Bloom-Based Filters for Hierarchical Data

G. KOLONIARI
University of Ioannina, Greece

E. PITOURA
University of Ioannina, Greece.

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 summarize hierarchical data and support regular 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.

Keywords

XML, Bloom Filters, Peer-to-Peer Systems, Indexing, Signature Files

1 Introduction

Peer-to-peer (P2P) systems have become very popular as a way to effectively share huge, massively distributed data collections. Since XML has evolved as the new standard for data representation and exchange on the Internet, we consider the case in which each peer stores and wishes to share XML documents or XML-based descriptions of its provided services. Such documents must be efficiently indexed, queried and retrieved. XML query languages share the ability to query the structure of the data through path expressions.

A single query on a peer may need results from a large number of others, thus we need a mechanism that finds peers that contain relevant data efficiently. In this paper, we consider a distributed index, in which each peer stores indices for routing a query in the system. Such indices should be small and scalable to a large number of peers and data. Furthermore, since peers will join and leave the system at will, these indices must support frequent updates efficiently.

Bloom filters [1] can be used as indices in such a context. They are hash-based indexing structures designed to support membership queries. When querying XML data, we wish besides their content to exploit their structure as well. However, Bloom filters are unable to represent hierarchies, and thus support the

evaluation of regular path expressions. To this end, we introduce two novel data structures, Breadth and Depth Bloom filters, which are multi-level structures that support efficient processing of regular path expressions including partial path and containment queries. We also consider two approaches for the distribution of the filters, one based on a hierarchical structure and one based on horizons. We have implemented the proposed data structures, and our experiments show that they both outperform simple Bloom filters in discovering the appropriate resources.

2 Preliminaries

2.1 XML Service Description and Querying

We assume a P2P system where peers store XML documents or XML-based descriptions of services. An XML document comprises a hierarchically nested structure of elements that can contain other elements, character data and attributes. Thus, XML allows the encoding of arbitrary structures of hierarchical named values. This flexibility allows peers to create descriptions that are tailored to their services. In our data model, an XML document is represented by a tree. Fig. 1 depicts an XML service description for a printer and a camera provided by a peer and the corresponding XML tree.

Definition 1 (XML tree): *An XML tree is an unordered labeled tree that represents an XML document. Tree nodes correspond to document elements while edges represent direct element-subelement relationships.*

We distinguish between two main types of queries: membership and path queries. *Membership queries* consist of logical expressions, conjunctions, disjunctions and negations of attribute-value pairs and test whether a pair exists in a description. *Path queries* refer to the structure of the XML document. These queries are represented by regular path expressions expressed in an XPath-like query language. In this paper, we focus on the efficient evaluation of path queries represented as regular path expressions that consist of label paths.

Definition 2 (label path): *A label path of an XML tree is a sequence of one or more slash-separated labels, $l_1/l_2/\ldots/l_n$, such that we can traverse a path of n nodes ($n_1 \ldots n_n$), where node n_i has label l_i and the type of node is element.*

We address the processing of queries that represent a path starting from the root element of the XML document, queries that represent only partial paths that can start from any point in the document, and queries that contain the containment operator (*) indicating that two elements in a path may not be immediately succeeding one another, but multiple levels may exist between them.

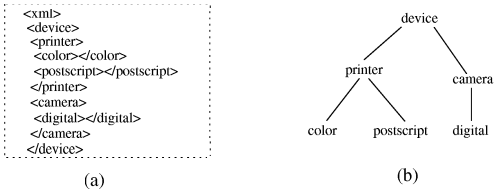

Figure 1: Example of (a) an XML document and (b) the corresponding tree

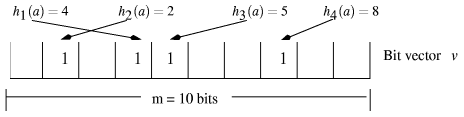

Figure 2: A (simple) Bloom filter with $k = 4$ hash functions

2.2 Bloom Filters

Bloom filters are compact data structures for probabilistic representation of a set that support membership queries ("Is element a in set A?"). Since their introduction [1], Bloom filters have seen many uses such as web cache sharing [2] and query filtering and routing [3, 4]. Consider a set $A = \{a_1, a_2,..., a_n\}$ of n elements. The idea 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, …, h_k, each with range 1 to m. For each element $a \in A$, the bits at positions $h_1(a)$, $h_2(a)$, …, $h_k(a)$ in v are set to 1 (Fig. 2). A particular bit may be set to 1 many times. Given a query for b, the bits at positions $h_1(b)$, $h_2(b)$, …, $h_k(b)$ are checked. If any of them is 0, then certainly $b \notin A$. 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" and it is the tradeoff for Bloom filters' compactness. The parameters k and m should be chosen such that the probability of a false positive is acceptable. To support updates of the set A we maintain for each location i in the bit vector a counter $c(i)$ of the number of times that the bit is set to 1 (the number of elements that hashed to i under any of the hash functions).

Bloom filters are appropriate as an index structure for resource discovery in terms of scalability, extensibility and distribution. However, they do not support path queries as they have no means for representing hierarchies. To this end, we introduce multi-level Bloom filters.

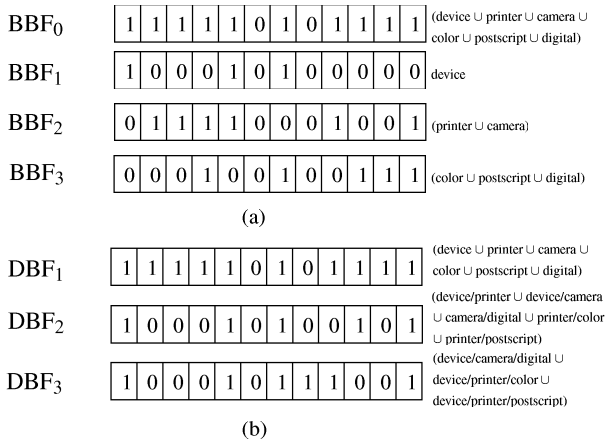

Figure 3: The multi-level Bloom filters for the XML tree of Fig. 1: (a) the Breadth Bloom filter and (b) the Depth Bloom filter

3 Multi-level Bloom Filters

We introduce two new data structures based on Bloom filters that aim at supporting regular path expressions. They are based on two alternative ways of hashing XML trees.

3.1 Breadth and Depth Bloom Filters

Let T be an XML tree with j levels and n elements and let the level of the root be level 1. The *Breadth Bloom Filter* (BBF) for an XML tree T with j levels is a set of simple Bloom filters $\{\text{BBF}_1, \text{BBF}_2, \ldots \text{BBF}_i\}$, $i \leq j$. There is one simple Bloom filter, denoted BBF_i, for each level i of the tree. In each BBF_i, we insert the elements of all nodes at level i. To improve performance and decrease the false positive probability in the case of $i < j$, we may construct an additional Bloom filter denoted BBF_0, where we insert all elements that appear in any node of the tree. For example, the BBF for the XML tree in Fig. 1 is a set of 4 simple Bloom filters (Fig. 3(a)). 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 s_i denote the size of BBF_i. As a heuristic, when we have no knowledge for the distribution of the elements at the levels of the tree, we set: $s_{i+1} = d\ s_i$, ($i < j$), where d is the average degree of the nodes, the elements inserted in each level i are d^{i-1} (Table 1). For equal size BBF_is, BBF_0 is the logical OR of all BBF_is, $1 \leq i \leq j$.

Depth Bloom filters provide an alternative way to summarize XML trees. We use different Bloom filters to hash paths of different lengths. The *Depth Bloom Filter* (DBF) for an XML tree T with j levels is a set of simple Bloom filters $\{\text{DBF}_1, \text{DBF}_2, \text{DBF}_3, \ldots, \text{DBF}_i\}$, $i \leq j$. There is one Bloom filter, denoted DBF_i, for each path of the tree with length *i-1*, (i.e., a path of i nodes), where we insert all paths of length *i-1*. For example, the DBF for the XML tree in Fig. 1 is a set of 3 simple Bloom filters (Fig. 3(b)). Note that we insert paths as a whole, we do not hash each element of the path separately; instead, we hash their concatenation. We use a different notation for paths starting from the root. This is not shown in Fig. 3(b) for ease of presentation.

Regarding the size of the filters, as opposed to BBF, all DBF_is have the same size, since the number of paths of different lengths is of the same order. The number of elements inserted in each level i of the filter is $n_i = 2d^i + \Sigma_{j=i+1}{}^{l} d^j$ for a tree with maximum degree d and j levels. The size is then $s_i = s/j$, where s is the given space available for the filter (Table 1).

3.2 Search Algorithms

BBF Search. The lookup procedure, that checks if a BBF matches a query, distinguishes between path queries starting from the root and partial path queries. In both cases, first we check whether all elements in the query appear in BBF_0 (if BBF_0 exists). Only if we have a match for all, we proceed in examining the structure of the path. For a root query: $a_1/a_2/\ldots/a_p$ every level i from 1 to p of the filter is checked for the corresponding a_i. The algorithm succeeds if we have a hit for all elements. For a partial path query, for every level i of the filter, the first element of the path is checked. If there is a hit, the next level is checked for the next element and the procedure continues until either the whole path is matched or there is a miss. If there is a miss, the procedure repeats for level i+1. For paths with the containment operator *, the path is split at the *, and the subpaths are processed. All matches are stored and compared to determine whether there is a match for the whole path. The algorithm is presented in detail in Fig. 4(a).

Let p be the length of the path and d be the number of levels of the filter. (We exclude BBF_0). In the worst case, we check d-p+1 levels for each path, since the path can start only until that level. The check at each level consists of at most p checks, one for each element. So the total complexity is $p(d\text{-}p+1)=O(dp)$. When the path contains the * operator, it is split into two subpaths that are processed independently with complexity $O(dp_1)+O(dp_2)<O(dp)$. The complexity for the comparison is $O(p^2)$, since we have at most $(p+1)/2$ *. For a path that starts from the root the complexity is $O(p)$.

DBF Search. The lookup procedure, that checks whether a DBF matches with a path query, first checks whether all elements in the path expression appear in DBF_1. If this is the case, we continue treating both root and partial paths queries the same. For a query of length p, every subpath of the query from length 2 to p is

Table 1: Bloom filter properties

	Number of Elements	Filter Size	False Positive Probability
SBF	n	s	$P = (1 - e^{-kn/m})^k$
BBF	$n_i = d^{i-1}$	$s_i = md^{i-1}$	$P_i = (1 - e^{-k/m})^k$
DBF	$n_i = 2d^i + \Sigma_{j=i+1}{}^l d^j$	$s_i = s/l$	$P_i = (1 - e^{-kl(2d^i + d^{i+1} + \cdots + d^l)/s})$

checked at the corresponding level. If any of the subpaths does not exist then the algorithm returns a miss. For paths that include the containment operator *, the path is split at the * and the resulting subpaths are checked. If we have a match for all subpaths the algorithm succeeds, else we have a miss. The algorithm is presented in detail in Fig. 4(b).

Consider a query of length p. Let p be smaller than the number of the filter's levels. Firstly p subpaths of length 1 are checked, then p-1 subpaths of length 2 are checked and so on until we reach length p where we have 1 path. Thus the complexity of the lookup procedure is p+p-1+p-2+...+1=$p(p+1)/2$=$O(p^2)$. This is the worst case complexity as the algorithm exits if we have a miss at any step. The complexity remains the same with * operators in the query. Consider a query with one *, the query is split into two subpaths of length p_1 and p_2 that are processed independently, so we have O(p_1^2)+O(p_2^2)<O(p^2).

3.3 False Positives

The probability of false positives depends on the number k of hash functions we use, the number n of elements we index, and the size m of the Bloom filter. The formula that gives this probability for Simple Bloom filters is [1]: $P = (1 - e^{-kn/m})^k$. We can use this formula to compute the false positive probability for a single element lookup in a level i of a multi-level Bloom filter, by substituting n and s with number of elements in the ith level of the filter (n_i) and its corresponding size (s_i) respectively. Table 1 summarizes our results for BBFs, DBFs and Simple Bloom filters (SBFs).

Using BBFs, a new kind of false positive appears. Consider the tree of Fig. 1 and the partial path query: camera/color. We have a match for camera at BBF_2 and for color at BBF_3; thus we falsely deduce that the path exists. The probability for such a false positive is strongly dependent on the degree of the tree. For DBFs we have a type of false positive that refers to queries that contain the * operator. Consider the paths: a/b/c/d/ and m/n. For the query: a/b/*/m/n, we split it to: a/b and m/n. Both of these paths belong to the filter so the filter would indicate a false match. Due to space limitations, we omit the analysis of the false positives probability which can be found in [5].

Lookup(Breadth Bloom Filter *BBF*, path expression *path*)
BBF = BBF_0, BBF_1, BBF_2, ..., BBF_i
Path = $a_1/a_2/\ldots/a_p$
1.for $i = 1$ to p /*check for all attributes of the path*/
2. if $a_i \neq *$ /*$a_1, \ldots, a_p$ in the BBF_0 the * is ignored*/
3. if no match(BBF_0, a_i) return(NO MATCH)
4. $p_1 = 1$
5. for $k = 1$ to p /*traverse the path and check for * to split the path */
6. if $a_k = *$
7. $n = p_1 + 2$
8. $p_1 = k - 1$
9. subpath(p_1)
10. if case(a) and no match subpath return NO MATCH
11. for all stored matches:
12. for $i = 1$ to number of paths
13. if all start_point(match i)>end_point(match $i + 1$)
14. return(MATCH)
15. return (NO MATCH)

Sub_Match(start point of subpath)
$p = p_1$
Case (a) path is a root path expression
1. for $j = 1, i = 1 to d, p$
2. if no match(BBF_j, a_i) return(NO MATCH)
/*check if the query is longer than the remaining levels*/
3. if $j + (p - i) < d$ return(NO MATCH)
4. store MATCH /*if 4 is reached we had a match for the whole path*/
Case (b) path is a partial path expression
1. for $j = 1$ to d /*for every level of the filter check */
2. if $j + p > d$ return /*if the query is longer than the remaining levels*/
3. for $i = 1 to p$ /*for every attribute beginning from the start of the query check by starting from level j*/
4. if $j + (p - i) > d$ return
5. if no match(BBF_j,a_i)
6. goto 1
7. else
8. $j = j + 1$
9. store MATCH /*if 9 is reached we had a match for the whole path*/

store (MATCH) /*stores the start and the end */
/* point of a match at two arrays*/
/*start_point and end_point*/

(a)

Lookup(DepthBloom Filter *DBF*, path expression *path*)
DBF= DBF_1, DBF_2, DBF_3, ..., DBF_i
Path = $a_1/a_2/\ldots/a_p$
/*check for all attributes of the path $a_1, \ldots, a_p$ in the DBF_1 the * is ignored when found in the path */
1. for $i = 1$ to p
2. if $a_i \neq *$
3. if no match(DBF_1, a_i) return(NO MATCH)
4. $p_1 = -1$
5. for $k = 1$ to p /*traverse the path and check for '*' to split the path */
6. if $a_k = *$
7. $n = p_1 + 2$
8. $p_1 = k - 1$
9. for $i = n$ to p_1 /*check all the subpaths of the path of length $k - 1$ */
10. for $j = 1$ to $p_1 - 1$ /* until the '*' */
11. if $i + j \leq p_1$ /*if any of the sub-paths do not exist return failure*/
12. if no match(DBF_j, $(a_i/a_{i+1}/\ldots/a_{i+j})$) return(NO MATCH)
13. else if $k = p$ /*if there is no path, or for the last part of the path*/
14. for $i = p_1 + 2$ to p /*after the last * we repeat the above procedure*/
15. for $j = 1$ to $p - 1$
16. if $i + j \leq p$
17. if no match($DBF_j, (a_i/a_{i+1}/\ldots/a_{i+j})$) return(NO MATCH)
18. return(MATCH) /* if statement 18 is reached we have a match*/

(b)

Figure 4: The lookup procedures for (a) Breadth and (b) Depth Bloom filters

3.4 Distribution

Multi-level Bloom filters are used to locate peers that may contain documents that match a path query in a P2P system. Such queries may originate at any peer. Each peer maintains a multi-level Bloom filter for the documents it stores locally and a summary filter for a set of its neighboring peers. This summarized filter facilitates the routing of a query only to peers that may contain relevant data. When a query reaches a peer, the peer checks its local filter and uses the summary filter to direct the query to other peers. Based on how the set of neighboring peers for which we maintain summarized filters is defined, we consider two approaches: the hierarchical and the horizon-based.

In the hierarchical organization, peers are organized into hierarchies which communicate by their roots through a main communication channel. Each peer has two filters: one *local filter* and, if it is a non-leaf peer, one with summarized data for all peers in its sub-tree (*merged filter*). Besides these filters, the root peers contain the merged filters of all other root peers. The hierarchical approach is described in [6].

Apart from the hierarchical approach, we consider an alternative in which peers form an undirected graph. Each peer has a set of neighbors, chosen from the participating peers closest to it in network latency. To distribute the filters across peers, we use *horizons* [7]. Each peer, apart from its local filter, holds data about peers within its horizon. Every peer stores one *merged filter*, for each of its neighbor links. Each such merged filter summarizes data for all peers at distance h (i.e., the horizon) through any path starting with this link. A detailed description of the horizon-based organization can be found in [8].

3.5 Semantic Knowledge

The quality of the method we described for summarizing XML documents depends on the element names used in the documents. The element names are given as input to the hash functions and the output, that is the bits that are set, are produced according to these names. However, often different words are used to describe the same concept, for example author and writer, for the author of an article. The hash functions in this case would return different outputs. However, having a query for an “author”, we would like to be able to return both “author” and “writer” as possible results.

Thus, a pre-processing phase is required before the documents are summarized by the multi-level Bloom filters. The basic idea behind this pre-processing phase is to map the element names contained in the documents into concepts from an ontological space such as WordNet [9]. WordNet distinguishes between *words* as literally appearing in texts and the actual *word senses*, the concepts behind the words. The mapping consists of looking up each element of the document in WordNet, identifying possible word senses and replacing the element name in

Table 2: Input parameters

Parameter	Default Value	Range
# of XML documents	200	-
Total size of filters	78000 bits	30000-150000 bits
# of hash functions	4	-
# of queries	100	-
# of elements per document	50	10-150
# of levels per document	4/6	2-6
Length of query	3	2-6

the document with them. Often, one word has multiple senses. To identify which sense best describes the element name, we have to take into account the other element names of the document, and find which is the correct sense in the context created by them [10]. Similarly, queries must also be pre-processed before looking them up in the filter.

4 Implementation and Experimental Results

We implemented the BBF and DBF data structures as well as an SBF (that just hashes all elements). For the hash functions, we used MD5 and for the generation of the XML documents the Niagara generator [11]. The number of levels of the BBF filters is equal to the number of levels of the XML trees, while for DBFs, we have at most 3 levels. In the case of the BBF, we excluded the optional BBF_0 filter. There is no repetition of element names in a single document or among documents. Queries are generated by producing arbitrary path queries, with 90% element names from the documents and 10% random ones. All queries are partial path queries and the probability of the "*" operator at each query is set to 0.05. Table 2 summarizes our parameters. We use as our performance metric the percentage of false positives, since the number of nodes that will process an irrelevant query depends on it directly.

Influence of filter size. In this experiment, we vary the size of the filters from 30000 to 150000 bits. The lower limit is chosen from the formula $k = (m/n)ln2$ that gives the number of hash functions k that minimize the false positive probability for a given size m and n inserted elements for an SBF : we solved the equation for m keeping the other parameters fixed. As our results show (Fig. 5(left)), both BBFs and DBFs outperform SBFs. For SBFs, increasing their size does not improve their performance, since they recognize as misses only paths that contain elements that do not exist in the documents. BBFs perform very well even

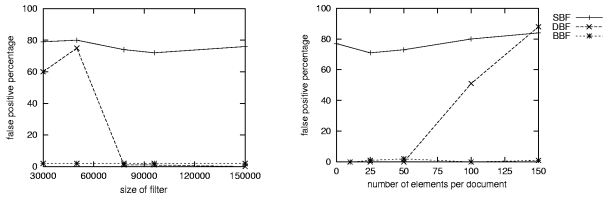

Figure 5: Influence of (left) filter size and (right) number of elements

for 30000 bits with an almost constant 6% of false positives, while DBFs require more space since the number of elements inserted is much larger than that of BBFs and SBFs (Table 1). However, when the size increases sufficiently, the DBFs outperform even the BBFs.

Using the results of this experiment, we choose as the default size of the filters for the rest of the experiments, a size of 78000 bits, for which both our structures showed reasonable results. For 200 documents of 50 elements, this represents 2% of the space that the documents themselves require. This makes Bloom filters a very attractive summary to be used in a P2P context.

Influence of the number of elements per document. In this experiment, we vary the number of elements per document from 10 to 150 (Fig 5(right)). Again, SBFs filter out only path expressions with elements that do not exist in the document. When the filter becomes denser as the elements inserted are increased to 150, SBFs fail to recognize even some of these expressions. BBFs show the best overall performance with an almost constant percentage of 1 to 2% of false positives. DBFs require more space and their performance decreases rapidly as the number of inserted elements increases, and for 150 elements, they become worse than the SBFs, because the filters become overloaded (most bits are set to 1).

Influence of the number of levels. In this experiment, we vary the number of levels of the documents from 2 to 6 (Fig. 6(left)). The queries are of length 3, except for the documents with 2 levels where we conduct the experiment with queries of length 2. The behavior of the SBF is independent of the number of levels of the documents, since they just hash all the elements irrespectively of their level. So they only recognize path expressions with elements not in the documents that account for about 30% of the given query workload. Both the BBF and DBF outperform SBFs with a false positive percentage below 7%. BBFs perform better for 4 and 5 levels. This is because the elements are more evenly allocated to the levels of the filter, while for fewer levels the filter has also less levels and it becomes overloaded. BBFs performance deteriorates for more levels because of the new kind of false positives we discussed in Section 3.3. DBFs perform very well for documents with few levels since they do not face the problem of BBFs. Their performance decreases for more levels but remains almost constant since we insert only subpaths up to length 3.

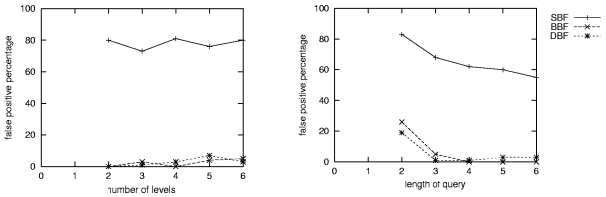

Figure 6: Influence of (left) number of levels and (right) query length

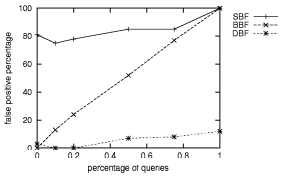

Figure 7: Varying query workload

Influence of the length of the queries. In this experiment, we vary the length of the queries from 2 to 6. Again, both BBF and DBF filters outperform Simple ones. The SBFs performance slightly improves as the query length increases but this is only because the probability for an element that does not exist in the documents increases. Both structures perform better for large path expressions, since if one level is sparse enough it is sufficient to filter out irrelevant queries. DBFs show a slight decrease in performance for a length of 5 and 6 since for documents with 6 levels the number of inserted elements increases and the filter becomes denser.
Workload. In most of our experiments, BBFs seem to outperform DBFs for a fraction of the space the latter require. The reason for using DBFs became evident in the third experiment where BBFs performance deteriorated because of the new kind of false positives they introduce. To clarify this, in this last experiment, we created a workload with queries consisting of such path expressions, that is, a workload that favors DBF filters. The percentage of these queries varied from 0% to 100% of the total workload. We included the SBF in the experiment only for completeness. BBF filters fail to recognize these misses and their percentage of false positives increases linearly to the percentage of these queries in the workload. However, DBFs have no problem with this kind of false positives and show better results. The slight increase of false positives in SBF and DBF is because as the number of these queries increases, the number of queries with elements that do not exist in the document decreases. When all queries are of this special form, the SBF has a percentage of 100% false positives and DBF of about 10%. Thus, we can conclude that one may consider spending more space in order to use DBFs, so as to avoid these false positives, while, when space is the key issue, BBFs are a more reasonable choice.

5 Filter Configuration

For our structures to show a low percentage of false positives so as to assist in query routing and provide accurate results while requiring a small space overhead, we need to tune the filter parameters so as to achieve the smallest false percentage possible with a small space and processing overhead. The parameters we have to consider are the number of filter levels, its size and the number of hash functions.
Levels of the Filter. When space is limited, we can limit the number of the filter levels j to be fewer than the levels of the XML documents we insert into the filter. BBFs require a small space overhead (Fig. 5(left)); thus we recommend not to omit any levels. However, if we had knowledge about the query workload we could limit the levels of the filter. The false positive percentage depends on the maximum level of the XML tree that a query may include (*MaxLevel*). If we knew that most queries would involve elements until MaxLevel, we could ignore the lower levels in order to gain efficiency for most part of the workload while sacrificing accuracy for a small fraction of it. If we have no knowledge about the elements distribution in the workload storing all the filters is more appropriate. For BBFs if we consider limiting the levels, the elements of the ignored levels are inserted into BBF_0. Given a query with length $p > j$, we check for the subpath of length j using the lookup algorithm of Fig. 4(left), and check for the remaining $p - j$ elements in BBF_0. If they exist we have a success. However, this increases the false positive ratio as it does not examine the structure of the remaining $p - j$ elements.

DBFs require much more space, thus, it is desirable to be able to limit the number of levels of a DBF when space is limited. If we had knowledge about the query workload, and more specifically about the maximum length (*MaxLength*) of most of the queries, we could save only MaxLength filters. In contrast to BBFs the level of the elements the query includes does not influence DBFs performance. Thus, storing only MaxLength levels can be very efficient. When, no knowledge of the query workload is available, space limitations might still force us to omit some levels. In the experiment of Fig. 6(right), we used documents with 6 levels and DBFs with only 3 levels, while queries were of length greater than 3. The experiment confirmed that ignoring some levels for the DBFs can be very efficient in saving space without increasing the false positives. The checks for all possible subpaths of a query, prove to be sufficient for recognizing a false positive even if they do not consider the whole length of the query.

We conducted another experiment using a document with 6 levels and 50 elements to examine how ignoring some levels when creating the DBF affects the false positive ratio (Fig. 8(left)). We used queries with length 5 and kept the size of each level fixed at 26000 bits. When using only 1 filter (length 1) the DBF behaves exactly as a Simple Bloom filter. By adding a second filter, we double its size but decrease the false positive ratio at half. As we can see using 3 levels is adequate since we achieve a false positive ratio of 3%. If we use 4 or more filters,

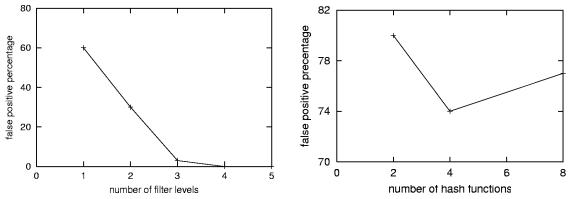

Figure 8: (left) Number of levels and (right) number of hash functions

then the false positive ratio drops to 0 but we also have to consider the significant increase in the filter size. If the ratio of 3% is acceptable, we can use only 3 filters and reduce the overall space of the filter, since a further decrease in the ratio would be small but the filter size would grow another 26000 bits.

Filter Size and Hash functions. We can choose to tune either the number of hash functions (k) or the size of the filter (s) according to the number of elements inserted in each level so as to decrease the false positive percentage. Since it would be easier for query processing, if we used the same number of hash functions for each level, we choose to keep k fixed and tune s. The number of hash functions is also desirable to be kept small for processing and time efficiency. We performed an experiment with varying number of hash functions for Simple Bloom filters to demonstrate that our choice of hash functions does not negatively affect the SBFs behavior. We used documents with 4 levels and 50 elements. As illustrated in Fig. 8(right), the percentage of false positives varies from 74% to 80% for 2 to 8 hash functions; thus we choose 4 hash functions for efficiency.

6 Related Work

Indexing methods for XML provide efficient ways to index XML, support complex queries and offer selectivity estimations. The method in [12] encodes paths as strings, and inserts them into an index that is optimized for string searching. Evaluating queries involves encoding the query as a search key string and performing a lookup in the index. The XSKETCH synopsis [13] relies on a generic graph-summary where each node only captures summary data that record the number of elements that map to it. Emphasis is given on the processing of complex path queries. APEX [14] is an adaptive path index that utilizes frequently used paths to improve query performance. It can be updated incrementally based on the query workload. The path tree [15] has a path for every distinct sequence of tags in the document. If it exceeds main memory space, nodes with the lowest frequency are deleted. In [16], a signature is attached to each node of the XML tree, in order to prune unnecessary sub-trees as early as possible while traversing the tree for a query. In [17], signatures are used to construct a tree index where components extracted from the documents are used as keys. A query signature based on the

components is used for traversing the tree and retrieving all documents that contain it. Compression or auxiliary data structures that add structural information to inverted lists are used in [18] to support information retrieval for XML data.

These structures are centralized and although they support path queries there is no intuitive way for their distribution. In contrast, in P2P systems, methods for finding peers that match a query are limited in handling membership queries. They construct indexes that store summaries of other nodes' data and provide routing protocols to propagate a query to relevant peers. The resource discovery protocol in [4] uses Simple Bloom filters as summaries. Servers, organized into a hierarchy modified according to the query workload, are responsible for query routing. Summaries are a single filter with all the subset hashes of the XML service descriptions up to a certain threshold. To evaluate a query, it is split to all possible subsets and each one is checked in the index.

7 Conclusions and Future Work

In this paper, we introduced two new hash-based indexing structures, based on Bloom filters, which represent hierarchies and exploit the structure of XML documents. Breadth and Depth Bloom filters are multi-level structures that are able to store large data sets in small space. Experiments showed that they both outperform Simple Bloom filters in answering path queries over hierarchically structured data. We also presented how these indices can be distributed in P2P systems. We plan to examine the system performance under different conditions and workloads. Future work includes the extension of the structures to incorporate values in the paths and of the data model to include XML graphs.

References

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

[2] L. Fan, P. Cao, J. Almeida, A. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. of ACM SIGCOMM Conference*, 254-265, 1998.

[3] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, D. Culler. Scalable Distributed Data Structures for Internet Service Construction. In *Proc. of 4th OSDI Symposium*, 173-184, 2000.

[4] T. D. Hodes, S. E. Czerwinski, B. Y. Zhao, A. D. Joseph, R. H. Katz. Architecture for Secure Wide-Area Service Discovery. In *Proc. of 5th Annual MobiCom Conference*, 24-35, 1999.

[5] G. Koloniari. Searching XML Documents in Peer-to-Peer Systems. *Master Thesis*, Department of Computer Science, University of Ioannina, 2003.

[6] G. Koloniari, E. Pitoura. Content-Based Routing in Peer-to-Peer Systems. To appear in *EDBT*, 2004.

[7] A. Crespo, H. Garcia-Molina. Routing Indices for Peer-to-peer Systems. In *Proc. of the 22nd ICDCS Conference*, 23-, 2002.

[8] G. Koloniari, Y. Petrakis, E. Pitoura. Content-Based Overlay Networks of XML Peers Based on Multi-Level Bloom Filters. *DBISP2P*, 2003.

[9] G. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11): 39–41, 1995.

[10] M. Theobald, R. Schenkel, G. Weikum. Exploiting Structure, Annotation and Ontological Knowledge for Automatic Classification of XML Data. *6th International Workshop on the Web and Databases*, 2003.

[11] http://www.cs.wisc.edu/niagara

[12] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, M. Shadmon. Fast Index for Semistructured Data. In *Proc. of 23rd VLDB Conference*, 436-445, 1997.

[13] N. Polyzotis, M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *Proc. of 28th VLDB Conference*, 466-477, 2002.

[14] C-W Chung, J-K Min, K. Shim. APEX: An Adaptive Path Index for XML Data. In *Proc. of ACM SIGMOD*, 436-445, 2002.

[15] A. Aboulnaga, A. R. Alameldeen, J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proc. of 27th VLDB Conference*, 591-600, 2001.

[16] S. Park, H-J Kim. A New Query Processing Technique for XML Based on Signature. In *Proc. of 7th DASFAA Conference*, 22-, 2001.

[17] W. Lian, N. Mamoulis, D. W. Cheung. A Filter Index for Complex Queries on Semi-structured Data. In *Proc. of 4th WAIM Confernce*, 397-408, 2003.

[18] N. Fuhr, N. Govert. Index compression vs. retrieval time of inverted files for XML documents. In *Proc. of the 11th ACM CIKM Conference*, 2002.

Georgia Koloniari is with the Department of Computer Science at the University of Ioannina, Greece GR-45110. E-mail: kgeorgia@cs.uoi.gr

Evaggelia Pitoura is with the Department of Computer Science at the University of Ioannina, Greece GR-45110. E-mail: pitoura@cs.uoi.gr