BLOOM-BASED FILTERS FOR HIERARCHICAL DATA

Georgia Koloniari, Evaggelia Pitoura

29– 2002

Preprint, no 29 – 02 / 2002

Department of Computer Science
University of Ioannina
45110 Ioannina, Greece

Bloom-Based Filters for Hierarchical Data

Georgia Koloniari, Evaggelia Pitoura

Department of Computer Science
University of Ioannina, Greece
{kgeorgia, pitoura}@cs.uoi.gr

Abstract. In this paper, we present two novel hash-based indexing structures, based on Bloom filters, called breadth and depth Bloom filters, which in contrast to traditional hash based indexes, are able to represent hierarchical data and support path expression queries. We describe how these structures can be used for resource discovery in peer-to-peer networks. We have implemented both structures and our experiments show that they both outperform the traditional Bloom filters in discovering the appropriate resources.

1 Introduction

The decreasing cost of communications technologies and the increase of computational power in many network-enabled devices create a massive infrastructure composed of highly diverse interconnected entities. In such an environment, data, resources and services are fragmented and distributed all over the network. We view such a system as a peer-to-peer network of interconnected data resources. A problem central to such systems is how to locate the desired information in the multitude of available peers. A single query on a peer may need results from a large number of others, thus a mechanism is needed that will efficiently identify the peers that may contain data relevant to the query.

XML is rapidly emerging as the new standard for data representation and exchange on the Internet. Therefore, we will assume that the data sources in the peer-to-peer network store XML documents that we wish to index, query and finally retrieve. To retrieve XML documents several query languages have been proposed. A common feature of such languages is the ability to query the structure of the documents through path expressions.

There are many mechanisms for resource location in peer-to-peer systems. In this paper, we consider a distributed index approach, in which indices are maintained at each peer to assist in directing the query to the peers that may provide relevant data. Such indexes should be small and able to scale to a large number of peers and documents. Furthermore, since peers will join and leave the system dynamically, these indexes must support frequent updates.

Bloom filters can be used as indexes in such a context. Bloom filters are hash-based indexing structures designed to support membership queries. Traditional Bloom filters are unable to represent hierarchies, and thus evaluate path expressions. However, when querying XML documents, we are most likely to use a query

document, and finally queries that allow the * operator in a path. The last kind of queries specifies that two elements (tags) in a path may not be immediately succeeding one-another, but multiple levels may be between them.

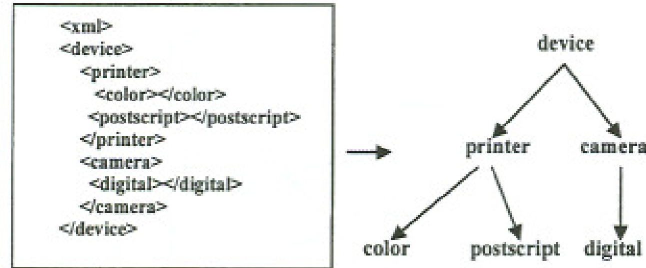

Fig. 1. An XML document and its tree

2.2 Bloom Filters

Bloom filters are compact data structures for probabilistic representation of a set in order to support membership queries (i.e., queries that ask: "Is element X in set Y?"). This compact representation is the payoff for allowing a small rate of *false positives* in membership queries; that is, queries might incorrectly recognize an element as member of the set. Since their introduction in [1], Bloom filters have seen various uses such as web cache sharing [2], query filtering and routing [3, 4], compact representation of a differential file [5] and free text searching [6].

Consider a set $A = \{a_1, a_2, \ldots, a_n\}$ of n elements (also called keys). The idea (illustrated in Figure 2) is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, $h_1, h_2, \ldots, h_k$, each with range $\{1,\ldots, m\}$. For each element $a \in A$, the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$ in v are set to 1. A particular bit might be set to 1 multiple times. Given a query for b, we check the bits at positions $h_1(b), h_2(b), \ldots, h_k(b)$. If any of them is 0, then certainly b is not in the set A. Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a "false positive". The parameters k and m should be chosen such that the probability of a false positive (and hence a false hit) is acceptable. Let BF be the Bloom filter for a set A, we denote the query for b, match(BF, b).

To support updates of the set A we maintain for each location l in the bit array a count $c(l)$ of the number of times that the bit is set to 1 (that is, the number of elements that hashed to l under any of the hash functions). All counts are initially 0. When a key a is inserted or deleted, the counts $c(h_1(a)), c(h_2(a)), \ldots, c(h_k(a))$ are incremented or decremented accordingly. When a count changes from 0 to 1, the

improve performance, we construct an additional Bloom filter denoted BBF_0. In this Bloom filter, we insert all attributes that appear in any node of the XML tree *T*.

For example, the BBF for the XML tree in Figure 1 is a set of four Bloom filters (Figure 3). In BBF_1 (i.e., the filter of the first level), we insert only the attribute device, in BBF_2, we insert attributes printer and camera, and finally in BBF_3, we insert attributes color, postscript and digital. In BBF_0, we insert the attributes that appear in any node of the tree.

Note that the BBF_is are not necessarily of the same size. In particular, since the number of nodes and thus keys that are inserted in each BBF_i ($i > 0$) increases at each level of the tree, we analogously increase the size of each BBF_i. Let $|BBF_i|$ denote the size of BBF_i. As a heuristic, we set: $|BBF_{i+1}| = d\ |BBF_i|$, $(0 < i < j)$, where d is the average out-degree of the nodes of the XML tree. For equal size BBF_is, BBF_0 is the logical OR of all BBF_is, $1 \leq i \leq j$.

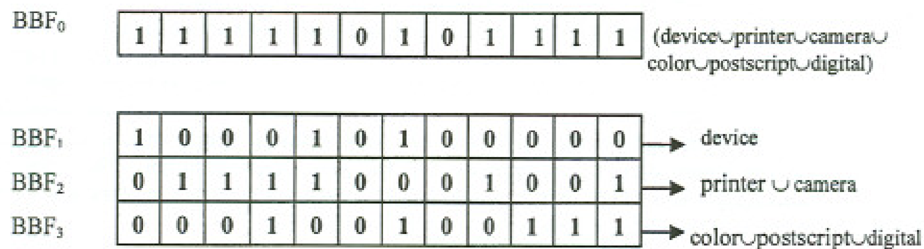

Fig. 3. The Breadth Bloom Filter (BBF) for the XML tree of Figure 1.

The look-up procedure that checks whether a BBF matches with a path query distinguishes between two kinds of path queries: path queries starting from the root level and partial path expressions. In both cases, first the algorithm checks whether all attributes in the path expression appear in BBF_0. Only if we have a match for all the attributes will the algorithm proceed to examine the structure of the path. For a root query: $//a_1/a_2/\ldots/a_p$ every row i from 1 to p of the filter is checked for the corresponding a_i. The algorithm is successful if we have a match for every attribute of the query. For a partial path expression, for every level i of the filter: the first attribute of the path is checked. If there is a match the next line is checked for the next attribute in the path and if we have a match this step continues until either we match the whole path or we have a miss. If we have a miss we return to the previous step for the row i + 1. For paths containing the * operator the algorithm skips as many levels of the filter as needed in order to match the next attribute in the query following the *. The algorithm is presented in detail in the Appendix (Figure 10).

Let p be the length of the path and d be the number of levels of the filter. (We exclude the Bloom on top). In the worst case, we check each level i, $i < p$, i times. That is $1 + 2 + \ldots + p - 1 = p \times (p - 1) / 2$. Then, for the remaining $d - p$ levels, we will have at most p accesses at each level. So the overall cost is: $p \times (d - p) + p \times (p - 1) / 2 = O(dp)$.

insertion and no levels can be skipped. Say we have the path A/*/B. The path is split in two paths where the * appears, (paths A and B) and both paths are checked with the above algorithm, to see whether there is a match. If we have a match for both paths the algorithm succeeds, else we have a miss. The algorithm is presented in detail in the Appendix (Figure 11).

Consider a query of length p. Let us suppose that p is smaller than the number of the levels of the filter, so that the query can be satisfied. The algorithm proceeds as follows: firstly p sub-paths of length 1 are checked, then $p - 1$ sub-paths of length 2 are checked and this goes on until we reach length p where we have 1 path. Thus the complexity of the look-up procedure is $p + p - 1 + p - 2 + \ldots + 1 = O(p^2)$. The complexity remains the same even if we have the * operator in the query.

Besides the false positive induced by the traditional Bloom filters, we have two additional types of false positives. The first is with path queries that start from the root. With the current description, we cannot distinguish such paths from partial ones. However, such false positives can be easily treated if we treat paths that start from the root differently by annotating such paths with a special symbol, e.g., path device\printer could be represented as \\device\printer. The other type of false positive refers to queries that contain the * operator. In particular, consider the following paths: a/b/c/d/e and a/k/f/m/n that are inserted in a depth Bloom (along with their sub-paths). For the query: a/b/*/m/n, our algorithm splits the path to: a/b and m/n. Both of these parts belong to the filter so the algorithm would indicate a match although the desired path does not exist.

Regarding the size of the filters, as opposed to BBF, all DBF_is have the same size. This is because the number of paths of different lengths is of the same order.

4 Distribution

We consider a peer-to-peer system where each peer (site) stores a set of XML documents. A client requests specific XML documents though a path expression query. Such queries may originate at any peer of the system. Bloom filters are used to locate the peers that may contain documents that match the query.

In particular, each peer maintains a multi-level Bloom filter (breadth or depth) for the documents stored locally at the peer. In addition, it also maintains a multi-level summary (depth or breadth) Bloom filter for a set of its neighboring peers. This summarized filter holds information about other peers in the system and thus facilitates the routing of a query only to peers that may contain relevant information. When a query reaches a peer, the peer first checks its local Bloom filter and then uses the summary filter to direct the query to other peers in the system that potentially hold relevant information.

The summarized Bloom filter of a set of Bloom filters is calculated by taking the bitwise OR for each level of the filters separately. Specifically, the summarized Bloom filter, Sum_BBF of two breadth Bloom filters BBF^k and BBF^m with i levels is a breadth Bloom filter Sum_BBF = {Sum_BBF_0, Sum_BBF_1, … Sum_BBF_i} with i levels such that: $Sum_BBF_j = BBF^k_j$ *BOR* BBF^m_j, $0 \leq j \leq i$ and *BOR* stands for bitwise OR. Similarly, we define the summary Bloom filter for depth Bloom filters.

the query. This procedure continues until the peers at the leaf nodes of the hierarchy are reached.

Updates that may occur in local data sources are propagated firstly to the associated local filter of the peer and then to the peer's parent. The parent updates its merged filter and propagates the update to its own parent. The update procedure continues until the root peer is reached. The root peer sends the update to all other root peers which in turn update the merged filter of the associated peer. Note that we need to propagate only the BBF_is (DBF_is) that are updated not the whole BBF (DBF).

4.2 Horizons

Apart from this hierarchical structure, we consider an alternative organization in which the participating peers are considered as forming an overlay network between them. Each peer has a set of neighbors, chosen from the participating peers closest to it in network latency. Each peer associates with each of its neighbors a probability of finding each document through this neighbor.

To distribute multi-level Blooms across this organization of peers we use the notion of *horizons*. Knowledge about all peers in the system may seem limiting in terms of scalability. By introducing the notion of horizon, a peer bounds the number of neighbors whose data it summarizes. In this organization, peers form an undirected graph and each peer additionally to its local index, holds information about peers at distance d from itself, where d is the radius of the horizon. In particular, every peer stores one filter, called *merged filter*, for each one of its neighbor links. Each such merged filter summarizes information for all peers at distance d through any path starting with this neighbor link (Figure 6).

In particular, every peer, when it enters the system, sends its local index together with a hop counter set to d to all of its neighbors. Every peer that receives a filter merges it with the other filters it has received through the same neighbor link (same neighbors). Then it decreases the hop counter by 1 and if the counter is not 0, it propagates the merged filter it has received to all of its own neighbors, except the one it was sent from. This way, the summary reaches all nodes at distance d, and thus every peer has now information about all other peers at distance d from itself. Every peer that receives the summary also sends back to the new peer its own local summary in order for it to construct its neighbors' summaries.

When a query is posed, and it cannot be satisfied locally, each filter for all the links of the node is checked and the query is propagated only through links that their filter gave a match. This time, however, it is not the immediate neighbor who is likely to possess the data the query wants, but one of its neighbors. The next neighbor is determined as before, by examining the filters of the current node.

If a query were to reach a server d hops from its source due to a false positive; there is no incentive to forward it further. This problem can be overcome either by backtracking or by the use of an exhaustive algorithm that searches the graph. Also to prevent a query from being caught in a cycle in the graph every query holds a list with the nodes it has already visited.

Updates are propagated the same way a summary of a node is propagated when it enters the system, so as to inform all nodes at distance d about the change in their

Bloom, we have also implemented a simple Bloom filter (SBF) that just hashes all attributes (similar to BBD_0 and DBF_0). For the hash functions, we have used the MD5 algorithm, which has known attractive characteristics. For the generation of the XML documents, we used the Niagara generator [13]. We performed our experiments on a Linux PC with 1000MHz Pentium IV processor and 256MB of RAM.

For the generation of the queries we have used 90% tags that appear in the generated XML documents and 10% random ones, and produced arbitrary path expressions. The name tags generated by the Niagara generator are quite similar and of small length and this deteriorates the performance of Bloom filters as it is more possible that they return similar values while they are hashed. All the path expressions were partial match queries and the possibility of having the containment operator at each query was set to 0.05.

Our metric is the percentage of false positives in the three structures (simple, breadth and depth Blooms). In the first experiment we vary the size of the filter to determine how this affects the filter's performance. In the three other experiments we keep the size of the Bloom filter fixed and vary the other parameters. Table 1 summarizes the parameters of interest.

Parameter	Default Value	Range
# of XML documents	200	-
total size of filter	992	80-2000
# of hash functions	4	
# of queries	200	
possibility of * per query	0.05	
# of elements per document	500	200-2500
# of levels per document	4 / 6	2-6
length of query	3	2-6

Table 1. Table of Parameters

Experiment 1: *Influence of filter size.*
The first experiment examines the influence of the size of the filter with respct to its performace, that is, the performance of false positives it presents. The document's structure is fixed with 500 elements and 4 levels of depth. The queries are of length 3. The range of the filter's size was from 80 to 2500 bits. Figure 7 illustrates the results.

Simple Bloom filters fail to recognize most of the false positives even when the size of the filter increases significantly to 2000 bits and its percentage does not fall under 70%. In contrast, both Breadth and Depth Blooms ouperfrom it, while requiring much less space. Breadth Bloom have at most 5% false positives even for the small space budget of 80 bits. This is because even if one row of the filter is sparse, that is enough for most of the irrelevant queries to be rejected. Depth Blooms require more space in order to achieve the same performance. For a size of less than 300 bits we have more than 40% false positives but with a size of more than 400 bits the performance improves significantly and it outperfroms even Breadth Blooms for a size more than 992 bits.

postives at about 3%. We have to note here, that as the number of the elements increase the percent of repetion in the tag names produced by the XML generator increases significantly. This is the main reason for the constant percentage of false positives.

Experiment 3: *Influence of the number of levels.*
In this experiment, we compare the three approaches with respect to the depth of the documents. The size of the filter is fixed to 992 bits, and the documents have 500 elements. The queries are of length 3, except for the documents with 2 levels where we conduct the experiment with queries of length 2.

Simple Blooms have the worst overall performance, which is independent of the document's structure (depth). Depth Blooms perform better for documents with few levels, less than 4 while their performance deteriorates when the depth increases. This is because, as the rows of the filter increase fewer bits are allocated to each row, while the number of paths to be inserted increases and the filter becomes overloaded. In contrast, Breadth Blooms perform very well even for 6 levels. While the number of bits allocated to every row decreases in Breadth Blooms as well, the number of elements inserted becomes also smaller, in contrast with Depth Blooms where it becomes larger. The increase in the depth results in a more sparse distribution of the same number of elements in more levels of the document.

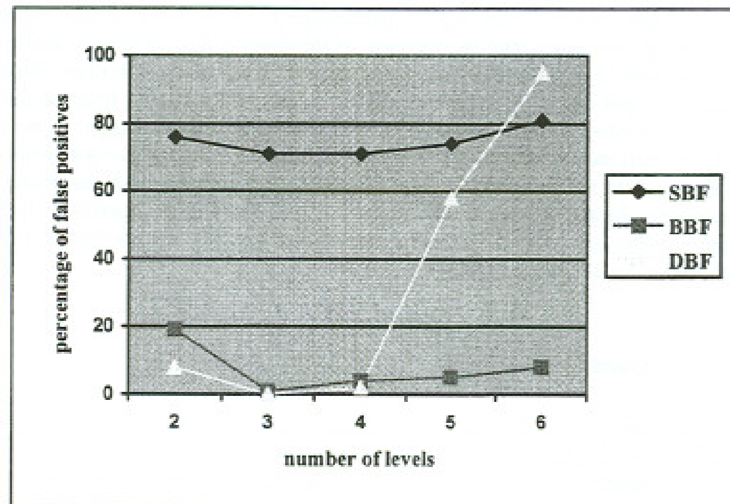

Fig. 8. Experiment 3: number of levels

Experiment 4: *Influence of the length of the queries.*
The parameter examined in this last experiment, is the length of the queries. The structure of the document is fixed, with 6 levels and 500 elements. The size of the filter is also fixed to 992 bits. The queries range from length of 2 to 6.

labeled, directed graph structure S(G), where: (1) each node u in the structure corresponds to a subset of data nodes in a partitioning of the graph (*extent* of u) that have the *same label* and, (2) an edge (u, v) in the graph is represented in the structure as an edge between the nodes whose extents contain the two endpoints u and v. To enable selectivity estimates for complex path expressions, each node u of S(G) only captures summary information about G in the form of a *count* that records the number of elements in G that map to u. XSKETCH synopses are specific instantiations of this generic graph-synopsis model that record some additional edge-label information to capture localized *stability* conditions across synopsis nodes. Emphasis is given in the evaluation of complex path expressions and there is no mention on the way the updates are handled.

APEX [9] is an adaptive path index for XML data. APEX does not keep all paths starting from the root and utilizes frequently used paths to improve the query performance. APEX also has a nice property that it can be updated incrementally according to the changes of query workloads. In APEX, frequently used path expressions in query workloads are taken into account using the sequential pattern mining technique so that the cost of query processing can be improved significantly. The path tree [11] is a tree, which represents the structure of an XML document. Every node of the tree corresponds to a path in the XML document and has a child for every distinct tag name of the elements of that path. For every sequence of tag names in the XML document there is only one path in the tree. Apart from the name of the tag of the elements in every node the number of the elements, the node's frequency, is also stored. For the estimation of the selectivity of a path, the path tree is traversed and searched for all the nodes with a label same as the first label of the query. From every such node the tree is traversed and the query's labels are matched with those of the nodes. The result is a set of nodes which correspond to the path of the query. The selectivity of the query is the total frequency of these nodes. This tree however, may exceed the given main memory space, therefore a path tree summary is computed, by deleting the nodes with the lowest frequency in the tree. We add *-nodes in the tree to store as much information as possible for the deleted nodes.

All these structures are as we mentioned centralized and there is no intuitive way for their use in a distributed environment. Although they support the valuation of complex path expression they cannot be employed in distributed environments.

In peer-to-peer systems, there were many methods developed in order to find the peers that contain data relevant to a query posed at some peer. These methods construct indexes that store summaries of other nodes information and additionally provide routing protocols to propagate the query to the relevant sites. In this line of research, emphasis is given to the distribution of the summaries across the peers' network. However these structures answer simple queries that consist of combinations of attribute-value pairs and do not address the evaluation of path expressions.

Perhaps the resourced discovery protocol most related to our approach is the on in [12]. SDS servers are responsible for routing the queries a client poses. They are organized into a hierarchical organization that is modified according the workload of the servers in order to achieve a load-balance among all the servers. Each server stores a summary of its children in the hierarchy, with the form of a Bloom filter constructed by the union of its children summaries. Services are described in XML documents, and their summary is constructed by computing subset hashes of the

4. Hodes, T.D., Czerwinski, S.E., Zhao, B.Y., Joseph, A.D. and Katz, R.H. Architecture for Secure Wide-Area Service Discovery. Wireless Networks.
5. Mullin, J.K A second look at Bloom Filters. Communications of the ACM, 26 (8). 570-571.
6. Ramakrishna, M.V. Practical performance of Bloom Filters and parallel free-text searching. Communications of the ACM, 32 (10). 1237-1239.
7. Brian Cooper, Moshe Shadmon. The Index Fabric: Technical Overview. RightOrder Inc 2001.
8. Brian F. Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, Moshe Shadmon. Fast Index for Semistructured Data. In Proc. of VLDB 2001.
9. Chin-Wan Chung, Jun-Ki Min, Kyuseok Shim. APEX: An Adaptive Path Index for XML Data. ACM SIGMOD '2002.
10. Neoklis Polyzotis, Minos Garofalakis. Structure and Value Synopses for XML Data Graphs. In Procs. of VLDB 2002.
11. Ashraf Aboulnaga, Alaa R. Alameldeen and Jeffrey F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. VLDB 2001.
12. Hodes, T.D., Czerwinski, S.E., Zhao, B.Y., Joseph, A.D. and Katz, R.H. Architecture for a Secure Service Discovery Service. Mobicom '99.
13. http://www.cs.wisc.edu/niagara

```
    Lookup(DepthBloom Filter DBF, path expression path)
      DBF= {DBF0, DBF1, DBF2, ... DBFi-1}
      Path = a1/a2/.../ap

      /* check for all attributes of the path a1, ..., ap in the BBF0 */
      /* the * is ignored when found in the path */
1.    for i = 1 to p
2.        if ai ≠ *
3.            if no match(DBF0, ai) return(NO MATCH)
4.    p1 = - 1
5.    for k = 1 to p  /*traverse the path and check for '*' to split the path */
6.        if ak = *
7.            n = p1 + 2
8.            p1 = k - 1
9.            for i = n  to p1      /*check all the subpaths of the path of length k-1 */
10.               for j = 1 to p1 - 1   /* until the '*'  */
11.                   if i + j ≤ p1
                          /*if any of the sub-paths do not exist reurn failure*/
12.                       if no match(DBFj + 1, (ai / ai+1 /.../ ai+j)) return(NO MATCH)
13.       else if k = p               /*if there is no path, or for the last part of the path*/
14.               for i = p1 + 2  to p  /*after the last * we repeat the above procedure*/
15.                   for j = 1 to p - 1
16.                       if i + j ≤ p
17.                           if no match(DBFj + 1, (ai / ai+1 /.../ ai+j)) return(NO MATCH)
18. return(MATCH)    /* if statement 18 is reached we have a match*/
```

Fig 11. Depth Bloom Filter Lookup procedure