Bloom-Based Filters for Hierarchical Data

Georgia Koloniari and 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 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 Simple Bloom filters in discovering the appropriate resources.

1. Introduction

Peer-to-peer 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.

Bloom filters 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 efficient 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 [14]. 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 peer-to-peer system where peers store XML documents or XML-based descriptions of the services that they provide. 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

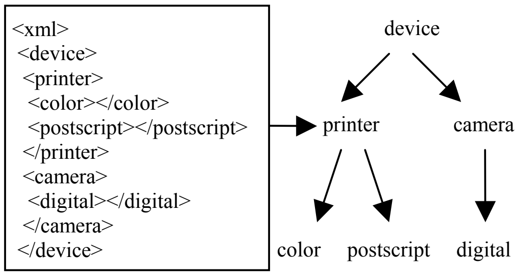
XML document is represented by a tree. Fig. 1 depicts an XML service description for a printer and a camera provided by a peer and the corresponding XML tree.

Definition 1 (XML tree): *An XML tree is an unordered labelled 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 concentrate 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.



```
<xml>
 <device>
  <printer>
   <color></color>
   <postscript></postscript>
  </printer>
  <camera>
   <digital></digital>
  </camera>
 </device>
```

Fig. 1: An XML document and its tree

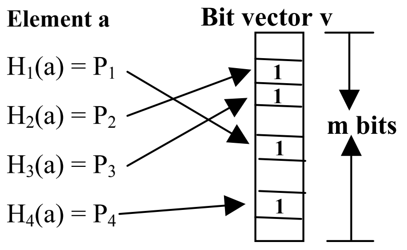


Fig. 2: A Bloom filter with 4 hash functions

2.2 Bloom Filters

Bloom filters are compact data structures for probabilistic representation of a set that support membership queries ("Is element *X* in set *Y*?"). Since their introduction [1], Bloom filters have seen many uses such as web cache sharing [2], query filtering and routing [3, 4] and free text searching [5].

Consider a set $A = \{a_1, a_2, \ldots, a_n\}$ of *n* elements. The idea (Fig. 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 to *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 may be set to 1 many 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" and it is the payoff 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 *l* in the bit array a count $c(l)$ of the number of times that the bit is set to 1 (the number of elements that hashed to l under any of the hash functions). All counts are initially set to 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 corresponding bit is turned on. When a count changes from 1 to 0 the corresponding bit is turned off.

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.

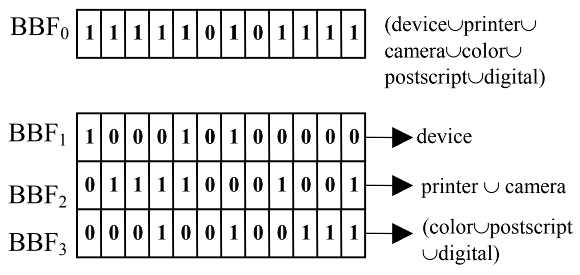
3. Multi-level Bloom Filters

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

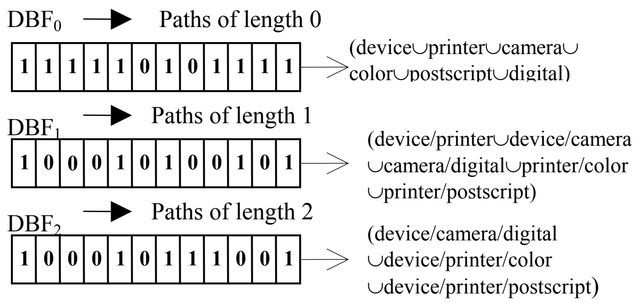
Let an XML tree *T* with *j* levels, and let the level of the root be level 1. The Breadth Bloom Filter (BBF) for an XML tree *T* with *j* levels is a set of Bloom filters {BBF_0, BBF_1, BBF_2, …, BBF_i}, $i \leq j$. There is one Bloom filter, denoted BBF_i, for each level *i* of the tree. In each BBF_i, we insert the elements of all nodes at level *i*. To improve performance, we construct an additional Bloom filter denoted BBF_0. In this Bloom filter, 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 four Bloom filters (Fig. 3).

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, when we have no knowledge for the distribution of the elements at the levels of the tree, we set: $|BBF_{i+1}| = d\,|BBF_i|$, ($i < j$), where *d* is the average degree of the nodes. For equal size BBF_is, BBF_0 is the logical OR of all BBF_is, $1 \leq i \leq j$.



BBF0 1 1 1 1 1 0 1 0 1 1 1 1
(device∪printer∪camera∪color∪postscript∪digital)
BBF1 1 0 0 0 1 0 1 0 0 0 0 0
device
BBF2 0 1 1 1 1 0 0 0 1 0 0 1
printer ∪ camera
BBF3 0 0 0 1 0 0 1 0 0 1 1 1
(color∪postscript∪digital)

Fig. 3: The BBF for the XML tree of Fig. 1.



DBF0
Paths of length 0
1 1 1 1 1 0 1 0 1 1 1 1
(device∪printer∪camera∪color∪postscript∪digital)
DBF1
Paths of length 1
1 0 0 0 1 0 1 0 0 1 0 1
(device/printer∪device/camera∪camera/digital∪printer/color∪printer/postscript)
DBF2
Paths of length 2
1 0 0 0 1 0 1 1 1 0 0 1
(device/camera/digital∪device/printer/color∪device/printer/postscript)

Fig. 4: The DBF for the XML tree of Fig. 1

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 Bloom filters {DBF_0, DBF_1, DBF_2, …, DBF_{i-1}}, $i \leq j$. There is one Bloom filter, denoted DBF_i, for each path of the tree with length *i*, this is having (i + 1) nodes, where we insert all paths of length *i*. For example, the DBF for the XML tree in Fig. 1 is a set of three Bloom filters (Fig. 4). 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. 4 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 maximum number of keys inserted in the filter is order of d^j for a tree with maximum degree d and j levels.

3.2 Search Algorithms

BBF Search: The look-up 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. 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 sub-paths 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 the Appendix.

Let p be the length of the path and d be the number of levels of the filter. (We exclude the Bloom on top-it). 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 $p(d – p + 1)=O(dp)$. When the path contains the * operator, it is split into two sub-paths 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 look-up procedure, that checks whether a DBF matches with a path query, first checks whether all elements in the path expression appear in DBF_0. If this is the case, we continue treating both root and partial paths queries the same. For a query of length p, every sub-path of the query from length 2 to p is checked at the corresponding level. If any of the sub-paths 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 sub-paths are checked. If we have a match for all sub-paths the algorithm succeeds, else we have a miss. The algorithm is presented in detail in the Appendix.

Consider a query of length p. Let p be smaller than the number of the filter's levels. Firstly p sub-paths of length 1 are checked, then $p – 1$ sub-paths of length 2 are checked and so on until we reach length p where we have 1 path. Thus the complexity of the look-up procedure is $p + p – 1 + p – 2 +\ldots+ 1 = p(p + 1)/2 = O(p^2)$. 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 sub-paths 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]: $P = (1 - e^{-kn/m})^k$.

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 [13].

4. Distribution

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

Each peer maintains a multi-level Bloom filter for the documents it stores locally. It also maintains a multi-level Bloom filter summary 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 Bloom filter and uses the summary filter to direct the query to other peers.

To calculate the summarized filter of a set of filters we take the bitwise OR for each of their levels. The summarized filter, Sum_BBF of two Breadth Blooms BBF^k and BBF^m with i levels is a Breadth Bloom Sum_BBF = {Sum_BBF_0, Sum_BBF_1,…, Sum_BBF_i} with i levels where: $Sum\_BBF_j = BBF^k_j$ *BOR* BBF^m_j, $0 \leq j \leq i$ and *BOR* stands for bitwise OR. Similarly, we define the summary for Depth Blooms.

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. Note that we are interested in providing all the results for a query. Our mechanisms are easily extensible to locating the best k results.

4.1 Hierarchical Organization

In the hierarchical approach (Fig. 5), a set of peers is designated as *root peers* that are connected to a main channel that provides communication.

Each peer has two Multi-level Bloom filters: one for the local documents, called *local filter* and, if it is a non-leaf peer, one with summarized data for all peers in its sub-tree, called *merged filter*. To compute the merged filters the leaf peers send their local filters to their parent. The parent merges its children filters and produces its merged filter. Then, it summarizes its merged filter with its local one and propagates this to its parent. We continue until the root peers are reached. Besides these two multi-level Bloom filters, the root peers contain the merged filters of all other root peers.

To process a query, each peer checks its local filter. If there is a match, it checks its local documents. It also propagates the query to its parent and if there is a match with its merged filter, to its children. When the query, reaches a root peer, the root peer checks the merged filters of the other root peers. The query is propagated to all root peers that match it. Each of the matched root peers propagates the query to its children whose merged filters match the query. The procedure continues until the peers at the leaf nodes of the hierarchy are reached.

Updates 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 corresponding merged filter. Note that we need to propagate only the BBF_is (DBF_is) that are updated not the whole BBF (DBF).

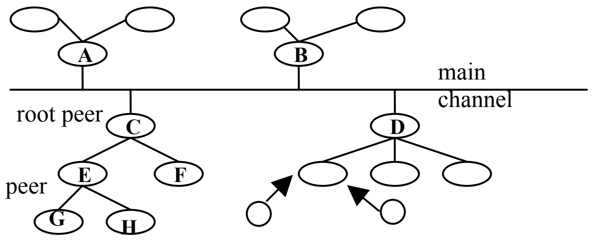


Fig. 5: Hierarchical Organization.

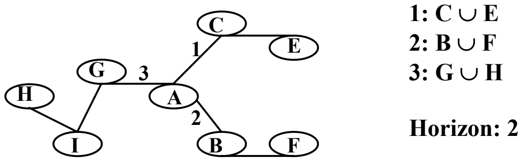


Fig. 6: Horizons.

4.2 Horizons

Apart from the hierarchical approach, we consider an alternative in which peers are forming an undirected graph. Each peer has a set of neighbors, chosen from the participating peers closest to it in network latency. To distribute multi-level Blooms across peers, we use *horizons*. Knowledge about all peers is limiting in terms of scalability. By introducing horizons, a peer bounds the number of neighbors whose data it summarizes. Each peer, apart from its local index, holds data about peers within its horizon. Every peer stores one filter called *merged filter*, for each of its neighbor links. Each such merged filter summarizes data for all peers at distance *h* through any path starting with this link (Fig. 6).

Definition 5 (Distance): *The distance between two nodes N_i and N_j, $d(N_i, N_j)$ is the number of hops on the shortest path from N_i to N_j in the overlay network.*

Definition 6 (horizon): *A node N has a horizon of h if it stores peer summaries only for nodes N_i for which the distance $d(N, N_i) \leq h$, where h is the radius of the horizon.*

Every peer, when entering the system, sends its local index together with a counter set to *h* to all of its neighbors. Every peer that receives a filter merges it with the summary of that link. Then it decreases the hop counter by 1 and if the counter is not 0, it propagates the merged filter to all of its neighbors, except the one it was sent from. Thus, the summary reaches all nodes at distance *h*, and all peers have data about all other peers in their horizon. Every peer that receives the summary replies with its own local summary, so the new peer can construct its neighbors' summaries.

When a query is posed, each filter for all the links of the node is checked and the query is propagated only through links that gave a match. The next neighbor is determined as before. If a query reaches a peer *h* hops from its source due to a false positive; there is no incentive to forward it further. This can be overcome either by backtracking or by using an exhaustive algorithm that searches the graph. To prevent a query from being caught in a cycle, the query holds a list with the nodes it has already visited.

Updates are propagated the same way with summaries, so as to inform all nodes at distance *h* about the change. This approach requires more space than the hierarchical organization, since it stores many different merged filters and not a single summary of all as in the hierarchical organization.

5. Implementation and Experimental Results

We implemented both the BBF and DBF data structures in C. We also implemented a Simple Bloom filter (SBF) that just hashes all elements. We limited the inserted path expressions in Depth Bloom to be at most of length 3, that is, the Depth Bloom only has three levels. Also, we excluded the bloom on top that is only used for performance reasons since it requires more space and it would deteriorate Breadth's performance for a given space overhead. For the hash functions, we used MD5 and for the generation of the XML documents the Niagara generator [11]. The repetition on the names of the elements was set to 0 between the elements of a single document as well as between all the documents. Queries were generated by producing arbitrary regular paths, with 90% elements from the documents and 10% random ones. All queries were partial paths and the probability of the containment operator at each query was set to 0.05. Table 1 summarizes our parameters. We have chosen as our metric the percentage of false positives, since the number of nodes that will process an irrelevant query strongly depends on it.

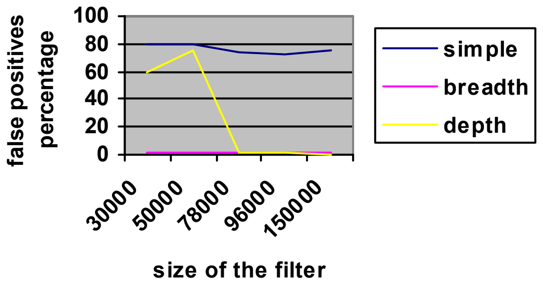
Parameter	**Default Value**	**Range**
# of XML documents	200	-
Total size of filter	78000	30000-150000
# of hash functions	4	
# of queries	100	
# of elements per document	50	10-150
# of levels per document	4 / 6	2-6
Length of query	3	2-6

Table 1: Table of Parameters

Experiment 1: Influence of filter size.

We examine the influence of the size of the filter with respect to false positives. The document's structure is fixed with 50 elements and 4 levels. The queries are of length 3. The size of the filters varies from 30000 bits to 150000 bits. The lower limit was chosen from the formula that gives the number of hash functions *k* that minimize the false positives probability for a given size *m* and *n* inserted elements for a Simple Bloom filter: $k = (m / n)\ln 2$. We solved the equation for *m* keeping the other parameters fixed. The goal of this experiment is to show that even if we increase the size of the filter significantly, Simple Blooms cannot correctly recognize path expressions.

The results show that both Breadth and Depth Blooms outperform Simple Blooms even for only 30000 bits. In addition, in contrast with Simple Blooms where the increase in the size results in no improvement in their performance, the multi-level structures exploit the extra space. Simple Blooms are only able to recognize as misses paths that contain elements that do not exist in the documents. Breadth Blooms perform very well even for 30000 bits with an almost constant 6% of false positives, while Depth Blooms require more space since the number of the elements inserted is much larger than that of Breadth and Simple Blooms. However, when the size increases sufficiently, Depth Blooms outperform even Breadth Blooms and produce no false positives.



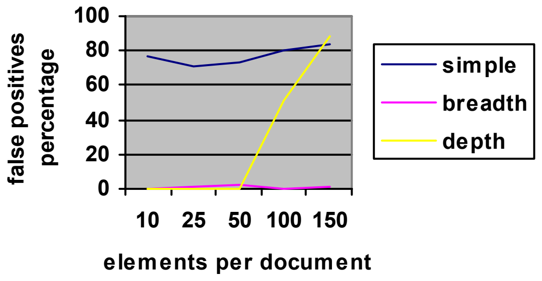


Fig. 8: Experiment 1: varying the size of the filters

Fig. 9: Experiment 2: number of elements

Using the result of the first experiment, we choose as the default size of the filters for the rest of the experiments, a size of 78000 bits, where 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 peer-to-peer context.

Experiment 2: Influence of the number of elements per document.

We compare the filters with respect to the number of elements per document. The size of the filter is fixed to 78000 bits, and the documents have 4 levels. Queries have length 3 and the elements vary from 10 to 150.

Once again, Simple Bloom is only able to recognize path expressions with elements that do not exist in the document. Even for 10 elements where the filter is very sparse, Simple Blooms have no means to recognize hierarchies. When the filter becomes denser as the elements inserted are increased to 150, Simple Blooms fail to recognize even some of these expressions. Breadth Blooms show the best overall performance with an almost constant percentage of 1 to 2% of false positives. Depth Blooms require more space and their performance rapidly decreases as the number of inserted elements increases, and for 150 elements they become worse than Simple Blooms because the filters become overloaded (most bits are set to 1).

Experiment 3: Influence of the number of levels.

In this experiment, we compare the three approaches with respect to the number of levels of the documents. The size of the filter is fixed to 78000 bits, and the documents have 50 elements. The levels vary from 2 to 6. 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' behavior is independent of the number of levels of the documents, since they just hash all their elements irrespectively of the level that they belong to. So they only recognize path expressions with elements not in the documents that account for about 30% of the given query workload. Both Breadth and Depth Blooms outperform them with a false positive percentage below 7%. Breadth Blooms perform better for 4 to 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.

Also false positives of a new kind appear for Breadth Blooms. If we had a tree that had the following paths: /a/b/c and /a/f/l then Breadth Bloom would falsely recognize as correct the following path: /a/b/l. Depth Blooms do not have this problem as they would check for all possible sub-paths /a/b/l, /a/b, /b/l, and would find a miss for the last one. That is why they perform very well for documents with few levels. Their performance decreases for more levels but remains almost constant since we insert only sub-paths up to length 3, while Breadth Blooms deteriorate further for 6 levels.

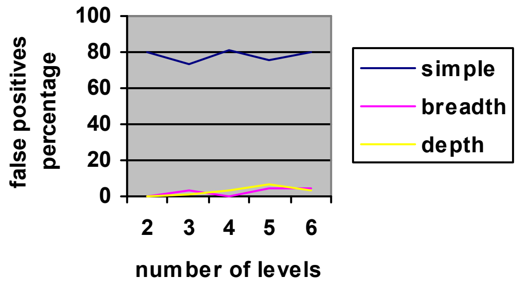


Fig. 10: Experiment 3: number of levels

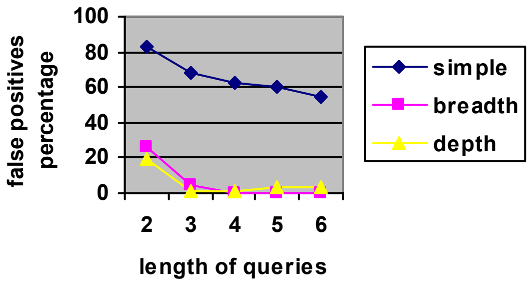


Fig. 11: Experiment 4: varying query sizes

Experiment 4: Influence of the length of the queries.

The parameter examined in this experiment is the length of the queries. The structure of the document is fixed, with 4 levels and 50 elements, for queries of length 2 to 4 and 6 levels for queries with length 5 and 6. The size of the filter is also fixed to 78000 bits.

Once again, both multi-level Blooms outperform Simple ones. The Simple Blooms 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. Depth blooms 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.

The last two experiments also show that though we limited the number of filters to three for the Depth bloom, it is still able to show a very good performance although all the elements (all possible sub-paths with length larger than 3) are not inserted. The checks of all possible sub-paths up to length 3 are able to recognize most of the misses, so we conclude that we can limit the number of levels of the filter without a significant loss in performance if we have limited space.

Experiment 5: Workload

In most of our experiments, Breadth Blooms seem to outperform Depth Blooms for a fraction of the space the second require. The reason for using Depth Blooms became evident in Experiment 3. Breadth Blooms fail to recognize a new kind of false positives such as the example we described in Experiment 3. To clarify this, in this last experiment, we created a workload with queries consisting of such path expressions, that is, a workload that favors Depth Bloom filters. The percentage of these queries varied from 0% to 100% of the total workload. The size of the filter is fixed to 78000 bits; the documents have 4 levels and 50 elements. We included the Simple Bloom in the experiment only for completeness.

Breadth Blooms fail to recognize these misses and their percentage of false positives increases linearly to the percentage of these queries in the workload. However, Depth Blooms have no problem of recognizing this kind of false positives and show much better results. The slight increase of the percentage of false positives in Simple and Depth Blooms is because as the number of these special queries increases, the number of queries with elements that do not exist in the document decreases. When all queries are of this special form (thus, there are no queries with elements that do not exist in the documents), the Simple Blooms has a percentage of 100% false positives and Depth of about 10%. Thus, we can conclude that one may consider spending more space in order to use Depth Blooms, so as to avoid these false positives, while, when space is the key issue, Breadth Blooms are a more reasonable choice.

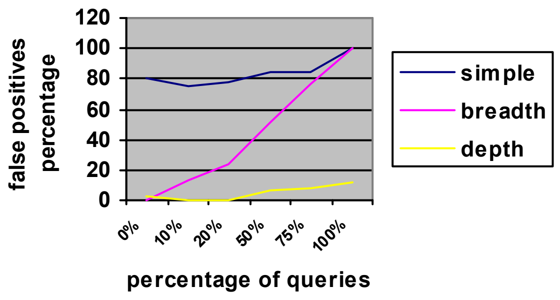


Fig. 12: Experiment 5: varying the workload type

6. Related Work

We consider two lines of research related to our work: indexing XML documents and resource discovery in peer-to-peer systems. Indexing methods for XML provide efficient ways to index XML, support complex queries and offer selectivity estimations.

The method in [6, 7] 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 [9] 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 [8] 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 [10] 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 [12], 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.

These structures are centralized and although they support complex path queries there is no intuitive way for their distribution. In contrast, in peer-to–peer 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, which are 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. 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 documents. We also presented how these indices can be distributed in peer-to-peer systems. We plan to examine the system's performance under different conditions and workloads. Future work also 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, pages 13(7): 422-426, July 1970.

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

[3] S.D. Gribble, E.A. Brewer, J.M. Hellerstein, D. Culler. Scalable Distributed Data Structures for Internet Service Construction. In Procs of the Fourth Symposium on Operating Systems Design and Implementation, 2000.

[4] T.D. Hodes, S.E. Czerwinski, B.Y. Zhao, A.D. Joseph, R.H. Katz. Architecture for Secure Wide-Area Service Discovery. Mobicom '99.

[5]M.V. Ramakrishna. Practical performance of Bloom Filters and parallel free-text searching. Communications of the ACM, 32 (10). 1237-1239.

[6] B. Cooper, M. Shadmon. The Index Fabric: Technical Overview. RightOrder Inc 2001.

[7] B.F. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, M. Shadmon. Fast Index for Semistructured Data. VLDB 2001.

[8] CW. Chung, JK. Min, K. Shim. APEX: An Adaptive Path Index for XML Data. ACM SIGMOD '2002.

[9] N. Polyzotis, M. Garofalakis. Structure and Value Synopses for XML Data Graphs. VLDB 2002.

[10] A. Aboulnaga, A.R. Alameldeen, J.F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. VLDB 2001.

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

[12] S. Park, HJ. Kim. A New Query Processing Technique for XML Based on Signature. DASFAA 2001.

[13] G. Koloniari and E. Pitoura. Bloom-Based Filters for Hierarchical Data. Univ. of Ioannina, Computer Science Dept, Technical report, TR-2002-29, 2002.

[14] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-peer Systems. *In ICDCS, 2002*

Appendix A: The lookup procedures for Breadth and Depth Bloom filters.

```
Lookup(Breadth Bloom Filter BBF, path expression path)
        BBF= {$BBF_0$, $BBF_1$, $BBF_2$, … $BBF_i$}
        Path = $a_1/a_2/…/a_p$

/* check for all attributes of the path $a_1, …, a_p$ in the $BBF_0$
the * is ignored when found in the path */
1.   for $i = 1$ to $p$
2.       if $a_i \neq$ *
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
11.              return NO MATCH
12.    for all stored matches:
13.        for i = 1  to number of paths
14.            if   all  start_point(match  i)>end_point(match
     i+1)
15.                return(MATCH)
16.  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 if the query is longer than the remaining
levels*/
2.   if $j + p > d$ return
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($BBF_j$,$a_i$)
5.                   goto 1
7.               else
8.                   $j = j + 1$
9.       store MATCH  /*if 9  is reached we had a*/
10.      goto 1            /*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*/
```

```
Lookup(DepthBloom Filter DBF, path expression path)
        DBF= {$DBF_0$, $DBF_1$, $DBF_2$, … $DBF_{i-1}$}
        Path = $a_1/a_2/…/a_p$

/* check for all attributes of the path $a_1, …, a_p$ in the $BBF_0$
the * is ignored when found in the path */
1.   for $i = 1$ to $p$
2.       if $a_i \neq$ *
3.           if no match($DBF_0$, $a_i$) 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.           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 reurn failure*/
12.                      if no match($DBF_{j+1}$, ($a_i / a_{i+1} /…/
     a_{i+j}$)) 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($DBF_{j+1}$, ($a_i / a_{i+1}
     /…/ a_{i+j}$)) return(NO MATCH)
18.  return(MATCH)  /* if statement 18 is reached we
     have a match*/
```