bloom filter (Ada-BF): efficient utilization of the classifier with application to real-time information filtering on the web," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, in NIPS '20. Vancouver, BC, Canada: Curran Associates Inc., 2020.

[12] Y. Wu *et al.*, "Elastic Bloom Filter: Deletable and ExpandableFilter Using Elastic Fingerprints," *IEEE Transactions on Computers*, p. 1, Mar. 2021, doi: 10.1109/TC.2021.3067713.

[13] A. Bishop and H. Tirmazi, "Adversary Resilient Learned Bloom Filters." [Online]. Available: https://eprint.iacr.org/2024/754

[14] T. Gerbet, A. Kumar, and C. Lauradoux, "The Power of Evil Choices in Bloom Filters," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 101–112. doi: 10.1109/DSN.2015.21.

[15] P. Maini, Z. Feng, A. Schwarzschild, Z. C. Lipton, and J. Z. Kolter, "TOFU: A Task of Fictitious Unlearning for LLMs." [Online]. Available: https://arxiv.org/abs/2401.06121

[16] T. Al Mahmud, N. Jebreel, J. Domingo-Ferrer, and D. Sánchez, "Dp2unlearning: An Efficient and Guaranteed Unlearning Framework for Llms," 2025, doi: 10.2139/ssrn.5217160.