Functional Bloom Filter, Better than Hash Tables

Hayoung Byun, *Hyesook Lim
Ewha Womans University
*hayoung77@ewhain.net, *hlim@ewha.ac.kr*

Abstract

Hash tables have been widely used in many applications, which need to return values corresponding to each input key. However, hash-based data structures have an intrinsic problem of collision, where different keys have the same index of a hash table. As the load factor of the hash table increases, the number of collisions increases. Elements that could not be stored because of the collision cause failures in returning values. Variant structures such as multi-hashing, cuckoo hashing, and d-left hashing have been studied, but none of the structures solve completely the collision problem. In this paper, we claim that a functional Bloom filter can replace a hash table. While the hash table requires to store each input key itself or the signature of each input key in addition to the return value, the functional Bloom filter can store the return value only, since different combinations of Bloom filter indexes can work as the signature of each input key. Performance evaluation results show that the functional Bloom filter is more efficient than hash-based data structures in storing more number of elements into a fixed size memory and hence in producing less failures.

Keywords: key-value structure, hash table, Bloom filter, functional Bloom filter

1. Introduction

A key-value data structure that returns a value corresponding to an input key has been used in many fields, since it is simple and extensible. As a representative key-value data structure, hash tables have been popularly applied to many different applications. Hashing converts an input key string to an index used to point an entry of a hash table. The pointed entry stores both the key itself (or the signature of the key) and a return value. Collisions occurring in hash indexes are an intrinsic problem of hash-based data structures. To reduce the number of hash collisions, multi-hashing, cuckoo hashing, and *d*-left hashing have been studied [1–3].

A functional Bloom filter can satisfy the key-value operation required by various applications [4]. In this paper, we claim that a function Bloom filter is more effective than a hash table in storing a large amount of data to a limited size of memory. Unlike the hash table, the functional Bloom filter needs to store return values only, since different combinations of Bloom filter indexes can work as the signature of each input key. Experiments have shown that as the load factor increases, the functional Bloom filter works better than other hash tables.

2. Hash-Based Key-Value Structure

The key-value data structure returns the value corresponding to a key, and the structure generally stores a key and the corresponding value pair. In this paper, we propose a more efficient structure for processing and retrieving the vast amounts of information in a restricted memory by constructing and comparing key-value structures on a fixed size of memory. Figure 1 shows the hash-based key-value data structures compared in this paper. We assume that the hash tables store signatures (instead of keys) and values corresponding to each key.

A multi-hashing table [1] uses multiple hash functions instead of a single hash function for each key to reduce the number of collisions. Figure 1(a) shows the multi-hashing table with two hash functions and two entries per bucket. Even if the multi-hashing is used, overflows can occur if all the entries of the pointed buckets are occupied. To resolve the problem, we assume that a linked list pointer is used and the linked list pointer is inserted to the second bucket. However, if there is a loop in the linked list, the new element cannot be saved, and unsaved elements cause search failures and false negatives.

When a new element is inserted, cuckoo hashing [2] solves hash collisions by pushing the old element (pre-occupied in the pointed entry) to another location. The cuckoo hash table shown in Figure 1(b) uses two hash tables and two hash functions, and each element has a single index per table. If a collision occurs in the second table, this process is repeated until the pushed element finds an empty

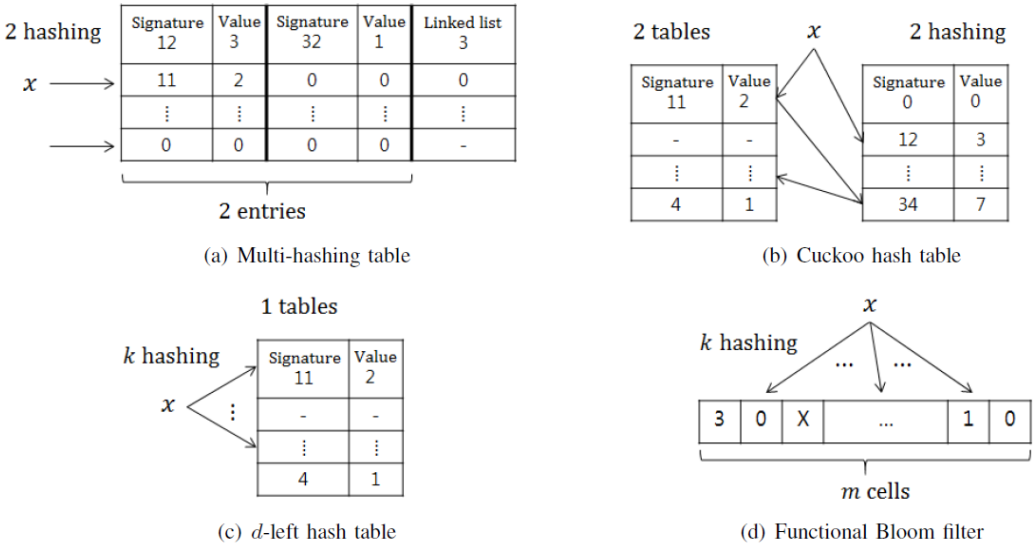

Figure 1 : Hash-based key-value data structures

bucket. If there are two loops in inserting an element, the element cannot be saved, and then unsaved elements cause search failures and false negatives.

Figure 1(c) shows a *d*-left hash table [3], where each bucket includes a single entry and the number of hash functions *d* is equal to *k*. Since we assume the *d*-left hash table with a single entry per bucket, an element is stored in the leftmost empty bucket. If collisions occur in all *k* entries, the element cannot be saved. Unsaved elements cause search failures and false negatives.

A functional Bloom filter (F-BF) [4] is a variant of the standard Bloom filter to return a value corresponding to a key as well as membership information. Figure 1(d) shows a functional Bloom filter with *m* cells and *k* hash indexes. In the functional Bloom filter, the signature of each key is not stored and only the return value is stored in each cell, since various combinations of hash indexes can serve as the signatures. When programming an element into the F-BF, if the cell pointed by a BF index is already programmed by a value other than the return value of the element under programming, the cell is denoted as a collided cell. Querying for the F-BF is only performed for non-collided cells. The querying of an input returns three types of results; a *negative*, a *positive*, and an *indeterminable*. The *negative* means that non-collided cells pointed by the indexes of the input include at least a zero value or different values. The *positive* means that non-collided cells include the same value. The *indeterminable* means the F-BF fails to return a value because every cell is collided. The functional Bloom filter can also return a *false positive*. The false positive and the *indeterminable* results are search failures that occur in the functional Bloom filter.

3. Performance Evaluation

We evaluate and compare the performance of four hash-based data structures using a specific example of name lookup in NDN, which requires the key-value data structure. Simulations are carried out using random URLs provided by Web Information Company ALEXA [5]. Set *S* (2^{17}) is stored in the hash tables or the functional Bloom filter. The number of URLs in input set *U* used in searching is three times the number of URLs in set *S*. We assume 14 types of return values, and hence the size of the return value is 4 bits.

We first construct a multi-hashing table with the load factors (α) of 0.5, 0.6, 0.8, and 1. Then we construct a cuckoo hash table, a *d*-left hash table, and a functional Bloom filter, using the same amount of memory as the multi-hashing table.

Table 1 shows the average and the worst-case number of memory accesses in inserting and searching an element when the load factor is 1. For the functional Bloom filter, the average and the worst-case number of accesses are the same, since the number of memory accesses is always equal to *k*.

Table 1: The number of memory accesses

		Multi	Cuckoo	*d*-left	F-BF
insertion	**Avg.**	12.1	64.1	2.1	8
	worst	359	832	8	8
search	**Avg.**	44.3	1.84	6.0	8
	worst	359	2	8	8

Figure 2 shows search failure rates according to load factors. The search failure rate is the most important performance metric of key-value data structures. Search failures in the hash tables result in false negatives. In case of the functional Bloom filter, the number of search failures is the summation of false positives and *indeterminables*. As the load factor increases, the F-BF shows the best performance in the search failure rates. The search failure rate of the functional Bloom filter is less than 0.5%, even when the load factor is close to 1.

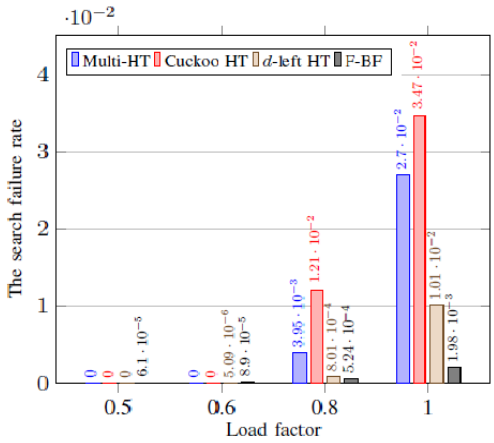

Figure 2 : The search failure rates

4. Conclusion

In this paper, we describe four different hash-based data structures and compare their performances according to load factors. It is shown that the functional Bloom filter is an excellent alternative to hash tables, when storing information more compactly in a limited amount of on-chip memories.

Acknowledgement

This research was supported by the National Research Foundation of Korea (NRF), NRF-2014R 1A2A1A11051762, NRF-2015R1A2A1A15054081, and NRF-2017R1A2B4011254. This research was also supported by the Ministry of Science, ICT and Future Planning (MSIP), Korea, under the Information Technology Research Center (ITRC) support program (IITP-2017-2012-0-00559) supervised by the Institute for Information & communications Technology Promotion (IITP).

References

[1] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," *in Proc. IEEE INFOCOM*, Vol.3, pp. 1454-1463, 2001.

[2] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, Vol. 51, No. 2, pp. 122–144, 2004.

[3] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," *in Proc. ESA*, pp. 684–695, Zurich, Switzerland, September 2006.

[4] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Beyond Bloom filters: From approximate membership checks to approximate state machines," *in Proc. ACM SIGCOMM*, pp. 315–326, September 2006.

[5] Alexa the Web Information Company, http://www.alexa.com/