Dynamic Reordering Bloom Filter

Da-Chung Chang[1], Chien Chen[12], and Mahadevan Thanavel[1]
[1]Department of Computer Science and [2]Information Technology Service Center
National Chiao Tung University, Hsinchu, Taiwan
jcus.cs98g@nctu.edu.tw, chienchen@cs.nctu.edu, tmahadevan1991@gmail.com

***Abstract*—In order to check a membership in multiple sets of bloom filter in a dynamic bloom filter, a sequential search is usually used. Since the distribution of queried data is unpredictable because the distribution has a feature of temporal locality. Therefore more search cost is incurred if queried data is stored in the peer which is corresponded to the Bloom Filter has lower query priority. In this paper, we introduce Dynamic Reordering Bloom Filter that can save the cost of searching Bloom Filter by dynamically reorder the searching sequence of multiple bloom filters in a dynamic bloom filter with One Memory Access Bloom Filter (OMABF) and checked in the order saved in Query Index (QI). The performance of the system is evaluated by Markov Chain. Simulation results show that our scheme on average has 43% better in searching performance comparing with the sequential methods, which is verified via three different trace log files.**

Keywords—bloom filter, dynamic bloom filter, one memory access bloom filter, temporal locality, distributed system.

I. INTRODUCTION

Bloom Filter has the benefit in saving memory space and checking membership in constant time [1]. It helps applications to increase their performance with fewer extra resources consumption. Many variants of Bloom Filter [3] are proposed for applying to different purpose into research fields. Among them, the following three different kinds of bloom filters are induced us think about to present our proposed work.

Firstly, to achieve lower false positive ratio with slowly increasing memory access times, One Memory Access Bloom Filter (OMABF) is an alias of Bloom-1 Filter proposed [4] whereby memory access speed may become a bottleneck when much lower false positive ratio is required for some applications. The basic idea is to replace mapping a datum to k bits randomly selected from the bit vector by mapping a datum to k bits in a block. A block is defined as continuous 'w' bits and can be fetched from memory to the processor in one memory access. Secondly, applications of dynamic data set are more common than static one in the real world, such as distributed hash table of peer-to-peer, network traffic measurement, etc., Dynamic Bloom Filter (DBF) is therefore proposed [5] which can be applied to manage dynamic data set in the manner of distributed system. It consists of three components: (i) a Bloom Filter Counter for recording the number of Bloom Filters included by DBF (ii) many Bloom Filters for managing dynamic data set (iii) many capacity counters for each Bloom Filter. Sequentially searching Bloom Filters is used to perform membership check in DBF. The searching order starts from number BF1. In distributed system, DBF maps one Bloom Filter to one peer. Once information of peer needs to be modified, the corresponding Bloom Filter is rebuilt. And finally, Time-Dependent Bloom Filter (TMBF) [6] depicts the issue of temporal locality. TMBF queries data in the order which is opposite to DBF. The result of TMBF can save up to 20% of extra query cost than DBF.

Recent researches for membership check in multiple Bloom Filters uses sequential search because of no relationship between any two data stored in the Bloom Filters. For mitigating the cost, the above discussed related Bloom Filters induced us to propose a concise scheme to immediately modify the query priority of each Bloom Filter by referencing distribution of temporal locality. In order to save search cost, we introduce a scheme Dynamic Reordering Bloom Filter that can dynamically reorder the searching sequence of multiple bloom filters in a dynamic bloom filter. Higher query priorities are assigned to the Bloom Filters which include frequently queried web documents. These Bloom Filter are called frequent Bloom Filter. There is an occurrence of overhead as more own ranked data are stored in a particular block. This is can be reduced by making Query Index that contains query priority of many bloom filters based on OMABF to improve the effect of search. The performance of the system is evaluated by Markov chain and then compared with simulation results with three trace log files such as NASA web server [17], eDonkey tracker [18] and web proxies [19] respectively. As a result, we show about 43% of search cost saved.

The rest of this paper is organized as follows: Section II introduces the proposed work as Dynamic Reordering Bloom Filter. Section III discusses the simulation results of the proposed work. Section IV ended up with the conclusion.

II. DYNAMIC REORDERING BLOOM FILTER

In this section, we introduce our proposed scheme how to achieve lower cost for checking and maintaining membership in multiple bloom filters. Two factors are considered: (i) Policy of changing the query priority of Bloom Filters (ii) Reducing

the overhead caused by changing the order. Our proposed work improves the effect of search based on TMBF.

A. Policy of Changing Query Priority

There is no information that can be used to build relations between data which are set in different Bloom Filters because incoming data are unpredictable. Therefore, an optimal searching scheme does not exist. Fortunately, queried data has a feature of temporal locality. The feature of the data can help to filter out which data are more frequently queried. For this reason, query order of the data can be sorted by their popularities. Our policy is considered that the query order of a Bloom Filter can be promoted one level up once a queried datum reflecting the popularity of the datum, must be assigned with higher query order after a time interval. Then, the searching cost is reduced because frequently queried Bloom Filter has become a higher query order. Figure 1 shows our idea to promote query order of a BF. Let us consider a red BF, initially, it is located in third query order. Then, the BF is perspective one query order when a queried datum C coming. Although successive other queried data are help to promote they own BF, and the query order just switch between two adjacent BFs. But the perspective on query order of a BF still depends on the popularity of the BF. For an instance, red BF has more popular than others as the datum C coming successively. However, switching any two Bloom Filters for changing their query order will incur too much overhead to practice. Therefore, the idea of OMABF is then introduced for amortizing the above issue.

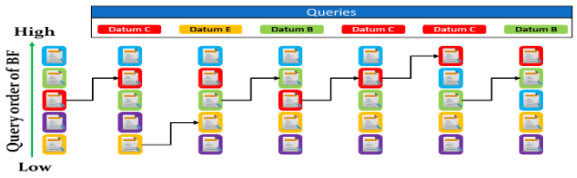


Fig. 1: Query order changing of multiple BFs

B. Improvement of System Performance with OMABF

Multiple memory access times are required for membership check of Bloom Filter because bits which are used to represent a datum are randomly distributed to multiple memory blocks. The number of memory access can be reduced if the bits are rearranged to fewer memory blocks. We replace original Bloom Filters with OMABFs in order to achieve this goal. OMABF takes one or more blocks for setting a datum. The bits are evenly shared by the blocks. Therefore, memory access times can be significantly reduced. Besides, the blocks can be regarded as a tiny Bloom Filter for measuring and managing purposes. Compared with original Bloom Filter, a more accurate result of the popularity of data can be obtained by measuring the block. In the same time, blocks can be assigned with different query order respectively. Therefore, the most popular blocks gain higher query order and create better performance. Queried data keep promoting their own BF and leading the query order of two BFs always exchange. Therefore, the more overhead is caused. Figure 2 shows the architecture of replacing BFs with OMABFs. OMABF consists of multiple homogeneous blocks. Each block only stores a few data of the OMABF. Therefore, a really popular datum just help to promote its' own block. Other blocks still have the same query order and do not compete with each other. However, the problem of updating Bloom Filter may be caused by switching blocks for changing query order, because no information can be used to trace where a block is. Thus, actually moving blocks is impractical. A concise data structure is introduced for solving this problem with the punishment of little extra memory space.

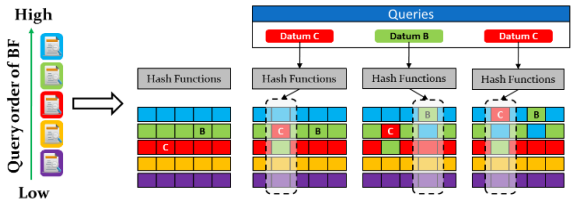


Fig. 2: Architecture of replacing BFs with OMABFs

C. Query Index

Extra overhead is created when data move from the original Data Block to another. Two costs are considered: (i) extra memory access times for exchanging a Data Block between two Bloom Filters (ii) Updating incorrect memory blocks. Based on these considerations, we propose an idea of indirectly querying to avoid moving the block. Query priority of each Data Block is recorded in a Query Index (QI). Query Index(QI) is an integer array for recording query priorities of specific blocks.

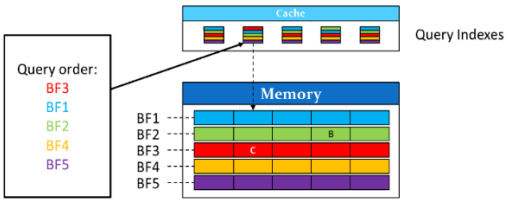


Fig. 3: Architecture of adding Query Index

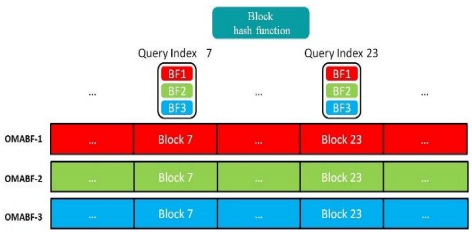


Fig. 4: Architecture of Dynamic Reordering Bloom Filter

Therefore, querying priority can be changed with the data stored in the index. First, initialize the array values with ascending numbers to represent original query priorities for each Bloom Filter. When checking membership, Bloom Filters are verified according to order saved in Query Index. Once a Bloom Filter includes queried document, the order of a value associated with the Bloom Filter is exchanged with previous one in Query Index. Then, the Bloom Filter is checked early in next round. Figure 3 is the architecture after adding QI. Since

QI is much smaller than BFs, the QI can be stored in cache memory. Each QI manages the query order of blocks which are located in the same position mapping by the hash function. The final architecture of Dynamic Reordering Bloom Filter is shown in figure 4. Block hash function is used to choose the blocks which are in the same position but across many OMABFs when accessing a datum. The blocks are checked following the number reported by Query Index when querying a datum. Query Index may be modified once a block includes the queried datum. The evaluation of our model is justified in the following section.

III. SIMULATION RESULTS

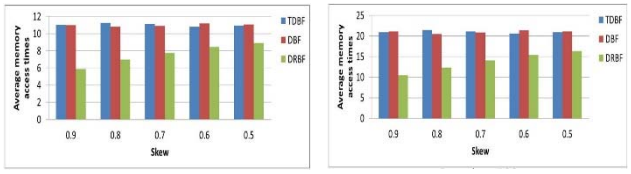
The proposed work considering a policy of changing query priority is evaluated by the comparison with Markov Chain. The performance of a couple of existing frameworks such as DBF and TMBF are discussed with one artificial data set and three real trace logs. In the following content, dynamic bloom filters are used for representing DBF, TMBF or DRBF. Each one of dynamic bloom filters uses Bloom-2 Filters to manage their data set. Artificial data shows the performance in dynamic bloom filters under different skew. Three real trace logs come from different application fields.

A. Artificial Data Set

Table I shows parameters used to produce artificial data set. The foundation of the data set is downloaded from Open Directory Project [14] maintained by Netscape of American OnLine. There are 3.6 million uniform resource locators (URL) in the set. The tested URLs are randomly (not duplicate) multiplied with the numbers produced by Zipf's law under different skews (called α) for creating queried URLs. Each dynamic bloom filter is assigned with two kinds of Query Depth for verification. Query Depth is defined as the maximum number can be checked for one queried datum. Obviously,

TABLE I: PARAMETERS FOR SIMULATION TEST DATA

Parameters	Value
Popularity distribution	Zipf's law
Skew (α)	0.9 ~ 0.5 (step by -0.1)
Data set	10,000 (From domain name set [20])
Queried data set	1,000,000 (According to Zipf's law)
Capacity of BF	500, 1000
Memory size per datum	30 bits
Memory size of each Query Index	128 bits (5 bits for an OMABF)
Average data per block	8
Memory size	300000 bits 300000 bits (DBF + TMBF) 316,000 bits (DRBF + Query Index)
Memory access times per query	2
The number of hash functions	21



(a) Query Depth is ten (b) Query Depth is twenty

Fig. 5: Performance for various α

DRBF can reduce the cost of membership check with increasing α in Figure 5 (a). When α is 0.9, DRBF reduces at most 46% memory access times for membership check. This is because significant bias distribution makes Popular Bloom Filters in higher query priority occurs in higher probability. For others dynamic bloom filters, they have similar performance because the popular queried URLs in the data set are uniformly distributed. At the same time, they execute membership check with sequential search and have no opportunity to gain better performance under bias query distribution. In Figure 5 (b), DRBF gains more benefit when Query Depth is twenty. However, the improvement of the average memory access times is limited even when Query Depth is set as 20 because the distribution of queried URLs dominates the performance. When α=0.9, DRBF can reduce at most 50% memory access times.

B. NASA Web Server Trace Log File

NASA web server trace log is downloaded from [17]. The profile of the log is shown in Table II. Fields of the log file are client ID, timestamp, request URL, and so on. Data of the request URL field are converted to two data sets called URL set and Queried URL set. The former consists of distinct URLs extracted from the log file and are set into dynamic bloom filters. Later it is used for querying dynamic bloom filters. Query Depth is set as thirty. Data of URL set are evenly set into Bloom Filters with ascending timestamp order. It is because the number of new URLs seldom appears every day. More than 95% queried URLs appear during the first day. Others rarely-queried URLs are dynamically created when users interact with web pages. In Figure 6, TMBF has worse performance under this kind of distribution, because it checks membership from the most recent Bloom Filter to the oldest one. On the contrary, DBF checks membership from the oldest Bloom Filter to the most recent one. The idea perfectly matches the distribution and gains significant benefit. The proposed work also gains large benefit because the distribution has an extreme high degree of bias.

TABLE II: PROFILE OF NASA WEB SERVER TRACE LOG

Parameters	Value
Log date	July 1, 1995
During	31 days
URL set	15,350 URLs
Queried URL set	1,569,850 URLs

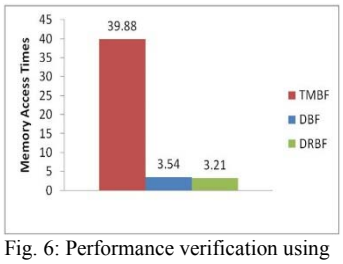


Fig. 6: Performance verification using trace log by NASA web server

C. eDonkey tracker trace log file

Trace log file is downloaded from [18]. The log file records events of peers which communicate with the server. We sample 1% of the file hash codes from the log file to verify the performance of dynamic bloom filters. File hash code is a MD5 hash code for representing the signature of a piece which is

requested or shared by the peers. Two data sets are created from the 1% file hash codes. The first data set stores to file hash codes which are not repeated. The second data set stores the full sample data. The profile of the sample data is shown in Table III. Query Depth is set as twenty-seven because the date of the file hash codes which is firstly discovered is uniformly distributed in every day. The distribution shows a serious appearance of bias. TMBF gains a large benefit from the distribution because of its membership check scheme. Similarly, our proposed work also gains large benefit because our scheme uses the same query priority as TMBF. Verification of the log file is shown in Figure 7.

TABLE III: PROFILE OF E-DONKEY TRACE LOG FILE

Parameters	Value
Log date	November 1, 2006
During	27 days
URL set	2,307,512 file hash codes
Queried URL set	2,335,718 URLs

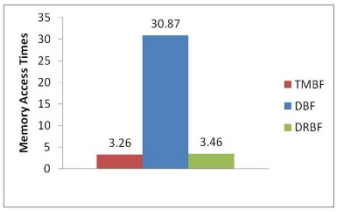


Fig. 7: Performance verification of dynamic bloom filter by eDonkey

D. Web Proxies Log Files

Trace log files are downloaded from [18]. These log files are created by web proxies in the United States. The format of these files follows Squid [19]. The profile of these trace log files is shown in Table IV. Eight trace log files are mixed and reallocated to ten Bloom Filters. Each group is assigned as 37,034 URLs for simulating the operation of a cooperative web proxy. The performance of the dynamic bloom filters is shown in Fig. 8. In the experiment, the order of queried URLs is the same as trace log files. URLs set into dynamic bloom filters are randomly distributed. Eight Bloom filters at most need to be checked when executing membership check for a queried URL. The performance is similar to the first experiment. The proposed work has better performance than the other two. For the same reason, the distribution of queried data has feature of the temporal locality. At most 43% cost of membership check is saved.

TABLE IV: PROFILE OF WEB PROXIES TRACE LOG FILES

Parameters	Value
Log date	January 9, 2007
During	1 day
URL set	296,269 URLs
Queried URL set	1,415,075 URLs

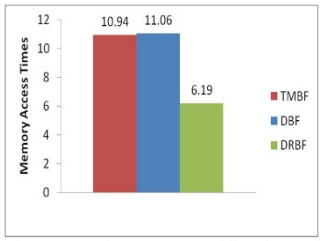


Fig. 8: Performance verification of dynamic bloom filter by web proxy

IV. CONCLUSION

Recently, many researches for membership check in multiple Bloom Filters uses sequential search because of the lack of relationship between any two data stored in the different Bloom Filters. The cost of membership check is increased substantially if many frequently queried web documents are stored in the Bloom Filters which have lower query order. This paper focuses on this issue to propose a concise scheme Dynamic Reordering Bloom Filter for reducing the average cost of membership check. With the temporal locality characteristics in web requests, popular queried documents can be assigned with higher query priority in checking multiple bloom Filters. And query priority of each Data Block of the OMABF is recorded in a Query Index (QI). Our proposed work always have better performance since dynamically change the query order of a BF once it is hit by a queried datum. We demonstrated our results with three real trace logs and web proxies to comparing the performance of our proposed work with linear and reverse query order. Hence, search cost of popular data can be rapid dropped down and save more memory access times.

REFERENCES

[1] B. H. Bloom, "Space/time tradeoffs in hash coding with allowable errors", *Communications of the ACM,* Vol.13, No.7, July 1970.

[2] A. Z. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A Survey", *Internet Mathematics*, Vol.1, no.4, 2004.

[3] S. Tarkoma, C. E. Rothenberg, and E. Lagerpetz, "Theory and Practice of Bloom Filters for Distributed Systems", *IEEE Communications Surveys and Tutorials, Vol.14, No.1,* First Quarter 2012.

[4] Y. Qiao. T. Li, and S. Chen, "One memory access Bloom filters and their generalization", *INFOCOM, Proceedings IEEE,* Jun. 2011.

[5] D. Guo, J. Wu, H. Chen, Y. Yuan and X. Luo, "The dynamic bloom filters", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 22, No.1, Jan 2010.

[6] O. Rottenstreich, and I. Keslassy, "The Bloom Paradox: When Not to Use a Bloom Filter", *IEEE Transactions on Networking,* Vol.23, No.3, Jun. 2015.

[7] J. H. Mun and H. Lim, "New Approach for Efficient IP Address Lookup using a Bloom filter in Trie-Based Algorithms", *IEEE Transactions on Computers,* Vol.65, No.5, May 2016.

[8] P. Reviriego, K. Christensen, and J.A. Maestro, "A comment on "Fast Bloom filters and their generalization", *IEEE Transactions on Parallel and Distributed Systems,* Vol.27, No.1, Jan.2016.

[9] S. Jin, and A. Bestavros, "Source and Characteristics of Web temporal locality", *Proceeding 8th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication System*, 2000.

[10] A. Mahanti, D. Eager, C. Williamson, "Temporal Locality and its impacts on web proxy cache performance", *ELSEVIER Performance Evaluation*, pp.187-203, 2000.

[11] https://en.wikipedia.org/wiki/Markov_chain

[12] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The Variable-increment Counting Bloom filter", *IEEE Transaction on Networking,* Vol.22, No.4, Aug. 2014.

[13] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better Bloom filter", *in Proceedings European Symposium o Algorithms, Springer,* vol.14, pp.456-467, 2006.

[14] Open Directory Project, from http://rdf.dmoz.org/

[15] M. Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom filter forwarding architecture for large organizations", *in proceedings ACM CoNEXT,* pp.313-324, Dec. 2009.

[16] NASA web server trace log file, from http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

[17] eDonkey trace log file, from http://fabrice.lefessant.net/traces/edonkey2/

[18] Web proxies trace log files, from ftp://ftp.ircache.net/Traces/DITL-2007-01-09/

[19] Squid, from http://www.comfsm.fm/computing/squid/FAQ-6.html