Cardinality Computing: A New Step Towards Fully Representing Multi-sets by Bloom Filters

Jiakui Zhao, Dongqing Yang, Lijun Chen, Jun Gao, and Tengjiao Wang

School of EECS, Peking University, Beijing 100871, China

Abstract. Bloom Filters are space and time efficient randomized data structures for representing (multi-)sets with certain allowable errors, and are widely used in many applications. Previous works on Bloom Filters considered how to support insertions, deletions, membership queries, and multiplicity queries over (multi-)sets. In this paper, we introduce two novel algorithms for computing cardinalities of multi-sets represented by Bloom Filters, which extend the functionality of the Bloom Filter and thus make it usable in a variety of new applications. The Bloom structure presented in the previous work is used without any modification, and our algorithms have no influence to previous functionality. For Bloom Filters support cardinality computing in addition to insertions, deletions, membership queries, and multiplicity queries simultaneously, our work is a new step towards fully representing multi-sets by Bloom Filters. Performance analysis and experimental results show the difference of the two algorithms and show that our algorithms perform well in most cases.

1 Introduction

Insertions, deletions, membership queries, multiplicity queries, and cardinality computing are five important operations and queries over multi-sets. In order to represent multi-sets as fully as possible by sketches, the sketches must support the five operations and queries simultaneously. Bloom Filters are space and time efficient randomized data structures for representing (multi-)sets with certain allowable errors, and previous works [1], [2], [3] considered how to support insertions, deletions, membership queries, and multiplicity queries. For fully representing multi-sets, Bloom Filters must support cardinality computing in addition to insertions, deletions, membership queries, and multiplicity queries simultaneously. In this paper, we consider cardinality computing and introduce two algorithms (FCount and BFCDist) to compute the cardinalities of multi-sets represented by Bloom Filters. The FCount algorithm uses membership queries to count the distinct elements over multi-sets, and the BFCDist algorithm uses pure probabilistic method to compute the cardinalities. Our work extends the functionality of the Bloom Filter and thus make it usable in a variety of new applications. From functionality extension’s point of view, our work can be seen as a new step towards fully representing multi-sets by Bloom Filters.

* Supported by State Key Laboratory of Networking and Switching Technology, NSFC Grant 60473051 and 60503037, and NSFBC Grant 4062018.
1.1 Previous Bloom Filters

Standard Bloom Filters. B. H. Bloom [1] introduced the Bloom Filter in 1970, which is called the Standard Bloom Filter in this paper. It uses $k$ hash functions $h_1, h_2, \ldots, h_k$ to hash elements into a bit vector $V$ of size $m$, and supports insertions and membership queries over (multi-)sets. Initially, all bits are set to 0; for each inserted element $e$, the bits at positions $h_1(e), h_2(e), \ldots, h_k(e)$ are set to 1. For an element $e$, its membership can be checked by examining the bits at positions $h_1(e), h_2(e), \ldots, h_k(e)$. An element $e$ is reported to be contained in the set iff all the $k$ bits had been set to 1. This method allows a small probability of producing false positive error (a positive result may be returned for an element $e$ which actually is not contained in the set), but no false negative error.

Counting Bloom Filters. L. Fan et al. [2] introduced an extension of the Standard Bloom Filter for supporting deletions in 1998, which is called the Counting Bloom Filter in this paper. It replaces the bit vector $V$ by a counter vector $C$. Initially, all counters are set to 0; an insertion (deletion) of an element $e$ will increase (decrease) the counters at positions $h_1(e), h_2(e), \ldots, h_k(e)$ by 1. Membership checking is the same as that in the Standard Bloom Filter, except that an element $e$ is reported to be contained in the set iff the $k$ counters are all not zero-valued. To maintain the compactness of the Counting Bloom Filter, the counters are limited to 4 bits, which is shown to be statistically enough to encode the number of the elements hashed to the same position. However, Counting Bloom Filters are not adequate when dealing with multi-sets in which identical elements may appear hundreds and thousands of times.

Spectral Bloom Filters. Saar and Matias [3] introduced the Spectral Bloom Filter for supporting multiplicity queries over multi-sets in 2003. The algorithms for evaluating multiplicity queries are Minimum Selection, Minimal Increase, and Recurring Minimum. In Minimum Selection, things are the same as that in the Counting Bloom Filter; for an element $e$, the minimum of the values of the counters at positions $h_1(e), h_2(e), \ldots, h_k(e)$ is selected as the multiplicity. In Minimal Increase, when performing an insertion of an element $e$, only increasing the counters at positions $h_1(e), h_2(e), \ldots, h_k(e)$ whose values are the minimum of the $k$ counters. Minimal Increase has lower error rate and lower error size; unfortunately, it does not support deletions. Recurring Minimum is similar to Minimum Selection except that it uses a two-level Spectral Bloom Filter to reduce the error rate. After performing an insertion (deletion) of an element in the Primary Spectral Bloom Filter, if it has single minimum among the $k$ counters, performing the insertion (deletion) in the Secondary Spectral Bloom Filter. When evaluating a multiplicity query for an element, if it has recurring minimum in the Primary Spectral Bloom Filter, return the minimum; otherwise, performing the evaluation in the Secondary Spectral Bloom Filter. Since through observations, the elements which have single minimum in the Primary Spectral Bloom Filter have high error rate in evaluating multiplicity queries. In [3], a data structure called string-array index is used to maintain the counter vector, which has good space, access, and query performance, and supports multi-sets in which identical elements may appear hundreds and thousands of times.