kBF: a Bloom Filter for Key-Value Storage with an Application on Approximate State Machines

Sisi Xiong*, Yanjun Yao*, Qing Cao*, and Tian He†

*Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN, US
Email: {sxiong, yyao9, cao}@utk.edu

†Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN, US
Email: {tianhe}@cs.umn.edu

***Abstract*—Key-value (k-v) storage has been used as a crucial component for many network applications, such as social networks, online retailing, and cloud computing. Such storage usually provides support for operations on key-value pairs, and can be stored in memory to speed up responses to queries. So far, existing methods have been deterministic: they will faithfully return previously inserted key-value pairs. Providing such accuracy, however, comes at the cost of memory and CPU time. In contrast, in this paper, we present an approximate k-v storage that is more compact than existing methods. The tradeoff is that it may, theoretically, return a null value for a valid key with a low probability, or return a valid value for a key that was never inserted. Its design is based on the probabilistic data structure called the "Bloom Filter", which was originally developed to test element membership in sets. In this paper, we extend the bloom filter concept to support key-value operations, and demonstrate that it still retains the compact nature of the original bloom filter. We call the resulting design as the kBF (key-value bloom filter), and systematically analyze its performance advantages and design tradeoffs. Finally, we apply the kBF to a practical problem of implementing a state machine in network intrusion detection to demonstrate how the kBF can be used as a building block for more complicated software infrastructures.**

I. Introduction

Key-value (k-v) storage has been used as a crucial component for many different network applications, such as social networks, online retailers, and cloud computing [1], [2]. Example implementations include Dynamo [3], Cassandra [4], Memcached [5], Redis [6], and BigTable [7]. By storing most data in the main memory, these implementations allow query, update, and delete operations of key/value pairs. The particular keys and values can be highly flexible: an online retailer can use the keys to represent the product catalog IDs, and the values to represent their associated metadata, such as its category or the latest updated price. Because of its importance, k-v storage has been heavily fine-tuned for the best performance in terms of cache usage, load balancing, and response time.

One observation of these different key-value storage services is that they are deterministic. For example, if a key-value pair was previously inserted, a query on the key should always return its value. Although this is desired, doing so requires storing and processing complete information of the keys and values, which introduces overhead. Therefore, in this paper, we develop a highly compact, low-overhead, but approximate k-v storage service, by taking inspiration from the bloom filter [8] (see Section II for a brief survey). However, despite of their usefulness, bloom filters are designed for testing set memberships, not key-value operations. Therefore, our goal is to develop an enhanced version of the bloom filter, so that it is able to support key-value operations. Specifically, it should support these following APIs: **insert**, **update**, **delete**, and **query**. Our goal is to make this data structure highly compact, by making the tradeoff that we allow false positives to occur, just like the bloom filter. This means that querying a non-existent key may return a value that does not actually belong to it. Although this may sound counter-intuitive at first, we argue that there exists applications that such approximate results are still acceptable: for example, an online shopping service may store product ID and its category as key-values to answer users' queries. If the kBF is used, occasionally querying a product ID that does not exist may return a category that is still valid. This, however, will not be a problem because such false products are fictional in the first place, and adding additional information to it, such as category, does not make them valid. By using the kBF, on the other hand, allows us to speed up such query processing considerably and to narrow down the list of products quickly.

Developing this data structure, however, is particularly challenging for two reasons. First, the original bloom filter uses bit arrays to keep track of the membership of elements. The keys and values, however, are much more irregular in length, and can not be directly stored into typical bloom filters. Second, the original bloom filter does not support deletions. Although later research, such as the counting bloom filter [9], partially addressed this problem by using counters to replace bits of a typical bloom filter, it only keeps the frequency of elements instead of the values of elements themselves.

Because of these challenges, adding the key-value support is not straightforward. One naive approach is that we can store, for each k-v pair, an item that represents the string

978-1-4799-3360-0/14/$31.00 ©2014 IEEE

$(key, value)$ into the bloom filter. However, to query a key later, all the possible values must be tried to construct possible strings for membership tests. This approach is clearly not scalable when there is a larger number of possible values, hence not practical.

A more recent approach [10] considers the problem under the assumption that values are limited in range, and are distributed within $[1, V]$, where V is a positive integer. The approach then modifies the conventional bloom filter to use cells instead of bits to hold the values directly. If there is a collision in a cell, this cell is then marked as "Don't Know (DK)". To query the value for a key, if at least one of its k hashed locations has not been marked as DK, the query will be successful. On the other hand, if all cells are marked as DKs, then the query returns a DK to the user. The problem with this approach is that it is not designed for arbitrary key-value string pairs, and it will also encounter reduced performance (returning many DKs) when more cells become populated. We demonstrate the latter problem in Section VI by comparing this approach with the kBF through an application study.

The approach we present in this paper aims to address these problems by supporting k-v operations with predictable performance and accuracy. We call it the "key-value bloom filter (kBF)". In particular, it has the following three key contributions: first, to address the challenge of arbitrary key-value pairs, we propose a method to encode the values into a special type of binary encodings that can fit into the cells of bloom filters easily. These encodings are designed to be resilient to collisions, i.e., insertions and queries can still be effectively handled when one or more collisions occur in a cell. In particular, the decoding allows using k hashed locations collaboratively, rather than using any single one of them, so that the successful decoding ratio can be greatly improved. Second, to address the challenge to handle a very large number k-v pairs, we design the kBF to be elastic, so that its capacity can grow and shrink as needed while ensuring that the desired query performance is achieved. To this end, we develop growth and compaction operations on the kBF to support its capacity changes. Third, to address the challenge to achieve predictable performance, we systematically analyze the capacity and decoding ratio of the kBF to demonstrate its performance limits. We derive closed form results to this end, so that we can closely monitor the runtime performance of the kBF to ensure a satisfactory quality of service.

In summary, kBF represents a novel type of the bloom filter that supports key-value operations using compact memory storage. To further illustrate its effectiveness, we demonstrate through a specific application example: we use it as a building block to enforce TCP state transition rules by developing an approximate concurrent state machine (ACSM). Using ACSMs, a router can efficiently keep track of many regular expression matchings simultaneously to detect potential intrusions.

The remaining of this paper is organized as follows. Section II presents the related work. Section III describes the problem formulation and the design of the kBF. Section IV analyzes its performance tradeoffs. Section V evaluates the performance of the kBF through experiments. Section VI develops an application of the kBF for detecting TCP flag transitions. Finally, Section VII concludes this paper.

II. Related Work

In this section, we describe related work in three parts: first we describe the original Bloom Filter design, then we describe its variants, and finally, we describe the related work to the network applications of the Bloom Filter.

The bloom filter, originally developed by Burton H. Bloom [8], is a space efficient randomized data structure that answers the question about membership tests. Recently it has received great attention in the networking area [11], [12], [13]. Specifically, the bloom filter allows insertions and queries of elements in sets, by hashing an element to k different locations in a bit array of m bits. To add an element, each of the k bits is set to 1. To query it, each of the k bits is tested against 1, and any 0 found will tell that the element is not in the set. In this way, no false negatives will occur, but false positives are possible, since all k bits might have been set to 1 due to other elements have been hashed to the same positions. The bloom filter is highly compact: it needs 10 bits to store each element to achieve a false positive rate of 1%, independent of the number and size of the inserted elements. Therefore, in situations where only limited on-chip RAM is available, a bloom filter becomes particularly useful.

After the original Bloom Filter was proposed, many variants followed [13], [14], [15]. One relevant work is the counting bloom filter [9], which has m counters along with m bits. This way, the CBF can support not only deletion operations, but also frequency queries. However, the CBF is not designed for key-value operations, hence, is also significantly different from our work.

In recent years, the Bloom Filter has been widely used in the context of network intrusion detection and measurement. Examples include identifying heavyhitters [16], iceberg queries [17] and packet attribution [18]. One typical application is approximate state machine, which means that one wants to monitor a flow's state in a finite state machine. For example, in [10], video congestion control and P2P traffic analysis are utilized to investigate the performance problems. Different from the approach presented here, however, this approach makes more strict assumptions on the range of values, and does not support arbitrary value strings.

III. Design of kBF

In this section, we introduce the design of the kBF. We first present the problem formulation. Then, we present an overview of its structure. Finally we present a detailed description of its components and related algorithms.

A. The Problem Formulation

We first present the problem formulation. Assume that we have a collection of n key-value pairs (k_i, v_i), where $i \in [0, n-1]$. The keys and values can be arbitrary strings. We

aim to develop kBF to support the following four operations for the stored k-v pairs:

- **insert**(*key*, *value*) //insert a key-value pair
- **update**(*key*, *new_value*) //update a value for a key
- **query**(*key*) //query the value for a key
- **delete**(*key*) //delete a key and its associated value

B. Architecture Overview

In this section, we present the architecture of kBF, which is shown in the Figure 1. This figure focuses on the insertion and query operations, and we will describe the delete and update operations later. The overall procedure works as follows. When $(key, value)$ pairs are inserted, the algorithm first performs an one-to-one conversion from their values to encoded binary strings, using a secondary kBF (s-kBF) as an assisting component. The pairs of the keys and their encodings are then inserted into the main kBF, which serves as the primary storage for the incoming data. On the other hand, if a key is provided for a query operation, the main kBF will return a total of k encoded strings. These strings are fed into a decoding algorithm to obtain the corresponding encoding for the key, which is further converted into its original value using a polynomial regression based algorithm. The constructed $(key, value)$ pair will be returned to the user.

In the following sections, we describe the details of this process, including how the encodings are formulated, the details on specific operations, and the growth/compaction of kBFs.

C. Encodings of Values

The central idea of the kBF is that it maps the values, represented by a set $V = \{v_1, v_2, ..., v_n\}$, into a set of binary strings. Specifically, such binary strings, denoted as $b[v_i]$, are constructed according to the following two rules:

- Each value v_i has a unique string $b[v_i]$.
- The XOR result of any two strings, i.e., $b[v_i] \oplus b[v_j]$, should be unique among themselves, as well as to the values of $b[v_i]$.

Given n values, the number of their pairwise combinations is $C(n, 2)$, or $\frac{n(n-1)}{2}$. Therefore, the minimum length of the binary string, as P, must conform to:

$$2^P \geq \frac{n(n-1)}{2} + n$$

For example, if there are only four values that need to be encoded, a total of four bits is sufficient, by encoding them as $\{0001, 0010, 0100, 1000\}$, so that their combinations will not cause any collisions. On the other hand, if $n = 100$, the theoretically minimal P is 13 ($2^{13} > C(100, 2) + 100$). Avoiding collisions is important because this way, the pairwise XOR results can be uniquely decoded. The entire approach is slightly similar to the CDMA encoding scheme where the XOR operation is also used. We emphasize that the procedure for finding encodings only runs once and is done offline. Therefore, powerful computational resources can be used to find a sufficient number of valid encodings for future uses.

Note that sometimes the minimal value of P may not be achieved. For example, if there are 7 values, the minimum P is 5. However, through an exhaustive search, with 5 bits, there is no encoding scheme to fulfill the two requirements mentioned above. Therefore, we develop a greedy algorithm for finding encodings when the number of values is large. We do not adopt the exhaustive search due to computational overhead considerations. The algorithm is shown in Algorithm 1. As illustrated, we start the search by setting the first encoding to 1. We then increase the successive encoding repeatedly by 1. If there is no collision, then the new encoding is admitted into the set of found encodings. Otherwise, the next encoding is tested. Figure 2 shows the gap between the theoretical minimal value and the actual value. We also find that even for a large number of encodings, the gap is quite small. For example, there is a theoretical lower bound of $P = 28$ for $2^{14}(16,384)$ encodings, and the greedy method is able to find a solution with $P = 30$.

Algorithm 1 Search Algorithm for Encodings

```
1:  procedure ENCODING SEARCH(n)
2:      v_0 = 1
3:      insert v_0 into a bloom filter BF
4:      counter ← 2
5:      for j = 1 → n − 1 do
6:          while True do
7:              v_j = counter
8:              calculate the XOR result for v_j
                    and v_i, for i ∈ [0, j − 1]
9:              if there is no collision for v_j then
10:                 Insert v_j and all XOR results into BF
11:                 counter ← counter + 1
12:                 break while
13:             else
14:                 counter ← counter + 1
15:             end if
16:         end while
17:     end for
18: end procedure
```

The Algorithm 1 also adopts an optimization in steps 3 and 9 to speed up the search process for a large n. This optimization is based on the classic bloom filter. Specifically, it will insert all admitted encodings, as well as their pairwise XOR results, into a conventional bloom filter. For every new encoding being tested, its combinations with existing encodings are queried against this bloom filter to determine if it has already been inserted. If yes, then there is a high probability that a collision has occurred. The algorithm then proceeds to the next encoding. If a negative result is returned by the bloom filter, it is guaranteed that there is no collision for this new encoding because the conventional bloom filter design never returns false negatives. This optimization allows each new encoding to be admitted or rejected in constant time. Therefore, it considerably speeds up our search process.

D. Conversions from Values to Encodings

We next describe how values are converted into encodings, which involves a secondary kBF (s-kBF) that operates on

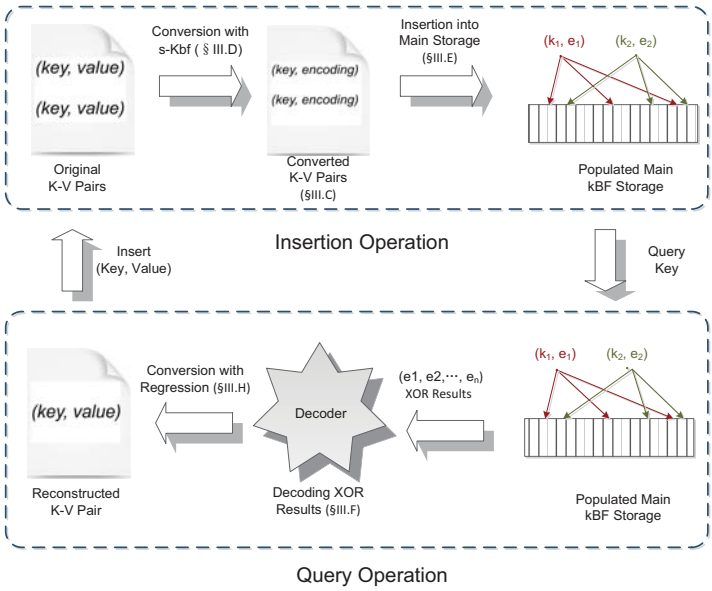

Fig. 1. The kBF algorithm architecture, note that when applicable, the corresponding section in the paper is located under the algorithm block

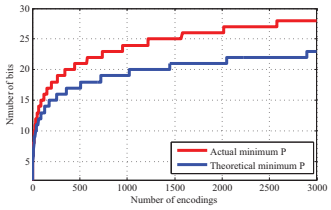

Fig. 2. Relation between the size of encodings and the number of bits

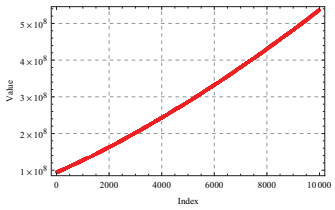

Fig. 3. Encoding value trend

values instead of keys. Specifically, whenever a value needs to be converted, it is queried against the s-kBF to decide if it has already been assigned an encoding. If yes, then the encoding will be used. Otherwise, a new encoding is obtained from the pool of available encodings, and is assigned to this value. The pair of $(value, encoding)$ is then inserted into the s-kBF for later queries. Meanwhile, the reverse pair of $(encoding, value)$ is stored in a separate lookup table for later conversions from encodings to values. Because s-kBF only stores $(value, encoding)$ mappings, it is much smaller than the main kBF. Its operations are exactly the same as the main kBF, as described in the next section.

E. Operations of the kBF

We now describe the central operation blocks of kBF, i.e., their operations. Different from a conventional bloom filter, each cell in the kBF consists of two components: a counter and a possibly superimposed encoding result. The counter keeps track of how many encodings have been inserted: 0 means the cell is empty, 1 means one encoding has been inserted, and so on. The encoding part, on the other hand, contains either an original encoding, or the XOR results of two or more encodings that are mapped to the same cell. In practice, we use a 32-bit cell with a 3-bit counter and a 29-bit encoding.

Algorithm 2 kBF Insert Algorithm

```
1:  procedure INSERT(x)                          ▷ Insert operation
2:      for j = 1 → k do
3:          i ← h_j(x)
4:          if B_i.counter == 0 then
5:              B_i.counter ← B_i.counter + 1
6:              B_i.value ← S_i
7:          end if
8:          if B_i.counter > 0 then
9:              B_i.counter ← B_i.counter + 1
10:             B_i.value ← B_i.value XOR S_i
11:         end if
12:     end for
13: end procedure
```

When the **insert** occurs, kBF first finds k hashed cells. The counter for each cell is increased, and the encoding is superimposed into the cells by performing the XOR operation with the existing contents stored by each cell. Algorithm 2 describes this process.

The second operation, **query**, works as follows. For each of the k cells, it will obtain the superimposed encodings as well as their associated counters. The original encoding can be recovered as long as one of the following two rules is satisfied:

- if one cell has a counter of 1 and stores an original encoding,
- if all cells have counters of more than 1, but the intersection of their stored encoding sets is unique.

As an example for the second rule, suppose we have two cells, both of which contain the superimposition of two encodings. If the first cell contains X and Y, and the second cell contains X and Z, then the original encoding that hashes to both cells must be X. Algorithm 3 describes this process. Note that we will describe the details of the decoding process of step 6 in the next section.

Algorithm 3 kBF Query Algorithm

```
1: procedure QUERY(x)                                   ▷ Query operation
2:     for j = 1 → k do
3:         i ← h_j(x)
4:         Add B_i.value to StateQueue
5:     end for
6:     State ← Decoding(StateQueue)
7:     return State
8: end procedure
```

We next describe the **delete** operation. This operation is based on our observation that for any encoding a, a XOR $a = 0$. Therefore, we can describe this procedure in Algorithm 4.

Algorithm 4 kBF Delete Algorithm

```
1: procedure DELETE(key, encoding)                      ▷ Delete operation
2:     for j = 1 → k do
3:         i ← h_j(key)
4:         if B_i.counter > 0 then
5:             B_i.counter ← B_i.counter − 1
6:             B_i.value ← B_i.value XOR encoding
7:         else
8:             report error
9:         end if
10:    end for
11: end procedure
```

Algorithm 5 kBF Update Algorithm

```
1: procedure UPDATE(key, encoding)                      ▷ Update operation
2:     encoding_old ← Query(key)
3:     for j = 1 → k do
4:         i ← h_j(key)
5:         if B_i.counter > 0 then
6:             B_i.value ← B_i.value XOR encoding_old
7:             B_i.value ← B_i.value XOR encoding
8:         end if
9:     end for
10: end procedure
```

Finally, we can implement the **update** algorithm by first querying the key to obtain its encoding, then delete the key with its encoding, and finally insert the key with its new encoding. The process can be combined together and Algorithm 5 describes its details.

F. Decoding Superimposed Encodings

We next describe how to obtain original encodings from their superimposed results. Given the way we constructed the encodings, we can always obtain unique results if only two of them are superimposed. For more than two encodings' XOR results, we may obtain multiple possible solutions.

Therefore, the critical step is, given the number of items and their XORed result, how to obtain all possible combinations of original encodings. We first consider the simple case: if there is an XOR result for two encodings X and Y, then there are two possibilities: if $X = Y$, the XOR result is 0. This is impossible to decode because any X may be possible. If $X \neq Y$, then a unique set of $\{X, Y\}$ can be found. By pre-constructing a bloom filter that has all encodings, we can find this unique set in $\mathcal{O}(N)$, by iterating through all items, calculating its XOR result with $X \oplus Y$, and checking if the result can be found in the pre-constructed bloom filter.

We now consider the more complicated case where three encodings are hashed to the same cell. Since we only know their XOR result, and the encoding scheme we designed earlier does not guarantee that the encodings give unique values when three or more of them are combined, we only provide an opportunistic approach for three-item decoding. For more than three items that are mapped to the same cell, we consider the cell to be non-decodable [1].

The opportunistic algorithm works as follows. For all available encodings, we store all the pair-wise XOR results of them into a bloom filter L. We denote the XOR result as R, and we know that $R = X \oplus Y \oplus Z$, for unknown X, Y, and Z. During the decoding phase, we iterate through all encodings. For each encoding E, we calculate $E \oplus R$. If $E = X$, we know that $E \oplus R = Y \oplus Z$. Therefore, we can use the filter L to check if $E \oplus R$ exists. If yes, then there is a hit, and we can find Y and Z as they are unique. Finally, we return the set of all found (X, Y, Z) as the result.

G. Growth and Compaction of kBFs

Just like a normal BF, a kBF has its capacity in terms of how many item insertions it is able to support at most. Although its capacity can be statically allocated if we know the maximum number of k-v pairs, in real applications, it happens that we do not know such information in advance. Therefore, we present the growth and compaction operations of kBFs for dynamic operations.

For the growth operation, we monitor the number of inserted k-v pairs for a constructed kBF. Whenever this reaches near its maximum, we can allocate another kBF of the same size for new k-v pairs. On the other hand, if we detect that an existing kBF has too few active k-v pairs after repeated deletions, we can start the compaction operation. This operation is facilitated by the bit-vector nature of kBFs. Given two k-v sets, suppose that they are represented by two kBFs, L_1 and L_2, we can calculate the kBF that represents the union set $L = L_1 \cup L_2$ by taking the XOR operation of their kBF cells: $Cell_L = Cell_{L1} \oplus Cell_{L2}$. For the counters, we can add them together to become the counter for the new cells. Observe that a tradeoff of this operation is that at the same time it saves memory space in compaction, it will lose some information during the XOR operations.

H. Conversion from Encodings to Values

The final step in the operation is to convert encodings to values for query results. To this end, recall that we maintained a table of $(encoding, value)$ mappings when encodings are created for values. In this table, all encodings in the table are sorted in the ascending order to simplify the lookup process later.

[1]Note that this cell may become decodable again when a delete operation occurs, causing one item from this cell to be canceled out.

To save memory accesses,given the encoding, instead of using a conventional binary search to find the value for an encoding, we follow a regression approach in this step. Specifically, as illustrated in Figure 3, valid encoding values usually form a curve that can be approximated with a polynomial function. We choose a quadratic function for this approximation, i.e., we find $y = f(x)$, where $x \in [0, n-1]$ as the index, and y is the encoding value. We then find the inverse function $x = f^{-1}(y)$, so that we can calculate the index given the value of the encoding. Once the index is found, the string for the value in $(key, value)$ pairs can be directly found by using the index to access the mapping table.

However, one challenge is that the f function is not 100% accurate. To find the true location after calculating the index, we search from the index by observing that the average step increase of the encoding values can be determined in advance. Then, based on the difference of the currently found encoding and the target encoding, we can divide it by the average step of encodings, and move the index accordingly, until the index finds the true encoding value. We describe this procedure through an example. Suppose we have a table of $10,000$ encodings, and find its quadratic function as $f(x) = 92884900 + 32952.9x + 1.151x^2$. The average step of all encodings is 44294, which is pre-determined. Let us suppose, in one query, the encoding returned is 237551267. According to this formula, the first index is found as 3868. By accessing the encoding corresponding to the location 3868, we find it as 237525822, which has an error of 25445 compared to the target encoding being searched. By using the average step size, the index will search using a step of 1. After two steps, the true index is found at 3870. This way, only three memory accesses are needed to find the encoding index and its associated value, which is much faster than the binary search method with an average number of memory accesses of 14.

IV. Analysis of kBF

In this section, we analyze the capacity and error rate of kBF using a theoretical analysis. The challenge of this analysis is that a bloom filter is constructed using several parameters, including its size m, the number of hashing functions k, and the number of k-v pairs n. It has been pointed out that to minimize the false positive rate, there exists an optimal k given a pair of n and m, where $k_{opt} = \frac{m}{n}\ln(2)$ [12]. On the other hand, to maintain the false positive rate of the filter below a threshold p, we know that

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

This formula shows that the parameter m must grow linearly with the size of n, or conversely, given an m, there exists an upperbound of n, over which the false positive rate can no longer be sustained. We can therefore define the following concept of capacity.

Definition The p-capacity of a bloom filter is defined as the maximum number of items that can be inserted without violating the false positive probability p.

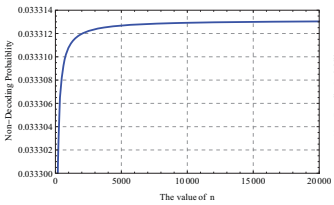

Fig. 4. Relation between n and the non-decodable probability

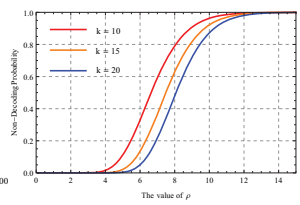

Fig. 5. Relation between ρ and the non-decodable probability

It is clear that the p-capacity can be derived using

$$\text{p-capacity} = -\frac{m(\ln(2))^2}{(\ln(p))}.$$

Also observe that when p-capacity is reached, the optimal number of hashing functions k is only related to p, as $k = -\frac{\ln(p)}{\ln(2)}$. In the kBF, whenever there are too many items inserted, we will allocate a new kBF with the same size. Therefore, we can guarantee that the false positive rate of each kBF will not be larger than p.

Now we derive the distribution of encoding superimposition for a kBF. Assume that this kBF has been inserted with n items. We use $c(i)$ to denote the number of encodings that are inserted into the ith cell. If this number is 3 or more, we consider that this cell is non-decodable. The probability that this counter is incremented j times is a binomial random variable as

$$P(c(i) = j) = \binom{nk}{j}(\frac{1}{m})^j(1 - \frac{1}{m})^{nk-j}.$$

Therefore, the probability that any counter is at least j is

$$P(c(i) \geq j) = \sum_{i=j}^{nk}\binom{nk}{i}(\frac{1}{m})^i(1 - \frac{1}{m})^{nk-i}.$$

Although it is relatively hard to obtain the closed form results for this particular function, we simplify it by observing in our setting, the value of nk and m are both quite large. Therefore, we can use the extreme limits of the formulas (by calculating $n \to \infty$ and $m \to \infty$) to approximate their original forms. We also use numerical results to demonstrate that this approach is indeed accurate.

The key observation we use to simplify the derivation comes from [19]. The result is concerned with the urn-ball model, of which our model of a bloom filter is a special case. Specifically, it states that if n balls are randomly assigned into m urns, and that each ball is equally likely to fall into any of the urns, suppose we use M_r to denote the number of urns containing r balls after the assignments are completed, we have that

$$E[M_r] = m\binom{n}{r}(\frac{1}{m})^r(1 - \frac{1}{m})^{n-r} \quad (r = 0, 1, ..., n).$$

If $n, m \to \infty$, with $nm^{-1} \to \lambda < \infty$, then,

$$\lim_{n\to\infty} E[m^{-1}M_r] = \frac{\lambda^r}{r!}e^{-\lambda}.$$

Apparently, for the case of a bloom filter, we have nk

hashing operations. Therefore, we should replace n in the formula above with nk instead. Also, by observing that $P(c(i) = j) = M_r/m$, we know that

$$\lim_{n\to\infty} P(c(i) = j) = \frac{\lambda^j}{j!}e^{-\lambda}.$$

Next, we consider the scenario that a bloom filter has not yet reached its p-capacity. Therefore, we have,

$$\frac{nk}{m} \leq \ln(2)$$

On the other hand, if a bloom filter has exceeded its p-capacity, we can define an additional parameter, ρ, as the capacity coefficient. That is, we can set

$$\frac{nk}{m} = \ln(2) \times \rho$$

Based on this, we can obtain that

$$\lim_{n\to\infty} P(c(i) = j) = \frac{(\rho \ln(2))^j}{j!} \times 2^{-\rho}.$$

Next, we can estimate the probability of three or more encodings combined together, which we deem as non-decodable. Note that this is a simplified over-estimate, because for such cases, we can still obtain multiple candidate sets, and it is possible that we can decode them with more computational overhead. Therefore, the results here serve as a lower-bound (a pessimistic value) on the capacity of a kBF. We can find this probability by

$$\sum_{j=3}^{n} P(c(i) = j) = \frac{2^{-\rho-1}\left(2^{\rho+1}\Gamma(n+1, \rho(\ln(2)))\right)}{\Gamma(n+1)} - \frac{2^{-\rho-1}\left(\rho^2\left(\ln(2)^2\right) + \rho(\ln(4)) + 2\right)\Gamma(n+1)}{\Gamma(n+1)}.$$

In this formula, the Γ stands for the gamma function. Its limit happens to be closed form as

$$\begin{aligned} P_n(\rho) &= \lim_{n\to\infty} \sum_{j=3}^{n} P(c(i) = j)) \\ &= 2^{-\rho-1}\left(-\rho^2\left(\ln(2)^2\right) + 2^{\rho+1} - \rho(\ln(4)) - 2\right). \end{aligned}$$

To verify, for the non-decodable probability of a single cell, we calculate the numerical results and plot them in Figure 4. Observe that the actual non-decodable probability for a single cell is concentrated around 0.0333313, which is the same as the predicted value of $P_n(1)$ as 0.0333132. This results shows that for a single cell, if the kBF has not reached its p-capacity, the probability that it has three or more encodings stored is no more than 3.33%, which is independent of the value of p.

Now we calculate the global decodability. We can mathematically write this as

$$1 - [P_n(\rho)^k + k \times P_n(\rho)^{k-1} \times (P(c(i) = 2))].$$

The results for this probability with different k values are plotted in Figure 5. Observe here, for $k = 10$, to maintain that virtually all decoding operations as successful (success rate is almost 1), we can only overload ρ to be less than 2.

V. Experiment Evaluation

A. The Evaluation Model

In this section, we systematically present the evaluation of the kBF. Due to the probabilistic nature of the kBF, we first focus on its errors. Specifically, there are three types of errors: false positives, false negatives, and incorrect outputs. Next, we focus on the performance of the kBF in terms of its memory overhead. A larger memory allocation (where the m increases) will provide a higher capacity, which in turn reduces ρ and the non-decodability probability.

B. Workload Generation

We generate the workload for the experiments based on the conclusions from a realistic study by researchers at Facebook [2]. Specifically, they studied several Memcached pools, and found the statistical distribution of the largest pool that contains general purpose key-value pairs. The key-size distribution in terms of bytes was found to be Generalized Extreme Value distribution with parameters $\mu = 30.7984$, $\sigma = 8.20449$, and $k = 0.078688$. The value-size distribution, starting from 15 bytes, were found to be Generalized Pareto with parameters $\theta = 0$, $\sigma = 214.476$, and $k = 0.348238$. The first 15 bytes follow a discrete distribution with a specific table (shown in [2]).

We generate 10 million key-value pairs where the size of keys follow these reported statistical parameters. The values are intended to be the most frequent ones, where a total of 3000 unique values are used. Note that such frequent values will typically constitute a majority of the $(key, value)$ instances, for which the kBF is targeted at. In this evaluation, we keep the specific keys and values random, so that the evaluation results are the most generic. The number of keys n inserted to the kBF is 10 million, and we use a false positive probability p to be between 0.001 to 0.000001 when we construct the bloom filter. Therefore, the number of hash functions k, the size of kBF m and the maximum load factor ρ can be decided. To test the case when kBF has been overloaded, we also conduct the experiments for the size of kBF m' to be $m/2$ to analyze the performance difference. The entire kBF takes between $0.5G$ to $1.5G$ bytes of RAM to build on a modern workstation, depending on the parameter selection.

C. Evaluation Results

First, to obtain the false negative error rate and the incorrect output rate, we start by inserting all 10 million keys into a constructed kBF, and then query each key for its value. We find that in this case, the incorrect output error is always zero. This can be explained by that as the filter has not been saturated, all decodings are correct if they are decodable at all. The false negative error rate, on the other hand, is plotted in Figure 6. According to the figure, the false negative errors almost do

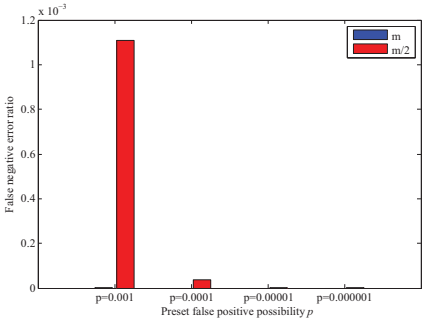

Fig. 6. False negative error ratio

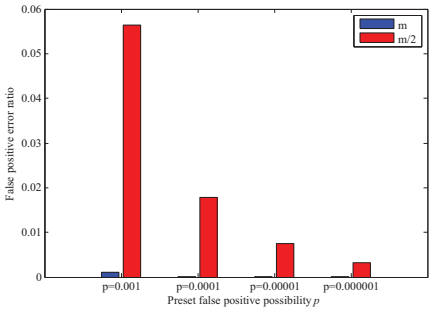

Fig. 7. False positive error ratio

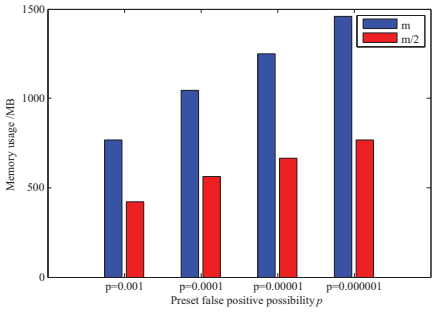

Fig. 8. Memory usage comparison

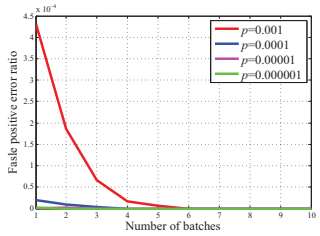

Fig. 9. False positive ratios after each batch deletion, kBF size = m

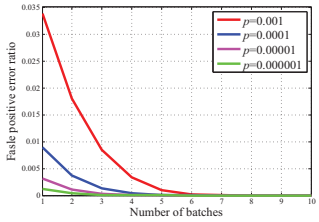

Fig. 10. False positive ratios after each batch deletion, kBF size = $m/2$

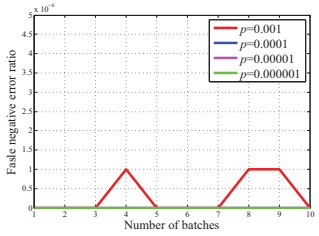

Fig. 11. False negative ratios after each batch update, kBF size = m

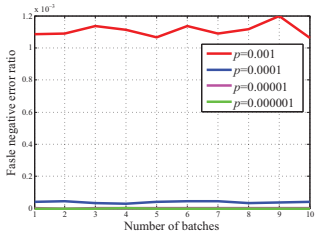

Fig. 12. False negative ratios after each batch update, kBF size = $m/2$

not exist if the p-capacity is not violated. On the other hand, if the size of the filter m is decreased by half, for larger p values, the false negative errors are more frequent, meaning that the decoding process gives more null results. This is as expected, as in this case, the filter is over-crowded.

To obtain the false positive error rate, i.e., the rate of obtaining valid values for incorrect keys, we generate another 10 million non-existent keys, and query them over a kBF that is populated with the first 10 million keys. If the kBF ever returns a valid encoding, we consider this as a false positive error. The results are shown in Figure 7. Observe that if the p-capacity is not violated, the false positive rate is close to the value of p (the conventional bloom filter false positive rate) in their order of magnitude. On the other hand, if the size of m is reduced by half, the false positive rate becomes higher, as we expected. We next plot the overhead in terms of memory usage in Figure 8. Observe that a smaller p leads to a larger memory usage, which obtains, in turn, better performance in terms of error rates. We next investigate the effects of deletions of keys. Specifically, we delete the 10 million inserted keys in batches, each has 1 million keys. We then query the deleted keys after each deletion, and plot the false positive ratios. Figure 9 and Figure 10 show the results. Observe that for a smaller p, the performance tends to be much better.

Finally, we evaluate the effects of update operations. Similar to the delete operations, we update keys with new values in batches. Note that here, instead of calculating the false positives, we are interested in false negatives, which refer to null values or incorrect values. The results are plotted in Figure 11 and Figure 12. Observe again that the performance will be much better for smaller p values and larger m sizes.

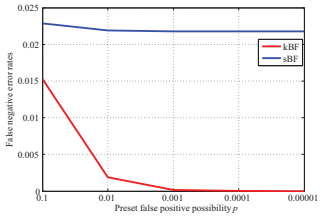

Fig. 14. False negative error rates

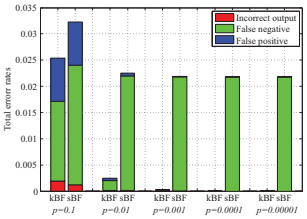

Fig. 15. Total error rates

VI. Application Case Study: TCP Flow Analysis

In this section, we describe how to use the kBF for a real application. We implement an Approximate Concurrent State Machine (ACSM) [10] based on the kBF, and compare it with the original approach in [10], which we refer to as the state-based bloom filter (sBF). For our evaluation, we choose a real dataset from CAIDA [20], which includes an hour length of traffic data. The pair of the source and destination IP address is used as flow-id, and the TCP flag is used as the state.

Specifically, the experiment has the goal of locating suspicious TCP flows by using TCP flags. This technique has been utilized in different network monitoring scenarios, such as SNORT database [21] and TCP SYN flooding attacks [22]. Whenever the specified TCP flags indicate potential problems, a warning can be generated. For example, when the "RST" bit and "FIN" bit are set, which means to reset and to terminate the TCP connection, it may indicate potential attack [22].

To detect such problems, we emulate the state transitions of TCP flows with ACSMs. Whenever a flow is encountered, we query the flow on its state. If it is a new flow, we insert this flow and its state into the kBF (or sBF [10]). If this flow is old, we will selectively update its state depending on the flow information. When a flow terminates, we delete its state

information. All insert, update, query, and delete operations are readily supported by both kBF and sBF, because in this case, the number of states is usually very small. To compare them, we choose four different preset false positive probability p from 0.1 to 0.00001 to conduct experiments. The total number of flows, n, is 908522.

The first type of errors, false positives, is related to the membership tests of bloom filters. Here, a new flow is detected, and queried against the filter. However, the kBF or the sBF returns that the flow is old, and will fail to insert the flow and its state information, due to a false positive error. The second and third types of errors, false negatives and incorrect outputs, indicate that the kBF or sBF returns a null state or an incorrect state for a valid flow. This may be caused by decoding failures, or for those flows that failed to be inserted due to false positives.

The performance gap between the kBF and the sBF is mainly in false negative errors. According to Figure 14, as the preset p value decreases, false negative errors of kBF decreases dramatically. However, false negative errors of sBF almost stay constant, due to that it simply returns null value for those cells with two or more flows. Furthermore, in its update process, false negative errors will accumulate due to previous overlappings, which leads to almost invariant false negative errors even though the size of bloom filter increases. The kBF, in contrast, still maintains part of the flow information even when three or more encodings are superimposed, as individual encodings can still be recovered later if delete operations occur. Figure 15 shows the total errors of kBF and sBF. Again, we observe that kBF performs much better in terms of reducing errors.

Finally, in terms of finding suspicious flows with certain TCP flags, after querying state of each flow, we find that there are 900 and 1658 flows with the flags of FIN and RST accordingly in the dataset. These flows can be marked with suspicious for further analysis.

VII. Conclusions

In this paper, by using the classic bloom filter as a base design, we extend it into a approximate key-value storage scheme called the kBF. We present a comprehensive investigation on the algorithm design of the kBF, analyze its performance in storing large datasets, and evaluate its performance in both synthetic workloads and a real application study. According to our experiment results, the kBF is highly compact, and supports insertion, query, update and deletion operations with adjustable error ratios. Compared to deterministic schemes, the kBF is more suitable to be implemented in devices with limited RAM space and timing constraints, as long as approximate results are tolerated by application semantics. We also demonstrate, through an application case study for detecting suspicious TCP flows, the kBF performs much better than the related approach in the literature in terms of error rates. Therefore, we believe that our study of the kBF can be beneficial for fast and low overhead key-value storage purposes in a wide range of applications.

VIII. Acknowledgement

The work reported in this paper was supported in part by the National Science Foundation grant CNS-0953238, CNS-1017156, CNS-1117384, and CNS-1239478.

References

[1] V. Vasudevan, M. Kaminsky, and D. G. Andersen, "Using Vector Interfaces To Deliver Millions Of Iops From A Networked Key-value Storage Server," in *Proceedings of the ACM SOCC*, 2012.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis Of A Large-scale Key-value Store," in *Proceedings of the ACM SIGMETRICS*, 2012.

[3] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo : Amazon's Highly Available Key-value Store," in *Proceedings of the ACM SOSP*, 2007, pp. 205–220.

[4] Apache Foundation, "Cassandra Website," http://cassandra.apache.org/.

[5] "Memcached Website," http://www.memcached.org/.

[6] "Redis Website," http://redis.io/.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] B. H. Bloom, "Space / Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, 1970.

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, Jun. 2000. [Online]. Available: http://dx.doi.org/10.1109/90.851975

[10] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Beyond Bloom Filters : From Approximate Membership Checks to Approximate State Machines," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2006, pp. 315–326.

[11] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2003.

[12] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and Practice of Bloom Filters for Distributed Systems," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[13] T. Chen, D. Guo, Y. He, H. Chen, X. Liu, and X. Luo, "A Bloom Filters Based Dissemination Protocol In Wireless Sensor Networks," *Journal of Ad Hoc Networks*, vol. 11, no. 4, pp. 1359–1371, 2013.

[14] O. Rottenstreich and I. Keslassy, "The Bloom Paradox: When Not To Use A Bloom Filter?" in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

[15] B. Donnet, B. Gueye, and M. A. Kaafar, "Path Similarity Evaluation Using Bloom Filters," *Journal of Computer Networks*, vol. 56, no. 2, pp. 858–869, 2012.

[16] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha, "The blue active queue management algorithms," *IEEE/ACM Trans. Netw.*, vol. 10, no. 4, pp. 513–528, Aug. 2002. [Online]. Available: http://dx.doi.org/10.1109/TNET.2002.801399

[17] Q. G. Zhao, M. Ogihara, H. Wang, and J. J. Xu, "Finding global icebergs over distributed data sets," in *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ser. PODS '06. New York, NY, USA: ACM, 2006, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1142351.1142394

[18] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer, "Single-packet ip traceback," *IEEE/ACM Trans. Netw.*, vol. 10, no. 6, pp. 721–734, Dec. 2002. [Online]. Available: http://dx.doi.org/10.1109/TNET.2002.804827

[19] N. Johnson and S. Kotz, *Urn Models and Their Applications: An Approach to Modern Discrete Probability Theory*. John Wiley and Sons Inc, 1977.

[20] "CAIDA Website," http://caida.org/.

[21] "SNORT Website," http://snort.org/.

[22] B. Harris and R. Hunt, "Tcp/ip security threats and attack methods," *Computer Communications*, vol. 22, no. 10, pp. 885 – 897, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S014036649900064X