2015 International Conference on Computational Science and Computational Intelligence

Bloom Filter Tree for Fast Search of Tree-Structured Data

Mengyu Wang, Ying Zhu
Faculty of Business and Information Technology
University of Ontario Institute of Technology
Oshawa, Canada
mengyu.wang@uoit.ca, ying.zhu@uoit.ca

***Abstract*—We consider the problem of searching for a data element in a tree-structured data set (e.g., XML). We propose a method which is more efficient than tree traversal and which still retains all the important metadata information that would be lost in the naive method of linear list search. We compute a bloom filter for each interior node of the tree, essentially building a bloom filter tree to enhance the original data tree. Using the bloom filters, we can do fast search by pruning out entire subtrees from being searched. We present a theoretical analysis of the search complexity of selective placement of bloom filters in the tree, which leads to an optimal placement strategy. Our experiments verify the efficiency of our method.**

Keywords. data search, information systems, tree-structured data

I. Introduction

We consider data that are structured as trees, examples include XML trees and DOM trees. The leaf nodes in the data tree are the data elements; the interior nodes contain metadata as well as possibly data elements. That is, the set of data elements reside in the leaf nodes and possibly in the interior nodes, while the metadata reside in the interior nodes. We consider the problem of searching for a particular data element x in the set. If we simply take the leaf nodes and represent them as a list of elements, and then search this list, we would obtain a boolean answer of whether x is in the data set or not. We would not find out any information related to the tree structure of the data: where x is in the tree and any of the metadata associated with x. We would lose vital information encapsulated in the tree organization, and therefore lose its original intent and advantage. To retain the tree location and metadata for x, we must conduct the search on the tree. The naive way of doing a breadth-first or depth-first tree traversal would yield x's location(s) in the tree and hence all its associated metadata. The cost of such a traversal is that every node in the tree must be visited and every leaf node also compared with x.

We propose a more efficient method to find x and obtain its metadata and location(s) in the tree. At each interior node, we compute a bloom filter for the entire subtree rooted at this node. By checking the bloom filter, we can eliminate the possibility that x is in that subtree and altogether forego the traversal or search in that subtree. This way, every subtree not having x as one of its leaf nodes does not get traversed or searched at all, thus saving the unnecssary cost. Essentially we construct a tree of bloom filters; the tree has the same topology as the data tree, minus its leaf nodes. A bloom filter is simply a bit vector of m bits. Each computed bloom filter can easily be stored at the corresponding interior node in the original data tree. We construct the bloom filter tree from bottom up. First, we compute the bloom filters for the lowest-level (topologically speaking) interior nodes, i.e., those with leaf nodes as children. The bloom filters of upper interior nodes are obtained by taking the union of the bloom filters of all their children. Proceeding thus from bottom up, we build the entire bloom filter tree right up to the root.

The search for x, in contrast, begins at the top. We first check the bloom filter at the root to see if x exists in the entire tree. If it does, then we go down one level to check the bloom filters of the children of the root, to see if x is in any of these subtrees. If a check yields positive, then we know that most likely x is in that subtree and proceed to keep checking in that subtree top-down, in the same fashion. If a check yields negative, then we know with certainty that x is not in that subtree and can prune out that subtree. Throughout this top-down pruning search process, we store the path(s) that lead to data element x. For every instance of x found in the data tree, the path to it from the root is obtained and this path gives the location and metadata information. The answer to the query for x is therefore not just a boolean, but a set of paths (obviously empty set means no instance of x found).

As an illustrative example, suppose we have a collection of text data — these could be documents or forum posts or just pieces of text that users contribute to a common repository. This data is naturally organized into a tree, the children of the root node may represent the different content types, the next levels down may represent the timestamps, and the further down the breakdown may be by user ID, etc. The leaf nodes contain the actual text data. By building the bloom filters for this tree, we can efficiently search for particular text items and for instance, find out which user contributed them and when.

II. Related Work

The bloom filter data structure is space-efficient which simply represents a set of data for membership queries [2], and has been widely used in various applications such as overlay collaboration [5] and network intrusion detection [12], [22], [21][21]. Several variants [11], [17], [16], [4], [20] have been proposed for performance speedup and space efficiency. Retouched bloom filters trade off false positives[6] and false negatives [10]. The work of [13] simplifies implementation by

978-1-4673-9795-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CSCI.2015.30

using two hash functions. Algorithms in [7], [16] approximate representations of multisets.

A closely related work is that bloom filters were used as summaries for the set of content on a node to aid global collaboration in peer-to-peer networks, eg. [14],[18], [19], [1]. A query is a search for specific data generated from the node to determine whether the query is present in the bloom filter for the corresponding node, also aiming to route path queries based on node?s content. Web caches [11], [15] use bloom filters as the compact representations for the set of cached files. Each cache periodically sends the summary of itself to all the other members of the distributed cache. Receiving all the summaries, the cache node will have an overview of the set of cached files stored in the aggregated cache. Due to the potential for representing objects in memory, the bloom filter data structures have been used to summarize the contents of stream data, to support explicit state model checking of finite-state transition systems [3], [9],[8]. One typical application is approximate state machine, which monitors a flow's state in finite-state transition systems.

III. Bloom filters

We briefly present the background on bloom filters. Given a set S of N elements, and an integer x, the problem is to determine whether x is in S.

A bloom filter uses k hash functions, $h_i, i = 1, \ldots, k$, each with the same range $[0, \ldots, m-1]$. To add an element x to S, set all k bits of $h_i(x), i = 1, \ldots, k$ to 1 in the bit vector. Given x, if not all these k bits are 1, then it is known with certainty that $x \notin S$; otherwise, it is with high probability that $x \in S$ (if $x \notin S$, then this is a false positive). With a pre-specified set size N and false positive rate, one can find the required size of bit vector m. Even for a low false positive rate of 0.1, m is only approximately $10N$ bits. This is considered constant space because m does not depend on the data size of each element in S.

IV. Bloom filter tree

In this section, we describe the construction and the search of the bloom filter tree. We are given a data tree T and build co-existing bloom filter tree. The storage of the bloom filter tree is simply enhancing T by storing a bloom filter (bit vector) at each of its interior nodes. The assumptions of T are: (1) Every leaf node is a data element. (2) Every interior node contains metadata and possibly a data element. (3) The data set S of interest for search consists of data elements at the leaves and the interior nodes. (4) There are n data elements in T.

Given the number of elements n and a desired false positive probability f, we can find the number of bits m required for the bloom filter to represent the set of elements, using the equation [5]: $m = -(n \ln f)/(\ln 2)^2$. In the performance evaluation, we use the false positive probability of 0.1 throughout our experiments. Given m and n, the optimal number of hash functions k can be found by the equation [5]: $k = (m/n) \ln 2$. Optimality is in the sense of minimizing false positive probability.

We construct the bloom filter tree from bottom up. Each bloom filter generated has m bits and uses k hash functions. We begin with computing a bloom filter for each group of leaf nodes sharing the same parent, and storing this bloom filter at the parent node. After processing all the leaf nodes, the parent nodes of the leaves (i.e., the bottom-most interior nodes) each has a bloom filter. We then move one level higher up and build bloom filters for those nodes that have as children the nodes already assigned bloom filters. Consider such an interior node u, let $c_i, i = 1, \ldots, l$ be its l children with their respective bloom filters $b_i, i = 1, \ldots, l$. The set of data represented by u should be the union of all the data sets represented by $\{c_i\}_1^l$. A nice property of bloom filters is that the union of a collection of sets is represented by the bloom filter computed by the bitwise OR of the filters of each of these sets. That is, the bloom filter for u is simply: b_1 `OR` b_2 `OR`$\ldots$`OR` b_l, where `OR` is the bitwise OR operator. We compute bloom filters for each level of interior nodes, moving up the tree until one is computed and stored at the root.

When we search for a data element x, we begin at the root of the tree. We check if x is in the bloom filter at the interior node u, if yes, then we recursively search for x in each of the children of u; otherwise, we prune out the subtree rooted at u and do not search it any further. Every time we find x in a bloom filter, we record the interior node visited in the search result. The construction and search of the bloom filter tree are shown in pseudocode in Algorithm 1 and 2, respectively.

V. Optimal placement in a sparse bloom filter tree

We consider now the problem of using a *sparse* bloom filter tree instead of the full one in which every node in the data tree is indexed by a bloom filter. In order to reduce the amount of overhead associated with computing and maintaining a full bloom filter tree, we can choose to index only a subset of the nodes. The problem becomes: Given the number of bloom filters k that we wish to generate, how do we find an optimal placement for them in the tree? Optimality is defined in terms of search efficiency, or equivalently, by the total number of nodes scanned or accessed before the target item is found or deemed to not exist in the tree. Incidentally, our analysis also yields the exact reduction in search complexity achieved by adding b bloom filters to the tree.

To calculate the search complexity, we need the parameters: $n =$ total number of nodes in the tree; $q =$ probability of any given data item being searched for is in the tree (); $\alpha =$ number of children each interior node has.

We reason that we only need to consider the cases of $b = 0, 1, 2$ because the analysis process can be thought of as being *recursive*, with each of the subtrees rooted at level-1 nodes (children of the root) being treated recursively as a tree.

We begin with the cases of $b = 0, 1$, that is, the cases of using no bloom filter and using only one. For search complexity, we

Algorithm 1: Constructing the bloom filter tree

```
Data: data tree T, n, m, k
/* n,m,k are parameters for creating a bloom filter: capacity, number of bits, number
of hash functions, respectively                                                    */
Result: bloom filter u.bf for each interior node u
/* build bloom filter tree by calling recursive function buildBF(·) and passing the
root node to it                                                                    */
buildBF(root node of T, n, m, k) /* definition of the recursive function buildBF   */
Function buildBF (node u){
u.bf := create-bloom-filter(n,m,k)
if u's children are leaf nodes then
    for each child c of u do
        add data element c to u.bf
    end
end
else /* bitwise OR of bloom filters of all u's children, recursively computing them */
    u.bf := bitwiseOR_{c child of u} buildBF (c,n,m,k)
end
}
```

Algorithm 2: Search for element x in bloom filter tree

```
lookFor (root node of T, x, result) /* definition of recursive function lookFor   */
Function lookFor (node u, data element x, result){
if u's children are leaf nodes then
    if x is in u.bf then
        append u in result
        return
    end
    else
        return
    end
end
else
    if x is in u.bf then
        append u in result
        for c in u's children do
            lookFor (c,s,result)
        end
    end
end
}
```

use the expected number of nodes scanned in the search. For $b = 0$, whether the data item being searched for is in the tree or not, the expected number of nodes scanned is the same:

$$q \cdot \frac{n}{2} + (1 - q) \cdot n$$

For $b = 1$, we first calculate the search complexity of placing the bloom filter at the root node. If the data item is in the tree, the search complexity remains the same as above; but if it is not, then the bloom filter at the root would indicate so and eliminate the search altogether.

$$q \cdot \frac{n}{2} + (1 - q) \cdot 0 = q \cdot \frac{n}{2}$$

We observe that by adding just one bloom filter (at the root), the expected number of nodes scanned is reduced by $(1 - q) \cdot n$. In the uninformed case of $q = 1/2$, search is improved by scanning $n/2$ fewer nodes on average.

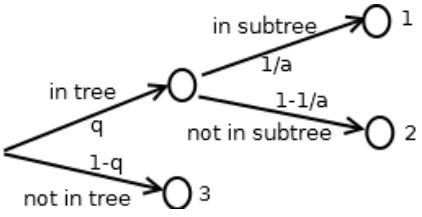

Figure 1. 1 bloom filter at a level-1 node.

Suppose instead, we place the bloom filter at a level-1 node (i.e., one of the children of the root). The decision tree in 1 shows the three different possible outcomes, which are at

the leaf nodes of this decision tree. For instance, if the data item is in the tree and in the subtree rooted at the level-1 node that has the bloom filter — outcome 1; this results in a search of only this subtree, the expected number of nodes scanned is hence $1/2$ of the number of nodes in the subtree, $\approx (1/2)n/\alpha$. If it is in the tree but not in the subtree of this level-1 node, then it must be in one of the other $(\alpha-1)$ subtrees of level-1 nodes — outcome 2. The edges are labeled by the probabilities of events. For outcome 1, the number of nodes scanned is limited to searching one subtree — the one rooted at the level-1 node with the bloom filter. For outcome 2, the bloom filter will indicate that the item is not in that subtree, so all the remaining subtrees will be searched until the item is found. Since the bloom filter is at a level-1 node, outcome 3 will require scanning every single node in all the subtrees but one, until it is concluded that the item is not in the tree. The expected number of nodes scanned is therefore given below.

$$q \cdot \frac{1}{\alpha} \cdot \frac{n}{2\alpha} + q \cdot \frac{\alpha-1}{\alpha} \cdot \frac{n(\alpha-1)}{2\alpha} + (1-q) \cdot \frac{n(\alpha-1)}{\alpha}$$
$$= q \cdot \frac{n(1+(\alpha-1)^2)}{2\alpha^2} + (1-q) \cdot \frac{n(\alpha-1)}{\alpha}$$

To compare with zero bloom filter, we subtract this from the other to obtain

$$q \cdot \Big[\frac{n}{2} - \frac{n(1+(\alpha-1)^2)}{2\alpha^2}\Big] + (1-q) \cdot \Big[n - \frac{n(\alpha-1)}{\alpha}\Big]$$
$$= q \cdot \frac{n(\alpha-1)}{\alpha^2} + (1-q) \cdot \frac{n}{\alpha}$$
$$\approx q \cdot \frac{n}{\alpha} + (1-q) \cdot \frac{n}{\alpha}$$

We compare selection of level-1 node with root node for placing the bloom filter, to obtain the difference:

$$-q \cdot \frac{n(\alpha-1)}{\alpha^2} + (1-q) \cdot \frac{n(\alpha-1)}{\alpha}$$

If this expression evaluates to a positive number, then placing the bloom filter at the root performs better than placing it at a level-1 node; otherwise, the level-1 node is the better choice. It is interesting to note that with different values of q, it may be one or the other. The root does not always give the better performance. For example, letting $q = 0.8$ and $\alpha = 3$, the above expression evaluates to a negative number, meaning placing the bloom filter at a level-1 node gives better performance than at the root. **We draw the conclusion: If it is known *a priori* that the search item is likely to be found in the tree, then it is better to place the bloom filter at a level-1 node, not at the root. Otherwise, it is optimal to place the bloom filter at the root. (Conclusion I)**

Generalizing this, we obtain the search complexity for placement at a level-k node. The probabilities on the rightmost two arrows in 1 would change to $1/\alpha^k$ and $1-1/\alpha^k$, respectively. Therefore we have the following:

$$q \cdot \frac{1}{\alpha^k} \cdot \frac{n}{2\alpha^k} + q \cdot \Big(1 - \frac{1}{\alpha^k}\Big) \cdot \Big(\frac{n}{2} - \frac{n}{2\alpha^k}\Big) + (1-q)\Big(n - \frac{n}{\alpha^k}\Big)$$
$$= q \cdot \frac{n((\alpha^k-1)^2+1)}{2\alpha^{2k}} + (1-q) \cdot \frac{n(\alpha^k-1)}{\alpha^k}$$
$$= q \cdot n\Big[\frac{(\alpha^k-1)^2}{2} - (\alpha^k-1)\Big] + \frac{n(\alpha^k-1)}{\alpha^k}$$

For increasing k, this quantity clearly increases since the $(\alpha^k - 1)$ term is easily dominated by the $(\alpha^k-1)^2/2$ and $(\alpha^k-1)/\alpha^k$ terms. In brief, this quantity achieves the minimum value at $k = 1$ over all values for $k >= 1$. **The conclusion is that bloom filter placement at a level-1 node is always better than at a level-k node for any $k > 1$. (Conclusion II)**

Next, we consider the case of $b = 2$ bloom filters to be placed in the tree. We compare the search complexity for (i) 1 at root and 1 at level-1, and (ii) 1 at root and 1 at level-2. Using the decision tree reasoning as above, we obtain the following.

$$(i): \quad q \cdot \frac{1}{2} \cdot \frac{n^2}{\alpha^2}\Big[1 + (\alpha+1)^2\Big]$$
$$(ii): \quad q \cdot \frac{1}{2} \cdot \frac{n^2}{\alpha^4}\Big[1 + (\alpha^2-1)^2\Big]$$

Subtracting them, (ii)−(i), gives the difference

$$n^2\Big[\frac{1+(\alpha^2-1)^2}{\alpha^4} - \frac{1+(\alpha-1)^2}{\alpha^2}\Big]$$
$$= n^2\Big[\frac{2(\alpha^3-2\alpha^2+1)}{\alpha^4}\Big]$$

Since $\alpha^3 - 2\alpha^2 + 1 > 0, \forall \alpha \geq 2$, placement (ii) always has a larger search complexity than (i), hence **the conclusion is that it is more optimal to place the first bloom filter at root and the second one at level-1 than to skip a level for the second one. (Conclusion III)**

Now we put our conclusions together. To be most generally applicable, we assume a q value that is not too lopsided. Conclusion I requires that we place a bloom filter at the root. Conclusion III indicates that the next bloom filter should be placed at a level-1 node for optimal search efficiency. Applying Conclusion II, the next bloom filter should be placed at another level-1 node, and this is applied until all but one of the level-1 nodes have bloom filters assigned (the last one does not need one, since the complement of all its siblings serve the same purpose). Then, we apply these *recursively* to each of the subtrees rooted at the level-1 nodes. Because the equivalent q value for each level-1 node (root of their own subtree) is known to be small (it is size of one subtree over size of all subtrees), Conclusion I dictates that we place bloom filters at the level-2 nodes (all except one). And so on and so forth, until we reach the quota of b bloom filters in total to be placed. Based on our analysis, this incurs the minimum search complexity and thus optimum search efficiency.

VI. Experiments

We implemented the bloom filter tree in Python to evaluate its performance. For the data tree, we randomly generated data elements for the leaf nodes and the interior nodes of the tree using parameters of tree height h, number of children for each interior node at levels less than h, and number of leaf nodes for each interior node at level h. Then using the generated data tree as input, we constructed the bloom filter tree as described in IV. To search for a given data element, we implemented the bloom filter tree search also as described in IV. For comparison, we

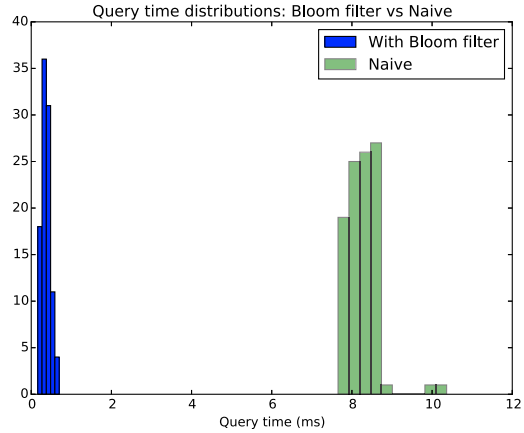

Figure 2. histogram of query time distribution

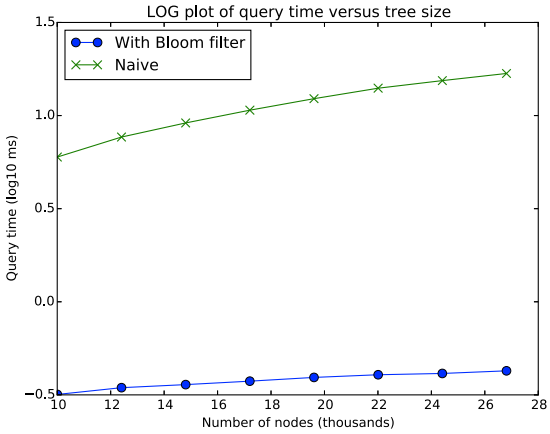

Figure 3. varying tree size versus query time

Figure 4. effect of tree height on query time

implemented the naive method of tree traversal for searching for a given data element.

We first compare the search time of using the bloom filter tree and of doing the naive tree traversal. We randomly select 1000 data elements from the leaf nodes of the data tree and run searches for them by using bloom filters and by tree traversal. The histograms of the search times for these 1000 searches for both methods are shown in Fig.2. As can be seen in the histograms, the search time for the bloom filter tree is consistently and substantially more efficient than the naive method. For this experiment, the tree size was around 20,000 nodes with height of 7.

In the second experiment, we varied the tree size from 10,000 to 30,000 and measured the average query time over 1000 queries, for both methods. Figure 3 shows the log plot of the query time. One may observe that with bloom filter index, the query time is improved by 1 to 2 magnitudes.

We further investigated the robustness of our method for trees of varying structures. We exprimented with shallow trees to deep trees, varying the height from 2 to 10. The log plot of query time is shown in Fig. 4. Not only is our method more efficient in query time, but also as the tree height increases, the gap grows even more rapidly. This further highlights that bloom filters brings in pruning the search space.

We also considered the scenario of selectively placing bloom filters in the tree. We investigated the performance when we change the proportion of nodes that have bloom filters. The proportion varies from sparsely indexed to densely indexed trees, numerically from 0 to 1. As we can see in Fig. 5, even with only 40% of the nodes indexed, our method outperforms the naive one; and as in the previous experiment, the improvement grows with the density increase.

In the experiments presented so far, we have consistently searched for data items known to be already in the tree. In real life, we also need to consider missed queries, namely queries that do not match any data in the tree. Given the bloom filter property, we have greater pruning power with missed queries, thus yielding better performance. In Fig. 6, we show the performance for workloads consisting of various proportions of missed queries. As expected, the higher the ratio of missed queries, the more efficient the search time our method exhibits.

VII. Conclusion

We have proposed a method to efficiently search for data items in data sets that are organized into hierarchical tree structures. Our method relies on assigning bloom filters to interior nodes of the tree. Using these bloom filters, our search through the tree can prune out subtrees, thereby greatly reducing search time. In order to reduce space requirement, we also explored the scenario of assigning bloom filters only to a fraction of the interior nodes. We further present a theoretical analysis of the selective placement of bloom filters to a subset of nodes and propose an optimal placement strategy. Our experiments

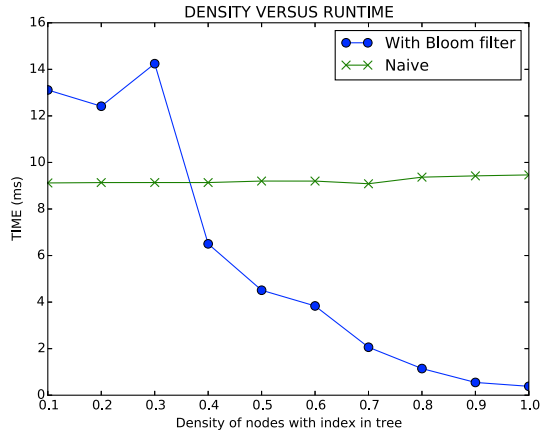

Figure 5. density of indexed nodes versus query time

Figure 6. effect of types of querying the work load on query time

show that our method is more efficient than the naive methods by orders of magnitude. They also show that placing bloom filters to only 40% nodes would be sufficient to attain search efficiency. Performance is consistent over varying tree sizes and heights.

REFERENCES

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[2] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrah, Sushil Singh, and George Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. *ACM SIGCOMM Computer Communication Review*, 36(4):315–326, 2006.

[4] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Algorithms–ESA 2006*, pages 684–695. Springer, 2006.

[5] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[6] Ken Christensen, Allen Roginsky, and Miguel Jimeno. A new analysis of the false positive rate of a bloom filter. *Information Processing Letters*, 110(21):944–949, 2010.

[7] Saar Cohen and Yossi Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 241–252. ACM, 2003.

[8] Peter C Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In *Formal Methods in Computer-Aided Design*, pages 367–381. Springer, 2004.

[9] Peter C Dillinger and Panagiotis Manolios. Fast and accurate bitstate verification for spin. In *Model Checking Software*, pages 57–75. Springer, 2004.

[10] Benoit Donnet, Bruno Baynat, and Timur Friedman. Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *Proceedings of the 2006 ACM CoNEXT conference*, page 13. ACM, 2006.

[11] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[12] Wu-chang Feng, Kang G Shin, Dilip D Kandlur, and Debanjan Saha. The blue active queue management algorithms. *IEEE/ACM Transactions on Networking (ToN)*, 10(4):513–528, 2002.

[13] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[14] Georgia Koloniari and Evaggelia Pitoura. Content-based routing of path queries in peer-to-peer systems. In *Advances in Database Technology-EDBT 2004*, pages 29–47. Springer, 2004.

[15] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[16] Abhishek Kumar, Jun Xu, and Jia Wang. Space-code bloom filter for efficient per-flow traffic measurement. *Selected Areas in Communications, IEEE Journal on*, 24(12):2327–2339, 2006.

[17] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

[18] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 21–40. Springer-Verlag New York, Inc., 2003.

[19] Sean C Rhea and John Kubiatowicz. Probabilistic location and routing. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1248–1257. IEEE, 2002.

[20] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. The variable-increment counting bloom filter. In *INFOCOM, 2012 Proceedings IEEE*, pages 1880–1888. IEEE, 2012.

[21] Alex C Snoeren, Craig Partridge, Luis A Sanchez, Christine E Jones, Fabrice Tchakountio, Beverly Schwartz, Stephen T Kent, and W Timothy Strayer. Single-packet ip traceback. *IEEE/ACM Transactions on Networking (ToN)*, 10(6):721–734, 2002.

[22] Qi George Zhao, Mitsunori Ogihara, Haixun Wang, and Jun Jim Xu. Finding global icebergs over distributed data sets. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 298–307. ACM, 2006.