General Adaptive Memory Allocation for Learned Bloom Filters

You Shang
Chongqing University
Chongqing, China
you.shang@stu.cqu.edu.cn

Xiang He
Chongqing University
Chongqing, China
hxcquer@163.com

Ruiyuan Li*
Chongqing University
Chongqing, China
ruiyuan.li@cqu.edu.cn

Yingying Sun
Chongqing University
Chongqing, China
Yingying.Sun@cqu.edu.cn

Guanyao Li
Beijing Normal-Hong Kong Baptist University
Zhuhai, China
guanyaoli@uic.edu.cn

Guangchao Yang
Chongqing University
Chongqing, China
gchao_yang@cqu.edu.cn

Junbo Zhang
JD Intelligent Cities Research
JD iCity, JD Technology
Beijing, China
msjunbozhang@outlook.com

Yu Zheng
JD Intelligent Cities Research
JD iCity, JD Technology
Beijing, China
msyuzheng@outlook.com

ABSTRACT

Membership testing, which determines whether an element belongs to a set, is widely used in fields like database systems and network applications. Bloom Filters (BFs) can solve this problem efficiently but suffer from high False Positive Rates (FPRs) and large memory requirements for massive datasets. Learned Bloom Filters (LBFs), combining a learning model with a backup Bloom Filter, mitigate these issues by capturing data distributions. However, the critical problem of memory allocation between the learning model and the backup filter has usually been overlooked, despite its significant impact on LBF performance under constrained budgets.

To this end, we propose Gama, the first General Adaptive Memory Allocation framework for LBFs as far as we know. Gama introduces two memory allocation strategies: Loop-Based method and Bayesian-Based method. Loop-Based method evaluates all configurations at each training epoch, making it well-suited for scenarios with tight memory constraints. However, it faces efficiency challenges under large memory budgets due to the requirement for exhaustive evaluations. In contrast, Bayesian-Based method efficiently navigates the search space through probabilistic exploration, which reduces the number of configurations evaluated and significantly improves the efficiency while maintaining FPRs. Furthermore, we propose a hybrid approach that combines their strengths to dynamically adapt to different constraints. Experiments on three real-world datasets show that Gama can achieve a relative performance improvement of 69% in terms of FPR in the best case.

KEYWORDS

Learned Bloom Filter, Approximate Membership Testing, Bayesian Optimization

*Corresponding author

CIKM '25, November 10–14, 2025, Seoul, Republic of Korea.

ACM ISBN 979-8-4007-2040-6/2025/11
https://doi.org/10.1145/3746252.3761196

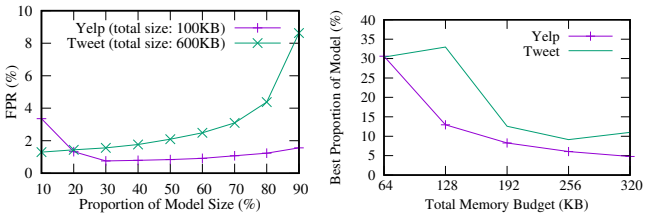
ACM Reference Format:
You Shang, Xiang He, Ruiyuan Li, Yingying Sun, Guanyao Li, Guangchao Yang, Junbo Zhang, and Yu Zheng. 2025. General Adaptive Memory Allocation for Learned Bloom Filters. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management (CIKM '25), November 10–14, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3746252.3761196

1 INTRODUCTION

Membership testing, the pivotal problem of determining whether an item x belongs to a set S, plays a critical role in accelerating query-driven tasks across multiple domains like database systems [17, 28, 29] and networking [1, 5, 6, 10, 12, 14, 15, 18, 26, 30, 36, 38, 45]. For example, in database systems, membership testing accelerates point query execution by pre-filtering non-existent items, reducing I/O and network overhead [6]. A key challenge in these scenarios is managing the trade-off between memory consumption and query performance. To address this, approximate data structures like Bloom Filters (BFs) [3] have become an essential tool for accelerating membership testing under resource constraints.

Traditional BF [3] utilizes several hash functions to map each item into certain bits in a bitmap. However, BF cannot capture the distribution of a given set, leading to limited performance in terms of FPR (false positive rate). When the size of dataset is large and the memory budget is constrained, the FPR of a BF could be rather high. Although there are numerous BF variants [11, 13, 16, 22, 27, 43, 44, 48, 51] proposed, they have not made a notable breakthrough yet.

To break this limitation, Learned Bloom Filter (LBF) [24] incorporates a learning model to capture the data distribution, significantly reducing FPR under the same memory budget. An LBF consists of a learning model that predicts membership likelihood and one or more backup Bloom Filters to handle uncertain predictions. However, existing works largely focus on optimizing the structure of LBFs [8, 35, 42], but rarely consider the memory allocation between the learning model and backup BF(s). We find that memory allocation imposes significant effects on the overall performance of an LBF. As shown in Fig. 1(a), given a memory budget, when the memory proportion of the learning model changes from 10% to 90%, the FPRs on two real-world datasets vary significantly. Therefore, it is of great importance to find the best memory allocation strategy for an LBF under a given total memory budget.



(a) FPR vs Various Allocation. (b) Best Allocation vs Various Budgets.

Figure 1: Motivation of Memory Allocation for LBF.

It is challenging to employ the best memory allocation on LBFs. **Challenge I**, the relationship between the memory allocation and the overall performance of an LBF is complicated. On one hand, as shown in Fig. 1(b), for a certain dataset, under different total memory budgets, the best memory allocation strategies are different. On the other hand, for different datasets, under the same memory budget, their best memory allocation strategies are different as well. **Challenge II**, how to train a learning model that occupies a given amount of memory? We should make full use of the given memory to build a learning model to fit the dataset. **Challenge III**, memory allocation proportion is continuous, and the learning model along with its hyper-parameters can be also considered as continuous variables. Enumerating the entire parameter space in a brute-force manner is time-consuming, if not infeasible.

To this end, this paper proposes Gama, the first holistic **G**eneral **A**daptive **M**emory **A**llocation framework for LBFs to the best of our knowledge. The objective of Gama is to minimize the overall FPR of an LBF under a given memory budget. Specifically, to address **Challenge I**, given a dataset and a memory budget, Gama employs a multiple-epoch process to fully explore the possible memory allocation strategies, and select the best LBF from the constructed LBFs. To resolve **Challenge II**, in each epoch, instead of first assigning some memory and then training a learning model, Gama first determines the structure (along with the parameters) of the learning model that occupies the fixed memory space, and then assigns the left memory to the backup BF(s). To avoid the time-consuming process of constructing LBFs, we propose a method that does not require the actual construction of the LBFs (Section 3.2). To address **Challenge III**, we make the best of the training process to change the memory occupation of the learning model, thus avoiding enumerating the infinite and continuous memory allocation strategies and parameters. Moreover, we transform the exploration process into a knob tuning problem and introduce Bayesian Optimization [37] to further enhance Gama's efficiency. In specific, Gama introduces two memory allocation strategies: Loop-Based method, which ensures precise control over memory utilization by incrementally training, and Bayesian-Based method that uses probabilistic exploration to optimize memory allocation, which enables it to achieve FPRs close to those of Loop-Based method under large memory budgets while significantly improving efficiency. To combine their strengths, Gama also includes a hybrid method, which balances efficiency and FPR better. Overall, the contributions of this paper are as follows:

(1) We propose Gama, the first general adaptive memory allocation framework for LBFs. Gama introduces two strategies: Loop-Based method, offering precise memory allocation and suitable for tight memory budgets, and Bayesian-Based method, exploring the memory allocation more efficiently and achieving close FPRs under large memory budgets.

(2) We develop a hybrid method that combines the strengths of both strategies, allowing for effective optimization of the LBFs' FPR across various memory budgets.

(3) We propose a novel method to quickly determine the optimal parameter of LBF and its corresponding FPR, eliminating the need to fully construct the LBF, thus reducing evaluation overhead.

(4) We conduct extensive experiments on three real-world datasets to evaluate the effectiveness of Gama. In the best case, Gama achieves a relative FPR improvement of 69% over the original LBF.

The remainder of this paper is organized as follows. In Section 2, we give some preliminaries. We introduce our proposed Gama in detail in Section 3 and Section 4, corresponding to Loop-Based and Bayesian-Based Memory Allocation, respectively. In Section 5, we discuss how to combine the advantages of Loop-Based and Bayesian-Based Memory Allocation and how Gama can be applied to other LBF variants. The experimental results are presented in Section 6. In Section 7, we summarize the related works. Finally, we conclude this paper in Section 8.

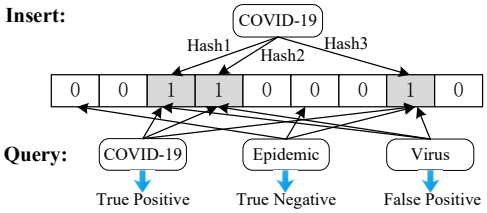


Figure 2: Bloom Filter.

2 PRELIMINARIES

In this section, we first give some background knowledge, and then present the problem definition.

2.1 Background Knowledge

Bloom Filter. Bloom Filter (BF) is used to determine whether the item x is in set S. It consists of a fixed-size bitmap and k independent hash functions. Initially, all bits of the bitmap are set to 0. As shown in Fig. 2, when an item is inserted into the set, k hash functions are used to map the item to k bits in the bitmap, setting them to 1. In order to determine whether the item x exists, it uses the same k hash functions to map x to k bits of the bitmap, and then checks whether all corresponding bits are 1. If so, it is highly possible that x exists in S. The situation where the item exists and the BF determines that it exists is called a **true positive** (e.g., "COVID-19" in Fig. 2). If there is at least one 0 bit, x definitely does not exist in S (e.g., "Epidemic" in Fig. 2), which is called a **true negative**. Since the hash function has a certain collision probability, when all corresponding bits are 1, x may not exist in S (e.g., "Virus" in Fig. 2), which is called a **false positive**. BF does not have any **false negative**. Given a total size μ_B of bitmap and k hash functions, if we insert n_B items into a BF, then its FPR F_B is expected to be [23]:

$$F_B = [1 - (1 - 1/\mu_B)^{kn_B}]^k \approx (1 - e^{-kn_B/\mu_B})^k \quad (1)$$

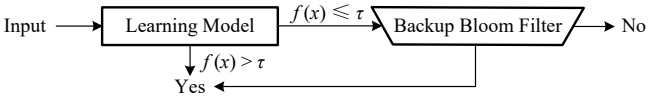


Figure 3: Structure of Learned Bloom Filter.

When choosing near-optimal $k = \mu_B \cdot ln2/n_B$, we have:

$$F_B = (0.5)^{\mu_B \cdot ln2/n_B} \tag{2}$$

Learned Bloom Filter. BF occupies a large amount of memory and has a high FPR when the dataset is large, so Learned Bloom Filter (LBF) is proposed, which combines a learning model and a backup BF. As shown in Fig. 3, by using some available training data for pre-training, the learning model can determine whether any given query item x belongs to the set S based on the characteristics of the data. In specific, LBF first sets a threshold τ. Then the item x is sent into the learning model, obtaining the prediction score $f(x)$. If $f(x) > \tau$, x is considered to exist in the set S. Otherwise, x will be sent into the backup BF for further determination. The FPR F_L of an LBF is defined as:

$$F_L = F_M + (1 - F_M) \cdot F_B \tag{3}$$

where F_M represents the FPR of the learning model and F_B is the FPR of the backup BF. This formula means that: for a negative item, there are two ways it can be mistakenly classified as a positive item. First, the learning model may predict it as positive with a probability of F_M. Second, the learning model may predict it as negative, but the BF incorrectly identifies it as positive (i.e., $(1 - F_M) \cdot F_B$).

There is a trade-off between the learning model and the backup BF. Intuitively, if the learning model is complex enough, its prediction accuracy tends to be high, leading to a small F_M. However, a more complex learning model usually occupies larger memory, hence resulting in less memory allocated to the backup BF. Therefore, the FPR F_B of the backup BF is supposed to be larger according to Equ. (2). In this paper, we propose a framework to find the optimal memory allocation for the learning model and backup BF.

2.2 Problem Definition

Given a memory budget μ, the memory allocation problem on LBFs aims to find the optimal combination of learning model memory μ_M^* and BF memory μ_B^*, such that the overall FPR F_L of the LBF is minimized, i.e.,

$$min\ F_L \quad s.t.\ \ \mu_M^* + \mu_B^* \leq \mu, \mu_M^* \geq 0, \mu_B^* \geq 0 \tag{4}$$

3 LOOP-BASED MEMORY ALLOCATION

To construct a Learned Bloom Filter, we need to determine the model structure and the threshold. In this section, we propose the basic Loop-Based memory allocation method consisting of α epochs, as shown in Fig. 4. In each epoch, there are two main steps, i.e., *Model Training* and *Threshold Selection*. In *Model Training* step, we construct a learning model and train it to fit the training dataset. In *Threshold Selection* step, we select the optimal threshold with the minimum FPR in this epoch, forming the locally best learning model. After α epochs, we select the globally best learning model among all locally best learning models.

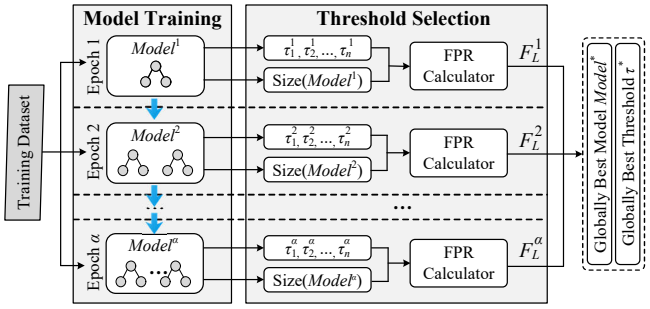


Figure 4: Framework of Loop-Based Memory Allocation.

3.1 Model Training

Model Training involves two main tasks: training a model that can efficiently and effectively perform binary classification, and adjusting the memory proportion of the learning model during the training process. In this step, we focus on the choice of learning model and how memory is allocated between different epochs.

Choice of Learning Model. A good learning model in an LBF should meet the following requirements. First, the model should be lightweight enough, i.e., it takes up little memory. Second, the accuracy of the learning model should be as high as possible for binary classification tasks. Third, the model should enlarge itself as epochs increase to avoid retraining it from scratch.

As a light-weight gradient boosting decision tree model, LightGBM [21] can meet all of the three requirements. First, LightGBM applies a histogram algorithm, binning continuous feature values into discrete bins to reduce its memory consumption. Second, the structure of LightGBM ensures its good ability to handle the classification tasks. Third, its gradient boosting algorithm counts in studying the residual to enlarge itself, which makes full use of the knowledge of the last epoch without retraining the model from scratch. Therefore, we choose LightGBM as our default model.

Memory Variation Between Epochs. Next, we introduce how we alter the memory consumption of LightGBM between epochs. In each iteration, LightGBM trains a new decision tree and then inserts it into the model. More exactly, during training, LightGBM splits the best nodes that it found with the constraints of maximum tree depth. The growing number of nodes and trees makes the size of the model larger, and choosing the leaf nodes that decrease the loss function most makes sure that we get the best LightGBM model. Hence, as the iteration goes on, the size of the learning model gets larger, and we can enumerate the possible memory allocations.

3.2 Threshold Selection

In each epoch i, after performing *Model Training* step, we obtain a learning model $Model^i$ that well fits the training dataset. In this step, we need to find an optimal threshold that bridges the learning model and the backup BF, determining which items need to be inserted into the backup BF. To achieve this, one intuitive idea is to evenly partition the prediction score space into several regions, and then select from them a threshold that yields the best performance. However, we find this method leads to a poor performance. The reason could be that the distribution of prediction scores is extremely

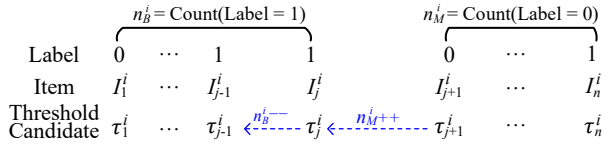


Figure 5: n_M^i and n_B^i Calculation ($\tau_j^i \le \tau_{j+1}^i$ for $1 \le j < n$).

uneven. If we select a threshold from the evenly-partitioned regions, we may miss many candidates in the dense regions.

To this end, we propose to select the threshold as follows. For each item j in the dataset, $Model^i$ can produce a prediction score τ_j^i. We select from $\{\tau_1^i, \tau_2^i, ..., \tau_n^i\}$ the best threshold τ^i that contributes to the minimum FPR F_L^i of the LBF, where n is the total number of items in the training dataset. For each threshold τ_j^i, it requires to calculate the FPR $F_{L,j}^i$ of the LBF. The intuitive method is to insert all items in the dataset into the LBF, and then measure what the FPR is. However, this method may suffer from two drawbacks. First, for any item that is judged not to exist by $Model^i$, it would be actually inserted into the backup BF, which is relatively time-consuming. Second, there are n possible threshold candidates, and for each candidate, we might insert at most n items into the backup BF, leading to a time complexity of $O(n^2)$.

To accelerate the FPR calculation, we make two optimizations. **Opt I**: We do not actually insert the items into the backup BF. As depicted in Equ. (3), given a threshold τ_j^i in epoch i, to obtain the FPR $F_{L,j}^i$ of the LBF, we only need to know the FPR F_M^i caused by $Model^i$ and the FPR F_B^i caused by the backup BF. F_M^i can be calculated by n_M^i/n_{neg}, where n_M^i is the number of negative items but $Model^i$ regards them as positive, and n_{neg} is the total number of negative items in the training dataset. F_B^i can be calculated by Equ. (2) if we know the number n_B^i of items inserted into the backup BF and the memory size μ_B^i of the backup BF. μ_B^i is assigned by $\mu - Size(Model^i)$, where μ and $Size(Model^i)$ are the memory budget and the memory occupation of $Model^i$, respectively. **Opt II**: We propose a sorting-based method to calculate n_M^i and n_B^i efficiently. First, we sort the items in the training dataset ascendingly by the prediction scores. Then, we traverse the threshold candidates from large to small. Fig. 5 gives an example. Suppose τ_j^i is the current candidate for checking. n_B^i is the number of items labeled as existing whose prediction scores are smaller than or equal to τ_j^i, while n_M^i is the number of items labeled as not existing whose prediction scores are greater than τ_j^i. Since the candidates are already sorted, the results of τ_j^i can be fully utilized for the results of τ_{j-1}^i, which accelerates the overall calculation. As shown in Fig. 5, when checking τ_j^i, as the label of the item I_{j+1}^i is 0, we can remain n_B^i unchanged and increase n_M^i by 1 based on the results of τ_{j+1}^i. Similarly, when checking τ_{j-1}^i, as the label of the item I_j^i is 1, we can maintain n_M^i unchanged and decrease n_B^i by 1 based on the results of τ_j^i.

3.3 Summary

Algorithm 1 presents the pseudo-code of Loop-Based Memory Allocation. Lines 1-3 are the initialization process, where we start with a pure classical BF without any learning model. μ_M^* and μ_B^* record the currently best memory sizes for the learning model and the backup BF, respectively, and F_L^* is the currently smallest FPR. Lines 4-15 are epoch loops, each of which consists of two steps: *Model Training* (Lines 5-6) and *Threshold Selection* (Lines 7-15). Note that in the *Model Training* step, if the size of $Model^i$ is larger than the budget μ, we break the loop (Line 6) and return the final result (Line 16). In *Threshold Selection* step, we start j in $(n-1)$ instead of in n (Line 9), because $j = n$ is equivalent to the case where we use a pure classical BF.

Suppose the number of epochs is α. As we need to sort the prediction scores in each epoch, the time complexity of Algorithm 1 is $O(\alpha \cdot n \cdot logn)$. Besides, we need to record the prediction scores, so its space complexity is $O(n)$.

Algorithm 1: Loop-Based Memory Allocation

Input: Training Dataset D, Memory Budget μ
Output: Best Memory Allocation (μ_M^*, μ_B^*)

```
/* Initialization                                                    */
1  n_neg ← |{I ∈ D|I.Label = 0}|; n_B^0 ← |{I ∈ D|I.Label = 1}|;
2  μ_M^* ← 0; μ_B^* ← μ; n ← |D|;
3  F_L^* ← (0.5)^{μ_B^* · ln2 / n_B^0}; // Equ. (2), the FPR of a pure BF
4  while true do // Suppose the number of current epoch is i
       /* Model Training                                             */
5      Train a learning model Model^i that fits D based on Model^{i-1};
6      if Size(Model^i) > μ then break;
       /* Threshold Selection                                        */
7      Get and sort the prediction scores {τ_1^i, τ_2^i, ..., τ_n^i} of items in D;
8      μ_B^i ← μ − Size(Model^i); n_M^i ← 0; n_B^i ← n_B^0;
9      for j ← (n − 1) to 1 do
10         if I_{j+1}^i.Label = 0 then n_M^i ++ ;
11         else n_B^i − − ;
12         F_M^i ← n_M^i / n_neg; F_B^i ← (0.5)^{μ_B^i · ln2 / n_B^i}; // Equ. (2)
13         F_L^i ← F_M^i + (1 − F_M^i) · F_B^i; // Equ. (3)
14         if F_L^i < F_L^* then
15             F_L^* ← F_L^i; μ_M^* ← Size(Model^i); μ_B^* ← μ_B^i;
16 return (μ_M^*, μ_B^*);
```

4 BAYESIAN-BASED MEMORY ALLOCATION

4.1 Motivation

Loop-Based memory allocation method can effectively find the optimal allocation strategy under the memory budget μ. However, if μ is large enough, there might be excessive epochs, which requires too much time to obtain a good memory allocation strategy. Note that the construction efficiency of LBF is vital for some critical scenarios, e.g., in database systems, if the datasets are updated, we need to rebuild the LBF efficiently to guarantee the correctness of the query results and the availability of the systems.

The memory allocation problem of LBFs can be also deemed as a knob tuning problem. Therefore, we can leverage the knob tuning methods for this problem. Given a memory budget μ, we try to find a proportion $\eta = \mu_M^*/\mu$ for the learning model that minimizes the FPR of the LBF, where $\eta \in [0, 1]$ is a continuous variable. However, as described in Section 1, it is challenging to train a learning model that occupies exactly a given amount of memory. To this end, instead of finding a good proportion η, we propose to search for the optimal number of epochs in Algorithm 1.

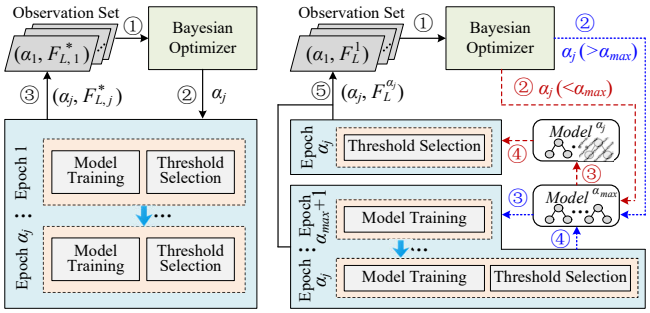


Figure 6: Bayesian-Based Memory Allocation.

4.2 Intuitive Method

Bayesian Optimizer [49] is widely-used in knob tuning problems. Compared with reinforcement learning-based methods and deep learning-based methods [50], it requires fewer samples to efficiently achieve high-quality knob settings, which makes it particularly suitable for the scenarios where the knob number is small. In our problem, there is only one parameter (i.e., α) for tuning. Besides, it requires fast prediction and small memory occupation. Therefore, we adopt Bayesian Optimizer as our knob tuning method.

Fig. 6(a) presents the intuitive method, which is an iterative process. First, Bayesian Optimizer predicts a number α_j of epochs based on the current observation set. Then, we launch a Loop-Based Memory Allocation process with α_j epochs, obtaining the globally minimum FPR $F^*_{L,j}$. Third, we add $(\alpha_j, F^*_{L,j})$ to the observation set. The three steps are repeated until the maximum number N_{BO} of iterations is reached. Finally, the number α^* corresponding to the minimum FPR F^*_L in the observation set is returned[1].

4.3 Proposed Method

There are two shortcomings in the intuitive method. First, for each predicted α_j, the Loop-Based Memory Allocation method would trigger α_j epochs from scratch, which is still time-consuming. Second, in each epoch, we need to perform the Threshold Selection step, in which many costly predictions and an expensive sorting operation are executed.

To address these issues, we propose a novel Bayesian-Based memory allocation method, as shown in Fig. 6(b). Compared with the intuitive method shown in Fig. 6(a), there are three main optimizations. **Opt I**: We make full use of the historical model structures. Note that we adopt LightGBM as the learning model, so $Model^{i+1}$ is obtained by simply adding a new decision tree to $Model^i$, but keeping the former i decision trees unchanged. Therefore, we cache the model structure $Model^{\alpha_{max}}$ obtained by the historically maximum number α_{max} of epochs. Suppose in the current iteration, the Bayesian Optimizer recommends a epoch number of α_j. If $\alpha_j < \alpha_{max}$, we can obtain $Model^{\alpha_j}$ by simply removing the last $(\alpha_{max} - \alpha_j)$ decision trees in $Model^{\alpha_{max}}$, i.e., the red dashed arrows in Fig 6(b). If $\alpha_j > \alpha_{max}$, we first get $Model^{\alpha_j}$ based on $Model^{\alpha_{max}}$ through only $(\alpha_j - \alpha_{max})$ epochs, and then replace $Model^{\alpha_{max}}$ with $Model^{\alpha_j}$, i.e., the blue dotted arrows in Fig 6(b). **Opt II**: We skip some operations in some epochs. As shown in Fig 6(b), if $\alpha_j < \alpha_{max}$, we only perform Threshold Selection step in epoch α_j. If $\alpha_j > \alpha_{max}$, we only perform Model Training step between epoch $(\alpha_{max}+1)$ and epoch $(\alpha_j - 1)$. **Opt III**: We change the components of an observation. We regard $(\alpha_j, F_L^{\alpha_j})$ as the new observation, where $F_L^{\alpha_j}$ is the FPR of the LBF built in epoch α_j. This is because we could not obtain the global minimum FPR if we employ the previous two optimizations. However, as our experiments show, this change can also lead us to the best memory allocation strategy.

[1]In fact we return (μ^*_M, μ^*_B) that can be uniquely determined by α^*.

Algorithm 2: Bayesian-Based Memory Allocation

Input: Training Dataset D, Memory Budget μ, Iteration Num N_{BO}
Output: Best Memory Allocation (μ^*_M, μ^*_B)

```
/* Initialization */
1  n_neg ← |{I ∈ D|I.Label = 0}|; n^0_B ← |{I ∈ D|I.Label = 1}|;
2  μ*_M ← 0; μ*_B ← μ; n ← |D|; F*_L ← (0.5)^{μ*_B·ln2/n^0_B}; // Equ. (2)
3  α_max ← 0; OS ← {(0, F*_L)}; // Observation Set
4  for j ← 1 to N_BO do // Bayesian Iteration
5      Predict α_j using Bayesian Optimizer based on OB;
6      if α_j > α_max then
7          for i ← (α_max + 1) to α_j do
               /* Model Training */
8              Train a model Model^i that fits D based on Model^{i-1};
9              if Size(Model^i) > μ then break;
10             α_max ← i;
11         Cache Model^{α_max};
           /* Threshold Selection */
12         Get F_L^{α_max} using Lines 7-13 in Algorithm 1;
13         if F_L^{α_max} < F*_L then
14             F*_L ← F_L^{α_max};
15             μ*_M ← Size(Model^{α_max}); μ*_B ← μ − μ*_M;
16         Add (α_max, F_L^{α_max}) to OB;
17     else if α_j < α_max then
18         Obtain Model^{α_j} from Model^{α_nax};
           /* Threshold Selection */
19         Get F_L^{α_j} using Lines 7-13 in Algorithm 1;
20         if F_L^{α_j} < F*_L then
21             F*_L ← F_L^{α_j}; μ*_M ← Size(Model^{α_j}); μ*_B ← μ − μ*_M;
22         Add (α_j, F_L^{α_j}) to OB;
23 return (μ*_M, μ*_B);
```

Algorithm 2 shows the pseudo-code of Bayesian-Based Memory Allocation. In Lines 1-3, it is similar to that of Algorithm 1, but we also initialize the current maximum number α_{max} of recommended epochs and the observation set OB. Lines 4-22 show the iteration process of Bayesian Optimizer, the logic of which is similar to that of Fig. 6(b). Note that we do not consider the case where $\alpha_j = \alpha_{max}$, since we can avoid this case through Bayesian Optimizer according to the observation set. We need to train the model for at most α_{max} times, and perform the Threshold Selection step at most N_{BO} times. Therefore, the time complexity of Algorithm 2 is $O(\alpha_{max} + N_{BO} \cdot n \cdot logn)$, where n is the number of items in the training dataset. If we ignore the memory occupied by the cached model (Algorithm 1 should also store one model for the next model training), the space complexity of Algorithm 2 is the same as that of Algorithm 1, i.e., $O(n)$.

5 DISCUSSION

5.1 Hybrid of the Two Proposed Methods

Bayesian-Based method is efficient by avoiding many Threshold Selection steps when the memory budget is relatively large. However, if the memory budget is too small, the additional computational overhead introduced by Bayesian Optimizer will exceed the saved overhead, leading to a poorer performance than Loop-Based method. To this end, we propose a hybrid method that combines the strengths of both Loop-Based method and Bayesian-Based method. The rationale is that we can already achieve a satisfactory memory allocation strategy with few epochs (e.g., ≤ 5) using Loop-Based method if the memory budget is small enough, so we do not need the Bayesian Optimizer. Another advantage is that we can initialize the observation set and the cached model with the results of the Loop-Based method, which is helpful for Bayesian Optimizer.

Algorithm 3: Hybrid-Based Memory Allocation

Input: Training Dataset D, Memory Budget μ, Iteration Num N_{BO}, Maximum Epoch Num N_{LB} of Loop-Based Method
Output: Best Memory Allocation (μ_M^*, μ_B^*)

Perform Loop-Based method with at most N_{LB} epochs;
if Loop-Based method has already reached memory budget μ **then**
  return (μ_M^*, μ_B^*) obtained from Loop-Based method;
$\alpha_{max} \leftarrow N_{LB}$;
Initialize $Model^{\alpha_{max}}$ and OS obtained from Loop-Based method;
Perform Bayesian-Based method with at most N_{BO} iterations;
return (μ_M^*, μ_B^*) obtained from Bayesian-Based method;

Algorithm 3 shows the pseudo-code of Hybrid-Based Memory Allocation, which is self-explanatory. In our paper, we set $N_{LB} = 5$. Since N_{LB} is usually small, i.e., the overhead of Loop-Based method can be ignored, the time complexity and space complexity of Algorithm 3 are the same as those of Algorithm 2.

5.2 Other Learning Model Structures

We use LightGBM as the default learning model, due to its light weight and capacity for incremental growth across different epochs. However, our framework is model-agnostic. For instance, our framework operates by adding neurons to an Artificial Neural Network (ANN) or decision trees to a Random Forest [4]. A performance comparison across these model structures is presented in Section 6.

5.3 Gama on LBF Variants

Throughout this paper, we mainly describe our Gama framework on the classical LBF. There are also other LBF variants [8, 35, 42], where a learning model is followed by multiple BFs. The biggest difference between these variants and the classical LBF is that there are multiple thresholds to determine which BF should be inserted into for the items. Therefore, for these variants, their model training steps keep unchanged, while the Threshold Selection steps need to be adjusted accordingly to find the best combination of the threshold candidates. The paper [39] proposes a dynamic programming-based method to find the best combination of thresholds efficiently. For all the variants mentioned above, we have experimentally validated the effectiveness and stability of Gama on them. Section 6.3 provides a detailed presentation of the experimental setup and results for Gama on the LBF variants.

5.4 Limitation of Gama

Since LBFs do not support item deletions, Gama on LBFs can not either. Besides, if the distribution of upcoming items is different from their historical distribution, the performance of LBFs (along with Gama) will deteriorate. All these limitations stem from the intrinsic shortcomings of LBFs. One optional solution is to rebuild the LBFs. Since Gama includes multiple strategies to accelerate the reconstruction, these limitations motivate Gama better.

6 EVALUATIONS

6.1 Experimental Settings

Datasets. We assess the performance of our proposed framework Gama using three real-world datasets: Celestial Objects Dataset (COD) [25], Yelp [20] and URL [40]. Specifically, COD, derived from the Sloan Digital Sky Survey (SDSS), comprises classifications of celestial objects based on their optical and spectroscopic properties. Yelp, containing user reviews of businesses, is used to analyze consumer feedback and sentiment. URL comprises malicious and benign URLs. The Yelp dataset we use is filtered to remove the extremely sparse regions from the original dataset to reduce the length of the encoding. All evaluations use the URL dataset by default. Table 1 lists the number of samples.

Table 1: Information of Datasets

Dataset	Yelp	URL	COD
# Positive Samples	384,602	223,088	2,498,077
# Negative Samples	249,886	428,118	2,980,291

Metrics. We evaluate the performance using two metrics: FPR and construction time. FPR measures the performance of the LBF. Construction time, on the other hand, assesses the efficiency of the model in terms of the time required to build the data structure.

Settings. By default, we use LightGBM [21] as the learning model and Gama uses Hybrid-Based Memory Allocation. The comparison method's learning model is the optimal binary classification model through a training process. The default learning model size is set to 39KB, as specified in [39]. The total memory budget is configured to range from 64KB to 320KB with an increment of 64KB. The remaining memory is allocated to the backup BF. The experiments are conducted on servers equipped with a 4-core CPU, 128GB RAM, and 4T disk. All methods are implemented in Python language with version 3.10. All source codes are published [2]. Please refer to the README documentation for comprehensive details regarding the embeddings of the associated datasets.

We conduct Gama on **LBF** [24] and two variants **Sandwich LBF** (abbreviated as **SLBF**) [35] and **PLBF** [39]. We refer to LBF optimized by Gama as G-LBF, Sandwich LBF optimized by Gama as G-SLBF, and PLBF optimized by Gama as G-PLBF. Although Ada-BF [8] is an advanced LBF variant, its prohibitive construction time

[2] https://github.com/Spatio-Temporal-Lab/LBF-Gama

Table 2: FPR Comparison (%)

Dataset	Method	Memory Budget				
		64 KB	128 KB	196 KB	256 KB	320 KB
URL	LBF	17.685	1.624	0.311	0.091	0.053
	G-LBF	**5.471**	**0.974**	**0.170**	**0.060**	**0.039**
Yelp	LBF	1.245	0.684	0.503	0.463	0.453
	G-LBF	**1.048**	**0.682**	**0.403**	**0.295**	**0.216**
COD	L BF	9.971	4.868	3.380	2.588	2.048
	G-LBF	**7.608**	**4.529**	**3.217**	**2.487**	**1.962**

Table 3: Construction Time Comparison (s)

Dataset	Method	Memory Budget				
		64 KB	128 KB	196 KB	256 KB	320 KB
URL	LBF	**0.534**	**0.661**	**0.768**	**0.869**	**0.952**
	G-LBF	0.790	1.754	2.544	2.707	3.076
Yelp	LBF	7.318	**7.465**	**7.666**	**7.759**	**7.875**
	G-LBF	**7.069**	9.707	12.135	13.027	13.983
COD	LBF	7.968	**8.094**	**8.108**	**8.323**	**8.653**
	G-LBF	**7.877**	12.261	13.195	17.748	22.102

(e.g., nearly 20 hours on the COD dataset) renders it impractical for real-world applications. Given its structural similarities with PLBF, we chose PLBF as the basis for our improvements. Regarding the experimental construction, the original LBF variants are implemented using open-source repositories [3,4], and the parameters are also set to the default ones in the open source repositories.

6.2 Overall Performance Comparison

To verify the effectiveness of Gama, we use three datasets to compare the FPRs of different variants of LBFs both before and after optimization by Gama under different memory budgets. Table 2 and Table 3 show the FPR and construction time comparison of LBF and G-LBF, respectively, from which we can find results below:

(1) Gama helps LBF in decreasing its FPRs in all circumstances. In the best case for the URL dataset, G-LBF reduces the FPR from 17.685% to 5.471%, which is only 30.9% of the original LBF's FPR. This improvement stems from Gama's ability to find the near optimal allocation strategy for different datasets and total memory budgets, thereby enhancing the overall performance of the model.

(2) Gama achieves exponential reduction in FPR across the majority of experimental scenarios. In rare cases where the improvement margin appears relatively modest, this can be attributed to the fact that the default 39KB memory budget for the learning model already represents a near-optimal configuration for those specific datasets and total memory budget combinations. Notably, Gama demonstrates consistently strong performance across all other scenarios. These results substantiate two critical findings: the importance of optimal memory allocation strategy and Gama's exceptional adaptability to diverse total memory budgets.

(3) It takes some extra construction time for Gama to find an optimal memory allocation strategy. However, the construction time increases at a linear rate, which represents an acceptable trade-off when compared to the exponential reduction in FPR. This is particularly justified as the decreased FPR significantly reduces the runtime overhead of the indexing structure.

Overall, Gama demonstrates its adaptability and reliability by effectively optimizing memory allocation for LBF, consistently reducing FPR across diverse real-world datasets and memory budgets compared to the original LBF.

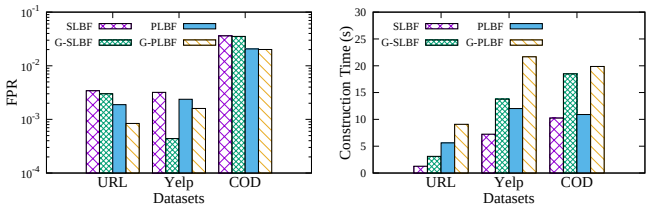


Figure 7: Performance on LBF Variants.

6.3 Performance on Different LBF variants

To demonstrate Gama's extendability on different structures of LBF, we conduct Gama on two classical LBF variants: **SLBF** adds a BF before LBF to catch most negatives. **PLBF** divides the score space into several regions and assigns backup BFs with different FPRs to each region. Here we set the memory budget to 192KB. From the experimental result shown in Fig. 7, we can observe that: 1) Gama helps LBF variants in decreasing their FPRs in all circumstances. 2) The trade-off between FPR and construction time observed in Gama on the original LBF still holds for the LBF variants. 3) G-PLBF generally has a lower FPR than G-SLBF, but this comes with a longer construction time. In summary, This experiment shows Gama's robustness in different structure of LBF's variants.

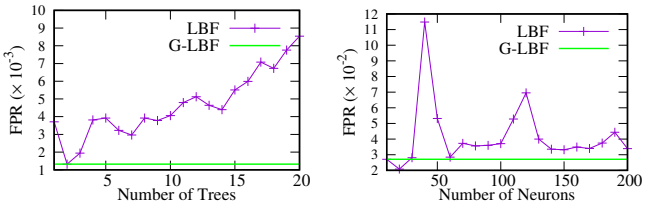
6.4 Performance on Different Classifiers.

To demonstrate Gama's versatility, we evaluate its performance with Random Forests [4] and ANN under a memory budget of 200KB. For ANN, Gama adjusts the memory allocation ratios by changing the number of neurons, and for Random Forests (abbreviated as RF), Gama changes the number of decision trees. As shown in Fig. 8, we can observe that:

(1) Regardless of the specific learning model used, the overall FPR of an LBF fluctuates significantly with the proportion of the total memory budget allocated to it. As the number of trees and neurons are varied, the LBF's FPR ranges from 0.0013 to 0.0085 and from 0.02 to 0.11, respectively. This demonstrates that effective memory allocation is crucial for LBFs, regardless of the underlying learning model.

(2) Gama can provide the near lowest FPR with different learning models, showing its wide applicability and effectiveness. The G-LBF, when configured with a RF model, achieves a minimum FPR of 0.0006, a result determined from the performance trend as the number of trees is varied. While the ANN-based LBF does not achieve the absolute minimum FPR, it attains the second-best performance with a value of 0.027. This demonstrates the success of our Bayesian-Based method in achieving a deliberate balance between the construction time and a highly competitive FPR.

[3]https://github.com/atsukisato/FastPLBF/tree/main
[4]https://github.com/RaimondiD/LBF_ADABF_experiment



(a) G-LBF on Random Forest. (b) G-LBF on ANN.

Figure 8: Performance on Different Classifiers.

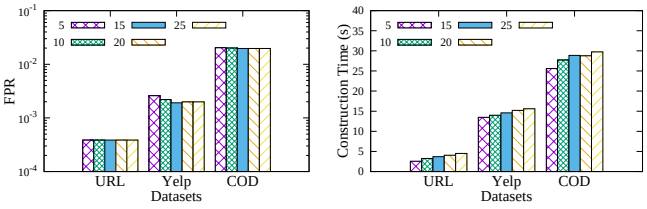
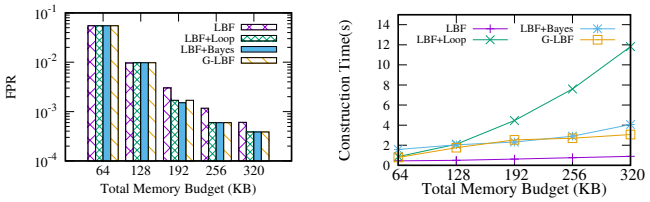
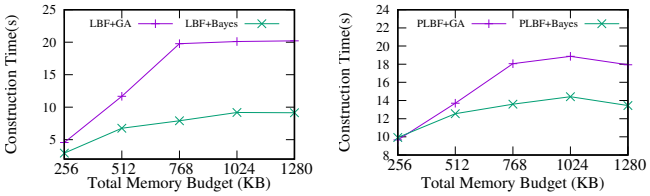


Figure 9: Performance on Different N_{BO}.



(a) FPR vs Memory Budget. (b) Construction Time vs Mem Budget.

Figure 10: Performance on Different Allocation Methods.



(a) Construction Time vs Mem Budget. (b) Construction Time vs Mem Budget.

Figure 11: Performance on Different Tuning Methods.

6.5 Parameter Study on Gama.

As mentioned in Algorithm 2, N_{BO}, the number of iterations for Bayesian Optimization, plays a crucial role in the performance of the Bayesian Optimization process itself. We evaluate the performance of Gama under varying values of N_{BO} to analyze how the hyper-parameter of Bayesian Optimization might influence Gama's performance. We adopt Bayesian-Based memory allocation in this experiment because the Hybrid-Based approach sometimes finds the optimal memory allocation during the Loop-Based phase, which diminishes the effectiveness of the Bayesian method. As a result, the role of N_{BO} is not well reflected. In this experiment, we set the memory budget to the maximum value in Section 6.2, i.e., 320KB.

Figure 9 illustrates the performance of Gama using the Bayesian-Based Memory Allocation method across different datasets with varying values of N_{BO}. The legend indicates N_{BO} values ranging from 5 to 25, with an increment of 5. As N_{BO} increases, the FPR shows a slight decrease, but it remains relatively stable overall, while the construction time shows a slight increase. The stable FPR demonstrates Gama's ability to maintain strong performance despite changes in relevant hyper-parameters. Meanwhile, the increased number of iterations results in more frequent threshold selection processes. Since the time complexity of threshold selection is relatively low, it does not significantly impact Gama's performance in terms of construction time. In summary, Gama demonstrates robust performance across varying number of iterations.

6.6 Ablation Study on Gama

We conducted an ablation study to demonstrate that Gama effectively integrates the Loop-Based method and the Bayesian-Based method, achieving a comparable FPR while significantly reducing construction time. The fixed 0.1 allocation ratio used in our evaluation was empirically determined to offer the best balance, based on preliminary tests sweeping from 0.1 to 0.9 on a 64KB budget.

Figure 10 shows the experimental results, where LBF+Loop denotes the LBF optimized using the Loop-Based method, while LBF+Bayes denotes the LBF optimized using the Bayesian-Based method. In terms of the trends, as the total memory budget increases, the FPR for all methods decreases, as shown in Fig. 10(a). However, Gama consistently achieves the lowest FPR values across all memory budgets. Specifically, in the optimal scenario with a total memory budget of 256KB, the FPR for LBF is approximately 0.0011, while G-LBF further reduces the FPR to around 0.00059, roughly half of LBF's FPR. These results demonstrate the significant advantage of Gama in reducing FPRs. Regarding construction time, as shown in Fig. 10(b), the curves for all methods exhibit an increasing trend with the total memory budget. Among them, Loop-Based method performs better under low memory budgets, while the Bayesian-Based method is more efficient with higher memory budgets. Gama, however, combines the strengths of both Loop and Bayesian methods. As a result, Gama achieves low FPRs and maintains low construction times under all memory budgets, showcasing its efficiency and robustness.

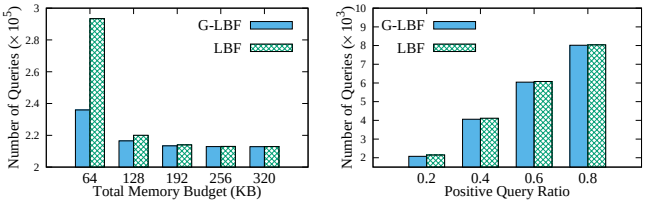
In summary, the results demonstrate that Gama is highly effective, significantly reducing the FPR while maintaining efficient construction time. This makes Gama an ideal choice for scenarios requiring both FPR and construction efficiency.

6.7 Performance on Different Tuning Methods.

We evaluate different tuning algorithms for LBFs. As shown in Fig. 11, we compare Genetic Algorithm (GA) [19] and Bayesian Optimization across LBF and PLBF. Given their comparable FPRs, the two methods are compared solely on construction time.

In this experiment, we use larger memory budgets than in previous tests to better highlight the time complexity of the tuning algorithms. As shown in both Fig. 11(a) and Fig. 11(b), Bayesian optimization (LBF+Bayes and PLBF+Bayes) outperforms Genetic Algorithm (LBF+GA and PLBF+GA) in terms of construction time, especially as the memory budget increases. LBFs tuned with GA exhibit a more rapid increase in construction time than those tuned with Bayesian optimization, primarily due to the inherent computational overhead of GA's evolutionary process.

In summary, this experiment demonstrates that Bayesian optimization is more effective for memory allocation on LBFs.



(a) #Queries vs Memory Budget. (b) #Queries vs Positive Query Ratio.

Figure 12: Performance on Database Query.

6.8 Performance on Database Query.

To evaluate the query acceleration performance of Gama, we use the total number of database queries as the primary metric. This choice is motivated by the need to isolate the filter's effectiveness from confounding factors, such as buffer pool state, which can introduce variability into execution times. Specifically, the query process begins with the pre-filter determining the potential existence of a query URL. A full database lookup proceeds only if the filter returns a positive match; otherwise, the database query is skipped. Figure 12 illustrates the reduction in the number of executed database queries when augmenting LBF with Gama.

Figure 12(a) compares the query counts of G-LBF and LBF under varying memory budgets. As the memory budget decreases from 320KB to 64KB, the query count for G-LBF consistently remains lower than that of LBF. At the most constrained budget of 64KB, LBF requires 24.4% more query executions than G-LBF. To assess performance robustness, we investigated the effect of varying the positive-query ratio, as depicted in Fig. 12(b). While a higher proportion of positive queries predictably increases the query load for both methods, G-LBF's outperformance of LBF remains constant.

In conclusion, Gama provides a robust enhancement to LBF's query acceleration capabilities, maintaining its effectiveness across a wide spectrum of memory budgets and positive-query ratios.

7 RELATED WORKS

Since Kraska et al. [24] proposed the LBF as a data structure for indexing problems, subsequent research has pursued two main optimization strategies: modifying its core architecture (e.g., the number or types of BFs), and tuning its operational parameters. Our Gama framework falls into the latter approach.

7.1 Data Structure Optimization of LBF

Since LBF is formed by the combination of a learning model and a BF, most researchers optimize it from a data structure view. A widely used approach is to increase the number of BFs in the LBF to take full advantage of the learning model. Sandwich LBF [35] introduces an initial BF before the learning model to handle some easy samples, preserving the learning model for more complex queries. Subsequent studies question the poor usage of the predicted probability scores of the learning model in initial approaches, and apply multiple backup BFs to distinguish different score ranges for further optimization. Ada-BF [8] divides the score spectrum lower than the threshold into distinct regions, each of which is associated with its own BF. Hence, it can tune the number of hash functions differently in different regions to adjust the FPR adaptively. Furthermore, PLBF [42] partitions the entire score space into multiple regions and assigns a separate backup BF with different FPRs to each region. When a query item arrives, the learning model utilizes its region and the corresponding backup BF to determine whether it exists.

Reflecting its growing recognition as an index structure, much research has focused on extending LBF's applicability by modifying its core components. Li [31] and Chen [7] extend LBF to handle multi-key queries, which contain a value-interaction-based multi-key classifier and a multi-key BF. To ensure the performance of LBF in streaming scenarios, Liu proposes Stable LBFs [32], which addresses this issue by combining the classifier with updatable backup filters. Stable LBFs introduce a similar level of FPR but save more space. To address the challenges in the approximate membership query on data streams, Learned Cuckoo Filters (LCF) [41] adaptively maintains cuckoo filters with the assistance of a well-trained oracle that learns the frequency feature of the data within the stream. Furthermore, LBF for large-scale membership query [47], spatial data [46], and incremental workloads [2] have also been introduced with specially designed learning models and structures.

7.2 Parameter Optimization of LBF

Alongside the proliferation of new LBF architectures, recent works have increasingly focused on parameter optimization to enhance performance and construction efficiency. Malchiodi et. al. [33, 34] summarize an experimental framework to guide users in selecting between BF or LBF and the specific parameters of LBF. Dai et al. [9] demonstrate the necessity of parameter tuning for the learning models in LBF by testing the performance of different learning models in Ada-BF and PLBF under various parameters. As PLBF has an extremely slow construction speed, Sato proposes fast PLBF [39], which can be constructed much faster than PLBF by omitting the redundant construction of the Dynamic Programming tables.

As a parameter optimization, the Gama framework we proposed formalizes an updated LBF memory allocation problem and finds a near-optimal memory allocation with acceptable time overhead, significantly reducing the FPR of LBFs.

8 CONCLUSION

This paper proposes an effective general adaptive memory allocation framework Gama for LBFs. In particular, Gama consists of two effective memory allocation strategies: Loop-Based approach ensures precise memory allocation and is well-suited for scenarios with tight memory constraints, while Bayesian-Based method excels at exploring configurations efficiently under larger memory budgets. Building on these two methods, we further introduce a hybrid approach that combines their strengths and performs well under almost all memory constraints. Extensive experiments on three large-scale datasets validate the effectiveness of our framework, Gama, achieving a best-case FPR reduction of 69%.

ACKNOWLEDGMENTS

This paper is supported by the National Natural Science Foundation of China (62202070, 72242106), the Fundamental Research Funds for the Central Universities (2024CDJYDYL015), Major Basic Research Project of Shandong Provincial Natural Science Foundation (ZR2024ZD03), and the Guangdong Provincial / Zhuhai Key Laboratory of IRADS (2022B1212010006).

REFERENCES

[1] Nima Asadi and Jimmy Lin. 2013. Fast candidate generation for real-time tweet search with bloom filter chains. *ACM Transactions on Information Systems (TOIS)* 31, 3 (2013), 1–36.

[2] Arindam Bhattacharya, Srikanta Bedathur, and Amitabha Bagchi. 2020. Adaptive learned bloom filters under incremental workloads. In *Proceedings of the 7th ACM IKDD CoDS and 25th COMAD*. 107–115.

[3] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[4] Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001), 5–32.

[5] Hayoung Byun and Hyesook Lim. 2021. Learned FBF: Learning-Based Functional Bloom Filter for Key–Value Storage. *IEEE Trans. Comput.* 71, 8 (2021), 1928–1938.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[7] Haitian Chen, Ziwei Wang, Yunchuan Li, Ruixin Yang, Yan Zhao, Rui Zhou, and Kai Zheng. 2023. Deep Learning-Based Bloom Filter for Efficient Multi-key Membership Testing. *Data Science and Engineering* 8, 3 (2023), 234–246.

[8] 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. *Advances in neural information processing systems* 33 (2020), 11700–11710.

[9] Zhenwei Dai, Anshumali Shrivastava, Pedro Reviriego, and José Alberto Hernández. 2022. Optimizing learned bloom filters: How much should be learned? *IEEE Embedded Systems Letters* 14, 3 (2022), 123–126.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2018. Optimal bloom filters and adaptive merging for LSM-trees. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–48.

[11] Niv Dayan and Moshe Twitto. 2021. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.

[12] Fangming Dong, Pinghui Wang, Rundong Li, Xueyao Cui, Junzhou Zhao, Jing Tao, Chen Zhang, and Xiaohong Guan. 2025. Poisoning Attacks and Defenses to Learned Bloom Filters for Malicious URL Detection. *IEEE Transactions on Dependable and Secure Computing* (2025), 1–14.

[13] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.

[14] Shahabeddin Geravand and Mahmood Ahmadi. 2013. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks* 57, 18 (2013), 4047–4064.

[15] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. 2017. Bitfunnel: Revisiting signatures for search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 605–614.

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

[17] Huajun He, Zihang Xu, Ruiyuan Li, Jie Bao, Tianrui Li, and Yu Zheng. 2024. TMan: a high-performance trajectory data management system based on key-value stores. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4951–4964.

[18] Qiang He, Siyu Tan, Feifei Chen, Xiaolong Xu, Lianyong Qi, Xinhong Hei, Hai Jin, and Yun Yang. 2023. Edindex: Enabling fast data queries in edge storage systems. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 675–685.

[19] John H Holland. 1992. Genetic algorithms. *Scientific american* 267, 1 (1992), 66–73.

[20] Yelp Inc. 2024. Yelp Open Dataset. https://www.yelp.com/dataset.

[21] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[22] Yekun Ke, Yingyu Liang, Zhizhou Sha, Zhenmei Shi, and Zhao Song. 2025. DP-Bloomfilter: Securing Bloom Filters with Differential Privacy. *arXiv preprint arXiv:2502.00693* (2025).

[23] Adam Kirsch and Michael Mitzenmacher. 2006. Less hashing, same performance: Building a better bloom filter. In *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*. Springer, 456–467.

[24] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[25] Harikesh Kumar. 2023. Celestial Objects Dataset. https://www.kaggle.com/datasets/hari31416/celestialclassify.

[26] Jungwon Lee, Hayoung Byun, and Hyesook Lim. 2020. Dual-load Bloom filter: Application for name lookup. *Computer Communications* 151 (2020), 1–9.

[27] Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Siqiang Luo, Rong Gu, Tong Yang, and Guihai Chen. 2022. Seesaw counting filter: An efficient guardian for vulnerable negative keys during dynamic filtering. In *Proceedings of the ACM Web Conference 2022*. 2759–2767.

[28] Ruiyuan Li, Huajun He, Rubin Wang, Yuchuan Huang, Junwen Liu, Sijie Ruan, Tianfu He, Jie Bao, and Yu Zheng. 2020. Just: Jd urban spatio-temporal data engine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1558–1569.

[29] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Tianfu He, Jie Bao, Junbo Zhang, Liang Hong, and Yu Zheng. 2021. TrajMesa: A distributed NoSQL-based trajectory data management system. *IEEE TKDE* 35, 1 (2021), 1013–1027.

[30] Ruiyuan Li, Xiang He, Yingying Sun, Jun Jiang, You Shang, Guanyao Li, and Chao Chen. 2024. Spatio-temporal keyword query processing based on key-value stores. *Data Science and Engineering* (2024), 1–19.

[31] Yunchuan Li, Ziwei Wang, Ruixin Yang, Yan Zhao, Rui Zhou, and Kai Zheng. 2023. Learned bloom filter for multi-key membership testing. In *International Conference on Database Systems for Advanced Applications*. Springer, 62–79.

[32] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable learned bloom filters for data streams. *PVLDB* 13, 12 (2020), 2355–2367.

[33] Dario Malchiodi, Davide Raimondi, Giacomo Fumagalli, Raffaele Giancarlo, and Marco Frasca. 2023. A critical analysis of classifier selection in learned bloom filters: the essentials. In *International Conference on Engineering Applications of Neural Networks*. Springer, 47–61.

[34] Dario Malchiodi, Davide Raimondi, Giacomo Fumagalli, Raffaele Giancarlo, and Marco Frasca. 2024. The role of classifiers and data complexity in learned Bloom filters: insights and recommendations. *Journal of Big Data* 11, 1 (2024), 45.

[35] Michael Mitzenmacher. 2018. A model for learned bloom filters and optimizing by sandwiching. *Advances in Neural Information Processing Systems* 31 (2018).

[36] Ju Hyoung Mun and Hyesook Lim. 2015. New approach for efficient ip address lookup using a bloom filter in trie-based algorithms. *IEEE Trans. Comput.* 65, 5 (2015), 1558–1565.

[37] Fernando Nogueira. 2019. BayesianOptimization—A Python implementation of global optimization with Gaussian processes. *The source package can be found at https://github. com/fmfn/BayesianOptimization* (2019).

[38] Ripon Patgiri, Anupam Biswas, and Sabuzima Nayak. 2023. deepBF: Malicious URL detection using learned bloom filter and evolutionary deep learning. *Computer Communications* 200 (2023), 30–41.

[39] Atsuki Sato and Yusuke Matsui. 2024. Fast partitioned learned bloom filter. *Advances in Neural Information Processing Systems* 36 (2024).

[40] Manu Siddhartha. 2021. Malicious URLs Dataset. Kaggle. https://www.kaggle.com/datasets/sid321axn/malicious-urls-dataset [Online; accessed 22-December-2022].

[41] Yao Tian, Tingyun Yan, Ruiyuan Zhang, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. A Learned Cuckoo Filter for Approximate Membership Queries over Variable-sized Sliding Windows on Data Streams. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[42] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. 2021. Partitioned learned bloom filter. *International Conference on Learning Representations* (2021).

[43] Paul Walther, Wejdene Mansour, and Martin Werner. 2025. Extending the Applicability of Bloom Filters by Relaxing their Parameter Constraints. *arXiv preprint arXiv:2502.02193* (2025).

[44] Hancheng Wang, Haipeng Dai, Meng Li, Jun Yu, Rong Gu, Jiaqi Zheng, and Guihai Chen. 2022. Bamboo filters: Make resizing smooth. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 979–991.

[45] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[46] Meng Zeng, Beiji Zou, Xiaoyan Kui, Chengzhang Zhu, Ling Xiao, Zhi Chen, and Jingyu Du. 2023. PA-LBF: Prefix-Based and Adaptive Learned Bloom Filter for Spatial Data. *International Journal of Intelligent Systems* 2023, 1 (2023), 4970776.

[47] Meng Zeng, Beiji Zou, Wensheng Zhang, Xuebing Yang, Guilan Kong, Xiaoyan Kui, and Chengzhang Zhu. 2023. Two-layer partitioned and deletable deep bloom filter for large-scale membership query. *Information Systems* 119 (2023), 102267.

[48] Fan Zhang, Hanhua Chen, Hai Jin, and Pedro Reviriego. 2021. The logarithmic dynamic cuckoo filter. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 948–959.

[49] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards dynamic and safe configuration tuning for cloud databases. In *Proceedings of the 2022 International Conference on Management of Data*. 631–645.

[50] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2023. Automatic database knob tuning: a survey. *IEEE Transactions on Knowledge and Data Engineering* 35, 12 (2023), 12470–12490.

[51] Yikai Zhao, Yubo Zhang, Pu Yi, Tong Yang, Bin Cui, and Steve Uhlig. 2022. The Stair Sketch: Bringing more Clarity to Memorize Recent Events. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 164–177.