Adaptive Learned Bloom Filters under Incremental Workloads

Arindam Bhattacharya
Department of Computer Science and Engineering, IIT Delhi
arindam@cse.iitd.ac.in

Srikanta Bedathur
Department of Computer Science and Engineering, IIT Delhi
srikanta@cse.iitd.ac.in

Amitabha Bagchi
Department of Computer Science and Engineering, IIT Delhi
bagchi@cse.iitd.ac.in

ABSTRACT

The recently proposed paradigm of learned Bloom filters (LBF) seems to offer significant advantages over traditional Bloom filters in terms of low memory footprint and overall performance as evidenced by empirical evaluations over static data. Its behavior in presence of *updates to the set of keys* being stored in Bloom filters is not very well understood. At the same time, maintaining the false positive rates (FPR) of traditional Bloom filters in presence of dynamics has been studied and extensions to carefully expand memory footprint of the filters without sacrificing FPR have been proposed. Building on these, we propose two distinct approaches for handling data updates encountered in practical uses of LBF: (i) **CA-LBF**, where we adjust the learned model (e.g., by retraining) to accommodate the new "unseen" data, resulting in *classifier adaptive* methods, and (ii) **IA-LBF**, where we replace the traditional Bloom filter with its adaptive version while keeping the learned model unchanged, leading to an *index adaptive* method. In this paper, we explore these two approaches in detail under *incremental workloads*, evaluating them in terms of their adaptability, memory footprint and false positive rates. Our empirical results using a variety of datasets and learned models of varying complexity show that our proposed methods' ability to handle incremental updates is quite robust.

ACM Reference Format:
Arindam Bhattacharya, Srikanta Bedathur, and Amitabha Bagchi. 2020. Adaptive Learned Bloom Filters under Incremental Workloads. In *7th ACM IKDD CoDS and 25th COMAD (CoDS COMAD 2020), January 5–7, 2020, Hyderabad, India.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3371158.3371171

1 INTRODUCTION

Recently, Kraska et al. [8], suggested the use of a combination of machine learned model and an auxiliary structure as an alternative to the traditional index structures. Apart from the overall smaller index size, they argued that this approach offers indexes tuned for the observed key distributions in large datasets thus making them much more efficient than the index whose parameters are set to handle "common case" or worst-case key distributions. Experiments conducted over Google's internal workloads using neural network based machine learned models showed that this novel approach provides 15% to 36% size reduction over traditional Bloom filters and up to two orders of magnitude reduction over classical B+-Trees while maintaining similar levels of performance. However, a fundamental assumption made by [8] is that the data is static and does not vary over time, thus making the method suitable for read-only workloads. It is not immediately clear how to adapt it for write-heavy or mixed workloads that are often found in business and scientific databases [12].

In this paper, we address the issue of incremental workloads, involving insertions. We restrict our attention to the *learned Bloom filter* framework proposed by Kraska et al. We focused on studying Bloom filters due to their wide usage in various distributed big data frameworks ranging from Hadoop to Spark. They are very simple to implement and can be constructed efficiently resulting in their widespread usage in settings ranging from bioinformatics to social network analysis [5]. We do not study workloads involving deletions, as it requires additional overheads in case of Bloom filter and distracts from the focus of our solutions.

With insertion workloads, two kind of scenarios may arise. One where the future data comes from the same distribution as the original data on which the model is trained. In this situation, the learned Bloom filter performs fine. The issue arises when the data distribution increases. To demonstrate this, we conducted an experiment where we performed an experiment with synthetic Gaussian data. We initiated two learned Bloom filters with Gaussian data $D_{init} \sim \mathcal{N}(\mu_0, \sigma_0)$. We then insert $D_{ins} \sim \mathcal{N}(\mu_0, \sigma_0)$ to one LBF and $D'_{ins} \sim \mathcal{N}(\mu'_0, \sigma_0)$ to the other, where μ'_0 is shifted by one σ_0 in each feature direction. The false positive rate in the second scenario increases as seen in Figure. 1 because LBF fails to adapt to the changing distribution.

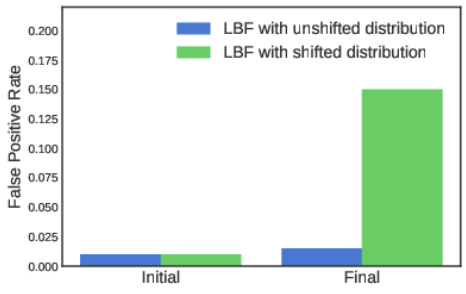


Figure 1: Effect of dynamic distribution on learned Bloom filter

The two solutions we propose differ in the way they handle dynamism by delegating the responsibility to adaptability differently.

CoDS COMAD 2020, January 5–7, 2020, Hyderabad, India

ACM ISBN 978-1-4503-7738-6/20/01…$15.00
https://doi.org/10.1145/3371158.3371171

In the classifier adaptive solutions (CA-LBF I and CA-LBF II), the adaptation happens with the classifier, while the Bloom filter is static and untouched. The classifier adapts by training incrementally (CA-LBF I) or partially (CA-LBF II). The retraining happens only on the batch of inserted data. The reason for this is two-fold. First, we may not have access to the entire database of existing data, and second, even if we have access to entire data, complete retraining may be prohibitively expensive.

In index adaptive solution (IA-LBF), the classifier remains untouched, and the Bloom filter is responsible for adapting to the new data. To do this we replace the standard Bloom filter with Dynamic Partition Bloom filter (discussed in Section. 2.2).

The reason we provide multiple solutions is because each solution provides trade-offs which may favor one kind of application but may be unsuitable for other. For this reason we applied our solutions to a range of datasets, each selected with particular characteristic that demonstrates strengths and weaknesses of the proposed solutions. The details are presented in Section. 5.1.

Contributions.

(1) We show that learned Bloom filters are not adaptable in presence of updates to the sets they are maintaining — i.e., they can not retain the false positive rates, have increasingly higher retraining costs, and necessitate full access to the data which may not be practical.
(2) We present two flavors of supporting updates in the learned Bloom filter (LBF) framework: (i) Classifier-adaptive LBF (CA-LBF), and (ii) Index-adaptive LBF (IA-LBF). The former trades off time (retraining) for compactness, whereas the later sacrifices memory for performance and time.
(3) We perform detailed empirical evaluation of LBF, CA-LBF and IA-LBF methods across a variety of datasets and demonstrate that while learned Bloom filters suffers in dynamic scenarios, our methods can handle dynamism while retaining the performance, by sacrificing time or memory.

Organization. In Section. 2 we provide background for Bloom filter, DPBF, and binary classifiers. We formally define the problem and describe our proposed solutions in Section. 3. In Section. 4 we describe the framework used to evaluate the performance on adaptive LBFs. The details of our experiments and results are presented in Section. 5.

2 BACKGROUND

2.1 Bloom filters

Bloom filters [2] are probabilistic space efficient data structures for approximate representation of sets that support set membership queries. A Bloom filter for representing a set $S = \{x_1, x_2, ..., x_n\}$ of n elements is defined by an array of m bits along with k independent hash functions $h_1, h_2, ..., h_k$ where $h_i : S \rightarrow 1, 2, ..., m$. All the elements of the array are initialized to 0. For each element $x \in S$, the bits $h_i(x)$, $1 \leq i \leq k$, are set to 1. Multiple elements may set one particular index, but only the first update is counted. When a query y is received, we check for the indices $h_i(y)$ for $1 \leq i \leq k$. If any of them is not set to 1, we can surely say that $y \notin S$. Otherwise we assume that $y \in S$, although it may be wrong with some probability. Thus, a Bloom filter may yield *false positives*, but no *false negatives*. The *false positive rate* is bounded by $f = (1 - \frac{1}{m})^{kn}$ [3].

Dynamic Bloom Filters. The bound of the false positive rate of a Bloom filter are defined for a static set of a given size. They do not adapt to a dynamic set where the cardinality of the set increases. In such scenario, the standard Bloom filter is unable to provide any guarantee on the false positive rate. Dynamic Bloom filters [6] addresses this limitation by using multiple Bloom filters. A DBF consists of s homogeneous standard Bloom filters of size m using k hash functions. The Bloom filters are associated with a capacity c based on the desired false positive rate. Initially, the structure has one Bloom filter, that is $s = 1$. It is set as the active Bloom filter. DBF performs insertions only on the active Bloom filter. As more elements are inserted and the active Bloom filter reaches its capacity, a new Bloom filter is appended and set as the active Bloom filter. Each standard Bloom filter in DBF thus stores a subset of the set S represented by the DBF. To perform a query on an element x, we check each of the standard Bloom filters in DBF. If x is not present in any of the s Bloom filters, we say $x \notin S$. Otherwise, $x \in S$ with some probability. The false positive rate is bounded by $f = (1 - (1 - e^{-k \cdot (n - c \cdot \lfloor n/c \rfloor / m)})^k)$. The false positive rate of DBP grows much slowly compared to standard Bloom filter beyond the capacity of the Bloom filter. While the growth of false positive rate is slower in DBF, it can still grow linearly with the size of the input set.

2.2 Dynamic Partition Bloom Filters

Dynamic Partition Bloom filters [14] is a data structure proposed by our group to address the issue of linear growth of FPR presented in DBF. DPBF can maintain a target false positive rate irrespective of how the size of the input set varies. In addition, DPBF provides much lower query time compared to DBF. It consists of a compressed partition tree (CPBPT) of depth d and a hash map of standard Bloom filters. The leaves of the partition tree are standard Bloom filters, called unit Bloom filters, that stores a subset of the stored set. The size of each unit Bloom filter is given by $m = \lceil -nk/ln(1 - f^{1/k}) \rceil$. This tree allows for efficient membership queries while keeping the memory utilization low. The hash map, called *populated unit Bloom filter list* (PUBFList) for a set $A \subset S$ is the set of all non empty unit Bloom filter that makes up A. It is used to handle set operations including insertion of new elements to DPBF.

In Figure. 2 we show an instance of a DPBF that is being used to store the set $\{5, 10, 17, 19, 22, 25, 31\}$ taken from the namespace

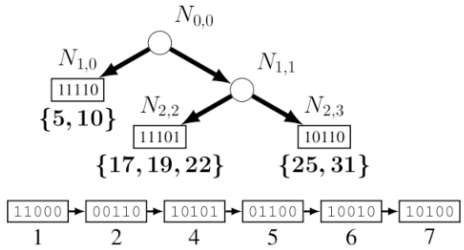


Figure 2: DPBF consisting the CPBPT and the PUBFList

$\{0, 1, \ldots 31\}$. The namespace is partitioned into 8 chunks of size 4, i.e., to a depth of 4, and the keys are inserted into the populated unit Bloom Filter map corresponding to the 8 chunks (note that in the example only 6 of these chunks are populated). The compressed tree is made to ensure that no leaf contains more than 4 keys in it. Note that the leaves $N_{2,2}$ and $N_{2,3}$ contain 2 and 3 keys respectively so they cannot be merged since that would create a node with 4 keys. To query this structure, for the key 18, say, we would find a path down to the leaf which accounts for the portion of the namespace containing 18, leaf $N_{2,2}$ in ths example, and query the Bloom filter at that leaf.

3 PROBLEM DEFINITION AND OUR SOLUTIONS

3.1 The Learned Set Membership Problem

The *set membership problem* is a fundamental problem in computing: Given a set S drawn from a universe U and a query $x \in U$, return YES if $x \in S$ and NO otherwise. A typical solution to this problem involves constructing an index structure that can answer this question efficiently, i.e., we construct an index structure $\mathcal{I}(S)$ and describe an algorithm that computes a function $f(\mathcal{I}, x)$ that returns the correct answer with some probability.

The *learned set membership problem with false positives* is a refinement of the set membership problem that seeks to compress knowledge about the set S into a classifier. A back-up index structure is added to compensate for inaccuracies introduced by the classifier. Formally speaking, a solution to the learned set membership problem with false positives comprises (i) a classifier C that classifies the elements of U into members of S and members of $U \setminus S$, and, (ii) an index structure $\mathcal{I}_N$ along with a function f that provides a solution to the set membership problem. The answer to the set membership query for $x \in U$ is given by a two stage process

- If $C(x)$ returns YES, return YES
- else return $f(\mathcal{I}_N, x)$,

i.e., if the classifier says that $x \in S$, we return YES, otherwise we check with the index structure and return the answer it returns. Clearly, for this solution to the work correctly $\mathcal{I}_N$ must be an index on

$$\{x \in S : C(x) = \text{NO}\},$$

i.e., on the false negatives of the classifier. Note that this algorithm *always* returns YES answers for the set

$$\{x \notin S : C(x) = \text{YES}\},$$

i.e., on the false positives of the classifier. We note that this solution returns false positives. To ensure that there are no false negatives the function $f(\mathcal{I}_N, x)$ must be of false negatives. In $f(\mathcal{I}_N, x)$ has false positives then the consolidated false positive rate of the solution is the false positive rate of the classifier along with the false positive rate of the index-based solution.

The dynamic version of the learned set membership problem poses a significant challenge since when S changes the accuracy of the classifier C could be severely affected and the index on the false negatives, $\mathcal{I}_N$, may require major changes. As such the dynamic version of the problem poses several new challenges that do not arise in the static version that was studied by Kraska et. al. [8]. In this paper we study the "insertion-only" case, i.e., we allow S to grow but do not allow deletions. Allowing deletions causes further complications which we postpone for now.

3.2 Evaluating a Solution to the Dynamic Learned Set Membership Problem

A solution to the learned set membership problem is evaluated on the following criteria: (i) *Query time*, (ii) the *space* taken, which includes the space needed for the the classifier and the index structure (iii) the accuracy of the answer returned, typically measured in terms of the *false positive probability*, i.e., the probability that a YES answer is correct, or the *false negative probability*, i.e., the probability that a NO answer is correct.The solutions we present in this paper are based on Bloom filters and so do not incur any false negatives and hence the focus is purely on the false positive probability. And finally, (iv) *Update time*, i.e., the time taken to accommodate for changes in S.

3.3 Three Bloom Filter-based Solutions to the Dynamic Learned Set Membership Problem

3.3.1 Baseline: Static Learned Bloom Filter (SLBF). Introduced by Kraska et. al., in this solution we maintain $\mathcal{I}_N$ as a fixed-size Bloom filter along with the classifier C. This solution will act as a baseline for us. Clearly here update time in zero, query time and space required remain the same as S grows, but the false positive rate suffers.

3.3.2 Classifier-Adaptive Learned Bloom Filter (CA-LBF). In this solution we change the classifier as the set grows. The first thought would be "just retrain the classifier" on the new version of S, but there are two issues here. The first issue is that we may not have database access, i.e., the elements inserted into S in the past may no longer be available to us. This will typically happen when S is defined over a data stream, e.g., Facebook Check-in is real time streaming data for which we may not have the access to centralized database at all times. The second issue is that as S grows in size the time taken to retrain the classifier may be prohibitively large, i.e., our update time may become unsustainably huge. Note that for the baseline SLBF the classfier training cost is ignored, deemed to be a one-time offline cost, but in the dynamic setting any classifier-adpative solution has to account for this.

With this in mind we suggest the following two versions of a solution.

CA-LBF I Suppose $S = \{s_i : 1 \leq i \leq n\}$ where the elements are indexed by the order in which they are inserted into S. Fix a $k > 0$ and train $m = \lfloor S/k \rfloor$ classifiers $C_i, 1 \leq i \leq m$, where C_i is a classifier for the batch $b_i = b_{i-1} \cup \{s_j : k \cdot (i-1)+1 \leq j \leq k \cdot (i-1) + k - 1\}$. A single fixed-size Bloom filter is maintained for as $\mathcal{I}_S$.
Query: Given an x, we compute $\bigvee_{i=1}^{m} C_i(x)$, i.e., we check if any of the classifiers claims that x belongs to its set. If S is not an exact multiple of k there may be a few elements of S not covered by any classifier. We can explicitly check for x amongst them.

CA-LBF II Given the same setup, we train $m = \lfloor S/k \rfloor$ classifiers $C_i, 1 \leq i \leq m$, where C_i is a classifier for the batch

$b_i = \{s_j : k \cdot (i-1) + 1 \leq j \leq k \cdot (i-1) + k - 1\}$. A single fixed-size Bloom filter is maintained for as $\mathcal{I}_S$.
Query: Same as CA-LBF I.

In this solution we define a chunk size k and every time k new elements arrive we train a new classifier exclusive for that chunk. We choose k small enough that the time taken to train the new classifiers is within the applications tolerance for updates. Clearly here the query time increases as S grows since the number of classifiers grows and we need to check the query against each one of them. The space needed for storing the classifiers also grows linearly with S.

Note that in *CA-LBF I*, we retain the knowledge of previous data when training on new data, effectively training a classifier for the entire data till now, whereas in *CA-LBF II*, each new classifier is trained from scratch and is responsible only for the chunk it sees. The reason behind the solutions is two-fold. First, it provides a trade-off between training time and false positive rate. This is because while training a new classifier from scratch takes longer for CA-LBF II, it can achieve lower false positive rate, since that classifiers are concerned with exclusive sets and do not add to the false positives of each other. On the other hand, the incremental training of CA-LBF I is fast but since one classifier targets a superset of previous classifiers, the false positives of each classifier adds up. Second reason is dependent on the data stream. If the distribution of the inserted data changes considerably, CA-LBF I will have harder time to adapt to the changes, whereas the new classifier in CA-LBF II will *overfit* on only the inserted data thus better adapting to the change. On the other hand, if the distribution is similar, CA-LBF II may spend all the extra time for little or no gain.

3.3.3 Index-Adaptive Learned Bloom Filter (IA-LBF). In this solution we train a classifier, C, when we have seen a certain number of elements of S and then we work with that classifier alone without either retraining it or adding new classifiers, i.e., as far as the classifier is concerned this scenario is exactly like SLBF. Instead we adapt at the index end. The currently best-known solution to the dynamic set membership problem using Bloom filters works by just adding new Bloom filters of a fixed size to the index as the set grows. These are the Dynamic Bloom Filters of Guo et. al. [6], and unfortunately even they suffer from the problem that the false positive rate increases as the set grows (see Theorem A.1 of [14]). So we use Dynamic Partition Bloom Filter [14], $\mathcal{I}_D$, which is able to maintain a target false positive rate for set membership in the insertion-only setting.

In IA-LBF all the cost is pushed to the index: query time and space all grow with the index. Update here involves checking if the new insertion returns YES or NO against C. If the answer is NO then we insert the new element into the index structure.

The architectures of these methods are presented in Figure. 3.

4 EVALUATION FRAMEWORK

The various possible trade offs imply that there is no *one size fits all* solution for the wide range of applications these data structures can be used for. In this section we list the parameters we examined, the metric we used to compare these parameters and their effect on the performance. As a result of the divergent objectives, we propose ways to adjust the classifiers to overfit the data and to tweak the classifier threshold to achieve desired metric (false positive rate, memory requirement). Here we explore the implications of such modifications and techniques to select such parameters. Experimental results are presented in Section 5.

4.1 Model Complexity

Machine learning models comes in a wide range of complexities. Model complexity is an intricate parameter which can be captured by various metrics such as model size, training time or VC dimensions. We use accuracy as a proxy for model complexity, with the assumption that the more complex models such as CNN will typically result in higher accuracy, given enough data. This definition aligns itself with other parameters such as space, time and false positive rate, and provides a practical heuristic for choice of models. The choice of model complexity is driven by the trade-off of time and space versus performance. In the case of dynamic LBF, this essentially leads to the trade-off between time versus space and performance.

Figure. 4 shows the scaled relative effect of the models from our experiments. It compares the relative model size with the relative training times. The size of the circle represents the standard deviation over multiple runs. Note that the complex models have larger number of parameters but the size of the model itself is insignificant compared to the size of the Bloom filter. Larger models, such as the one used in CIFAR takes considerably longer to train compared to smaller models such as the one used in LETOR. In Section 5 we present comparison of models of various complexities on various datasets and their effect on time, space and performance.

4.2 Time

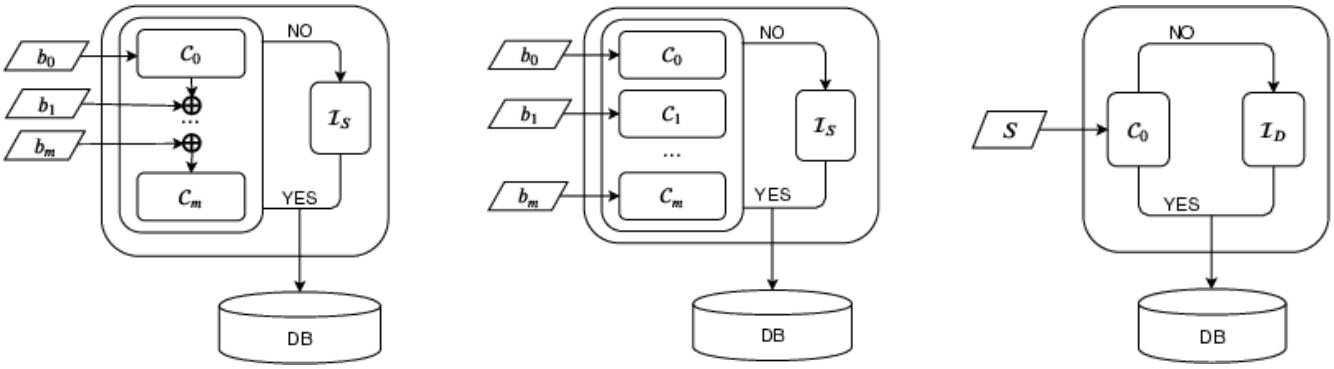
There are two distinct notions of time in the LBF setting. One is the time taken to train (and retrain if needed) the model, and the other is the time for the operations on the data structure such as membership query or insertion. The former is not directly observable in most application settings (except for temporary degradation of performance in IA-LBF), and thus allows for flexibility in choosing more complex models. The later measure of time is more critical and weighs in more heavily when considering the options, since while the former delay may lead to temporary increase in FPR, the later directly effects the query time of the data structure.

4.3 Memory

The memory used by the model is the sum of the model size and the memory of the overflow Bloom filter. The more memory we are willing to invest to handle dynamism, the better our performance will be. The trade-off between FPR and memory is shown in Figure. 5 which compares the solutions on relative space and false positive rate. The size of the circles represent the standard deviation over multiple runs. IA-LBF method achieves much lower false positive rates compared to CA-LBF methods, but at the cost of high memory usage. Section 5 discusses the implications of such trade-offs.

4.4 False Positive Rate

False positive rate of any LBF comes from the false positive rate of the classifier as well as that of the overflow Bloom filter. As mentioned in [13] any machine learned model can be attacked with



(a) CA-LBF I (b) CA-LBF II (c) IA-LBF

Figure 3: Adaptive Bloom filter Architectures

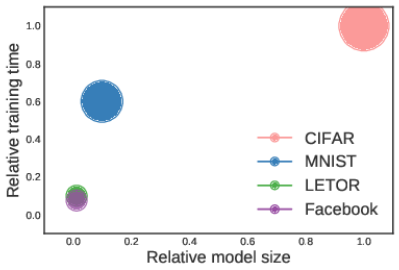


Figure 4: Effect of model complexity on time and space

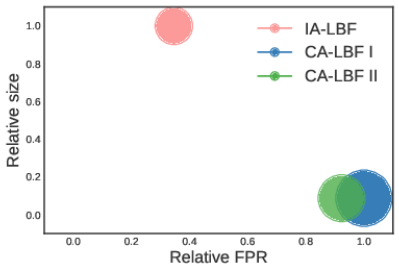


Figure 5: Relative trade-off between memory and FPR

adversarial examples to create arbitrarily high false positive rates. For our application, we define the false positive rate as the ratio of the samples generated from the distribution of negative data that were falsely classified as false positive by the structure.

5 EXPERIMENTS AND RESULTS

In this section we present the experiments conducted to evaluate the performance of each of our methods on a range of datasets of various characteristics. We study the methods on each dataset to provide a comprehensive insight on strengths and weakness of the methods in different scenarios, and the trade-off that they can provide. Based on the application, the data being inserted comes in a wide variety. We start with a relatively small and simple check-in dataset, where the power of classifier is evident, without much of it's overheads being apparent. We then gradually move on to more complex and larger datasets, where the size and complexity of data highlights the adaptability issue and demands a trade-off between false positive rates, memory and time.

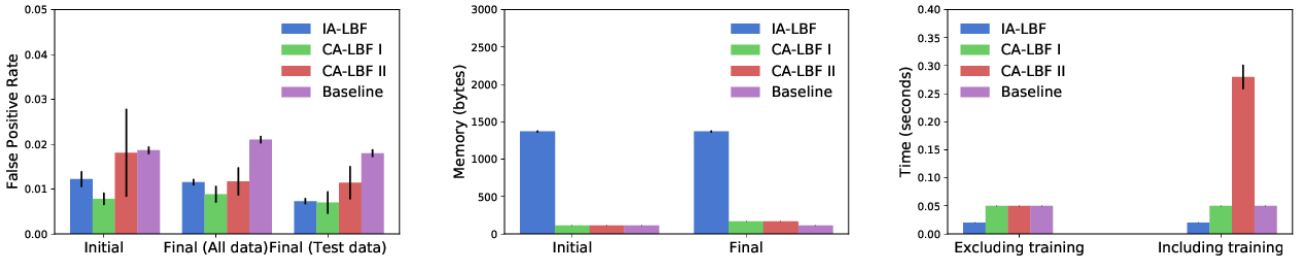
5.1 Datasets

For our experiments we selected datasets with different characteristics from different areas of applications. We have low dimensional datasets such as Facebook Check-in which allows us to use shallow models that are fast to train and execute, where the impact of underlying filters contribute to most of the time. Result on such datasets show that if classifier can adapt to the new data inexpensively, we need not invest on memory expensive indexes. On the other extreme, we have extremely high dimensional datasets such as images in CIFAR10 which are 32x32 dimensional. These datasets require complex models and significant investment in time. For such datasets we find that it may be preferable to invest on the index rather than spending significant time in adapting the classifier. Our datasets also varies in number of samples, ranging from 12000 images in CIFAR10 to 1M entries in LETOR. The size of the dataset effects both the quality of the trained model, the false positive rates of fixed size Bloom filters and the memory usage of DPBFs. The results show that the memory cost of such large datasets may be too detrimental for index adaptive methods thus favoring classifier adaptive solutions.

5.2 Facebook Check-In

Dataset. The dataset [4] consists of check-in information of 3861 users over a period of time. For the experiments we chose a subset of the dataset containing 20000 entries with 5 features. The data was split in half. One half was used to initialize the data structures and the other half was inserted later.

Classifier. We used a feed forward two layer neural network with *relu* activation implemented on Tensorflow [1]. Each hidden layer is of size 10. The classifier achieves an accuracy of 93% on the initialization set.



(a) False positive rate on entire data **(b) Memory usage** **(c) Average insertion time for 1000 elements**

Figure 6: Comparison of our methods on Facebook Check-in dataset

Observations. Figure. 6 shows the performance of our methods on this dataset. Due to relatively smaller size of dataset, the classifier achieves a relatively low accuracy. This causes higher initial false positive rates. But as we insert more data, the classifier of CA-LBF I and CA-LBF II learns the pattern better, thus reducing the false positive rate even when the number of elements inserted increases, for the same configuration for underlying Bloom filter. Compare this to the false positive rate of the baseline, which increases with insertion.

The insertion time of the data structures are similar if we ignore the training time. This may be reasonable if the training happens periodically in the background, where the effect is invisible to the end user. But the differences appear when we consider training times. CA-LBF II suffers significantly because it has to train from scratch a new model.

Finally, we observe that the memory usage of IA-LBF suffers significantly compared to other methods, both before and after insertion. While IA-LBF provides better FPR performance, for a small dataset, where the performance of other methods doesn't suffer too much from insertion, it is not a worthwhile trade-off.

5.3 LETOR

Dataset. We used MQ2008 dataset from LETOR v4.0 [15]. The subset we used consists of 1M query-results pairs with binary labels and 47 features. This is a much larger dataset and demonstrates the effects of insertion on memory and false positive rates clearly.

Classifier. We used a feed forward two layer neural network with *relu* activation implemented on Tensorflow. Each hidden layer is of size 50. The classifier achieves an accuracy of 96% on the initialization set.

Observations. Figure. 7 shows the performance of our methods on LETOR dataset. The initial false positive rate is more consistent because of higher accuracy of the classifier. But due to large number of insertions, the false positive rate shoots up, even with retraining, because of the increase in false negatives, which leads to more insertions to the bloom filter. The false positive rate goes up almost six times in CA-LBF methods whereas IA-LBF is able maintain almost the same FPR as the target.

While we see a huge memory overhead for IA-LBF in this dataset as well, the method still offers a desirable trade-off for an application that is willing to sacrifice memory in order to achieve better FPR.

Insertion time displays similar trend. The times are comparable if training time is ignored. Otherwise, CA-LBF II method suffers significantly.

The two cases above uses datasets with predefined features which are simple enough for relatively small models to provide significant advantages. Below we see cases where a lot of investment is required in the model to achieve a reasonable performance.

5.4 CIFAR-10

Dataset. We used a subset of CIFAR-10 dataset [9] consists of 12000 32x32 colour images of 2 classes, with 6000 images per class. There are 10000 training images and 2000 test images. The dataset was split into two. One half was used for initialization for each method with target false positive rate of 0.01, and the other half was inserted into the data structure later.

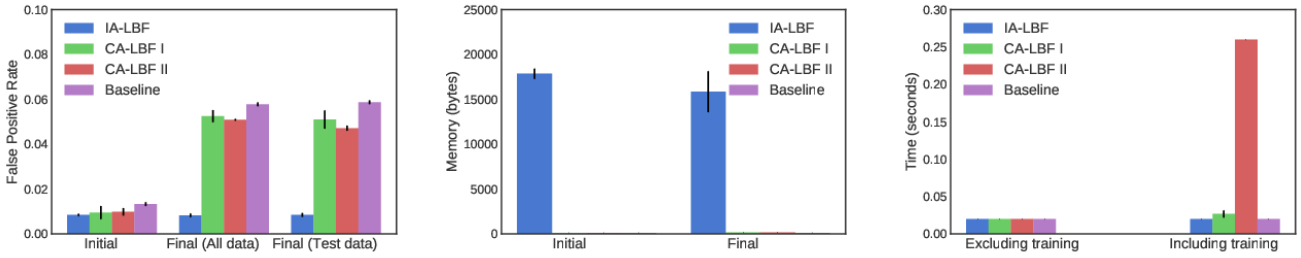
Classifier. ResNet [7] was used for the task. We used the tensorflow implementation from https://github.com/tensorflow/models/tree/master/official/resnet.

The model consisted of a 2D convolution layer with 16x16 window and shift of 3, 7 residual layers and a batch normalization layer and final fully connected softmax layer. The convolution layers used global average pooling. The layers use Relu activation. Momentum optimizer is used to optimize the neural network.

This classsifier achieves an accuracy of 97% on the dataset.

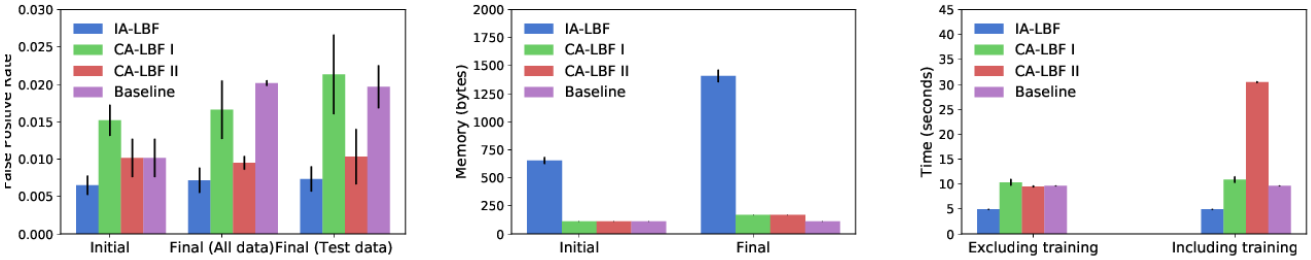
Observations. Figure. 8 shows the performance of each method. As expected, the false positive rates of each method increases with insertion. The method that performs the best in this metric is IA-LBF, because of the adaptability of the underlying DPBF. The CA-LBF methods suffer relatively due to insertion because i. Both classifiers contribute to the false positive rate, and, ii. the classifiers fail to adapt sufficiently to the new data. As discussed in Section 4.4, false positive rate is a tricky measure when machine learning is involved. Figure. 8a shows that the false positive rate suffers more if we measure it on previously unseen data.

IA-LBF uses significantly more memory compared to the other methods. This clearly presents us with a choice of trade-off. If the concerned application can afford a decay in performance of data



(a) False positive rate on entire data **(b) Memory usage** **(c) Average insertion time for 1000 elements**

Figure 7: Comparison of our methods on LETOR dataset



(a) False positive rate on entire data **(b) Memory usage** **(c) Average insertion time for 1000 elements**

Figure 8: Comparison of our methods on CIFAR 10 dataset

structure's false positive rate metric, we can save memory by using CA-LBF methods. On the other hand if the application is sensitive to the false positive rate, we have the option to sacrifice more memory to achieve the desired result.

Insertion time is roughly comparable for each method if we ignore the training times of CA-LBF methods. This is often reasonable assumption based on the application. But if we include the training time, the performance of CA-LBF II degrades significantly, because of higher training time when training from scratch.

5.5 MNIST

Dataset. The MNIST database of handwritten digits [10] has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image. For our experiments, we binarized the dataset by labelling all numbers from 0 to 4 as positive and those between 5 to 9 as negative. MNIST is also an image dataset like CIFAR-10 discussed in Section. 5.4, the few differences in characteristics like complexity of the classifier and the size of the data highlights some of the strengths and drawbacks of the proposed methods.

Classifier. LeNet [11] was used for the task. We used the tensorflow implementation from https://github.com/ganyc717/LeNet.

The model consisted of three blocks of 2D convolution layer, max pool layer and local response normalization layer followed by two blocks of fully connected, dropout layer and a final fully connected softmax layer. We used ADAM optimizer to optimize the network.

Note that the dataset is much larger that CIFAR 10 (Section. 5.4). Larger dataset, along with simple monochrome images allows a smaller model to reach accuracy of 95%.

Observations. Figure. 9 shows the performance of each method. Here we once again see the effect of inserting a larger set of data. IA-LBF is much more resilient in maintaining a reasonable false positive rate. While the CA-LBF methods are able to adjust, thus beating the baseline, they still incur a high penalty upon insertion.

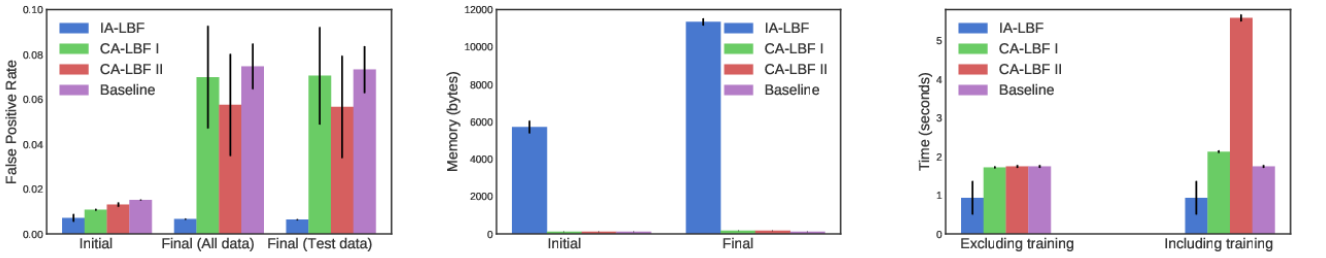
The memory usage issue of IA-LBF is more pronounced due to larger size of dataset in the MNIST dataset as compared to CIFAR data. This is a result of more examples being inserted into the data structure, both due to higher number and relatively lower accuracy as compared to CIFAR.

The insertion time is lower when compared to CIFAR due to the simpler model, CA-LBF methods, especially CA-LBF II suffers significantly.

5.6 Observations from the Case Studies

Based on our study of our methods on the different datasets we make the following observations.

Impact of Classifier Retraining Cost. A larger, more complex model leads to higher retraining cost. This is observed when we compare performance of our methods on CIFAR and Facebook



(a) False positive rate on entire data **(b) Memory usage** **(c) Average insertion time for 1000 elements**

Figure 9: Comparison of our methods on MNIST dataset

Check-in dataset. In Facebook Check-in, the average cost of insertion for 1000 elements is 0.3 seconds when we consider the retraining cost, whereas in CIFAR, the insertion cost goes up to 30 seconds. This may render CA-LBF methods infeasible for certain applications.

Impact of Classifier Adaptability. The ability for a classifier to adapt impacts both false positive rate and the memory usage of learned Bloom filters. From our experiments on LETOR dataset, we see that the false positive rates of CA-LBF methods increase from 0.01 to 0.07. This is because the classifier fails to adapt to the inserted set. This causes rise in false negatives of classifier in turn causing the Bloom filter to go over it's capacity and contributing to high false positive rate. In contrast, when the classifier is able adapt well to the changed distribution, CA-LBF method performs significantly better. This is demonstrated by the experiments on CIFAR dataset where the false positives for CA-LBF methods from 0.015 to 0.02!

Impact of Size of Insertion Workload. Another factor that affects the performance is the size of the incremental workload inserted. In our MNIST experiments, we see that the memory usage of IA-LBF method goes up from 6000 bytes to 12000 bytes. That is 100% increase in memory consumption. This is because a large number of elements are inserted and since the classifier is fixed, all the new false negatives are pushed to the DPBF. We contrast this with Facebook Check-in, where, even if the classifier adaptation is similar, the memory consumption hardly varies.

Impact of Classifier and Index Adaptability on False Positive Rate. The main difference between IA-LBF and CA-LBF methods is the way they adapt to change. In classifier adaptive methods the adaptation happens at the classifier, which implies that the method will perform well if the classifier is able to adapt well to the data. In contrast, IA-LBF adapts the index, which means even if the classifier fails to adapt to the change, this can provide stronger FPR guarantees at cost of memory. This is observed when we compare the performance of the methods in Facebook and LETOR. In LETOR, the classifier fails to adapt, and as a result, IA-LBF method greatly outperforms CA-LBF. But in a case where classifier can adapt, such as Facebook, the CA-LBF performs as well as IA-LBF, but the memory cost is an order of magnitude lower.

The observed phenomenon are illustrated in Figure. 4 and Figure. 5.

6 CONCLUSIONS

We present two distinct solutions which deals with adaptability of Learned Bloom filters for incremental workloads. In IA-LBF methods we replace the Bloom filter with DPBF and the adaptation happens at the index. In CA-LBF methods the adaption happens by retraining the classifier. The feasibility of CA-LBF methods are contingent on our access to database, and the time we can afford to perform retraining. The IA-LBF methods are limited by memory consumption. Our experiments demonstrates that both the methods outperform the baseline. This leads to the conclusion that simply using a one-time trained classifier in a static scenario is not enough, and that adaptability needs to be addressed directly as we have done in this paper.

CA-LBF methods that work by adapting the classifier to the changing set are, as expected, found to work well when the classifier can adapt well to the changes, and when the model is simple so that the cost of retraining is not too expensive. This indicates some clear limitations to the learned Bloom filter approach: it will not work across the board, we need to ensure that the classifiers are adaptable and that the application gives us the time needed to retrain. We also note here that for classifier adaptive methods to have the full power of the static technique we need to have full database access, which is not usually available except in certain specific situations. We note that the binary classifier has some inherent limitation when involved in a scenario where only the positive distribution changes. For that we can consider one-class classification methods, which we leave to future work.

Finally, IA-LBF methods work well if the classifier is unable to adapt to the new distribution of the inserted data, but there is a memory cost associated with them, and so the application designer is presented with a clear trade-off in terms of memory versus FPR when this method is compared to classifier-adaptive methods.

ACKNOWLEDGMENTS

This work was partially supported by IIT Delhi-IBM Research AI Horizons Network collaborative grant.

REFERENCES

[1] [n. d.]. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

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

[3] Andrei Broder and Michael Mitzenmacher. 2002. Network applications of bloom filters: A survey. In *Internet Mathematics*. Citeseer.

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

[5] Sainyam Galhotra, Amitabha Bagchi, Srikanta Bedathur, Maya Ramanath, and Vidit Jain. 2015. Tracking the Conductance of Rapidly Evolving Topic-Subgraphs. *PVLDB* 8, 13 (2015), 2170–2181.

[6] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. 2010. The dynamic Bloom filters. *IEEE Transactions on Knowledge and Data Engineering* 22, 1 (2010), 120–133.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[8] 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*. ACM, 489–504.

[9] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[10] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[11] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. 1999. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*. Springer, 319–345.

[12] Nathaniel Mcvicar, Chih-Ching Lin, and Scott Hauck. 2017. K-mer counting using Bloom filters with an FPGA-attached HMC. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 203–210.

[13] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Related Structures. arXiv:cs.DS/1802.00884

[14] Sidharth Negi, Ameya Dubey, Amitabha Bagchi, Manish Yadav, Nishant Yadav, and Jeetu Raj. 2019. Dynamic Partition Bloom Filters: A Bounded False Positive Solution For Dynamic Set Membership. *arXiv:1901.06493* (2019).

[15] Tao Qin and Tie-Yan Liu. 2013. Introducing LETOR 4.0 Datasets. arXiv:cs.IR/1306.2597