This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

kBF: Towards Approximate and Bloom Filter based Key-Value Storage for Cloud Computing Systems

Sisi Xiong, *Student Member, IEEE,* Yanjun Yao, *Student Member, IEEE,* Shuangjiang Li, *Student Member, IEEE,* Qing Cao, *Member, IEEE,* Tian He, *Senior Member, IEEE,* Hairong Qi, *Senior Member, IEEE,* Leon Tolbert, *Fellow, IEEE,* Yilu Liu, *Fellow, IEEE*

Abstract—As one of the most popular cloud services, data storage has attracted great attention in recent research efforts. Key-value (k-v) stores have emerged as a popular option for storing and querying billions of key-value pairs. So far, existing methods have been deterministic. Providing such accuracy, however, comes at the cost of memory and CPU time. In contrast, we present an approximate k-v storage for cloud-based systems that is more compact than existing methods. The tradeoff is that it may, theoretically, return errors. Its design is based on the probabilistic data structure called "bloom filter", where we extend the classical bloom filter to support key-value operations. We call the resulting design as the kBF (key-value bloom filter). We further develop a distributed version of the kBF (d-kBF) for the unique requirements of cloud computing platforms, where multiple servers cooperate to handle a large volume of queries in a load-balancing manner. Finally, we apply the kBF to a practical problem of implementing a state machine to demonstrate how the kBF can be used as a building block for more complicated software infrastructures.

Index Terms—Bloom filter, Key-value storage

1 Introduction

One of the primary challenges in modern cloud computing platforms is to provide highly scalable, efficient, and robust storage services for application needs. Among its various forms, key-value (k-v) stores have emerged as a popular option for storing and querying billions of key-value pairs [1], [2], [3] [4]. Examples of such services include Amazon Dynamo [5], Memcached [6] (used by Facebook and Twitter, etc.), Apache Cassandra [7], Redis [8], and BigTable by Google [9]. Furthermore, many cloud services now provide dedicated key-value stores, such as Amazon S3 in its EC2 framework, to allow developers to quickly take advantage of database services without worrying about the limitations of traditional SQL formats.

One observation of these different k-v storage services by existing cloud computing providers is that they are deterministic, which indicates a query of previous stored key should always return its correct value. Although this is certainly a desired feature, doing so requires storing and processing complete information of the keys and values, which introduces overhead. Therefore, we develop a highly compact, low-overhead, but approximate k-v storage service, by taking inspiration from the bloom filter [10]. However, despite of their usefulness, bloom filters are designed for testing set memberships, not key-value operations. 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 APIs that are common for k-v storage services, such as **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.

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 [11], 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.

We emphasize that there are indeed many scenarios where we do need 100% correctness in storage. This is not in conflict with our goals of developing the kBF and d-kBF, as we also observe that there exist

- *S. Xiong, Y. Yao, S. Li, Q. Cao, H. Qi, L. Tolbert and Y. Liu are with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, 37996. E-mail: {sxiong, yyao9, shuangjiang, cao, hqi, tolbert, liu}@utk.edu*
- *T. He is with the Deparment of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, 55455. E-mail: tianhe@cs.umn.edu*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

occasions where the application may have to be deployed under stringent time constraints and limited memory/storage space. A good example would be their use as a cache system, where using the kBF or d-kBF will allow a much larger size as the cache system and using much less time to handle queries. Another interesting application is where the data stored are inherently insensitive to the existence of errors, as the data themselves contain noise and errors that need to be filtered. One example is that we need to store large amount of data for further processing, such as network traffic traces and noisy scientific data such as instrument readings. In such cases, a second step of processing will be invoked once data are fetched. Therefore, it does not matter if the data may occasionally return non-decodable errors due to that they are inherently noisy and may contain errors. In such cases, the kBF/d-kBF can serve for the raw data storage directly.

The approach we present aims to address these problems by supporting key-value operations with predictable performance. In our previous work [12], we have investigated the feasibility of this approach with a prototype implementation. Specifically, we proposed a method to encode the values into a special type of binary encodings that can fit into the cells of bloom filters. 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 of the kBF. 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. In this paper, we extend the kBF into a comprehensive framework that has the following four new contributions:

- We extend our method in [12], and present a comprehensive set of programming APIs for cloud computing users to use kBFs as if they were objects in their software implementations. These new APIs include `CREATE`, `JOIN`, and `COMPRESS`, which support operations on multiple kBF objects. Such an extension greatly extends the flexibility of the kBF operations.
- Motivated by the unique requirements of cloud computing environments, we develop a distributed design of the kBF, the d-kBF, so that a user may use the familiar client-server model to invoke the kBF service remotely. This contribution makes the kBF to be scalable to massive datasets.
- To address the challenge to achieve predictable performance, we systematically analyze the computational and storage complexity of the kBF, including its capacity and decoding ratio to demonstrate its performance limits.
- We conduct an analysis of resilience to noise of the kBF. We model errors as a type of noise, where we analyze its statistic features of Euclidean distance, where we demonstrate that the results are consistent with the original error analysis.

In summary, the kBF represents a novel type of the bloom filter that supports key-value operations for cloud computing services. To further illustrate its effectiveness, we demonstrate through a specific application example: we use it as a building block for developing an approximate concurrent state machine (ACSM). Using ACSMs, a router can efficiently keep track of many regular expression matchings simultaneously to detect potential behavior anomalies and intrusions.

The remaining of this paper is organized as follows. Section 2 presents the related work. Section 3 describes the problem formulation and the design of the kBF. Section 4 analyzes its performance tradeoffs. Section 5 describes how the distributed version of the kBF is developed. Section 6 evaluates the performance of the kBF through experiments. Section 7 develops an application of the kBF for monitoring TCP flag transitions. Finally, Section 8 concludes this paper.

2 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 of the key-value stores in the cloud computing area.

The bloom filter, originally developed by Burton H. Bloom [10], is a space efficient randomized data structure that answers the question about membership tests. 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 insert 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 [11], 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

In recent years, key-value stores have been increasingly studied in the area of cloud computing platforms. For example, [16] presents a framework that allows secure outsourcing and processing of encrypted data over public key-value stores, and could automatically make use of multiple cloud providers for improving efficiency and performance. [17] presents M-Lock, which is a framework that helps accelerate distributed transactions on key-value stores in cloud computing platforms. Finally, [18] attempts to address the increasing software-I/O gap in key-value stores by using vector interfaces in high-performance networked systems as the basis for key-value storage servers, where they demonstrate that they can provide 1.6 million requests per second with a median latency below one millisecond thanks to the high speed of non-volatile memories. All these previous efforts are different from ours in the sense that they are targeting at accurate key-value stores while our approach is approximate by nature.

3 Design of kBF

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

3.1 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 then use the kBF to store these keys and values, where we treat each kBF similar to an object: it has its internal operations as well as interfaces. The kBF should support the following seven operations for the stored k-v pairs:

- `CREATE` (m, k) //create a new kBF based on configuration parameters
- `INSERT` (k, v, d) //insert a key-value pair (k, v) into the kBF of d
- `UPDATE` (k, v_{new}, d) //update a value for a key k to v_{new} in the kBF of d
- `QUERY` (k, d) //query the value for a key k in the kBF of d
- `DELETE` (k, d) //delete a key and its associated value in the kBF of d
- `JOIN` (d_1, d_2) //combine two kBFs d_1 and d_2, including all their keys and values
- `COMPRESS` (d) //compress a kBF d into half of the original size

We next present the architecture of the kBF, which is shown in the Figure 1. This figure focuses on the insert 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

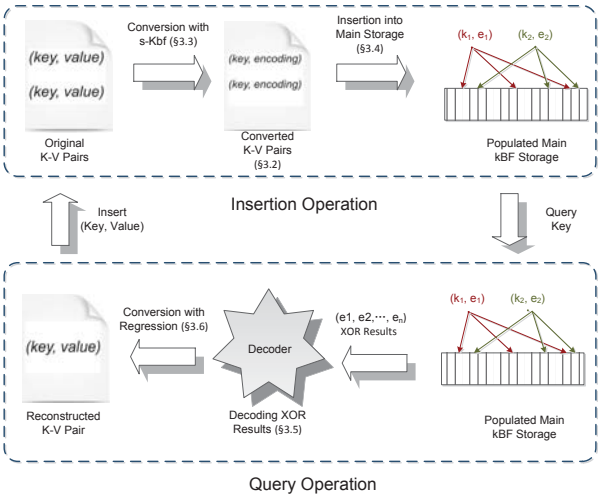

Fig. 1. The kBF algorithm architecture

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. 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.

3.2 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, denoted as $e[v_i]$, are constructed according to the following two rules:

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

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

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

Note that the minimal value of P may not be achieved. Therefore, we develop a greedy algorithm for finding the required encodings, as shown in Algorithm 1. 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

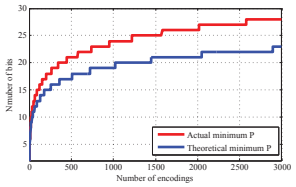

Fig. 2. Relation between the size of encodings and the number of bits

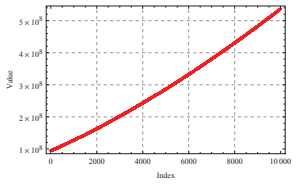

Fig. 3. Relation between the value of and index of encodings

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:     e_0 = 1
3:     insert e_0 into a bloom filter BF
4:     counter ← 2
5:     for j = 1 → n − 1 do
6:         while True do
7:             e_j = counter
8:             calculate the XOR result for e_j
                   and e_i, for i ∈ [0, j − 1]
9:             if there is no collision for e_j then
10:                Insert e_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. Specifically, it will insert all admitted encodings and their pair-wise 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.

3.3 Conversions from Values to Encodings

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

3.4 Operations of the kBF

We now describe the operations of kBFs. 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 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(key, e_value)        ▷ Insert operation
2:     for j = 1 → k do
3:         i ← h_j(key)
4:         B_i.counter ← B_i.counter + 1
5:         B_i.value ← B_i.value XOR e_value
6:     end for
7: end procedure
```

The typical kBFs support seven operations: `CREATE`, `INSERT`, `QUERY`, `UPDATE`, `DELETE`, `JOIN`, and `COMPRESS`. We first describe the `CREATE` operation. Whenever this API is invoked, an empty kBF object is initialized. Specifically, this API takes two parameters, k and m, which specify the dimensions of the underlying bloom filter.

We next describe the `INSERT` operation. When this function is invoked, the 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, where e_{value} represents the encoding of the value in k-v pair.

The third operation, `QUERY`, works as follows. For each of the k cells, it will obtain the superimposed encodings as well as their associated counters. Followed up with an decoding process, which will be discussed in Section 3.5.

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

Next, we can implement the `UPDATE` algorithm by first querying the key to obtain its encoding, then

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

Algorithm 3 kBF Query Algorithm

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

Algorithm 4 kBF Delete Algorithm

```
1: procedure DELETE(key, e_value)                ▷ 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 e_value
7:         else
8:             report error
9:         end if
10:    end for
11: end procedure
```

delete the key with its encoding, and finally insert the key with its new encoding. Algorithm 5 describes its details.

Just like a normal bloom filter, a kBF has its capacity in terms of how many items it is able to at most support. To avoid overflows of kBFs, 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 `COMPRESS` operation. This operation is facilitated by the bit-vector nature of kBFs, and implemented through the `JOIN` operation. Given two k-v sets, their `JOIN` operation works as follows. 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 each cell: $Cell_L = Cell_{L1} \oplus Cell_{L2}$. 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.

Algorithm 5 kBF Update Algorithm

```
1: procedure UPDATE(key, e_value)                ▷ Update operation
2:     e_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 e_old
7:             B_i.value ← B_i.value XOR e_value
8:         end if
9:     end for
10: end procedure
```

3.5 Decoding Superimposed Encodings

We next describe how to obtain original encoding giving the counters and values in k hashed cells, as shown in Algorithm 6. The simplest case is that one or more counters equal 1, which indicates only the queried key has been hashed to those cells and we simply return the encoding directly.

The next part is to obtain the original encoding which is the intersection of multiple superimposed results. We first consider the problem: if there is an XOR result, denote as V_1, how to find the unique set of $\{e_i, e_j\}$, such that $V_1 = e_i \oplus e_j$. By pre-constructing a bloom filter that has all encodings, we can find this unique set in $\mathcal{O}(N)$, by iterating through all encodings, calculating its XOR result with V_1, and checking if the result can be found in the pre-constructed bloom filter. We denote this process as $(e_i, e_j) = decode2(V_1)$. Therefore, our first choice is to utilize two different values from the set $Queue_2$, denoted as V_1, V_2, and conduct the decoding process, i.e., first conducting $(e_i, e_j) = decode2(V_1)$ and $(e_i, e_k) = decode2(V_2)$, and then finding the intersection e_i, which is the encoding with regard to the queried key. We denote this process as $e_i = decode22(V_1, V_2)$ and it also takes $\mathcal{O}(N)$.

However, when the kBF is too crowded, such a pair that shares one encoding may not exist. Our next choice is to use two different values, denoted as $V_1 \in Queue_2$, $V_3 \in Queue_3$. We develop an algorithm for finding their common encoding as shown in Algorithm 7. Note that here, we may encounter two exceptions that prevent us from finding the single common encoding if they occur: the first one is when the V_1 and V_3 have two common encodings, i.e. $e_i = e_m$, $e_j = e_n$; the second one is that we could not guarantee that the XOR result of four different encodings are distinct from each other. In both cases, we return with a response as "non-decodable". We compare the performance difference of only using first choice and using both two choices at the end of the evaluation section.

Here, we discuss the time complexity of the decoding algorithm. First, we consider Algorithm 7. Note that inserting all encodings and their XOR results to the two classic bloom filters takes $\mathcal{O}(N)$ and $\mathcal{O}(N^2)$, respectively. Furthermore, decoding V_1 takes $\mathcal{O}(N)$. Therefore, in total, the algorithm takes $\mathcal{O}(N^2)$, where N is the number of encodings.

For Algorithm 6, we first iterate k hash positions to get the $Queue_2$ and $Queue_3$ in $\mathcal{O}(1)$. The process $decode22(V_1, V_2)$ takes $\mathcal{O}(N)$, and the process $decode23(V_1, V_3)$ is Algorithm 7 itself, which takes $\mathcal{O}(N^2)$. Overall the decoding algorithm takes $\mathcal{O}(N^2)$.

3.6 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

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.

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.

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. We describe this procedure through an example. Suppose we have a table of $10,000$ encodings, and we 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 by empirical analysis. 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. Therefore, in terms of time complexity of regression method, we consider it as constant time, while binary search uses $\mathcal{O}(log(N))$.

Algorithm 6 Decoding

```
1: procedure DECODING(StateQueue)
2:     for r = 1 → k do
3:         if B_r.counter = 1 then
4:             return B_r.value
5:             break for
6:         else if B_r.counter = 2 AND B_r.value ≠ 0 then
7:             add B_r.value to Queue_2
8:         else if B_r.counter = 3 AND B_r.value ≠ 0 then
9:             add B_r.value to Queue_3
10:        end if
11:    end for
12:    if |Queue_2| ≥ 2 then
13:        choose V_1, V_2 ∈ Queue_2
14:        e = decode22(V_1, V_2)
15:    else if |Queue_2| = 1 AND |Queue_3| ≥ 1 then
16:        choose V_1 ∈ Queue_2, V_3 ∈ Queue_3
17:        e = decode23(V_1, V_3)
18:    else
19:        return Not Decodable
20:    end if
21:    return e
22: end procedure
```

Algorithm 7 Decode with V_1 and V_3

```
1: procedure DECODE23(V_1, V_3)                    ▷ V_1 = e_i ⊕ e_j, V_3 = e_m ⊕ e_n ⊕ e_k
2:     Insert all encodings to bloom filter B_1
3:     Insert all XOR results to bloom filter B_2
4:     V_4 = V_1 ⊕ V_3
5:     if V_4 is in B_1 then
6:         There are two common encodings, return Not decodable
7:     else
8:         (e_i, e_j) = Decode2(V_1)
9:         V_5 = e_i ⊕ V_3
10:        V_6 = e_j ⊕ V_3
11:        if V_5 is in B_2 AND V_6 is not in B_2 then
12:            return e_i
13:        else if V_6 is in B_2 AND V_5 is not in B_2 then
14:            return e_j
15:        else
16:            return Not decodable
17:        end if
18:    end if
19: end procedure
```

4 Analysis of kBF

In this section, we analyze the capacity and error rate of the 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 $k_{opt} = \frac{m}{n}\ln(2)$ [19]. 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 1: 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.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

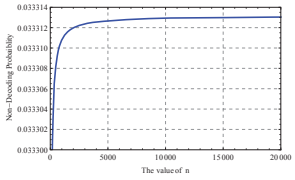

Fig. 4. Relation between n and the non-decodable probability

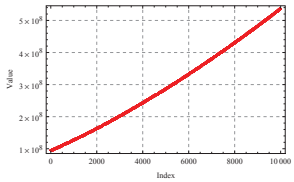

Fig. 5. Relation between ρ and the non-decodable probability

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

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

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 [20]. 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

$$\lim_{n\to\infty}P(c(i)=j)=\frac{(\rho\ln(2))^j}{j!}\times2^{-\rho}.$$

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 overestimate, 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

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

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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

5 Distributed Key-Value Bloom Filter

In this section, we describe how we develop a distributed key-value storage system based on the kBF that is tailored for the distributed nature of cloud computing platforms. Previous efforts in this domain are mostly deterministic. For example, Memcached [6] essentially is a distributed k-v store, and is utilized as a representative cloud computing memory caching system, which has a highly scalable architecture for clusters of servers. To compensate for an increasing I/O gap, many applications aim to improve the performance by adding a layer of caching systems between the disks and the CPUs. In cloud computing platforms, such caching systems are naturally distributed. However, Memcached's performance requires querying and processing all key-value pairs in their original forms. This guarantees that it will never lead to errors with the sacrifice of more memory consumption and time latency. In our work, we develop the support for key-value storage in a distributed manner. Different from the centralized kBF, such storage is no longer limited to a single machine or a storage device. Instead, the data are distributed over multiple servers that will collectively provide storage needs.

In our design of the distributed kBF, we follow a similar approach just like the centralized kBF in that we do not require the returned key-values to be 100% accurate, so that we can achieve a tradeoff between speed and error rates. We call the resulting design as d-kBF. The same as the kBF, the d-kBF could find its applications in distributed storage systems with a tolerance of low probability of errors. As mentioned above, one typical application is the cache system, the same with Memcached, whose primary purpose is to speed up system responses. In the following, we first present its architecture, followed by its implementation details.

5.1 Architecture of d-kBF

The architecture of d-kBF is shown in Figure 6, where the whole d-kBF is represented by multiple *slabs*, which is the same as the centralized kBF, stored on multiple slab servers. Here, each slab keeps track of a collection of (key,value) pairs. There is a frontend master server and multiple backend slab servers. The master server is responsible for allocating new slabs and deleting empty slabs, while at the same time, it keeps track of all slab locations. There are several slabs stored in the memory of one storage server. Note that for load-balancing purposes, slabs can also migrate between servers.

We next describe how we implement distributed operations in the d-kBF, including insert, query, delete and update, with the usage of slabs. We use a two-step hashing approach for this purpose. The first-step hashing locates a slab for the k-v pair based on the hashing value of the key. Once the slab is located, the client will query the master node on the current location of this slab. Then, the client will communicate with the slab server directly, where the second-step operation is the same as the centralized kBF.

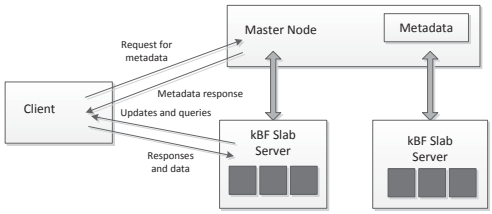

Fig. 6. Distributed Key-Value Bloom Filter Architecture

5.2 Load balance

In cloud-computing environments, we assume that each storage server has same storage capacity. To deal with load balance, we want to achieve the scenario where each server has nearly the same number of k-v pairs stored. At the beginning, the master will assign the same number of slabs to each storage server. Note that as long as the first step hashing function of the d-kBF operation is uniformly distributed, after insertion of first batch of data, the number of k-v pairs in each slab should be almost the same. Therefore, storage imbalances across slabs can be effectively prevented in the early stage of the d-kBF.

However, after more datasets and more operations are performed, some slabs will have more k-v pairs stored than others do. One approach to address this problem is slab migration, where we allow slabs to move between servers. However, finding the optimal solution to the slab assignment based on their load is NP-hard, as this is equivalent to the bin-packing or knapsack problem. Therefore, we develop a greedy algorithm to achieve query balances, as shown in Algorithm 8.

The load balance algorithm works as follows. The master node will perform re-balancing operations periodically with an interval of T. At the end of each interval, master node calculates the aggregate number of k-v pairs for each slab and each server. To measure the imbalance, we use a metric ρ, by dividing the standard deviation over the average value. The larger ρ is, the more imbalanced the server storage are. The goal of balance algorithm is to achieve a lower ρ, by migrating slabs from heavily loaded servers to the less loaded ones.

In each step of migration, we first choose two slabs, one is the most loaded slab in the most heavily loaded server, and the second one is the least loaded slab in the least loaded server. Before making the actual exchange, we calculate the new imbalance metric ρ_{new} after migration. If there is an improvement, i.e. a lower ρ, the master node informs the two slab servers

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

Algorithm 8 Query balancing algorithm

```
1: procedure QUERYBALANCE(q)
2:     counter ← 1
3:     calculate the number of total queries for each slab
4:     calculate the number of total queries for each server
5:     calculate the current load imbalance metric ρ
6:     while True do
7:         find one slab S_max on a most queried server
8:         find one slab S_min on the least queried server
9:         predicate the new imbalance metric ρ_new
10:        if ρ_new > ρ then
11:            break while
12:        else if counter ≥ C_max then
13:            break while
14:        else
15:            exchange the two slabs
16:            ρ ← ρ_new
17:            counter ← counter + 1
18:        end if
19:    end while
20: end procedure
```

to make exchanges, and updates their metadata accordingly. In addition, to avoid too much overhead, we allow a second parameter, C_{max}, that determines the maximum number of exchanges that can be incurred at the end of each period. When the maximum of slab exchange is achieved, the lowest ρ will be returned.

In terms of time complexity of Algorithm 8, we consider that both the number of servers and the number of slabs on each server is constant, therefore calculating the load imbalance metric ρ, as well as finding the most heavily loaded slab and sever takes constant time. Hence, the time complexity depends on the maximum number of exchange, C_{max}. Based on our experiments, $C_{max} = 10$ is an appropriate choice.

6 Experiment Evaluation

In this section, we first present the evaluation of operations of the kBF, including insert, query, delete, update, join and compress. Due to the probabilistic nature of the kBF, we focus on its errors. Specifically, there are three types of errors: false positives, which means that kBF returns an value for keys that haven't been inserted, false negatives, which means that the kBF returns null value due to non-decodable errors, and incorrect outputs, which means that kBF returns incorrect values.

Next, we focus on the time and memory efficiency of the kBF, by conducting comparison experiments with Memcached [6]. Although the design of Memcached and the kBF are quite different, they share several similar characteristics: both of them are in-memory key value store and support distributed deployment, and they are able to support inserting, querying and deleting one k-v pair in constant time. All the similarities make the comparison reasonable.

We then evaluate the two encoding options mentioned in Section 3.5. We also study noise analysis

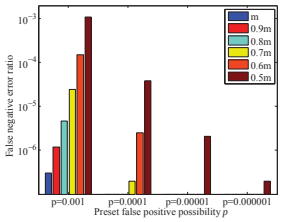

Fig. 7. False negative error ratio

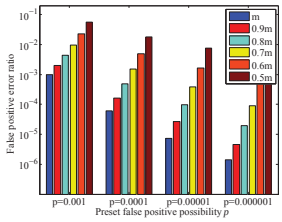

Fig. 8. False positive error ratio

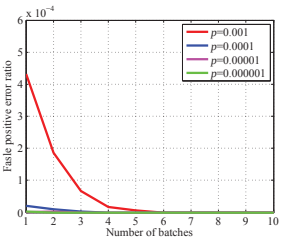

Fig. 9. False positive ratios after each batch deletion, kBF size = m

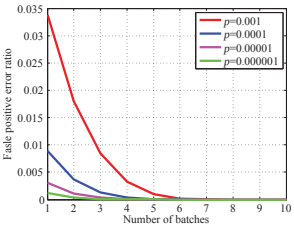

Fig. 10. False positive ratios after each batch deletion, kBF size = $m/2$

of the kBF, in which we demonstrate that the errors caused by the kBF is equivalent to the "intrinsic noise" of the dataset. Finally for the d-kBF, we evaluate the load balancing algorithm, and the experiment results show that by slab migration, we could achieve more balanced storage among servers.

6.1 Workload Generation

We generate the workload for experiments based on the conclusions from a realistic study 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 k-v pairs where the size of keys follow these reported statistical parameters, and 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. We keep the specific keys and values random, so that the evaluation results are the most generic. The number of k-v pairs n is 10 million, and we use a false positive probability p to be between 0.001 to 0.000001 when we construct the kBF. Therefore, the size of the kBF m and the number of hash functions k can be decided [19].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

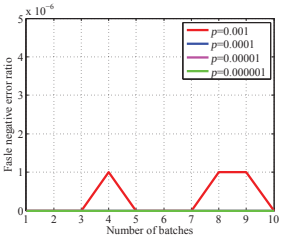

Fig. 11. False negative ratios after each batch update, kBF size = m

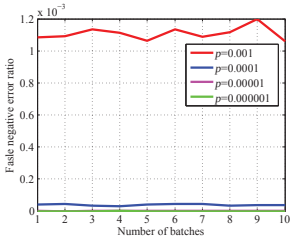

Fig. 12. False negative ratios after each batch update, kBF size = $m/2$

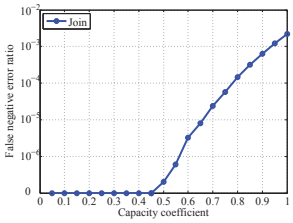

Fig. 13. False negative error ratios of the join operation

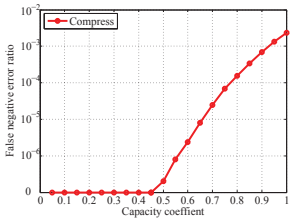

Fig. 14. False negative error ratios of the compress operation

6.2 The kBF Evaluation Results

We start by inserting 10 million keys into a constructed kBF, and then query each key for its value to obtain the error rates. To test the case when the kBF has been overloaded, we also conduct the experiments when gradually decreasing the size to half to analyze the performance difference. We find that in these cases, the incorrect output error is always zero when the kBF has not been saturated, indicating that the results are correct if they are decodable at all. The false negative error rate is plotted in Figure 7. According to this figure, the false negative errors almost do not exist if the p-capacity is not violated. On the other hand, if the size of the filter m is decreased linearly in steps from $0.9m$ to $0.5m$, the false negative errors are increasing exponentially, meaning that the decoding process gives more null results. This is as expected, as in this case, the kBF 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 the 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 8. 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, the false positive rate becomes higher, as we expected.

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 for normal and half size of kBF. Observe that for a smaller p, the performance tends to be much better.

We also evaluate the effects of update operations. Similar to the delete operations, we update keys with new values in batches and we are interested in false negatives, which refer to null 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.

We next evaluate the join and compress operations. Note that the compress operation is a special case of the join operation in the sense that it is implemented as the first half and second half of the kBF conducting the join operation.

We use the metric of capacity coefficient ρ mentioned in Section 4 to conduct the experiments with the following procedures. We use the ρ of two kBFs, d_1 and d_2, as the evaluation parameter, and conduct the join operation of these two kBFs, yielding the combined kBF as d_{join}. Next we query d_{join} with all the k-v pairs that have been inserted to d_1 and d_2. Note that since d_{join} loses some information compared to d_1 and d_2, we expect some queries will result in false negative errors (non-decodable errors), which are shown in Figure 13. Note that the k-v pairs inserted to d_1 and d_2 are different, which makes sense in a way that we don't insert duplicate k-v pairs in applications. According to the experiment results, when the capacity coefficient threshold is below 0.5, the aggregate capacity coefficient of d_{join} is under 1, the error rate is almost zero. On the other hand, when the ρs of d_1 and d_2 are beyond 0.5, the p-capacity of d_{join} is no longer maintained, where the number of false negative error increases exponentially. The experiment results are consistent with Figure 7, where we decrease the size of the kBF to intentionally violate the p-capacity rule, leading to more errors.

We also perform similar experiments for the compress operation, where we set the size of the kBF as an even number. Similar to the join operation, for different capacity coefficients, we combine the first half and second half of the kBF, and plot the errors shown in Figure 14. Due to the same reasoning, we have very similar results compared to join operation.

6.3 Comparison with Memcached

In order to demonstrate the time and memory efficiency of the kBF, we choose Memcached, one of state-of-art k-v stores as reference to conduct the comparison experiments. First, we compared the time efficiency, i.e, the time to insert, query and delete key-value pairs, and the results are shown in Figure 15. According to the results, the time of both the kBF and Memached increases almost linearly as the number of k-v pairs increases, which indicates constant time for each k-v pair. More importantly, the kBF is almost 10 times faster than Memcached.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

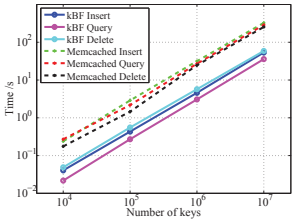

Fig. 15. Time efficiency comparison

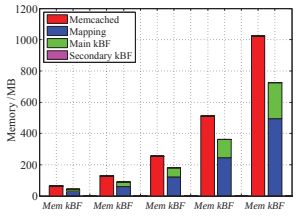

Fig. 16. Memory usage comparison

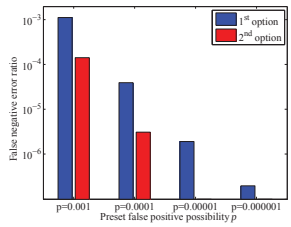

Fig. 17. False negative error ratio of two different decoding options

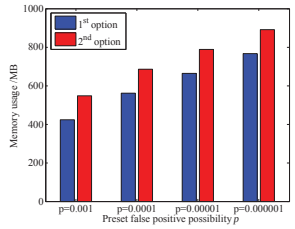

Fig. 18. Memory usage of two different decoding options

In terms of memory usage, we conduct the experiments in the following way: Memcached normally works as a cache system in that it first allocates fixed memory space for k-v stores and will delete the least recently used (LRU) key-value pairs when it is full. However, it also can be configured as a k-v store, and reports errors when the storage is full. We use the latter setting, and keep record on how many pieces of records it has stored. Next, we conduct the kBF experiments to store the same amount of data and then keep record of the memory usage of the kBF, where we show the results in Figure 16. Note that for the kBF, there are three types of memory usage, the main kBF, the secondary kBF and the $(encoding, value)$ mapping. The Memcached size is in the range from 64MB to 1024MB, and compared to Memcached, the overall size of kBF could save 30.33% memory on average.

6.4 Evaluation of two decoding options

As mentioned in design section, when there are overlapping encodings inserted to the same cell of the kBF, the procedure of decoding is needed. We have two options, either only decoding two superimposed encodings, or using the Algorithm 7. Intuitively, we get more non-decodable errors using the first option if the kBF is crowded, but we require more computation and memory resources for the second option. To evaluate their differences, we conduct the query operation with the same 10 million key-value pairs, and set the size of the kBF to be half of its normal size, so that the kBF is overloaded. We then compare the false negative error ratio and the memory used, and plot the results in Figure 17 and Figure 18. In the results, we obtain fewer false negative errors by almost an order of magnitude by using the second option, while we spend 21% more memory usage on average as additional overhead.

6.5 Resilience to Noise

Since the kBF would inevitably return errors, we evaluate how reliable the data is in performing certain APIs. If we regard errors caused by inaccurate data access as "query noise" as compared to the "intrinsic noise" that all data has, then as long as the query noise does not disrupt the behavior of the intrinsic noise, the queried data is considered "reliable".

We have conducted such a study where statistical features are extracted from both the accurate data and the query data through kBF to statistically compare the intrinsic differences as an indicator of data reliability. We used a real dataset from CAIDA [21], which includes an hour long network traffic data with the source and destination IP address serving as flow-id, and the TCP flag as the state. A total number of $n = 246,539$ unique flows were used. We then restructured the n data entries into a matrix of dimension 496×496 in a raster scan manner (so that the data can be viewed as an image) and used a non-overlapping patch size of 8×8 as the basic unit to extract statistical features of the data. The features we used are the popular Hu's invariant moments [22] where a set of seven values are calculated based on the pixel intensities (i.e., the data entry). The feature vector is formulated to be invariant under translation, scale, and rotation. Figure 19 shows the Euclidean distance between the feature vectors derived from any pair of patches belonging to the accurate data and the queried data, respectively, with the false positive probability being 0.01, 0.001, and 0.0001. We observe that as the false positive probability decreases, the Euclidean distance between patches from the two data sources show less and less number of sporadic peaks and eventually no peaks when $p = 0.0001$, showing the similarity between the two data sources in statistical measures.

6.6 Evaluation of d-kBF

For the evaluation of the d-kBF, we focus on the performance of designed load balance algorithm. Suppose each slab has the capacity of 10 million k-v pairs, and we have 5 severs, each of them has stored 10 slabs. Therefore, the overall d-kBF could store up tho 500 million k-v pairs. First, we assign a random number of k-v pairs to each slab, yet don't exceed its capacity, and calculate the initial imbalance ratio ρ. After conducting the load balance algorithm, we compare the resulted ρ to the original one. Figure 20 shows the number of k-v pairs each sever stores before and after the algorithm. As there are multiple steps of slabs migration, Figure 21 shows the variation of ρ during each step, where the ρ is keep decreasing.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

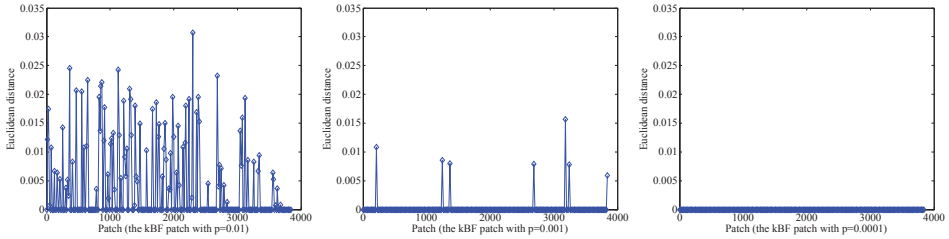

Fig. 19. The Euclidean distance of Hu's invariant moments of patches belonging to accurate data and queried data.

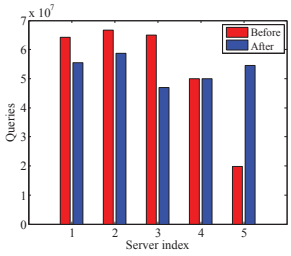

Fig. 20. Server storage

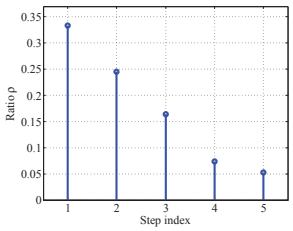

Fig. 21. Variation of ρ

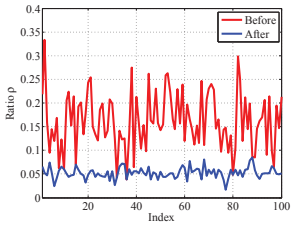

Fig. 22. Performance under different input

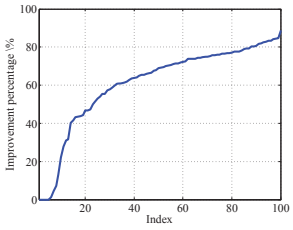

Fig. 23. Percentage improvement of ρ

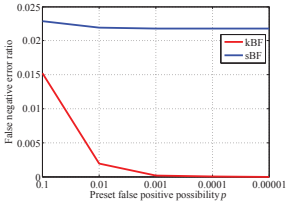

Fig. 24. False negative error ratio

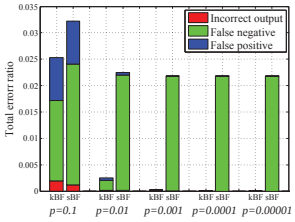

Fig. 25. Total error ratio

Next, we evaluate the performance of algorithm in different initial settings. We generate 100 sets of number of k-v pairs the servers stored and then conduct the algorithm. Figure 22 demonstrates the imbalance ratio ρ before and after the algorithm, and Figure 23 shows the sorted improvement percentage of ρ. The average improvement is 60.86%. Note that during some initial settings, there is no improvement since the original ρ is small enough, indicating the storage among all servers is fairly balanced. Other than that, we could gain large decrease of ρ, especially under those very imbalanced initial settings.

7 Application Case Study: TCP Flow Analysis

In this section, we describe how to use the kBF for a concrete application. We implement an approximate concurrent state machine (ACSM) [23] based on the kBF, and compare it with the original approach in [23], which we refer to as the state-based bloom filter (sBF). Note that we can use state machines for various purposes, ranging from user application behavior modeling, to large-scale and distributed intrusion detection. As a proof-of-concept study, for our evaluation, we use this ACSM to analyze a real dataset from CAIDA [21], 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. Note that such traffic analysis and intrusion detection is crucial for the security and integrity of modern cloud computing service providers, and for defending against large-scale attacks [24], [25] [26] [27] [28].

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 [29] and TCP SYN flooding attacks [30]. Whenever the specified TCP flags indicate potential problems, a warning can be generated.

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 the sBF [23]). 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 the kBF and the sBF. 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 performance gap between the kBF and the sBF is mainly in false negative errors. According to Figure 24, as the preset p value decreases, false negative errors of the kBF decreases dramatically. However, false negative errors of the 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

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 25 shows the total errors of the kBF and the sBF. Again, we observe that the 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.

8 Conclusions

In this paper, we present the design, implementation, analysis, and evaluation of the kBF, an approximate key-value store service for cloud computing platforms, by using the classic bloom filter as a base design. We present the design of the kBF including both centralized and distributed versions, 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 memory for fast speeds, 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 for a wide range of applications, and valuable especially for large-scale cloud service providers.

9 Acknowledgement

We appreciate the comments from Dr. Rajkumar Buyya on an early version of this paper. 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] L. Wei, H. Zhu, Z. Cao, X. Dong, W. Jia, Y. Chen, and A. V. Vasilakos, "Security and privacy for storage and computation in cloud computing," *Information Sciences*, vol. 258, pp. 371–386, 2014.

[4] L. Wei, H. Zhu, Z. Cao, W. Jia, and A. Vasilakos, "Seccloud: Bridging secure storage and computation in cloud," in *Distributed Computing Systems Workshops (ICDCSW), 2010 IEEE 30th International Conference on*, June 2010, pp. 52–61.

[5] 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.

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

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

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

[9] 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.

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

[11] 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

[12] S. Xiong, Y. Yao, Q. Cao, and T. He, "kbf: a bloom filter for key-value storage with an application on approximate state machines," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2014.

[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] E. Pattuk, M. Kantarcioglu, V. Khadilkar, H. Ulusoy, and S. Mehrotra, "Bigsecret: A secure data management framework for key-value stores," in *IEEE CLOUD*, 2013, pp. 147–154.

[17] N. Rapolu, S. Chakradhar, A. Hassan, and A. Grama, "Mlock: Accelerating distributed transactions on key-value stores through dynamic lock localization," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, June 2013, pp. 179–187.

[18] V. Vasudevan, M. Kaminsky, and D. G. Andersen, "Using vector interfaces to deliver millions of iops from a networked key-value storage server," in *SoCC*, 2012, p. 8.

[19] 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.

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

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

[22] M.-K. Hu, "Visual pattern recognition by moment invariants," *Information Theory, IRE Transactions on*, vol. 8, no. 2, pp. 179–187, 1962.

[23] 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.

[24] S. He, M. Ghanem, L. Guo, and Y. Guo, "Cloud resource monitoring for intrusion detection," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 2, Dec 2013, pp. 281–284.

[25] M. Ficco, L. Tasquier, and R. Aversa, "Intrusion detection in cloud computing," in *3PGCIC*, 2013, pp. 276–283.

[26] X. Zhang, F. Zhou, X. Zhu, H. Sun, A. Perrig, A. Vasilakos, and h. Guan, "Dfl: Secure and practical fault localization for datacenter networks," *Networking, IEEE/ACM Transactions on*, vol. 22, no. 4, pp. 1218–1231, Aug 2014.

[27] B. Liu, J. Bi, and A. Vasilakos, "Toward incentivizing anti-spoofing deployment," *Information Forensics and Security, IEEE Transactions on*, vol. 9, no. 3, pp. 436–450, March 2014.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2385063, IEEE Transactions on Cloud Computing

[28] Z. M. Fadlullah, T. Taleb, A. V. Vasilakos, M. Guizani, and N. Kato, "Dtrab: Combating against attacks on encrypted protocols through traffic-feature analysis," *IEEE/ACM Trans. Netw.*, vol. 18, no. 4, pp. 1234–1247, Aug. 2010. [Online]. Available: http://dx.doi.org/10.1109/TNET.2009.2039492
[29] "SNORT Website," http://snort.org/.
[30] 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

Sisi Xiong is currently a Ph.D. candidate majoring in Electrical Engineering in University of Tennessee, Knoxville, and she received the B.E. degree in Automation and the M.E. degrees in Control Science and Engineering from Beihang University, Beijing, China, in 2010 and in 2013. Her research interests include probabilistic data structures, distributed storage systems, probabilistic counting and big data processing.

Yanjun Yao received the B.S. degree in Software Engineering from Zhejiang University, Hangzhou, Zhejiang, China, in 2006, and the M.S. degrees in Security and Mobile Computing from Helsinki University of Technology, Espoo, Finland, and University of Tartu, Tartu, Estonia, in 2008, and the Ph.D. degree in Computer Science from University of Tennessee, Knoxville, TN, USA, in May 2014. Her research interests include energy-efficient network communication protocols, probabilistic data structures for data-aware networking systems, and mobile system implementations.

Shuangjiang Li received the B.S. degree in Measurement and Control in 2009 from the University of Science and Technology Beijing, Beijing, China and the M.S. degree in Computer Engineering in 2011 from the University of Tennessee, Knoxville. He is currently pursuing the Ph.D. degree in the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville. His research interests include signal and image processing, compressed sensing, mobile phone sensing and WSN.

Qing Charles Cao (M'10) is currently an assistant professor in the Department of Electrical Engineering and Computer Science at the University of Tennessee. He received his Ph.D. degree from the University of Illinois in October, 2008, and his Master's degree from the University of Virginia. His research interests include wireless sensor networks, embedded systems, and distributed networks. He is a member of IEEE, ACM, and the IEEE Computer Society. He has published more than 50 papers in various journals and conferences, and has served as technical committee members for more than 30 conferences held by IEEE and ACM.

Tian He is currently an associate professor in the Department of Computer Science and Engineering at the University of Minnesota-Twin City. He received the Ph.D. degree under Professor John A. Stankovic from the University of Virginia, Virginia in 2004. Dr. He is the author and co-author of over 150 papers in premier network journals and conferences with over 13,000 citations (H-Index 44). Dr. He has received a number of research awards in the area of n6tworking, including five best paper awards, NSF CAREER Award, K. C. Wong Award, and McKnight Land-Grant Professorship Award. Dr. He served a few program/general chair positions and on many program committees in international conferences , and also currently serves as an editorial board member for six international journals including ACM Transactions on Sensor Networks. His research includes wireless sensor networks, cyber-physical systems, intelligent transportation systems, real-time embedded systems and distributed systems.

Hairong Qi (S'97-M'00-SM'05) received the B.S. and M.S. degrees in computer science from Northern JiaoTong University, Beijing, China, in 1992 and 1995, respectively, and the Ph.D. degree in computer engineering from North Carolina State University, Raleigh, in 1999. She is now a Professor in the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. Her current research interests are in advanced imaging and robust collaborative processing in resource-constraint distributed environment and information unmixing. She is the recipient of the NSF CAREER Award. She also received the Best Paper Awards at the 18th International Conference on Pattern Recognition and the third ACM/IEEE International Conference on Distributed Smart Cameras. She recently receives the Highest Impact Paper from the IEEE Geoscience and Remote Sensing Society.

Leon M. Tolbert (S'88-M'91-SM'98-F'13) received the B.S., M.S., and Ph.D. degrees in electrical engineering from the Georgia Institute of Technology, Atlanta, in 1989, 1991, and 1999, respectively. He worked at Oak Ridge National Laboratory, Oak Ridge, TN, from 1991 until 1999. He was appointed as an Assistant Professor with the Department of Electrical and Computer Engineering, The University of Tennessee, Knoxville, in 1999. He is currently the Min Kao Professor in the Department of Electrical Engineering and Computer Science, The University of Tennessee. He is the UTK Campus Director for the National Science Foundation/Department of Energy Research Center, CURENT (Center for Ultra-wide-area Resilient Electric Energy Transmission Networks). He is also a Senior Research Engineer with the Power Electronics and Electric Machinery Research Center, Oak Ridge National Laboratory.

Yilu Liu (S'88-M'89-SM'99-F'04) is currently a Governor Chair Professor at the University of Tennessee, Knoxville and Oak Ridge National Laboratory. Prior to joining UTK/ORNL, she was a professor at Virginia Tech. Dr. Liu received her M.S. and Ph.D. degrees from the Ohio State University, Columbus, in 1986 and 1989. She received the B.S. degree from XiAn Jiaotong University. She led the effort to create the North American power grid monitoring network (FNET) at Virginia Tech which is now operated at UTK and ORNL as FNET/GridEye. Her current research interests include power system wide-area monitoring and control, large interconnection level dynamic simulations, electromagnetic transient analysis, and power transformer modeling and diagnosis.