A Privacy Preserving Model for Ownership Indexing in Distributed Storage Systems*

Tiejian Luo
University of Chinese Academy of Sciences
tjluo@ucas.ac.cn

Zhu Wang
University of Chinese Academy of Sciences
wangzhubj@gmail.com

Xiang Wang
University of Chinese Academy of Sciences
wangxiang11@mails.ucas.ac.cn

ABSTRACT

The indexing technique in distributed object storage system is the crucial part of a large scale application, where the index data structure may be published in many nodes. Here arises a problem on preserving the privacy of the ownership information while supporting queries on item locations with limited index space. Probabilistic data structure, such as the bloom filter which records the location of each item in distributed nodes, is one of the promising solutions. The data structure uses a hashed vector to index items on the nodes. In this paper we propose a Lightweight Bloom filter Array (LBA) indexing model which is compact in size and preserves ownership privacy. To tackle with the problem of examining wrong nodes in the lookup process, we find an optimal storage ratio of the bloom filters and reduce its false positive rate based on the observation of the user's access behavior in Internet applications. We use experiments to verify our proposed solution. In our experiment, the dataset consists of one billion items distributed in one hundred data nodes. The experiments show that our model can reduce the false checking times and save the index space significantly.

1. INTRODUCTION

With the rapid growth of the Internet, many online applications have been based on distributed storage systems which are composed of many single storage nodes. When a request for a certain item arrives at the system, the first step is to find which node contains the item. The indexing service, which is capable of recording ownership relation between nodes and items, is a key component in distributed systems. The node that holds the service is called the index node. It can be either a storage node that have the index function the same time as the storage role or an exclusive node that is only responsible for the indexing service. Because of the large total number of items, each storage node in the system is usually in charge of many items. However, in many distributed systems, the indexing data structure has to be deployed in many (even all) nodes in order to support fast lookup and high scalability. Therefore the index has to be very compact in size in order to be stored in the index nodes' memory. Moreover, in many distributed systems, an item can be a video slice, file fragment or even a piece of record. It needs several times of item lookup in the table to finish a meaningful service. Therefore, we need a high performance indexing technique with limited space consumption in distributed systems.

Private information grows with the increase of data volume and service type. Index data structure has to support the queries for resident nodes of an item while keeping the entire ownership information away from unknown requests. Since distributed systems may have more than one index nodes, it is sometimes unavoidable for the information transportation and indexing publishing, and hence unpredicted interception. In those cases, the security of the information stored in the index may be threatened. Compact, fast and secure index will become a key component in distributed applications.

Bloom filter[2] is a space-efficient probabilistic data structure for item representation and lookup in a set. When indexing space is limited, i.e. in the memory, the data structure offers fast item lookup with a low false positive rate. Many distributed systems that emphasize time efficiency are using bloom filters as their indexing technique when a small false positive rate is tolerable[8]. The data structure uses hash functions to map items onto several positions on a bit vector. It only stores the hashed bits and allows for hash collisions. The actual data is not stored on the vector. Without the knowledge of hash functions, the ownership information cannot be obtained from the vector. In this way, the privacy is preserved. Actually, many online systems[1, 4, 3] are using bloom filters for secure index. In our paper we try to optimize their usage with reference to user behavior.

The observation of user behavior indicates that in many applications, a small number of items attract a major part of user access. The phenomenon inspires us to be selective in index construction when space is limited. In this paper we provide a lightweight mechanism for bloom filter usage. The lookup procedures are given to guarantee effective item

locating.

The rest of the paper is organized as follows. Section 2 describes the related work of our research. In Section 3, we give the lightweight bloom filter construction and item lookup procedures. Experimental results are shown in Section 4. Finally, we present the conclusion and future work in Section 5.

2. BLOOM FILTER AND PURE BLOOM FILTER ARRAY (PBA) INDEX

2.1 Bloom Filter

Bloom filter[2] works as an index which records all elements of a set. We may assume that the set $S = \{x_1, x_2, ..., x_n\}$, which consists of n elements. A Bloom Filter vector (BFV), which consists of m bits, is used to represent elements of set S. All bits of the vector are set to zero initially. For each element, the algorithm uses k hash functions $\{h_i\}_{i=1...k}$ to map the element onto k positions of the vector and sets the bit on the position to 1. The k functions, ranging from 1 to m, are independent from each other and can map elements of the set S to a random place on the vector. During the insertion period, the algorithm maps all elements of the set to load the BFV with all the information of the elements.

In lookup procedure which we want to check whether an element y belongs to the set S, the algorithm uses the same hash functions to map y onto k locations and check whether all $h_i(y)$ equal to 1. If the answer is no, we conclude that y doesn't belong to S, otherwise, we say y belongs to S. The time complexity of bloom filter lookup is constant.

It needs to be mentioned that there is a probability that elements don't belong to S be judged as inside S by BF. That is to say, BF has a false positive rate. Research[6] shows that the false positive rate can be represented as follows:

$$f_{FP} = (1 - e^{-\frac{kn}{m}})^k \tag{1}$$

Study[6] also shows that f_{FP} reaches minimal value when

$$k = \frac{m}{n} ln2 \tag{2}$$

Then the false positive is minimized

$$f_{FP} = 0.6185^{\frac{m}{n}} \tag{3}$$

Since the bloom filter does not store the actual data, ownership information cannot be revealed without the knowledge of hash functions. The algorithm can be used as secure index in online applications[1, 4, 3]. Due to its simple structure and smooth integration characteristic, the mathematical format allows considerable potential improvement for system designers to develop new variations for their identical application requirements. In this paper, we focus on fast object lookup in distributed systems.

2.2 Pure Bloom Filter Array for Distributed Data Storage Index

Many distributed systems use Pure Bloom filter Array (PBA)[10] to support item index and lookup. The approach consists of a two-stage process: indexing building and item locating.

Index Building. For each node of the system, the index node builds a bloom filter for representing all of its items. These Bloom filters are loaded with all the items in the entire system and can act as an indexing system.

Item Locating. The object locating process is described below: when a query for a certain item arrives on the index node, the node first uses the bloom filters to find the approximate membership relations: it calculates with the bloom filter of each node and collects the results. The negative result of a certain bloom filter means that the queried item doesn't exist on the related node. The positive result means that the queried item exists on the node with a probability of $1 - f_{FP}$. Then the system queries the actual node whose bloom filter check result is positive to check whether the queried item exists in the node. In that way, the false positive occurrence is finally eliminated. Since the bloom filters have a constant time complexity, the method can reduce lookup time remarkably.

3. LIGHTWEIGHT BLOOM FILTER ARRAY FOR DISTRIBUTED STORAGE INDEX

3.1 Lightweight Bloom Filter Design

User behavior observation indicates that access for items varies between different objects. In many applications, the access frequency can be observed accurately. If ordered by access frequency, the top ranked items attract a large portion of user visits while the low ranked items absorb a very little part. That phenomenon shows us a way to increase bloom filter space use efficiency. Each node in the system builds a bloom filter for indexing the items of the node. It selects the highly ranked items and inserts them into the bloom filter. The bloom filters on the nodes forms an array, which is capable of recording item ownership information on nodes and plays the role of a distributed index. Though the total index space is limited, the data structure has a lower load and therefore a lower false positive rate. Queries for highly ranked (popular) items will have a more accurate response. Queries for low-ranked items will not receive a positive response from the bloom filter index. In those cases the system uses traditional lookup method to find the queried item directly in the storage nodes. Since most queries are for popular items, the overall false positive rate of the index can be reduced. The detailed bloom filter improvement method is described below. In that way, the index can perform with lower space consumption and preserve ownership privacy.

Index Building. The system sets a load factor β, which is the ratio of the loaded item to the total item number. Each node in the system orders the items by their access time. Then it inserts the items one by one from rank 1 until it reaches the load threshold. The bloom filters are gathered to form an array, which represents the ownership relation between items and nodes, and stored on the index nodes. For a system of totally N items, the bloom filter arrays index βN items.

Item Locating. When a query for a certain item arrives, the index node first calculates with the bloom filter of each node and collects the results. One possible situation after the calculation is that there is at least one positive result, it means that the queried item exists on the node with a probability of $1 - f_{FP}$. Then the system first queries the actual node whose bloom filter check result is positive to verify whether the queried item exists in the node. If it does exist on one of the positive nodes, the lookup procedure stops; otherwise it continues to check on the remaining negative nodes until it finds the queried item. The other possible situation after bloom filter calculation is that there is no positive match. Under that circumstance, the system looks up the item in each unsearched node directly. The lookup procedure is shown in Fig.1.

In the two steps of the lookup procedure, the bloom filter calculation takes place in index nodes' memory. The time consumption is rather low considering the O(C) time complexity of bloom filters. The vast majority of time cost comes from the node lookup process, in which the index node communicates with the storage node and the storage node check in its disk for queried items. So the time consumption is approximately linear to the average checking times of a query. In the many check in nodes, only one of the checking procedure can find the needed item. The rest nodes checking end without a match and waste a lot of time and system resource. Those redundant (false) checking times are the key factor that lowers system performance. The occurrence that the system looks up a query in a wrong node and finds no result is called the *false checking* in nodes.

3.2 Selection of Popular Items

In the design of LBA, the algorithm stores the top $\beta \times 100\%$ popular items in the bloom filter index. In actual processing, each node keeps a list of all its popular items and builds the lightweight bloom filter of its own. Then it transmits the index to the index nodes, where all bloom filters are stored in.

The selection of the popular items follows the same procedure of that with cache. The goal is to find the top $\beta \times 100\%$ items in each node efficiently with high accuracy. The cache selection and update algorithms are adequate for that task. Actually in [10], the authors use LRU cache scheme for selecting popular items.

It needs to mention that the item selection process takes place in each storage node, just like the cache does. It does not occupy the resource of the index node, which is responsible of storing the bloom filters already built on the storage nodes. Also, the index node does not have the entire knowledge of the ranking of items on the storage nodes.

3.3 Dynamic Items and Renewal Process

In online systems, the access pattern of items and visit frequency varies over time. A "hot" item may become unpopular after a certain period of time. The updating of popular list on each storage node follows the same way as caches do. After reaching a certain threshold, the node rebuilds the bloom filter index in the same way as [10] does. Then it transmits the new filter to the indexing node. In update mechanism, we use the mechanisms given in [10] and [9]. The effect of that method is given in [9].

4. EXPERIMENTAL EVALUATIONS

In this section we use experiments to show the index performance. The items and nodes are synthetic data. In all experiments we set node number s=100, the total number of items $N = 10^9$. The items are scattered randomly among s nodes, so each node has approximately $n = 10^7$ items. The probability that an object be allocated in one node is identical among all servers. The total query number $Q = 10^5$.

4.1 Queries

Observations show that several "hot" items attract a majority of user access, as stated in Zipf's law[7, 5]. In the experiment we assume that access for the entire corpus follows Zipf's distribution with total number $N = 10^9$. The queries reflect the real popularity of the corpus, so the queries follow the same distribution as the corpus. Actually, queries are generated automatically as a sampling set of Zipf's law with parameter $N = 10^9$.

The real query count of top 10000 hot items are plotted in Fig.2.

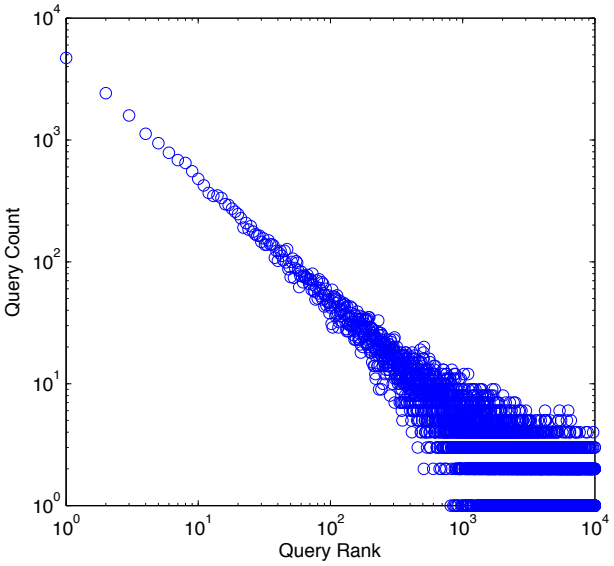

Figure 2: Query count for top 10000.

The figure is in a log-log scale. We can see that the first half part of the figure is a nearly straight line with slope=-1. That indicates the queries follow the Zipf's law. The second half consists of some irregular points because of the low value of the actual count.

4.2 Node Construction

In actually processing of bloom filter construction in storage node, we first order all items in the node by its popularity rank and pick the first βn items. Then we insert those items into its bloom filter. For easy deployment in the memory of the node, the bloom filter needs to be compact in size. In the experiment we set the length of the bloom filter vector

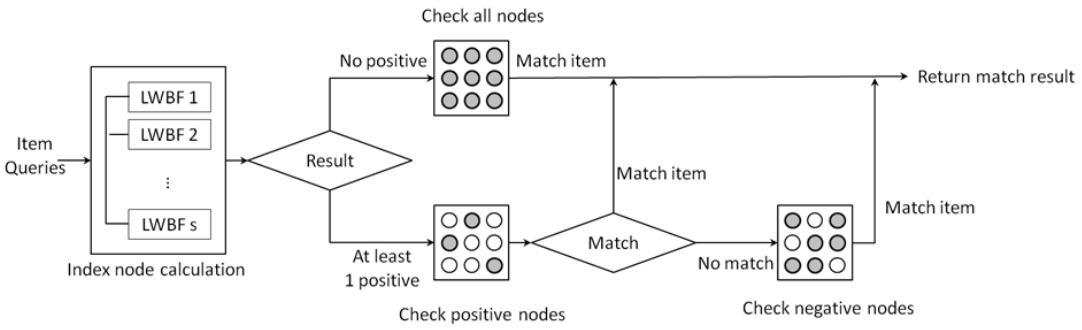

Figure 1: Item lookup procedure.

m=33554432 (use 3.36 bits for one item). The hash number of the bloom filters reaches the nearest integer of its optimal value in (2).

4.3 Experiments with Different β

In the experiment we want to analyze the impact of β. First, we want to know whether our proposal can actually improve the bloom filter index performance. Second, we are going to find out what the optimal load factor is in the lightweight bloom filter.

The range of β is $\{0.1, 0.2, ..., 1\}$. Here β=1 means that the bloom filters are fully loaded, which is equivalent to pure bloom filter array. For each β, we build the bloom filter index in each node and lookup all Q queries until we find a match. In the experiment, we count the total false checking times when we finish serving all the queries.

In order to find if our proposed method have a positive effect, we use the PBA approach and the direct lookup approach (in which nodes are checked one by one to find a queried item without using index) for a comparison. In the following experiment we repeat the indexing and querying procedure separately with the same system environment: index objects, queries, nodes, etc. Then we count the false checking times of each method. The false checking times in each experiment is plotted in Fig.3.

In the figure we can see that both LBA and PBA have an impressive improvement over direct lookup method (DL) considering false checking rate. That will reduce the overall system workload. Comparing PBA and LBA, we find that the LBA performs better than the PBA and reaches its optimization when β=0.4. The two methods come near when β approaches 1. When β=1, the LBA is fully loaded and is equivalent to PBA. The false checking rate of LBA reaches only 30 percent of that in PBA at optimal β. That will have a direct impact on system performance. Under that system parameter, when top forty percent popular objects are loaded, the false checking rate is 5.87, which means that on average the system will check 5.87% of all nodes before finding a query.

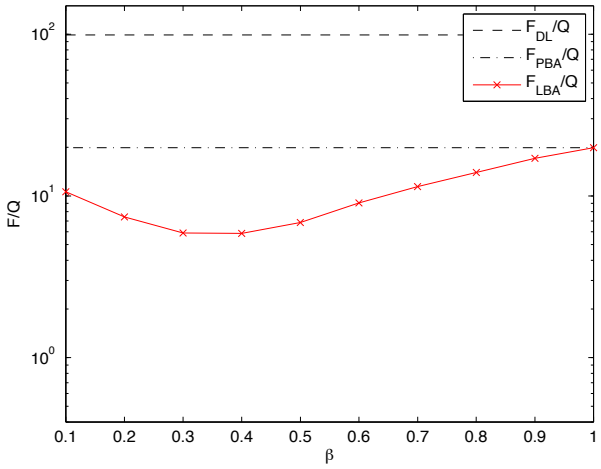

Figure 3: Overall false checking in node.

4.4 Space Consumption

Compared with the PBA, LBA can be more accurate when the index memory cost is very low. Equivalently, when reaching the same accuracy, the LBA can use less space than PBA. Take the example in the experiment, with one billion items distributed in one hundred nodes, a query will be checked on 5.87% of all nodes before finding its right location. The space needed for the indexing structure is only about 400MB using LBA (3.35 bits for one item), which is affordable in many distributed nodes. If we use PBA method instead, in order to achieve the same accuracy, we need about 700MB memory space. The space needed has reduced by 43%.

4.5 Distribution of False Checking Times

In the experiment, we record the false checking times of each query. Fig.4 shows the experimental false checking distribution when β=0.4. We see that most queries will find their match after ten false checking times. Some queries will have to go through all nodes to find a match.

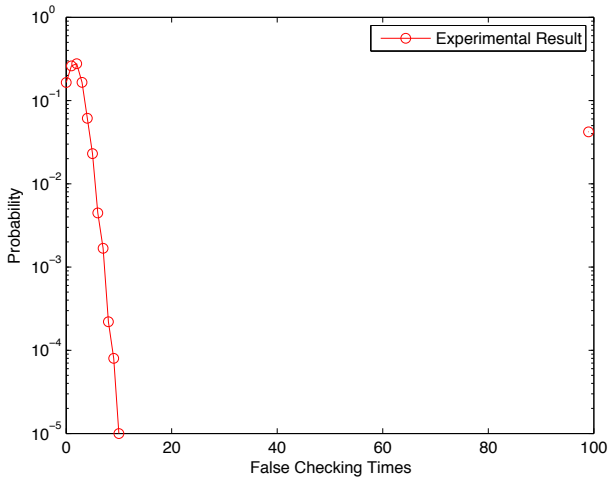

Figure 4: False checking (β=0.4).

Fig.5 shows the experimental false checking times distribution with different β. We can see that for every β, the false positive rate concentrates on two areas: the low false checking area (Bernoulli distribution) and s-1. When β increases, the mean of Bernoulli distribution increases while the false checking times s-1 decreases (when β=1 the false checking on s-1 is 0).

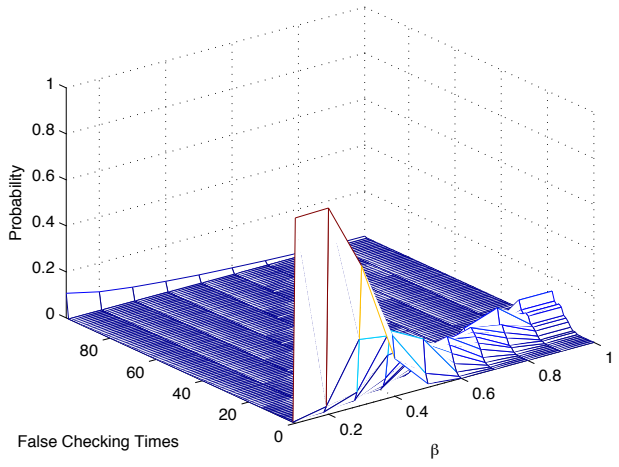

Figure 5: False checking distribution.

5. CONCLUSION

In the paper we have presented a privacy preserving model for ownership indexing in distributed storage systems - the lightweight bloom filter array based on the observation that user access for items varies from case to case in many Internet applications. We have pointed out a problem in the secure index - support fast ownership query with limited index space. We have changed the traditional bloom filter construction method by ranking the items and putting only a part of top visited items into the bloom filter array. Then the experimental evaluation has been conducted to show that the mechanism can reduce false checking times by 70 percent. We have demonstrated that our algorithm can improve the system performance while preserving the privacy of ownership relation in distributed systems when the index nodes' memory is limited.

The future work includes adopting the new algorithm to more complex indexing systems and adding new functions to the algorithm, i.e. item deletion. The improvement of lookup procedure will also continue in the development of the algorithm.

6. REFERENCES

[1] M. Bawa, R. J. Bayardo Jr, R. Agrawal, and J. Vaidya. Privacy-preserving indexing of documents on the network. *The International Journal on Very Large Data Bases*, 18(4):837–856, 2009.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[3] E.-J. Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[4] F. Jian-ming, X. Ying, X. Hui-jun, and W. Wei. Strategy optimization for p2p security using bloomfilter. In *International Conference on Multimedia Information Networking and Security (MINES'09)*, pages 403–406. IEEE, 2009.

[5] I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi. Modeling of cache access behavior based on zipf's law. In *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA '08)*, pages 9–15. ACM, 2008.

[6] J. K. Mullin. A second look at bloom filters. *Commun. ACM*, 26(8):570–571, 1983.

[7] P. Rodriguez, C. Spanner, and E. W. Biersack. Analysis of web caching architectures: hierarchical and distributed caching. *IEEE/ACM Trans. Netw.*, 9(4):404–418, 2001.

[8] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2012.

[9] Y. Zhu and H. Jiang. False rate analysis of bloom filter replicas in distributed systems. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP '06)*, pages 255–262. IEEE Computer Society, 2006.

[10] Y. Zhu, H. Jiang, J. Wang, and F. Xian. Hba: Distributed metadata management for large cluster-based storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 19(6):750–763, 2008.