Designing a New Bloom Filter-based Index for Distributed Data Management *

Tiejian LUO, Zhu WANG*, Fuxing CHENG, Xin ZHANG, Xiang WANG

University of Chinese Academy of Sciences, Beijing 100049, China

Abstract

Distributed architectures are widely used in Internet applications nowadays. In such systems, one of the key techniques is how to maintain an indexing data structure which records elements of each single node in the system. Bloom filter is one of the popular solutions. The beautiful mathematical format offers a fast and space-efficient solution for probabilistic membership presentation. In many Internet applications, user access for items follows Zipf’s law where a small number of items attract many visits. According to that phenomenon, we propose a selective insertion method of bloom filter to reduce the workload of BFs by finding an optimal load ratio. The experiments show that our new approach can reduce the false lookup time by 36% compared with the pure bloom filter approach.

Keywords: Indexing; Bloom Filter; Zipf’s Law; Selective Insertion

1 Introduction

The rapid growth of the Internet brings about the wide adoption of distributed storage systems in online applications. In such systems, when a request for a certain item (in this paper we interchangeably use “item” and “object”) arrives at the system, the first step is to locate the object storage place in the distributed nodes. Therefore, the distributed index, which is capable of recording items in each node, is a key component in all distributed systems. A typical index is the global table. The system simply records each item of the nodes in a large table, one item bounded with one storage location. Despite its simplicity in theory and adoption, the global table (GT) suffers from the large space consumption and low time-efficiency. The space complexity of GT is O(N), where N is the total number of objects. In the distributed system, the index space is a bottle neck for index service, which may be deployed in an ordinary node. Moreover, the time complexity for row-by-row search of GT is O(N). When the total number of items reaches one billion, the lookup procedure of one certain item will take several tens of seconds [1]. With the help of B-tree [2], the lookup performance can be improved (O(logN)). However, the insertion and deletion operation time will increase as a price for that.

*This work is funded in part by project Y11Z01YEH2 supported by China MOST, NSFC Project 61103131/F020511.

Corresponding author.

Email address: wangzhu09@mails.ucas.ac.cn (Zhu WANG).
Bloom filter [3] is a space-efficient probabilistic data structure for membership presentation of a set. When indexing space is limited, i.e. in the memory, the mathematical format offers fast item lookup with a low false rate. Actually, many online systems are using bloom filters as an index to obtain high performance [4, 5]. In this paper, we use the bloom filter as an index for distributed storage systems. We optimized the item insertion method to reduce the false rate of BFs and achieve high overall performance.

Internet user access pattern for objects can have influence on system performance. User access are observed to follow Zipf’s law [6], where a small number of objects attract a large portion of traffic. Consequently, there are some objects that are “hot” for users while other objects relatively “cold”. Using the same space for indexing all items will cause a waste for “cold” items, because they are seldom visited. In this paper, we try to insert selected items into the bloom filter index. We design a lookup procedure to make the search process both effective and efficient. Then we use both theoretical analysis and experiments to find the optimal system parameter.

The rest of the paper is organized as follows. Section 2 describes the related work of our research. In Section 3, we first give a selective bloom filter array (SBA) construction and item lookup procedures. Then, the theoretical analysis of the system performance is deducted. Experimental results are shown in Section 4. Finally, we present the conclusion and future work in Section 5.

2 Related Work

Here we list the previous work of other researchers, their contributions are also the foundations of our work.

2.1 Bloom filter

Bloom filter [3] works as an probabilistic index which records all elements of a set. Given a set with \( n \) elements, the bloom filter with length \( m \) and hash number \( k \) has a false positive rate [7]:

\[
f_{FP} = (1 - e^{-kn/m})^k\]

(1)

\( f_{FP} \) reaches minimal value when

\[
k = \frac{m}{n} \ln 2
\]

(2)

Then the false positive is minimized

\[
f_{FP} = 0.6185 \frac{m}{n}
\]

(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. Counting bloom filters [8, 9] can be used to improve network router performance [10] and for redundancy removal [11]. Other variations are adopted in state machines [12], Internet video [13], security index [14] and publish/subscribe networks [15], etc. In this paper, we focus on fast object lookup in distributed storage systems.
2.2 Zipf’s law

The basic user activity in Internet systems is to send a content query and receive response. The user requests in many online systems are observed to be differentiated between identical objects. That is to say, the frequency of user request for different objects varies from one item to another. The phenomenon is reported in many independent research papers. Work of [16, 17] claims that user requests for terms in information retrieval system follow Zipf’s law. Researchers in [18] find term frequency follows the Zipfian distribution. In Zipfian distribution [6], the top twenty percent of ranked items account for a major part of the total query probability. Zipf’s law states that the probability of a ranked object is inversely proportional to its rank.

\[ P_i = C \cdot \frac{1}{i} \]  \hspace{1cm} (4)

Here \( C \) is a constant. \( i \) is the rank of the object. \( P_i \) is the probability of the item occurrence. The total number of items is \( N \), which is very large. Using normalization condition and conclusions in [19] we have

\[ P_i = \frac{1}{i \cdot \ln N} \]  \hspace{1cm} (5)

The fact that many applications have differentiated user requests can have a notable impact on Internet system performance, especially on object indexing and lookup. A “hot” item which arouses many query requests can cause workload imbalance while a “cold” item may be wasting its allocated resource. Therefore, the differentiated query frequency for items complicates the indexing design. On the other hand, it may provide an opportunity for system architects to improve the performance by using the observed rule.

3 Index Design

In this section we present a selective bloom filter array construction method and lookup procedure. For comparison with our method, we first describe the pure bloom filter array index [20, 21].

3.1 Pure bloom filter array for distributed data storage index

Many distributed systems use the pure bloom filter array 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 method builds a bloom filter for representing all of its items. These bloom filters are then loaded with all the items in the entire system and can act as an indexing system.

**Item locating:** When a query for a certain item arrives, the system 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 rate is finally eliminated. Since the bloom filters have an O(C) time complexity, the method can reduce lookup time remarkably.

3.2 Selective bloom filter array for distributed storage index

Since the distributed systems have the need to support membership queries in distributed nodes, the indexing data structure have to be compact in size. When bloom filter vector length is limited, the large item number will cause high false rate and low system performance. We try to optimize bloom filter usage in distributed systems by presenting a selective bloom filter array mechanism.

Zipf’s law indicates that user access for items varies between different objects. 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 usage efficiency. We try to select the high ranked items and insert them into the bloom filter. Though the total index space is limited, the data structure has a lower load and therefore a lower false rate. Queries for high 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 we use traditional lookup method to find the queried item directly in the nodes. Since most queries are for popular items, the overall false rate of the index can be reduced. The detailed bloom filter improvement method is described below.

![Fig. 1: Lookup procedure for selective bloom filter array](image)

**Index building:** The system selects items to be inserted into the bloom filter index. It first decides a load factor $\beta$, which ranges from 0 to 1. The factor indicates the ratio of the number of loaded items to the total number of items. Each node of the system maintains a visit frequency of items. In the bloom filter construction, it inserts the top $\beta \times 100\%$ items into the data structure.
**Item locating:** A query for a certain object arrives at the system. The system first calculates the bloom filters of each node to find if there is a match. If one of the answers is yes, it means that there is a high probability that the item exists on the node. Then it checks the node to find the object. If the item does exist on the node, the lookup procedure stops. If all the BFs cannot find a match or the checking node procedure does not find the wanted item, it indicates that the object is a “cold” item and therefore not loaded in the BFs. Then the system broadcasts the query to all the nodes of the system and asks for a direct check-in-node procedure. On receiving the direct lookup request, each node looks up the item locally. The item locating process is illustrated in Fig. 1.

### 3.3 Theoretical analysis

We first define the system environment and parameters. Let $N$ be the total number of items of the system. $s$ is the node number. We assume that those objects’ distribution on the nodes is in uniform distribution, so each node has approximately $N/s$ items. Let $Q$ be the total number of queries. We assume that the queries are generated by Internet users and follows Zipfian distribution. The load factor (ratio of loaded items in BF to total items) is $\beta$. The vector length of each bloom filter is $m$ and the hash function number is $k$.

For a node with a load factor $\beta$, the bloom filter only stores $n = N\beta/s$ items. So the false positive rate is

$$ f_P = (1 - e^{-\frac{mN\beta}{ms}})^k $$

According to Eq. (2), when $k$ reaches optimal value,

$$ k = k_{opt} = \frac{ms}{N\beta} \cdot \ln2 $$

$f_P$ reaches its minimal

$$ f_P = f_{P_{opt}} = 0.6185 \frac{m}{N\beta} $$

$Q$ is the number of queries. Since queries follow Zipf’s law, the ratio that the queried item is indexed by the bloom filters (hit rate) is

$$ r = \sum_{i=1}^{N\beta} P_i = \frac{\ln(N\beta)}{\ln(N)} = 1 + \frac{\ln(\beta)}{\ln(N)} $$

The bloom filter data structure is small enough to be stored in the nodes’ memory. The time complexity is $O(C)$. Therefore, the time needed for the bloom filter calculation process can be very short. On the contrary, the node checking process (including network communication and lookup in actual node) may cost a lot of time. In the lookup process of one item, after the bloom filter lookup, the system will check the actual nodes to find the item. If there are false positive occurrences, the lookup of the query in the nodes will find no match. If the object is a “cold” item, the request will be broadcast to all nodes for lookup. Among those, only one will find the query. The false checking in nodes is the main time cost for locating an item in our distributed
system. Here we use theoretical analysis to find the false checking in nodes of three indexing methods.

3.3.1 Direct lookup method

For a baseline of the comparison, we first analyze the lookup procedure which uses pure checking in nodes without bloom filters. When a query arrives, the system checks all the nodes directly and returns the result of the query. For one query, the system checks \( s \) nodes; only one has the positive response. So the average false checking per query of the direct lookup algorithm is

\[
F_{DL}/Q = s - 1 \quad (10)
\]

3.3.2 PBA

The second baseline is Pure Bloom filter Array (PBA), in which bloom filters are loaded with all the items in the nodes. Therefore all queries will find at least one match in one of the bloom filters. The system first calculates the bloom filter of each node one by one and collects the result. Then it checks only the positive nodes and finds the result. The false checking time comes from the false positive rate of the bloom filters. In one node, \( Q/s \) queries are already loaded in its bloom filter because of the uniform item distribution between nodes. So the queries that can cause false positive rate are \( Q - Q/s \). The bloom filter’s false positive rate is Eq. (6) in which \( \beta = 1 \). The average false checking per query of PBA is

\[
F_{PBA}/Q = \frac{(Q - Q/s)s}{Q} f_P(\beta = 1) = (s - 1)f_P(\beta = 1) \quad (11)
\]

3.3.3 SBA

Now let us discuss the average false checking per query of Selective Bloom filter Array (SBA). In the bloom filter calculation process, queries that are loaded in one bloom filter are \( rQ/s \). Therefore the false checking caused by the bloom filter false rate is

\[
F_b/Q = \frac{(Q - rQ/s)s}{Q} f_P = (s - r)f_P \quad (12)
\]

It needs to mention that there exists a possibility that in a node, a query that is not loaded in its bloom filter receive a false positive response; but the queried item happens to be in the node outside the bloom filter content. That is to say, the bloom filter false positive rate miss-guides the lookup procedure and happens to find the item in the node. That bloom filter’s false positive judgment of items inside the node outside the bloom filter is not a false checking instance, because the lookup does find the item. It should be excluded from the false rate. The probability is

\[
F_d/Q = \frac{(1 - r)Q/s \cdot f_P \cdot s}{Q} = (1 - r)f_P \quad (13)
\]

After the bloom filter calculation and node lookup, \( rQ \) queries have already found their resident node. The queries misguided by the bloom filter but match in the node also find the answers.
The remaining \((1-r)Q - F_d\) queries will go through one-by-one node checking. The false checking rate during that process is

\[
F_c/Q = (1 - r)(s - 1)(1 - f_P)
\]  
(14)

The total false checking rate per query of selective bloom filter is

\[
F_{SBA}/Q = F_b/Q - F_d/Q + F_c/Q = (s - 1)(1 - r + rf_P)
\]  
(15)

Bring Eq. (9) into Eq. (15) we have

\[
F_{SBA}/Q = (s - 1)(f_P - \log N \beta + f_P \cdot \log N \beta)
\]  
(16)

We can see that false checking per query of direct lookup, PBA and SBA method have a multiplier \(s-1\) in common, which does not affect the comparison results between the three algorithms.

4 Experimental Evaluations

After the theoretical analysis, we use experiments to verify our deduction. We perform experiments to simulate the real online applications. Then we make comparison between the experimental result and the theoretical analysis. In all experiments we set node number \(s=10\), the total number of items \(N = 10^6\). The items are scattered randomly among \(s\) nodes, so each node has approximately \(n = 10^5\) 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

The entire corpus follows Zipf’s law with total number \(N = 10^6\). The queries reflect the real popularity situation of the corpus, so the queries follow the same distribution as the corpus. Actually, queries are formed as a sampling set of Zipf’s law with parameter \(N = 10^6\), not the query number \(10^5\).

4.2 Node construction

The item allocation in nodes follows uniform distribution. There are approximately \(n = 10^5\) items in each node. For each node, we build a bloom filter to index its popular items. The top \(\beta n\) items are indexed by the bloom filter. Theoretically, given that the queries follow Zipf’s distribution and the total item number is \(N = 10^6\), the probability that a queried item is indexed by bloom filter Array (hit ratio) is given in Eq. (9).

In the equation we find that larger \(\beta\) is, the higher the hit rate is and therefore the more likely that a query can find a positive answer in one of the bloom filters. However, later we will find that larger \(\beta\) also brings about higher false rate of the bloom filter and hence more false lookup.

In the actually processing of bloom filter construction, we first order all items in a node by their popularity rank and pick the first \(\beta n\) items. Then we insert those items into the node’s 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 \( m \) to be 524288 (about five times more than that of item number). The hash number of the bloom filters reaches the nearest integer of the optimal value in Eq. (7).

The theoretical analysis indicates that the false rate has a fast increase with \( \beta \). More items inserted, higher false rate. Note that when \( \beta \) reaches 1 (standard PBA), the false rate of bloom filters is very high (8.1%). That false rate will cause a lot of false checking in nodes by returning plenty of positive answers after bloom filter calculation.

### 4.3 Experiments with different \( \beta \)

In the experiment we want to analyze the impact of \( \beta \). First, we want to know whether our proposal can actually improve the bloom filter index performance. Second, we will see if the experimental result merges with our theoretical deduction. Third, we are going to find out what the optimal load factor is in the selective bloom filter. The range of \( \beta \) is \( \{0.1, 0.2, \ldots, 1\} \). Here \( \beta = 1 \) means that the bloom filters are fully loaded, which is equivalent to pure bloom filter array. For each \( \beta \), we build the bloom filter index in each node and lookup all \( Q \) queries until we find a match, as in Fig. 1. In the experiment, we count the false checking in nodes caused by bloom filters, the BF false judgment with correct position and the false checking in direct lookup process. Then we figure out the total false checking number when we finish serving all the queries.

When a query arrives, the system first calculates the bloom filter to find the match results. It then checks the nodes whose BF search result is positive. If it does not find the query in the “hot” list of the node, it causes an instance of false checking. The false checking in nodes caused by bloom filter false positive rate is given in Fig. 2.

We use experimental result to find the experimental \( F_b/Q \) and bring the system parameters into formula (12) to calculate the theoretical value. The figure shows that the two have a good agreement. The zero false rate in experimental result where \( \beta = 0.1 \) is caused by limited query number and extreme expected low false rate \( (10^{-10}) \). It can be seen that both rise with the increase of \( \beta \). In a bloom filter with fixed length, the more items loaded, the higher false rate and hence the higher false checking per query of the selective bloom filter array.

After the bloom filter checking process, the remaining queries that have not been found in nodes then go through a direct node checking process. The query is broadcast to all nodes and
each node looks it up in its own item list. When it does not find the query, it causes a false check. The false rate per query in direct lookup process is plotted in Fig. 3.

$F_c/Q$ is obtained the same process as $F_b/Q$. We can see that the experimental $F_c/Q$ merges well with the theoretical one. The decline of the curve shows that false checking in direct lookup decreases with the growth of $\beta$. The larger $\beta$, the more items indexed by the BF layer, and the less remaining items that can cause $F_c$. When $\beta=1$, $F_c/Q$ is zero because all items are stored in BF and will be found in the first BF lookup process.

Recall that there is a probability that an item misled by the BF happen to find a match in the item lookup process in the node. The corresponding $F_d/Q$ is given in Fig. 4.

We can see from the figure that with the increase of $\beta$, the probability first increases and then decreases. When $\beta=1$, $F_d/Q$ is zero because there is no items that are not indexed by the BFs, and hence no misled checking. The overall false checking per query of SBA is a linear combination of $F_b/Q$, $F_c/Q$, $F_d/Q$, as in Eq. (15). The experimental $F_{SBA}/Q$ is plotted below in Fig. 5.

In the figure, the theoretical false rate of SBA merges well with the experimental one, which is acceptable since all three components of the false rate have a proper fit. We can see that $F_d/Q$ is a very small portion of the overall false rate, always less than 1% of $F_{SBA}/Q$. The $F_{SBA}/Q$ is mainly composed of two parts: $F_b/Q$ and $F_c/Q$. With the growth of $\beta$, the first part increases while the second part decreases. The overall false rate obtains a minimum when $\beta=0.6$ under the combined influence of $F_b$ and $F_c$.

In order to find if our proposed method have a positive effect, we use the PBA approach and the direct lookup approach 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 of each method. The false checking number in each experiment is plotted in Fig. 6.

In the figure we can see that both SBA and PBA have an impressive improvement over direct lookup method considering false checking rate. That will reduce the overall system workload. Comparing PBA and SBA, we find that when the load factor $\beta$ is very small, the SBA is less effective than the PBA. When $\beta$ grows larger, the SBA performs better than the PBA and reaches its optimization when $\beta=0.6$. The two methods come near when $\beta$ approaches 1. When $\beta=1$, the SBA is fully loaded and is equivalent to PBA. The false rate of SBA has reduced by 36% than that of PBA at optimal $\beta$. That will have a direct impact on system performance. Under that
system parameter, when top sixty percent popular objects are loaded, the false checking rate is 0.43, which means that the system will check 1.43 nodes on average to find a queried item.

5 Conclusions and Future Work

In the paper we have used bloom filter as an index in the distributed storage system. We have presented a new bloom filter construction method, the selective bloom filter item insertion. The algorithm inserts “hot” items into the bloom filters to provide fast object lookup. We have also shown the lookup procedure for locating an object. Theoretical and experimental analyses have been conducted to verify the performance. The two results merged well in our paper. Further we found the optimal load ratio of the SBA index. It has been proved that our solution can achieve better performance than pure bloom filter array or direct lookup method when indexing space is limited.

The future work includes the improvement of the lookup procedure. The algorithm can be adjusted according to the actual system setting and user need, e.g. user access pattern, backup algorithm of the system, etc. The functional development of bloom filter algorithm will also continue in the future.

References