A Multi-Layer Bloom Filter for Duplicated URL Detection

Cen Zhiwang, Xu Jungang, Sun Jian
School of Information Science and Engineering
Graduate University of Chinese Academy of Sciences
Beijing, China
{cenzhiwang, xujungang, jiansun6000} @gmail.com

Abstract—**It is of great significance to improve the speed of data collecting and updating in a web crawler because there are a large number of web pages in Internet. A duplicated URL detection approach based on multi-layer bloom filter algorithm is proposed in this paper, which divides an entire URL into some layers and stores them in multi-layer bloom filter. The experimental result shows that the false positive of multi-layer bloom filter algorithm is significantly lower than that of classical bloom filter algorithm, while the efficiency of the former is almost the same as the later.**

Keywords-bloom filter; duplicated URL detection; web crawler; false positive

I. INTRODUCTION

With the rapid development of Internet, more and more websites appear at an alarming rate. According to the "Internet development report of China" issued by China Internet Network Information Center (CNNIC), the number of Chinese websites, this is, the number of domain name registered in China, is 840,000 in December 2006 [1] , and it had rapidly increased to 3.23 million in December 2009 [2]. Furthermore, according to the data from Netcraft (a famous Internet services company), the number of websites all around the world reached 233,848,493 in December 2009 [3]. It is an enormous challenge for a web crawler to deal with such large number of websites and web pages. Therefore, it is of great significance to improve the speed of data collecting and updating in a web crawler.

A web crawler, which can download the web page with its URL, usually starts downloading from the home page of the web site, and then analyzes the source code of the page, extracts all links in the page, gets web pages in the next layer by the new links, and repeats the work until the URL reaches the specified layer or no new link is found. Considering the huge number of web pages and the complicated structure between links, there will inevitably exist a large number of duplicated links. If all web pages are collected indiscriminately, it will wastes large amounts of network bandwidth and resources inevitably. Therefore, during the process of web page collecting, in order to improve the speed of collecting and avoid downloading a page multiple times, the web crawler needs to register pages that have been collected, and judges whether the next page has been collected or not. In the process of web page collecting, a page will be checked whether its URL address is already in the URL collection, if it is, this URL will be skipped, and otherwise, it will be added to the waiting queue of collecting.

Duplicated URL detection is an operation that is performed frequently when a web crawler is working. It is very important for a web crawler that the duplicated URL detection algorithm used in it has low space and time complexity. Because duplicated URL detection must be performed on every collected page, it is a time-consuming process for a web crawler that needs to collect a large quantity of pages. To solve this problem, this paper presents a duplicated URL detection approach for web crawler, which is based on multi-layer bloom filter algorithm. According to the layered characteristic of URL, the approach divides an entire URL into some layers and stores them in multi-layer bloom filter. The experimental result shows that false positive of multi-layer bloom filter algorithm is significantly lower than that of classical bloom filter algorithm, while the efficiency of the former is almost the same as the later.

The remainder of this paper is organized as follows. Section II describes the related work. Section III introduces the bloom filter algorithm and false positive probability. Section IV explains multi-layer bloom filter based duplicated URL detection approach. Based on the URL data set from sogou.com, section V presents the experimental results of multi-layer bloom filter, and analyzes the results compared with the basic bloom filter. Finally, section VI concludes and proposes the future work.

II. RELATED WORK

The simplest approach for duplicated URL detection is to store visited URLs first, and then traverse all the saved URL records to check up whether a new URL has been visited or not. However, it is a hard work for a web crawler because the web crawler needs to deal with a large quantity of web pages. For example, if every URL contains an average of 40 characters, it will take about 4GB space to save 100 millions URLs, and it is very time-consuming to check up whether a new URL has been visited among these URLs. Moreover, we can imagine the consumed time for that the web crawler has to traverse all these 100 millions URLs for each new URL. Therefore, many scholars and engineers all around the world devote themselves to research the mechanism of duplicated URL detection in order to greatly improve its efficiency. The main research on this is listed as follows.

Igloo, a distributed web crawler system developed by Ye Yun-ming and Yu Shui from Shanghai Jiao Tong University, uses a Trie tree to store URLs [4]. A Trie tree, also known as word search tree, is a tree structure used to save a lot of strings. The advantage of this method is that it can save a lot of storage space through using public prefix of strings. Igloo system stores two types of data: one is the string of URL; the other is the metadata of URL. The string of URL is stored in the internal nodes of the path from the root node to leaf nodes, and its metadata is saved in leaf nodes. Bai He and Tang Di-bin from Graduate University of Chinese Academy of Sciences also presented an improved Trie tree to store URLs, which can efficiently supports URL search, insertion and duplicated URL detection [5].

Allan Heydon and Marc Najorka from Compaq Systems Research Center use a fix-sized checksum of a URL for duplicated URL detection in their web crawler named Mercator [6]. This approach is to store the URL checksums only, which is constructed through cutting out the high-order bits hostname of URLs; the URLs from the same domain name are stored in close positions. In order to retrieve checksums of URLs faster, page replacement algorithm named Least Recently Used (LRU) is used for caching, and an in-memory cache of 2^{18} entries are used to store checksum. They claim that this approach has a good hit rate, and an average of 0.16 disk seeks per operation was observed. Hemant Balakrishnan from University of Central Florida presented the same method for duplicated URL detection in [7].

Vladislav Shkapenyuk and Torsten Suel from Polytechnic University use a Red-Black tree to store new URLs in the design of one web crawler. They initially keep the URLs in main memory in a Red-Black tree. When the tree grows beyond a certain size, write the sorted list of URLs into the disk, and then the main memory is used to store new URLs only, and periodically merge memory-resident data with disk-resident data, meanwhile, perform URL insertion and search operation [8]. Polytechnic Crawler not only stores the checksums of URLs, but also stores compressed full URLs.

The web crawler of Internet Archive uses classic bloom filter for duplicated URL detection [9]. In the beginning, this approach sets all bits of an in-memory large array to 0. When inserting a new URL, ten different hash functions are used to compute, and each corresponding bit in the array is set to 1. When checking whether a URL is in the URL-seen set, the ten hash function values are computed again, and if all the ten bits in the array are 1, then the URL is in the set. However, false positive may occur in this method. The probability of false positive is very small at the beginning, but increases as more and more URLs are inserted. Nevertheless, the classic bloom filter is a space-efficient data structure. The approach proposed in this paper is based on classic bloom filter, but multi-layer bloom filter decreases the probability of false positive a lot.

III. Classic Bloom Filter

Bloom filter, which was presented by Burton Howard Bloom in 1970, is a space-efficient data structure used to represent a set [10]. It uses a bit-array to represent a set and can easily tell whether an element is in the set. The bloom filter is more space and time efficient than general algorithm, while it has a very small false positive probability. False positive, a concept from Probability Theory, is that an element which is not in the set is assumed to be an actual member of the set. Because of the time and space efficiency, bloom filter has a high practical value. Since the algorithm was proposed, bloom filter algorithm is widely applied into various areas of computer systems, which is used to represent a huge set of data to improve query efficiency. Initially, bloom filter was applied to the database application, spell checkers and file operations [11] [12] [13]. With the development of the network and P2P (Peer to Peer) technologies, bloom filter is used into more areas, including collaborative P2P node interaction [14], resource routing [15], network measurement management [16], network intrusion detection [17], duplicated URL detection and etc.

Bloom filter algorithm is different from the traditional tree query algorithm and hash query algorithm, the required space of which is no matter with the size of an element, just depends the size of bit-array which is corresponding to the elements. Thus, bloom filter is suitable for space restriction situation with allowable errors. With the widely application of bloom filter, some varieties of bloom filter appears, such as counting bloom filter [18], compressed bloom filter [19], spectral bloom filter [20] and split bloom filter [21].

A. Classic Bloom Filter

Classic bloom filter consists of a large bit-array and several hash functions, which uses very little space to represent a large set, and can easily tell whether an element is in the set or not. Suppose a set $A = \{x_1, x_2, \cdots, x_n\}$ contains *n* elements. Bloom filter consists of an array of bits, initially all bits are set to 0. The bloom filter uses *k* independent random hash functions with range $\{0, \cdots, m-1\}$ to hash elements into an array of size *m*. Therefore, the information of the set is stored in the array of size *m*, which is named *V*.

To add an element *x*, feed it to each of the *k* hash functions to get *k* array positions $h_1(x), h_2(x), \cdots, h_k(x)$. Set the bits at all these positions to 1, that is $V[h_1(x)] = V[h_2(x)] = \cdots = V[h_k(x)] = 1$. A location can be set to 1 multiple times, but only the first change is contributing.

To query an element *x* (check whether it is in the set), feed it to each of the *k* hash functions to get *k* array positions $h_1(x), h_2(x), \cdots, h_k(x)$, if any bit at these positions are 0, the element *x* is not in the set; If all bits are 1, then the element *x* is in the set.

Hence, bloom filter may generate false positive, where it suggests that an element is in even though it is not. Element *x* that is not in the set is assumed in the set because *a*, *b*, *c* and other elements cause that all bits of element *x*'s *k* hash positions are 1.That is a false positive.

Classic bloom filter uses *k* hash functions to hash elements into a bit-array of size *m* with allowable false positive. False positive occurs when querying whether an element is in the set, so in practice we should estimate probability of false positive and design suitable classic bloom filter in order to reduce false positive.

B. The Probability of Bloom Filter's False Positive

Suppose that each hash function selects each array position with the same probability, and if the size of bit array is *m*, the probability of a certain bit that is not set to 1 by a certain hash function during the insertion of an element is $1-\frac{1}{m}$. The probability that the bit is not set to 1 by any hash function is $(1-\frac{1}{m})^k$. After *n* elements are inserted, the probability that a certain bit is still 0 is $(1-\frac{1}{m})^{kn}$. Therefore, the probability that the bit is set to 1 is $1-(1-\frac{1}{m})^{kn}$.

Now check whether an element *x* is in the set. Assume that $p_1, p_2, \cdots, p_k$ are *k* array positions produced by hash functions, and the probability that the bit in $p_i (1 \le i \le k)$ position is set to 1 is $1-(1-\frac{1}{m})^{kn}$. Thus, when all $p_i (1 \le i \le k)$ positions are 1, bloom filter will consider that the element is in the set, the probability of occurrence of this situation is

$$f(k,m,n) = (1-(1-\frac{1}{m})^{kn})^k \approx (1-e^{-kn/m})^k . \quad (1)$$

Obviously, the probability of false positive will decrease when *m* (the number of bits in the array) increases, and will increases when *n* (the number of inserted elements) increases. And when $k = (\ln 2)\left(\frac{m}{n}\right)$, the minimum value of *k*, the minimum probability of false positive is

$$f(k_{\min}) = (0.6185)^{\frac{m}{n}} . \quad (2)$$

The major advantage of bloom filter is space efficient and easily used to query. The time complexity of insertion operations is $O(k)$, and that of query is also $O(k)$. Because bloom filter needs a bit-array of size *m*, space complexity is $O(m)$.

IV. Multi-Layer Bloom Filter for Duplicated URL Detection

A. The Layered Characteristics of URL

One URL usually takes the form of $http://a_1/a_2/\cdots/a_L$ or $https://a_1/a_2/\cdots/a_L$. It uses "/" as the separator, by which the URL is divided into different layers (segment). This is very similar with the file path in operating system except the separator.

We leave the URL protocol identifier (*http* or *https*) out, because the web pages specified by $http://a_1/a_2/\cdots/a_L$ or $https://a_1/a_2/\cdots/a_L$ are the same one, only used protocols are different. So, we only consider the string behind *http://* or *https://* as processing object, that is $a_1/a_2/\cdots/a_L$ .Thus, duplicated URL detection is converted to to the detection of duplicated string like $a_1/a_2/\cdots/a_L$.

Suppose $S = a_1/a_2/\cdots/a_L$ .We regard *S* as a path from the root node to one leaf node in a tree, and $a_i (1 \le i \le L)$ is one internal node of the path, and it is considered as a segment of the URL. It is shown in Fig. 1.

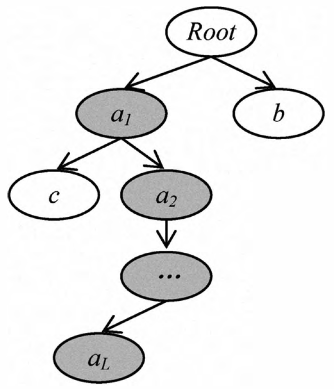


Figure 1. The tree structure of one URL

B. The Principle of Multi-Layer Bloom Filter

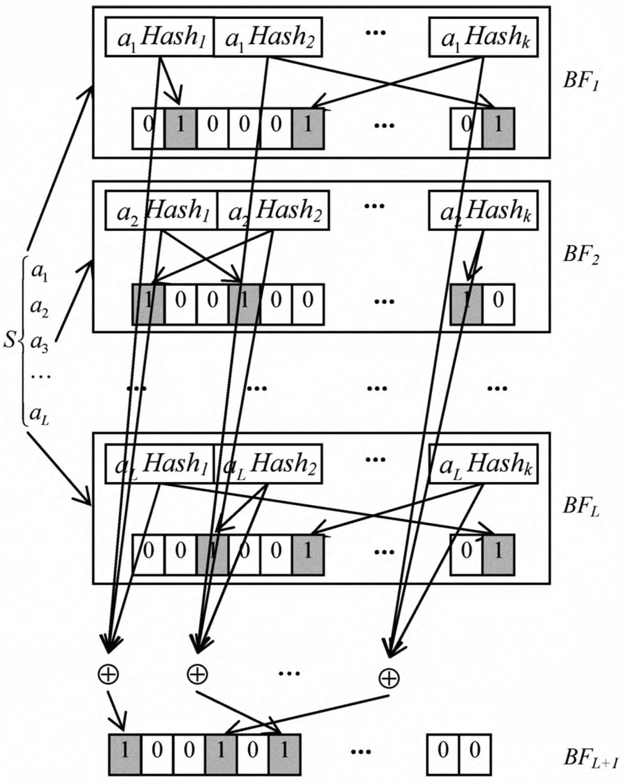
According to the layered characteristics of URL demonstrated in Fig. 1, if a certain $a_i (1 \le i \le L)$ in $S = a_1/a_2/\cdots/a_L$ is not in the set, so *S* is not in the set either. Multi-Layer bloom filter is based on this principle.

Because a certain $a_i (1 \le i \le L)$ in $S = a_1/a_2/\cdots/a_L$ represents one different layer of the tree, we can not simply use a single bloom filter to execute duplicated URL detection, but need to use multi-layer bloom filter to do it, and different layer bloom filter stores the corresponding layer of URL.

We assume that $BF^{k,m}$ is a classic bloom filter consisting of *k* hash functions and an *m*-bits array. And multi-layer bloom filter (MLBF) is composed of some identical bloom filters, and each classic bloom filter is used to record data in each layer. We assume that $MLBF^{L,k,m}$ is a multi-layer bloom filter which is composed of *k* hash functions and an *m*-bits array in *L* layer, that is $MLBF^{L,k,m} = \{BF_1^{k,m}, BF_2^{k,m}, \cdots, BF_L^{k,m}\}$.

When inserting a new URL, insert $a_i (1 \le i \le L)$ into i-th layer bloom filter $BF_i^{k,m} (1 \le i \le L)$ of $MLBF^{L,k,m}$. When checking whether *S* is in the set, check whether each $a_i (1 \le i \le L)$ in *S* is in $BF_i^{k,m} (1 \le i \le L)$ or not. If a certain $a_i (1 \le i \le L)$ is not in $BF_i^{k,m} (1 \le i \le L)$, we think that *S* is not in the set, otherwise, it is in the set. It is shown in Fig. 2. Similarly, multi-layer bloom filter also inevitably generates false positives.

MLBF can improve efficiency, but it can produce new false positives. For example, $S = a_1/a_2/\cdots/a_L$ is inserted into $MLBF^{L,k,m}$, and a_2 is stored in $BF_2^{k,m}$. Suppose that *b* is the second segment of another URL, storing in the second layer of $MLBF^{L,k,m}$. When querying $S' = a_1/b/\cdots/a_L$, MLBF will answer that S' is in the set, which is a false positive. The reason of this problem is that MLBF uses a single classic bloom filter to represent each single segment of URL, and there is no other structure to store the entire URL. In order to solve this problem, we can add an extra layer $BF_{L+1}^{k,m}$ into MLBF, which is used to store the integral information of level 1 to level L, but this layer does not contains k hash function, just contains an m-bits array, as shown in Fig. 2. When a new URL is inserted, the way to calculate the bit-position in $BF_{L+1}^{k,m}$ is to perform an XOR operation among corresponding bit-positions of *1*-layer segment to L-Layer segment.



a_1Hash_1 a_1Hash_2 ··· a_1Hash_k
0 1 0 0 0 1 ··· 0 1
BF_1
a_2Hash_1 a_2Hash_2 ··· a_2Hash_k
1 0 0 1 0 0 ··· 1 0
BF_2
S a_1 a_2 a_3 ··· a_L
a_LHash_1 a_LHash_2 ··· a_LHash_k
0 0 1 0 0 1 ··· 0 1
BF_L
$\oplus$ $\oplus$ ··· $\oplus$
1 0 0 1 0 1 ··· 0 0
BF_{L+1}

Figure 2. The principle of MLBF

Obviously, if $BF_1^{k,m}$ to $BF_L^{k,m}$ all produce a false positive, MLBF will produce false positive too. Suppose that the false positive probability of a classic bloom filter is $f(k,m,n)$. From Section 3.2 we can get that,

$$f(k,m,n) = (1-(1-\frac{1}{m})^{kn})^k \approx (1-e^{-kn/m})^k . (3)$$

Thus, the false positive probability of MLBF is

$$f_{MLBF}(k,m,n,L) = \prod_{i=1}^{L} f(k,m,n) = (f(k,m,n))^L . (4)$$

It can be seen that the false positive probability of MLBF is lower than that of the classic bloom filter.

C. *Duplicated URL Detection based on Multi-Layer Bloom Filter*

The URL insertion algorithm based on MLBF is shown in Fig. 3. When inserting an URL, input the string of URL $S = a_1/a_2/\cdots/a_L$ firstly. Secondly, calculate k hash address of each segment and set the bit in $BF_i^{k,m}$ to 1. Finally, calculate the bit-position in $BF_{L+1}^{k,m}$ by performing XOR operation among the bit-positions in Level *1* to Level L and set the bit to 1.

```
Algorithm1 Insert Item to MLBF
    Input: $S = a_1/a_2/\cdots/a_L$
    Output: void
    for i= 1 to L
      for j=1 to k
        Addr=Hash$_j$ (a$_i$)
        BF$_i$[Addr] = 1
        LAddr[j] = LAddr[j] $\oplus$ Addr
      end for
    end for
    for j=1 to k
      BF$_{L+1}$[LAddr[j]] = 1
    end for
```

Figure 3. Insertion algorithm based on MLBF

```
Algorithm2 Query Item in MLBF
  Input: $S = a_1/a_2/\cdots/a_L$
  Output: if $S$ in MLBF, return true, else return false
  //first, check BF$_1$ to BF$_L$
  for i=1 to L
    for j=1 to k
      Addr = Hash$_j$[a$_i$]
      if BF$_i$[Addr] != 1 then
        return false
        LAddr[j] = LAddr[j] $\oplus$ Addr
    end for
  end for
  //second, check BF$_{L+1}$
  for j=1 to k
    if BF$_{L+1}$[LAddr[j]] != 1 then
      return false
  end for
  return true
```

Figure 4. Query algorithm based on MLBF

The URL query algorithm based on MLBF is shown in Fig. 4. When querying the URL string $S = a_1 / a_2 / \cdots / a_L$, check whether all segments of URL are in the set from $BF_1^{k,m}$ to $BF_L^{k,m}$. If they are, calculate the bit-position in $BF_{L+1}^{k,m}$ by performing XOR operation among the bit-positions in Level *1* to Level *L* in $BF_{L+1}^{k,m}$, and then check whether all these bits are 1. If they are, *S* is considered in the set, otherwise it is not considered in the set.

V. Experimental Analysis and Evaluation

In this section, we did some experiments to evaluate the performance of multi-layer bloom filter (MLBF) algorithm compared with classic bloom filter (BF).

We adopted the URL data set offered by the research center of Sogou.com [22] in the experiments. The URL data set contains 3,537,380 URLs, and each URL has a unique ID. During the experiments, we insert the same URLs into classic bloom filter and MLBF at the same time, and then link each URL with its unique ID to produce a new URL. After all insertion, we check whether each of the new URLs is in classic bloom filter and MLBF, and count the number of URLs in set that produce false positive.

We implemented the evaluation application in C++ language. All of the experiments were performed on a PC with an Intel Pentium Dual-core E2160 1.80 GHz processor and 2 GB memory, running Windows XP professional operating system. The number of URLs which were inserted into classic bloom filter and MLBF are 1, 1.5, 2, 2.5 and 3 million separately. Because the ID of each URL is unique, the new URL generated by linking the URL with its ID is not the set. We use the new URLs as input for query, if the answer of each query is that the new URL is in the set, this is a false positive. The result of experiment is shown in Fig. 5. We set false positive rate to 0.1. The maximum number of elements n is 3,500,000 and the number of hash functions k is 3. According to (2), the size of bit-array m can be work out at 16,829,152. The layer number of the MLBF L is set as 4.

In Fig. 5, the false positive times of classic bloom filter and MLBF increases with the increase of the number of URLs inserted into the set. However, the false positive times of MLBF is in a low number, while the false positive times of classic bloom filter is in a high number. From this we can see that, MLBF can effectively reduce the occurrence of false positive.

As MLBF will perform operations in each level of bloom filter during insertion and query, time complexity of MLBF is a little higher than that of classic bloom filter. Fig. 6 shows the time comparison between these two algorithms when inserting URLs.

As shown in Fig. 6, we know that insertion time of BF and MLBF increases when the number of inserted URLs increases, while the insertion time of BF is less than that of MLBF. However, the insertion time of MLBF is not much longer than BF, and it is still acceptable because of its lower false positive.

Fig. 7 shows the comparison of the time that BF and MLBF take to query the same number of URLs. As can be seen from Fig. 7, the query time of BF is a little bit less than that of MLBF, while both of them are almost the same. Therefore, MLBF is almost the same as classic bloom filter in efficiency of query.

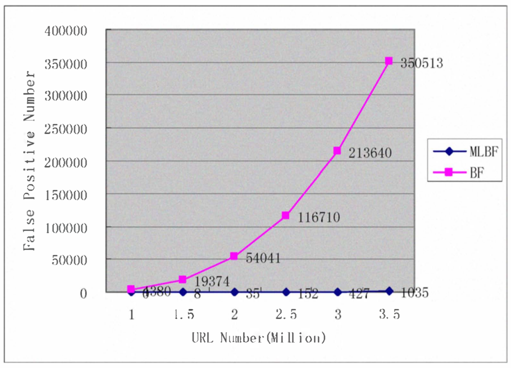


Figure 5. False positives comparison in BF and MLBF

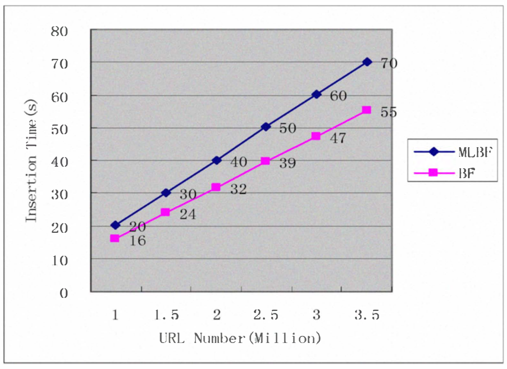


Figure 6. Insertion time comparison between BF and MLBF

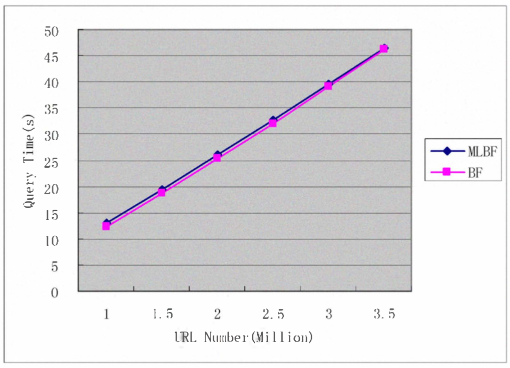


Figure 7. Query time comparison between BF and MLBF

VI. Conclusions and Future Work

In this paper, we introduced a duplicated URL detection algorithm based on multi-layer bloom filter, and compared the performance between multi-layer bloom filter and classic bloom filter through the experiments. The algorithm for duplicated URL detection is based on the classic bloom filter, and considering the layered characteristics of URL.

MLBF needs more memory space than classic bloom filter, but false positive times of MLBF is much lower than classic bloom filter, and the time that these two algorithms spend to insert and query URL are almost the same. Therefore, since the low false positive and high efficiency of MLBF algorithm, to use MLBF for duplicated URL detection in a Web Crawler is very valuable in practice,

MLBF proposed in this paper just runs on single computer currently. With the rapid growth of number of web pages, a single computer is unable to complete the task. Therefore, further work is to research how to apply MLBF on a distributed environment so that it can detect duplicated URLs much faster. Meanwhile, we plan to extend our work into duplicated web content detection in the future.

REFERENCES

[1] China Internet Network Information Center, "China Internet Development Report," http://research.cnnic.cn/img/h000/h10/attach200906151417190.doc, 2007.

[2] China Internet Network Information Center, "China Internet Development Report," http://www.cnnic.cn/uploadfiles/pdf/2010/1/15/101600.pdf, 2010.

[3] Netcraft, "December 2009 Web Server Survey," http://news.netcraft.com/archives/2009/12/index.html, 2009.

[4] Y. Yun-ming, Y. Shui, M. Fan-yuan, S. Hui, and Z. Ling, "On Distributed Web Crawler: Architecture, Algorithms and Strategy," Acta Electronica Sinica, vol. 30, Dec. 2002, pp. 2008-2011, doi: CNKI:SUN:DZXU.0.2002-S1-022.

[5] B. He, T. Di-bin, and W. Jin-lin, "Research and Implementation of Distributed and Multi-topic Web Crawler System," Computer Engineering, vol. 35, Oct. 2009, pp. 13-16, doi: CNKI:SUN:JSJC.0.2009-19-007.

[6] H. Allan and N. Marc, "Mercator-A Scalable, Extensible Web Crawler," World Wide Web, vol. 2, Dec. 1999, pp. 219-229, doi: 10.1023/A:1019213109274.

[7] B. Hemant, "A SNAPSHOT OF THE WEB," http://longwood.cs.ucf.edu/~hemant/report.pdf, 2004.

[8] S. Vladislav and S. Torsten, "Design and Implementation of a High-Performance Distributed Web Crawler," Proc. the 18th International Conference on Data Engineering (ICDE 2002), IEEE Press, Feb. 2002, pp. 357-368, doi: 10.1109/ICDE.2002.994750.

[9] B. Dustin, "Distributed High-performance Web Crawlers: A Survey of the State of the Art," Department of Electrical & Computer Engineering, University of California, San Diego, 2003.

[10] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," Communications of the ACM, vol. 13, Jul. 1970, pp. 422–426, doi:10.1145/362686.362692.

[11] M. K. James, "Optimal Semijoins for Distributed Database Systems," IEEE Transactions on Software Engineering, vol. 16, May 1990, pp. 558–560, doi: 10.1109/32.52778.

[12] M. D. McIlroy, "Development of a Spelling List," IEEE Transactions on Communications, vol. 30, Jan. 1982, pp. 91–99.

[13] L. L. Gremillion, "Designing a Bloom Filter for Differential File Access," Communications of the ACM, vol. 25, Sep. 1982, pp. 600–604, doi: 10.1145/358628.358632.

[14] C. Matias Francisco, P. Christopher, M. P. Richard, and N. D. Thu, "PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities," Proc. the 12th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2003), IEEE Press, Jun. 2003, pp. 236-246.

[15] R. C. Sean and K. John, "Probabilistic Location and Routing," Proc. the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002), IEEE Press, Nov. 2002, pp. 1248-1257, doi: 10.1109/INFCOM.2002.1019375.

[16] K. Abhishek, X. Jun, W. Jia, S. Oliver, and L. Li, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement," Proc. the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004), IEEE Press, Nov. 2004, pp. 1762-1773, doi: 10.1109/INFCOM.2004.1354587.

[17] S. C Dinesh, G. Zhi, B. Betul, and N. A. Walid, "Automatic Compilation Framework for Bloom Filter based Intrusion Detection," in Reconfigurable Computing: Architectures and Applications, vol. 3985, Berlin:Springer, 2006, pp. 413-418.

[18] L. Fan, P. Cao, A. Jussara, and B. Z.Andrei, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol," IEEE/ACM Transactions on Networking, vol. 8, Jun. 2000, pp. 281–293, doi: 10.1109/90.851975.

[19] M. Michael, "Compressed Bloom Filters," IEEE/ACM Transactions on Networking, vol.10, Oct. 2002, pp. 604–612, doi: 10.1109/TNET.2002.803864.

[20] C. Saar and M. Yossi, "Spectral Bloom Filters," Proc. the 2003 ACM SIGMOD International Conference on Management of Data, ACM Press, Jun. 2003, pp. 241-252, doi: 10.1145/872757.872787.

[21] X. Ming-zhong, D. Ya-fei, and L. Xiao-ming, "Split Bloom Filter," Acta Electronica Sinica, vol 32, Feb. 2004, pp. 241–245, doi: CNKI:SUN:DZXU.0.2004-02-014.

[22] SogouT-Link, "Sogou laboratory data download-SogouT-Link," http://www.sogou.com/labs/dl/t-link.html. 2008.