MINIMAL PERFECT HASHING AND BLOOM FILTERS MADE PRACTICAL

ABSTRACT

The Bloom filter is a space-efficient data structure for storing an approximation S' to a set S such that $S \subseteq S'$ and any element not in S belongs to S' with probability at most ϵ. This problem happens in many areas and a Bloom filter has widespread use. In this paper we present a practical implementation of a theoretical result that provides the same functionality of a Bloom filter for static sets and uses a near-optimal space data structure based on recent results on perfect hashing by Botelho et al. (2007). The implementation outperforms the standard Bloom filter in space usage and construction time for false positive rates lower than or equal to $\epsilon \leq 2^{-6}$ (≈ 1.56 out of one hundred). It also presents a better lookup time for $\epsilon \leq 2^{-4}$ (≈ 6.25 out of one hundred) and presents a constant lookup time that does not depend on ϵ. However, we note that there are other variants of Bloom filters that are as compact as the one we consider here for static key sets. In this work our objective is to show an implementation of a theoretical result that can be used in practice and the original Bloom filter is a standard baseline for comparison.

KEYWORDS

Bloom filter, minimal perfect hashing, data structures

1 INTRODUCTION

In this paper we study a data structure that supports membership queries to a set in an efficient way in terms of both space and lookup time, specially when space is at a premium. In many applications that need to store a set S of $n = |S|$ elements it is acceptable to include elements that appear to be in S but are not, which are referred to as *false positives* from now on. For example, if a Web crawler needs to check whether a Web server contains a desired page, it is acceptable that the page would not be there with a small probability, since the only cost is to make an unsuccessful request to the server. The data structure studied here allows to describe the content of a Web server using near-optimal space.

In any application where false positives do not cause significant problems, only an approximation of S can be stored instead of explicitly storing the set, thus providing a more compact representation that can fit in memory. If every element not in S is a false positive with probability at most ϵ, then the number of bits needed to store the approximation is roughly $n \log_2(1/\epsilon)$, whereas to avoid false positives one would need to store S in at least $n \log_2 n$ bits (Pagh et al., 2005).

Consider the set $S' \supseteq S$ the elements that are stored, including false positives. The set S' is chosen such that any element not in S belongs to S' with probability at most ϵ. A Bloom filter (Bloom, 1970) is an extremely compact and elegant data structure for representing a suitable set S'. Bloom filters yield a data structure that supports membership queries to a set S. Efficiency is achieved at the cost of a small false positive rate, but elements are always recognized as being in the set when they really are (there are no false negatives). It was invented by Burton Bloom in 1970 with the purpose of spell checking. Bloom filters are now widely used in practice in networks (Broder and Mitzenmacher, 2003), database management systems (Gremillion, 1982), distributed systems (Little et al., 2002), proxy cache services (Papapetrou et al., 2005), problems with strong requirements of lookup time and space usage (Jain et al., 2005; Chang et al., 2006).

More recently, several variants have been proposed in the literature. In (Mitzenmacher, 2001; Ficara et al., 2008), compression techniques were developed so that the compressed Bloom filters can be transmitted through a network using space very close to the optimal required. In (Hao et al., 2008), it is shown how to minimize the memory requirements in cases where the number of elements in the set is not known in advance, but the distribution of the number of elements is known.

A Bloom filter works by storing the set S as a bit array B_S where bits are turned on by a set of truly random hash functions. It begins as an array of all 0's. Each element x_i in the set S is hashed k times, with each hash yielding a bit location which is set to 1. To check if an element y is in the set, hash it k times and check whether the corresponding bits are set to 1.

The work in (Pagh et al., 2005) proposes a theoretical near-optimal space Bloom filter data structure using perfect hashing to deal with static key sets. This data structure was not practical for two reasons. First, only theoretical perfect hashing near-optimal space algorithms were known at the time. This was solved by new developments on perfect hash functions that require approximately 2.6 bits per key to be stored and can be evaluated in $O(1)$ time (Botelho et al., 2007). Second, it was not possible to represent a fully random hash function without wasting $O(n \log n)$ bits. This was solved by the split-and-share technique proposed in (Dietzfelbinger and Weidling, 2005; Dietzfelbinger, 2007), making possible the use of $o(n)$ bits to represent a fully random hash function.

The Bloom filter we implement is particularly important for a key set that remains fixed for a long period of time (i.e., might be considered a *static key set*) and allows the satellite data associated with each key to be updated as much as required. Also, it is especially important when the application requires a data structure as compact as possible. This is quite common for data warehouse applications (Seltzer, 2005) and distributed database applications (Broder and Mitzenmacher, 2003).

In this paper we propose a practical implementation of the theoretical result on Bloom filters by (Pagh et al., 2005) using the result on perfect hashing by (Botelho et al., 2007). The randomized data structure is built in linear time, is stored in near-optimal space and answers successful and unsuccessful queries in constant time. The implementation outperforms the standard Bloom filter in space usage and construction time for false positive rates lower than or equal to $\epsilon \leq 2^{-6}$ (≈ 1.56 out of one hundred). It also presents a better lookup time for $\epsilon \leq 2^{-4}$ (≈ 6.25 out of one hundred) and presents a constant lookup time that does not depend on the adopted false positive rate ϵ. However, it is important to mention that there are more efficient implementations of Bloom filters than the standard implementation we consider here. For instance, in (Putze et al., 2009), several replacements were proposed with the same functionality of Bloom filters considering space-efficiency, cache-efficiency and hash-efficiency. In this work our objective is to show an implementation of the theoretical result proposed in (Pagh et al., 2005) that can be used in practice and the original Bloom filter is a standard baseline for comparison.

The remainder of this paper is organized as follows. Section 2 presents the background required for this paper. Section 3 presents the minimal perfect hash Bloom filter data structure. Section 4 compares standard Bloom filters with the minimal perfect hash Bloom filter data structure. Finally, Section 5 concludes our work.

2 BACKGROUND

2.1 Bloom Filter

A Bloom filter (Bloom, 1970) is a data structure that represents an approximation of a key set $S = \{s_1, s_2, \ldots, s_n\}$ of size n, which is a subset of a key universe U of size u. To insert each key $s_j \in S$, k hash functions $h_i : U \to \{0, 1, \ldots, m-1\}$ are used to turn on k bits in a bit array B_S of size m. For that we set $B_S[h_i(s_j)]$ to 1 for $1 \leq i \leq k$ and let $f_j = \{h_1(s_j), h_2(s_j), \ldots h_k(s_j)\}$ for $1 \leq j \leq n$ denote the approximation of $s_j \in S$, referred to as a *signature* of s_j from now on. The value $m = kn \log_2 e$ minimizes the space requirement of this data structure when $k = \lceil \log_2 \left(\frac{1}{\epsilon}\right) \rceil$ (Bloom, 1970; Broder and Mitzenmacher, 2003). For instance, for $\epsilon = 2^{-6}$ we have a space usage of $\log_2 e \times \log_2 2^6$, which is approximately 8,64 bits per key. An m-bit Bloom filter used to represent S must answer true when we perform a membership query for every key $s \in S$ because it does not allow false negatives. However, it might also answer true for additional $\epsilon \times (u - n)$ elements not in S, considering a false positive rate of at most ϵ. This means that if B_S answers true for a membership query of a given element $x \in U$, then $x \in S$ with probability $1 - \epsilon$, otherwise (i.e., the answer is false) $x \notin S$ for sure.

Figure 1 illustrates the standard Bloom filter data structure. To search for a key s_j we compute its signature f_j and check if $B_S[i] = 1$ for each $i \in f_j$. Initially, all bits are set to 0 in the bit array B_S. Since k depends on ϵ, we note that the lookup time depends on ϵ. The smaller is ϵ the larger is the lookup time on average. The k hash functions are assumed to be fully random hash functions (that is, they have independent function values uniformly distributed). A fully random random hash function can be obtained by using an additional of $o(n)$ bits of space and the "split-and-share" technique presented in (Dietzfelbinger and Weidling, 2005; Dietzfelbinger, 2007). The standard Bloom filter is a sub-optimal data structure that is within a factor of $\log_2 e \approx 1.44$ of the space lower bound for an optimal Bloom Filter. It is also important to remark that it will require an additional of $o(kn)$ bits to represent k fully random hash functions.

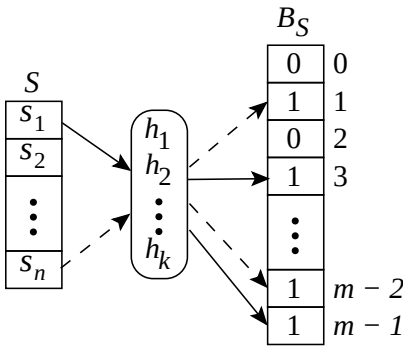


Fig. 1. Standard Bloom Filter (SBF).

Space Lower Bound for Bloom Filters Considering a universe U of size u, we need to associate an m-bit Bloom filter with each of the $\binom{u}{n}$ possible sets. Let $S \subseteq U$ be a fixed subset of size n and let B_S be the associated m-bit Bloom filter. Therefore, each B_S answers true for $n + \epsilon \times (u - n)$ elements and can be used to represent $\binom{n+\epsilon\times(u-n)}{n}$ sets. As there are 2^m different B_S, then we must have $2^m \binom{n+\epsilon\times(u-n)}{n} \geq \binom{u}{n}$. So, the information theoretic lower bound in bits to store any Bloom filter (Broder and Mitzenmacher, 2003) is:

$$m \geq \log_2 \frac{\binom{u}{n}}{\binom{n+\epsilon\times(u-n)}{n}} \geq n \log_2 \left(\frac{1}{\epsilon}\right).$$

2.2 Perfect Hashing

The study of perfect hash functions started in the early 1980s, when it was proved that the theoretic information lower bound to store a minimal perfect hash function is approximately 1.44 bits per key (Mehlhorn, 1984). Although the proof indicates that it would be possible to build an algorithm capable of generating optimal functions, no one was able to obtain a practical algorithm that could be used in real applications until the work by (Botelho et al., 2007). This work provided a practical construction of minimal perfect hash functions that require approximately 2.6 bits per key to be stored and can be evaluated in $O(1)$ time. The result from (Botelho et al., 2007) is used in (Botelho and Ziviani, 2007) to obtain a simple and very efficient external perfect hashing scheme for very large key sets. In (Botelho et al., 2011), it is shown that these new minimal perfect hash functions provide the best tradeoff between lookup time and space usage when compared to other hashing schemes used to index static key sets in internal memory.

Given a static key set S of size n, which is taken from a key universe U of size u, we shall say that a hash function $h : U \to \{0, 1, \ldots, m-1\}$ is a *perfect hash function* (PHF) for S if h is an injection on S, that is, there are no *collisions* among the keys in S: if x and y are in S and $x \neq y$, then $h(x) \neq h(y)$. The function h is used to index a hash table. Since no collisions occur, each key can be retrieved from the table with a single probe. A *minimal perfect hash function* (MPHF) is a PHF with the hash table size equal to the number of keys in S, the smallest possible range.

Space Lower Bound for Perfect Hashing There is an easy way to obtain quite good space lower bounds starting from the simple formulas in (Mehlhorn, 1984, Theorem III.2.3.6 (a)). There it was noted that the

length in bits to describe a perfect hash function for S must be at least

$$\log_2 \left(\frac{\binom{u}{n}}{\binom{\frac{u}{m}}{n} \times \binom{m}{n}} \right). \tag{1}$$

As our focus is on the cases where m is typically lower than $3n$, we applied Stirling's approximation $x! \approx x^x e^{-x} \sqrt{2\pi x}$ to derive a tighter approximation for the information theoretic lower bound to describe a PHF as:

$$\left(m - n + \frac{1}{2}\right) \log_2 \left(1 - \frac{n}{m}\right) - \left(u - n + \frac{1}{2}\right) \log_2 \left(1 - \frac{n}{u}\right), \tag{2}$$

and an MPHF as:

$$\left(n - u - \frac{1}{2}\right) \log_2 \left(1 - \frac{n}{u}\right) - \frac{1}{2} \log_2(2\pi n) \cdot \tag{3}$$

Considering $u \gg n$, for $m = 1.23n$ the lower bound for PHFs is approximately $0.89n$ bits and for $m = n$ the lower bound for MPHFs is approximately $1.44n$ bits.

3 MINIMAL PERFECT HASH BLOOM FILTER

By using the algorithm proposed in (Botelho et al., 2007) we can build an MPHF h for S so that the signature f_j of each key $s_j \in S$ can be stored in the entry $h(s_j)$ of an array H_S with n entries, i.e., $H_S[h(s_j)] = f_j$. But now, f_j is computed by a hash function $h' : U \to \{0, 1, \ldots, 2^{\lceil \log_2(1/\epsilon) \rceil} - 1\}$ chosen uniformly at random from a family of universal hash functions (Carter and Wegman, 1979). A family of hash functions $\mathcal{H}$ is defined as universal if for any pair of distinct elements $s_1, s_2 \in U$ and h' chosen uniformly at random from $\mathcal{H}$ then $\Pr(h'(s_1) = h'(s_2)) \leq \frac{1}{2^{\lceil \log_2(1/\epsilon) \rceil}} \leq \epsilon$. Thus, for any key set $S \subseteq U$ of n elements and any key $x \in U - S$, we obtain a false positive rate of at most ϵ, as follows:

$$\Pr(h'(x) \in H_S) \leq \sum_{j=1}^{n} \Pr(h'(x) = h'(s_j)) \leq \frac{n}{n 2^{\lceil \log_2(1/\epsilon) \rceil}} \leq \epsilon.$$

Figure 2 illustrates the minimal perfect hash Bloom filter data structure for n keys, which is indexed by the MPHF h and stores in each entry of H_S a 3-bit signature for each key and some additional bits for satellite data. We let b denote the maximum number of bits required to store the satellite data associated with each key. In this case, we use a universal hash function $h' : U \to \{0, 1, \ldots, 2^3 - 1\}$ to compute each signature f_j, $1 \leq j \leq n$, and the false positive rate is then equal to $\epsilon = 2^{-3}$.

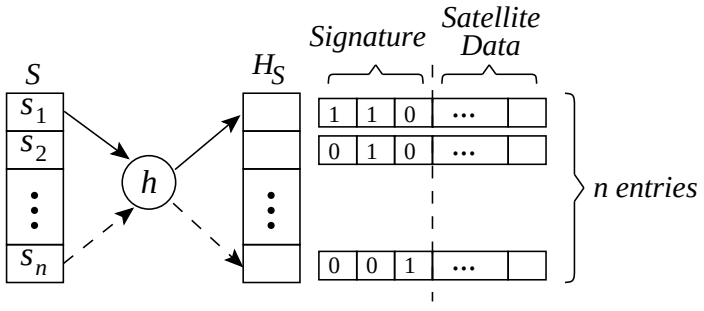


Fig. 2. Minimal perfect hash bloom filter (MPHBF) for a key set S of n keys. It uses a universal hash function $h' : U \to \{0, 1, \ldots, 2^3 - 1\}$ to compute a 3-bit signature for each key. The false positive rate is then equal to $\epsilon = 2^{-3}$.

Therefore, to obtain a practical implementation of the theoretical data structure proposed in (Pagh et al., 2005) we use the result from (Botelho et al., 2007) to store an MPHF that requires approximately $2.6n$ bits, a signature f_j for each key $s_j \in S$ of $\lceil \log_2(1/\epsilon) \rceil$ bits and, when required, the satellite data associated with each key. Thus, the randomized data structure can be stored in $n\left(\log_2\left(\frac{1}{\epsilon}\right) + c\right) + o(n)$ bits, where $c = 2.6$ bits when we are not storing satellite data and $c = 2.6 + b$ otherwise.

This is away from the information theoretic lower bound by an additive constant, which gives an near-optimal space randomized data structure while maintaining constant lookup time. It is also important to emphasize that the lookup time does not depend on the false positive rate as it happens for the standard Bloom filter presented in (Bloom, 1970).

4 EXPERIMENTAL RESULTS

In this section we evaluate the performance of the standard Bloom filter (SBF) and the minimal perfect hash Bloom filter (MPHBF). In the comparison of the two schemes we set different values for the false positive rate ϵ and considered the space usage, the construction time and the lookup time as metrics, which are defined as follows:

1. Space usage: represents the average number of bits per key used for each data structure. For the SBF data structure it corresponds to the space needed to store the bit array B_S plus the k fully random hash functions. For the MPHBF data structure it corresponds to the space required to store the MPHF plus n signatures of $\lceil \log_2(1/\epsilon) \rceil$ bits, one for each key.
2. Construction time: represents the average amount of time required to build the data structures. For the SBF data structure it corresponds to the amount of time required to insert n keys in the bit array B_S. For the MPHBF data structure it corresponds to the amount of time required to insert n keys in the hash table H_S plus the amount of time required to generate the MPHF used by the data structure. It is measured in microseconds per key (μs/key).
3. Lookup time: represents the average time required to check whether a key is present or not in the SBF or MPHBF data structures. It is measured in microseconds per key (μs/key). We divide the lookup time into *successful lookup time* for a successful search, and *unsuccessful lookup time* for an unsuccessful search.

The experiments were carried out on a computer running Linux kernel version 2.6, with a 64-bit Xeon Quad Core processor running at 2.0 gigahertz, 8 gigabytes of main memory, 12 megabytes of L2 cache (3 megabytes for each core), and a SATA II disk of 750 gigabytes. In order to estimate the number of trials for each experiment we used a statistical method for determining a suitable sample size (Jain, 1991, Chapter 13). The results are averages on 10 trials and were statistically validated with a confidence level of 99%.

The SBF and MPHBF algorithms were implemented using the C++ language. We used the hash function proposed in (Jenkins, 1997) for both data structures because it works well for input key sets usually found in practice. We used the code for building the MPHF required for the MPHBF algorithm from the CMPH library available at `http://cmph.sf.net`, which is an open source library that contains state-of-the-art algorithms to generate minimal perfect hash functions. The codes used for the experiments are available at `http://anonymous_for_submission`.

4.1 Key Sets

In our experiments we used two key sets: (i) a key set of $5,424,923$ unique query terms extracted from the AllTheWeb[1] query log, referred to as AllTheWeb key set; (ii) a key set of $115,967,966$ unique URLs

[1] AllTheWeb (`www.alltheweb.com`) is a trademark of Fast Search & Transfer company, which was acquired by Overture Inc. in February 2003. In March 2004 Overture itself was taken over by Yahoo!.

extracted from the webbase-2001 dataset, referred to as WebBaseURL key set, which is available at `http://law.dsi.unimi.it`. Table 4.1 shows the main characteristics of each key set, namely the shortest key length, the largest key length and the average key length in bytes.

Table 1. Characteristics of the key sets used for the experiments

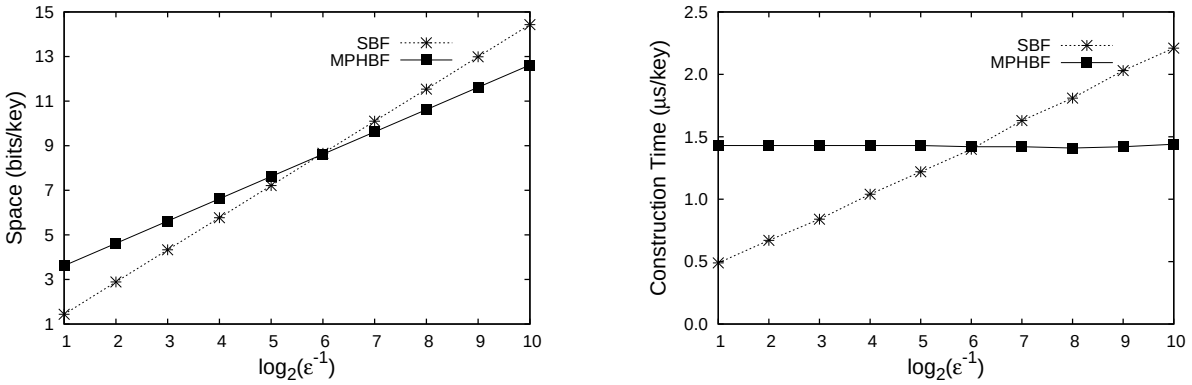
Key Set	n	Shortest Key	Largest Key	Average Key Length
AllTheWeb	$5,424,923$	2	31	17.46
WebBaseURL	$115,967,966$	10	10,212	60.17

Although we have done experiments with the AllTheWeb key set, we just present the results for the WebBaseURL key set because the same conclusions are achieved with both key sets. It is important to remark that both data structures have not presented any dependence on the key set nature, once the AllTheWeb and WebBaseURL key sets are completely different.

The values used for the false positive rate were: $\epsilon = \frac{1}{2^1}, \frac{1}{2^2}, \cdots, \frac{1}{2^{10}}$. For each value of ϵ we split the WebBaseURL key set and use $n = 113,139$ URLs to populate the data structures and to carry out successful searches. The other $L = n \times 2^{\lceil \log_2(\frac{1}{\epsilon}) \rceil}$ URLs are used to carry out unsuccessful searches.

4.2 Space Usage Analysis

Figure 3(a) shows that the space usage of both SBF and MPHBF data structures depends on the false positive rate ϵ. This result matches with the theoretic results presented in Section 2.1 and 3, respectively. It is important to remark that the MPHBF data structure outperforms the SBF data structure in space usage when the false positive rate gets lower than 2^{-6} (1.56%) approximately.



(a) Space usage in bits/key.

(b) Construction time.

Fig. 3. False positive rate versus space usage and construction time for both the SBF and MPHBF data structures for each value of ϵ and $n = 113,139$ URLs.

An important difference between the SBF and MPHBF data structures is the way space usage increases. For a decreasing ϵ, both the size of the bit array B_S and the number k of fully random hash functions increase in the SBF data structure, whereas only the space required to store the signatures increases in the MPHBF data structure. The number of bits required to store the MPHF h remains immutable. This characteristic can significantly impacts construction and lookup time once keeping B_S or the MPHF h in cache improves performance considerably.

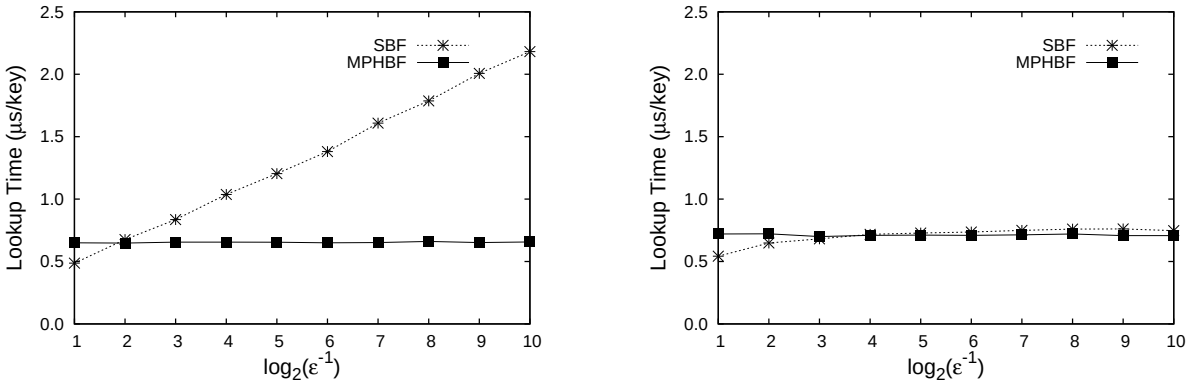
4.3 Construction Time Analysis

As well as it happens for the space usage metric, the construction time also depends on the false positive rate in both the SBF and MPHBF data structures. Figure 3(b) shows how the construction time scales as the false positive rate decreases in both data structures. It is interesting to note that the MPHBF data structure outperforms the SBF data structure for false positive rates lower than or equal to $\approx 2^{-6}$ (1.56%), even having to pay an extra price to pre-compute an MPHF to index the hash table H_S. We remark that the construction time metric is not as important as both the space usage and lookup time metrics for applications that involve static key sets.

4.4 Lookup Time Analysis

In this section we evaluate the lookup time performance for both successful and unsuccessful searches. For successful searches we used the same set of $n = 113,139$ URLs used to build the data structures. For unsuccessful searches we used $L = n \times 2^{\lceil \log_2(\frac{1}{\epsilon}) \rceil}$ URLs not in the set used to build the data structures.

Differently from the space usage and construction time metrics where performance depends on the false positive rate for both data structures, Figure 4 shows that the lookup time depends on ϵ only for the SBF data structure. For the MPHBF data structure the lookup time is constant and does not depend on the false positive rate. Furthermore, the lookup time is quite different for successful and unsuccessful searches in the SBF data structure. This can be explained by the fact that successful searches demand exactly $\lceil \log_2 \left(\frac{1}{\epsilon}\right) \rceil$ bits checking (i.e., all bits in the bit array B_S must be turned on), whereas unsuccessful searches demand the same amount of bit checking in the worst case. That is, when we find one bit turned off we quit the lookup.



(a) Successful searches.

(b) Unsuccessful searches.

Fig. 4. False positive rate versus lookup time for the SBF and MPHBF data structures populated with $n = 113,139$ URLs.

It is important to remark that the MPHBF data structure outperforms the SBF data structure in lookup time for successful searches when the false positive rate is lower than 2^{-2} (25%) approximately. The same thing happens for unsuccessful searches and a false positive rate lower than 2^{-4} (6.25%) approximately. The successful and unsuccessful searches correspond to an upper bound and to a lower bound of the execution time, respectively. In practice we will find something in between.

5 CONCLUSIONS

In this paper we have presented a thorough study of two data structures that are suitable for indexing internal memory in an efficient way in terms of both space and lookup time when we have a key set that

is fixed for a long period of time (i.e., a static key set), each key might be associated with a satellite data, and we can tolerate a small false positive rate. This is widely used in distributed databases, distributed data warehousing and networking applications (see (Broder and Mitzenmacher, 2003) for other examples).

In our study we have shown how to implement in practice a theoretical data structure presented in (Pagh et al., 2005) (MPHBF) that provides the same functionality as a standard Bloom filter but outperforms it in practice for small false positive rates. We have shown that the MPHBF data structure presents a constant lookup time that does not depend on the false positive rate, as it happens for a standard Bloom filter. We have also shown that the MPHBF data structure is more compact and can be built faster than a standard Bloom filter for false positive rates lower than or equal to 2^{-6} ($\approx 1.56\%$). It also provides a faster lookup time for successful and unsuccessful searches when the false positive rate is lower than 2^{-2} (25%) and 2^{-4} (6.25%), respectively. Future work will concentrate on reducing space usage without increasing lookup time.

REFERENCES

Bloom, B., 1970: Space/time trade-offs in hash coding with allowable errors. *Com. of the ACM*, **13**(7), 422–426.

Botelho, F., A. Lacerda, G. Menezes, and N. Ziviani, 2011: Minimal perfect hashing: A competitive method for indexing internal memory. *Information Sciences*, **181**(13), 2608–2625.

Botelho, F., R. Pagh, and N. Ziviani, 2007: Simple and space-efficient minimal perfect hash functions. In *WADS*, 139–150.

Botelho, F. and N. Ziviani, 2007: External perfect hashing for very large key sets. In *CIKM*, 653–662.

Broder, A. and M. Mitzenmacher, 2003: Network applications of bloom filters: A survey. *Internet Mathmatics*, **1**(4), 485–509.

Carter, J. and M. Wegman, 1979: Universal classes of hash functions. *Journal of Computer and System Sciences*, **18**(2), 143–154.

Chang, F., J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, 2006: Bigtable: a distributed storage system for structured data. In *OSDI*, 205–218.

Dietzfelbinger, M., 2007: Design strategies for minimal perfect hash functions. In *SAGA*, 2–17.

Dietzfelbinger, M. and C. Weidling, 2005: Balanced allocation and dictionaries with tightly packed constant size bins. In *ICALP*, 166–178.

Ficara, D., S. Giordano, G. Procissi, and F. Vitucci, 2008: Multilayer compressed counting bloom filters. In *INFOCOM*, 311–315.

Gremillion, L., 1982: Designing a bloom filter for differential file access. *Com. of the ACM*, **25**(9), 600–604.

Hao, F., M. Kodialam, and T. Lakshman, 2008: Incremental bloom filters. In *INFOCOM*, 1067–1075.

Jain, N., M. Dahlin, and R. Tewari, 2005: Using bloom filters to refine web search results. In *WebDB*, 25–30.

Jain, R., 1991: *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley, 685.

Jenkins, B., 1997: Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, **22**(9).

Little, M., S. Shrivastava, and N. Speirs, 2002: Using bloom filters to speed-up name lookup in distributed systems. *The Computer Journal*, **45**(6), 645–652.

Mehlhorn, K., 1984: *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag.

Mitzenmacher, M., 2001: Compressed bloom filters. In *PODC*, 144–150.

Pagh, A., R. Pagh, and S. Rao, 2005: An optimal bloom filter replacement. In *SODA*, 823–829.

Papapetrou, E., E. Pitoura, and K. Lillis, 2005: Speeding-up cache lookups in wireless ad-hoc routing using bloom filters. In *PIMRC*, 1419–1423.

Putze, F., P. Sanders, and J. Singler, 2009: Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, **14**.

Seltzer, M., 2005: Beyond relational databases. *ACM Queue*, **3**(3), 50–58.