Content-Based Routing of Path Queries in Peer-to-Peer Systems*

Georgia Koloniari and Evaggelia Pitoura

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

Abstract. Peer-to-peer (P2P) systems are gaining increasing popularity as a scalable means to share data among a large number of autonomous nodes. In this paper, we consider the case in which the nodes in a P2P system store XML documents. We propose a fully decentralized approach to the problem of routing path queries among the nodes of a P2P system based on maintaining specialized data structures, called filters that efficiently summarize the content, i.e., the documents, of one or more node. Our proposed filters, called multi-level Bloom filters, are based on extending Bloom filters so that they maintain information about the structure of the documents. In addition, we advocate building a hierarchical organization of nodes by clustering together nodes with similar content. Similarity between nodes is related to the similarity between the corresponding filters. We also present an efficient method for update propagation. Our experimental results show that multi-level Bloom filters outperform the classical Bloom filters in routing path queries. Furthermore, the content-based hierarchical grouping of nodes increases recall, that is, the number of documents that are retrieved.

1 Introduction

The popularity of file sharing systems such as Napster, Gnutella and Kazaa has spurred much current attention to peer-to-peer (P2P) computing. Peer-to-peer computing refers to a form of distributed computing that involves a large number of autonomous computing nodes (the peers) that cooperate to share resources and services [1]. As opposed to traditional client-server computing, nodes in a P2P system have equal roles and act as both data providers and data consumers. Furthermore, such systems are highly dynamic in that nodes join or leave the system and change their content constantly.

Motivated by the fact that XML has evolved as a standard for publishing and exchanging data in the Internet, we assume that the nodes in a P2P system store and share XML documents [23]. Such XML documents may correspond either to native XML documents or to XML-based descriptions of local services or datasets. Such datasets may be stored in local to each node databases supporting diverse data models and exported by the node as XML data.

* Work supported in part by the IST programme of the European Commission FET under the IST-2001-32645 DBGlobe project, IST-2001-32645

A central issue in P2P computing is locating the appropriate data in these huge, massively distributed and highly dynamic data collections. Traditionally, search is based on keyword queries, that is, queries for documents whose name matches a given keyword or for documents that include a specific keyword. In this paper, we extend search to support path queries that exploit the structure of XML documents. Although data may exhibit some structure, in a P2P context, it is too varied, irregular or mutable to easily map to a fixed schema. Thus, our assumption is that XML documents are schema-less.

We propose a decentralized approach to routing path queries among highly distributed XML documents based on maintaining specialized data structures that summarize large collections of documents. We call such data structures *filters*. In particular, each node maintains two types of filters, a *local filter* summarizing the documents stored locally at the node and one or more *merged filters* summarizing the documents of its neighboring nodes. Each node uses its filters to route a query only to those nodes that may contain relevant documents. Filters should be small, scalable to a large number of nodes and documents and support frequent updates.

Bloom filters have been used as summaries in such a context [2]. Bloom filters are compact data structures used to support keyword queries. However, Bloom filters are not appropriate for summarizing hierarchical data, since they do not exploit the structure of data. To this end, we introduce two novel multi-level data structures, *Breadth* and *Depth* Bloom filters, that support efficient processing of path queries. Our experimental results show that both multi-level Bloom filters outperform a same size traditional Bloom filter in evaluating path queries. We show how multi-level Bloom filters can be used as summaries to support efficient query routing in a P2P system where the nodes are organized to form hierarchies. Furthermore, we propose an efficient mechanism for the propagation of filter updates. Our experimental results show that the proposed mechanism scales well to a large number of nodes.

In addition, we propose creating overlay networks of nodes by linking together nodes with similar content. The similarity of the content (i.e., the local documents) of two nodes is related to the similarity of their filters. This is cost effective, since a filter for a set of documents is much smaller than the documents themselves. Furthermore, the filter comparison operation is more efficient than a direct comparison between sets of documents. As our experimental results show, the content-based organization is very efficient in retrieving a large number of relevant documents, since it benefits from the content clusters that are created when forming the network.

In summary, the contribution of this paper is twofold: (i) it proposes using filters for routing path queries over distributed collections of schema-less XML documents and (ii) it introduces overlay networks over XML documents that cluster nodes with similar documents, where similarity between documents is related to the similarity between their filters.

The remainder of this paper is structured as follows. Section 2 introduces multi-level Bloom filters as XML routers in P2P systems. Section 3 describes a hierarchical distribution of filters and the mechanism for building content-based overlay networks based on filter similarity. Section 4 presents the algorithms for query routing and update propagation, while Section 5 our experimental results. Section 6 presents related research and Section 7 concludes the paper.

2 Routers for XML Documents

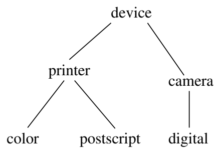
We consider a P2P system in which each participating node stores XML documents. Users specify queries using path expressions. Such queries may originate at any node. Since it is not reasonable to expect that users know which node hosts the requested documents, we propose using appropriately distributed data structures, called *filters*, to route the query to the appropriate nodes.

2.1 System Model

We consider a P2P system where each node n_i maintains a set of XML documents D_i (a particular document may be stored in more than one node). Each node is logically linked to a relatively small set of other nodes called its *neighbors*.

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

(a)



(b)

Fig. 1. Example of (a) an XML document and (b) the corresponding tree

In our data model, an XML document is represented by an unordered labeled tree, where tree nodes correspond to document elements, while edges represent direct element-subelement relationships. Figure 1 depicts an XML service description for a printer and a camera provided by a node and the corresponding XML tree. Although, most P2P systems support only queries for documents that contain one or more *keywords*, we want also to query the structure of documents. Thus, we consider *path queries* that are simple path expressions in an XPath-like query language.

Definition 1. (path query) *A path query of length p has the form ‘s_1 l_1 s_2 l_2 ... s_p l_p” where each l_i is an element name and each s_i is either / or // denoting respectively parent-child and ancestor-descendant traversal.*

A keyword query for documents containing keyword k is just the path query $//k$. For a query q and a document d, we say that q is satisfied by d, or *match(d, q)* is true, if the path expression forming the query exists in the document. Otherwise we have a *miss*. Nodes that include documents that match the query are called *matching nodes*.

2.2 Query Routing

A given query may be matched by documents at various nodes. Thus, central to a P2P system is a mechanism for locating nodes with matching documents. In this regard, there are two types of P2P systems. In *structured P2P* systems, documents (or indexes of documents) are placed at specific nodes usually based on distributed hashing (such as in CAN [21] and Chord [20]). With distributed hashing, each document is associated with a key and each node is assigned a range of keys and thus documents. Although, structured P2P systems provide very efficient searching, they compromise node autonomy and in addition require sophisticated load balancing procedures.

In *unstructured P2P* systems, resources are located at random points. Unstructured P2P systems can be further distinguished between systems that use indexes and those that are based on flooding and its variations. With flooding (such as in Gnutella [22]), a node searching for a document contacts its neighbor nodes which in turn contact their own neighbors until a matching node is reached. Flooding incurs large network overheads. In the case of indexes, these can be either centralized (as in Napster [8]), or distributed among the nodes (as in routing indexes [19]) providing for each node a partial view of the system.

Our approach is based on unstructured P2P systems with distributed indexes. We propose maintaining as indexes specialized data structures, called *filters*, to facilitate propagating the query only to those nodes that may contain relevant information. In particular, each node maintains one filter that summarizes all documents that exist locally in the node. This is called a *local filter*. Besides its local filter, each node also maintains one or more filter, called *merged filters*, summarizing the documents of a set of its neighbors. When a query reaches a node, the node first checks its local filter and uses the merged filters to direct the query only to those nodes whose filters match the query.

Filters should be much smaller than the data itself and should be lossless, that is if the data match the query, then the filter should match the query as well. In particular, each filter should support an efficient *filter-match* operation such that if a document matches a query q then filter-match should also be true. If the filter-match returns false, we say that we have a *miss*.

Definition 2. (filter match) *A filter F(D) for a set of documents D has the following property: For any query q, if filter-match(q, F(D)) = false, then match(q, d) = false,* $\forall\ d \in D$.

Note that, the reverse does not necessarily hold. That is, if filter-match(q, *F(D)*) = true, then there may or may not exist documents $d \in D$ such that match(q, d) is true. We call *false positive* the case in which, for a filter *F(D)* for a set of documents D, filter-match(q, *F(D)*) = true but there is no document $d \in D$ that satisfies q, that is $\forall\ d \in D$, match(q, d) = false. We are interested in filters with small probability of false positives.

Bloom filters are appropriate as summarizing filters in this context in terms of scalability, extensibility and distribution. However, they do not support path queries. To this end, we propose an extension called multi-level Bloom filters.

Multi-level Bloom filters were first presented in [17] where preliminary results were reported for their centralized use. To distinguish traditional Bloom filters from the extended ones, we shall call the former *simple* Bloom filters. Other hash-based structures, such as signatures [13], have similar properties with Bloom filters and our approach could also be applied to extend them in a similar fashion.

2.3 Multi-level 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 [3], Bloom filters have seen many uses such as web caching [4] and query filtering and routing [2,5]. 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 a false positive. It has been shown [3] that the probability of a false positive is equal to $(1 - e^{-kn/m})^k$. 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).

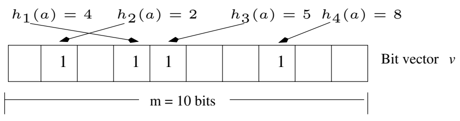
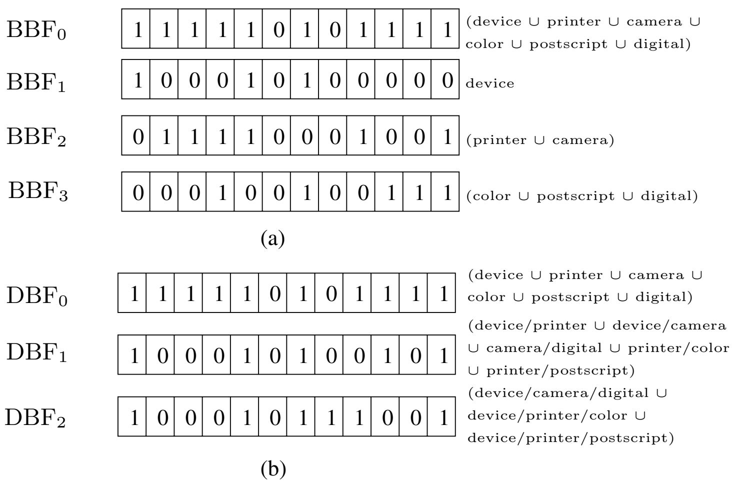


Fig. 2. A (simple) Bloom filter with $k = 4$ hash functions

Let T be an XML tree 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 simple Bloom filters $\{BBF_1, BBF_2, \ldots BBF_i\}$, $i \leq j$. 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)).

The *Depth Bloom Filter* (DBF) for an XML tree T with j levels is a set of simple Bloom filters $\{DBF_0, DBF_1, DBF_2, \ldots, DBF_{i-1}\}$, $i \leq j$. There is one Bloom filter, denoted DBF_i, for each path of the tree with length i, (i.e., a path of $i + 1$ nodes), where we insert all paths of length i. For example, the DBF for the XML tree in Fig. 1 is a set of 3 simple Bloom filters (Fig. 3(b)). Note that



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)
(a)
DBF0
1 1 1 1 1 0 1 0 1 1 1 1
(device ∪ printer ∪ camera ∪ color ∪ postscript ∪ digital)
DBF1
1 0 0 0 1 0 1 0 0 1 0 1
(device/printer ∪ device/camera ∪ camera/digital ∪ printer/color ∪ printer/postscript)
DBF2
1 0 0 0 1 0 1 1 1 0 0 1
(device/camera/digital ∪ device/printer/color ∪ device/printer/postscript)
(b)

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

we insert paths as a whole; we do not hash each element of the path separately. We use a different notation for paths starting from the root. This is not shown in Fig. 3(b) for ease of presentation.

The BBF filter-match operation (that checks whether a BBF matches a query) distinguishes between queries starting from the root and partial path queries. In both cases, if BBF_0 exists, the procedure checks whether it matches all elements of the query. If so, it proceeds to examine the structure of the path, else, it returns a miss. For a root query: $/a_1/a_2/.../a_p$, every level i from 1 to p of the filter is checked for the corresponding a_i. The procedure succeeds, if there is a match 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 match, the next level is checked for the next element and so on 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 ancestor-descendant axis //, the path is split at the // and the sub-paths are processed. The complexity of the BBF filter-match is $O(p^2)$ where p is the length (number of elements) of the query; in particular, for root queries the complexity is $O(p)$. The DBF filter-match operation checks whether all sub-paths of the query match the corresponding filters; its complexity is also $O(p^2)$. A detailed description of the filter match operations is given in [24].

3 Content-Based Linking

In this section, we describe how the nodes are organized and how the filters are built and distributed among them.

3.1 Hierarchical Organization

Nodes in a P2P system may be organized to form various topologies. In a *hierarchical organization* (Fig. 4), a set of nodes designated as *root nodes* are connected to a main channel that provides communication among them. The main channel acts as a broadcast mechanism and can be implemented in many different ways. A hierarchical organization is best suited when the participating nodes have different processing and storage capabilities as well as varying stability, that is, some nodes stay longer online, while others stay online for a limited time. With this organization, nodes belonging to the top levels receive more load and responsibilities, thus, the most stable and powerful nodes should be located to the top levels of the hierarchies.

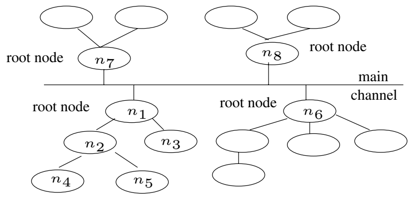


Fig. 4. Hierarchical organization

Each node maintains two filters: one summarizing its local documents, called *local filter* and, if it is a non-leaf node, one summarizing the documents of all nodes in its sub-tree, called *merged filter*. In addition, root nodes keep one merged filter for each of the other root nodes. The construction of filters follows a bottom-up procedure. A leaf node sends its local filter to its parent. A non-leaf node, after receiving the filters of all its children, merge them and produces its merged filter. Then, it merges the merged filter with its own local filter and sends the resulting filter to its parent. When a root computes its merged filter, it propagates it to all other root nodes.

Merging of two or more multi-level filters corresponds to computing a bitwise OR (BOR) of each of their levels. That is, the merged filter, D, of two Breadth Bloom filters B and C with i levels is a Breadth Bloom filter with i levels: $D = \{D_0, D_1, \dots D_i\}$, where $D_j = B_j$ BOR C_j, $0 \leq j \leq i$. Similarly, we define merging for Depth Bloom filters.

Although we describe a hierarchical organization, our mechanism can be easily applied to other node organizations as well. Preliminary results of the filters deployment in a non-hierarchical peer-to-peer system are reported in [18].

3.2 Content-Based Clustering

Nodes may be organized in hierarchies based on their proximity at the underlying physical network to exploit physical locality and minimize query response time. The formation of hierarchies can also take into account other parameters such as administrative domains, stability and the different processing and storage capabilities of the nodes. Thus, hierarchies can be formed that better leverage the workload. However, such organizations ignore the content of nodes. We propose an organization of nodes based on the similarity of their content so that nodes with similar content are grouped together. The goal of such content-based clustering is to improve the efficiency of query routing by reducing the number of irrelevant nodes that process a query. In particular, we would like to optimize *recall*, that is the percentage of matching nodes that are visited during query routing. We expect that content-based clustering will increase recall since matching nodes will be only a few hops apart.

Instead of checking the similarity of the documents themselves, we rely on the similarity of their filters. This is more cost effective, since a filter for a set of documents is much smaller than the documents. Moreover, the filter comparison operation is more efficient than a comparison between two sets of documents. Documents with similar filters are expected to match similar queries.

Let B be a simple Bloom filter of size m. We shall use the notation $B[i]$, $1 \leq i \leq m$ to denote the ith bit of the filter. Let two simple Bloom filters B and C of size m, their Manhattan (or Hamming) distance, $d(B, C)$ is defined as $d(B, C)$ = $|B[1]$ - $C[1]|$ + $|B[2]$ - $C[2]|$ + $\ldots$ + $|B[m]$ - $C[m]|$, that is the number of bits that they differ. We define the similarity, of B and C as $similarity(B, C) = m$ - $d(B, C)$. The larger their similarity, the more similar the filters. In the case of multi-level Bloom filters, we take the sum of the similarities of each pair of the corresponding levels.

We use the following procedure to organize nodes based on content similarity. When a new node n wishes to join the P2P system, it sends a join request that contains its local filter to all root nodes. Upon receiving a join request, each root node compares the received local filter with its merged filter and responds to n with the measure of their filter similarity. The root node with the largest similarity is called the *winner* root. Node n compares its similarity with the winner root to a system-defined *threshold*. If the similarity is larger than the threshold, n joins the hierarchy of the winner root, else n becomes a root node itself. In the former case, node n replies to the winner root that propagates its reply to all nodes in its sub-tree. The node connects to the node in the winner root's subtree that has the most similar local filter.

The procedure for creating content-based hierarchies effectively clusters nodes based on their content, so that similar nodes belong to the same hierarchy (cluster). The value of threshold determines the number of hierarchies in the system and affects system performance. Statistical knowledge, such as the average similarity among nodes, may be used to define threshold. We leave the definition of threshold and the dynamic adaptation of its value as future work.

4 Querying and Updating

We describe next how a query is routed and how updates are processed.

4.1 Query Routing

Filters are used to facilitate query routing. In particular, when a query is issued at a node n, routing proceeds as follows. The local filter of node n is checked, and if there is a match, the local documents are searched. Next, the merged filter of n is checked, and if there is a match, the query is propagated to n's children. The query is also propagated to the parent of the node. The propagation of a query towards the bottom of the hierarchy continues, until either a leaf node is reached, or the filter match with the merged filter of an internal node indicates a miss. The propagation towards the top of the hierarchy continues until the root node is reached. When a query reaches a root node, the root, apart from checking the filter of its own sub-tree, it also checks the merged filters of the other root nodes and forwards the query only to these root nodes for which there is a match. When a root node receives a query from another root it only propagates the query to its own sub-tree.

4.2 Update Propagation

When a document is updated or a document is inserted or deleted at a node, its local filter must be updated. An update can be viewed as a delete followed by an insert. When an update occurs at a node, apart from the update of its local filter, all merged filters that use this local filter must be updated. We present two different approaches for the propagation of updates based on the way the counters of the merged filters are computed. Note that in both cases we propagate the levels of the multi-level filter that have changed and not the whole multi-level filter.

The straightforward way to use the counters at the merged filters is for every node to send to its parent, along with its filter, the associated counters. Then, the counters of the merged filter of each internal node are computed as the sum of the respective counters of its children's filters. We call this method *CountSum*. An example with simple Bloom Filters is show in Fig. 5(a). Now, when a node updates its local filter and its own merged filter to represent the update, it also sends the differences between its old and new counter values to its parent. After updating its own summary, the parent propagates in turn the difference to its parent until all affected nodes are informed. In the worst case, in which an update occurs at a leaf node, the number of messages that need to be sent is equal to the number of levels in the hierarchy, plus the number of roots in the main channel.

We can improve the complexity of update propagation by making the following observation: an update will only result in a change in the filter itself if the counter turns from 0 to 1 or vice versa. Taking this into consideration, each node just sends its merged filter to its parent (local filter for the leaf nodes) and not the counters. A node that has received all the filters from its children creates its merged filter as before but uses the following procedure to compute the counters: it increases each counter bit by one every time a filter of its children has a 1 in the corresponding position. Thus, each bit of the counter of a merged filter represents the number of its children's filters that have set this bit to 1 (and not how many times the original filters had set the bit to 1). We call this method *BitSum*. An example with simple Bloom Filters is show in Fig. 5(c). When an update occurs, it is propagated only if it changes a bit from 1 to 0 or vice versa.

An example is depicted in Fig. 5. Assume that node n_4 performs an update; as a result, its new (local) filter becomes (1, 0, 0, 1) and the corresponding counters (1, 0, 0, 2). With CountSum (Fig. 5(a)), n_4 will send the difference (-1, 0, -1, -1) between its old and new counters to node n_2, whose (merged) filter will now become (1, 0, 1, 1) and the counters (2, 0, 1, 4). Node n_2 must also propagate the difference (-1, 0, -1, -1) to its parent n_1 (although no change was reflected at its filter). The final state is shown in Fig. 5(b). With BitSum (Fig. 5(c)), n_4 will send to n_2 only those bits that have changed from 1 to 0 and vice versa, that is (-, -, -1, -). The new filter of n_2 will be (1, 0, 1, 1) and the counters (2, 0, 1, 2). Node n_2 does not need to send the update to n_1. The final state is illustrated in Fig. 5(d). The BitSum approach sends fewer and smaller messages.

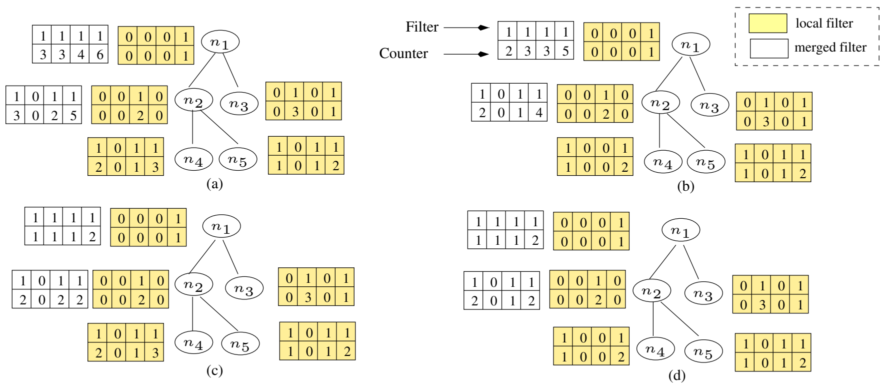


Fig. 5. An example of an update using CountSum and BitSum

5 Experimental Evaluation

We implemented the BBF (Breadth Bloom filter) and the DBF (Depth Bloom Filter) data structures, as well as a Simple Bloom filter (SBF) (that just hashes all elements of a document) for comparison. For the hash functions, we used MD5 [6]: a cryptographic message digest algorithm that hashes arbitrarily length strings to 128 bits. The k hash functions are built by first calculating the MD5 signature of the input string, which yields 128 bits, and then taking k groups of $128/k$ bits from it. We used the Niagara generator [7] to generate tree-structured XML documents of arbitrary complexity. Three types of experiments are performed. The goal of the *first set of experiments* is to demonstrate the appropriateness of multi-level Bloom filters as filters of hierarchical documents. To this end, we evaluate the false positive probability for both DBF and BBF and compare it with the false positive probability for a same size SBF for a variety of query workloads and document structures. The *second set of experiments* focuses on the performance of Bloom filters in a distributed setting using both a content-based and a non content-based organization. In the *third set of experiments*, we evaluate the update propagation procedures.

5.1 Simple versus Multi-level Bloom Filters

In this set of experiments, we evaluate the performance of multi-level Bloom filters. As our performance metric, we use the percentage of false positives, since the number of nodes that will process an irrelevant query depends on it directly. In all cases, the filters compared have the *same* total size. Our input parameters are summarized in Table 1. In the case of the Breadth Bloom filter, we excluded the optional Bloom filter BBF_0. The number of levels of the Breadth Bloom filters is equal to the number of levels of the XML trees, while for the Depth Bloom filters, we have at most three levels. There is no repetition of element names in a single document or among documents. Queries are generated by producing arbitrary path queries with 90% elements from the documents and 10% random ones. All queries are partial paths and the probability of the // axis at each query is set to 0.05.

Table 1. 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
Distribution of query	90% in documents	
elements	10% random	0%-10%

Influence of filter size. In this experiment, we vary the size of the filters from 30000 bits 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. 6(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 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. However, when the size increases sufficiently, the DBFs outperform even the BBFs. Note than in DBFs the number of elements inserted in each level i of the filter is about: $2d^i + \Sigma_{j=i+1}^{l} d^j$, where d is the degree of the XML nodes and l the number of levels of the XML tree, while the corresponding number for BBFs is: d^{i-1}, which is much smaller.

Using the results of this experiment, we choose as the default size of the filters for the rest of the experiments in this set, 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 computing context.

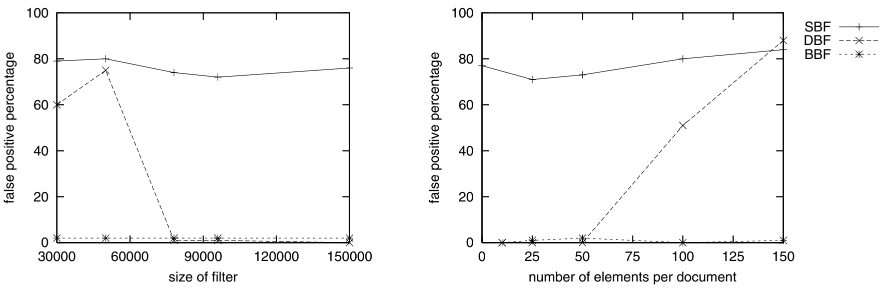


Fig. 6. Comparison of Bloom filters: (left) filter size and (right) number of elements per document

Influence of the number of elements per document. In this experiment, we vary the number of elements per document from 10 to 150 (Fig 6(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 rapidly decreases 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).

Other Experiments. We performed a variety of experiments [24]. Our experiments show that, DBFs perform well, although we have limited the number of their levels to 3 (we do not insert sub-paths of length greater than 3). This is because for each path expression of length p, the filter-match procedure checks all its possible sub-paths of length 3 or less; in particular, it performs (p - i + 1) checks at every level i of the filter. In most cases, BBFs outperform DBFs for small sizes. However, DBFs perform better for a special type of queries. Assume an XML tree with the following paths: /a/b/c and /a/f/l, then a BBF would falsely match the following path: /a/b/l. However, DBFs would check all its possible sub-paths: /a/b/l, /a/b, /b/l and return a miss for the last one. This is confirmed by our experiments that show DBFs to outperform BBFs for such query workloads.

5.2 Content-Based Organization

In this set of experiments, we focus on filter distribution. Our performance metric is the number of hops for finding matching nodes. We simulated a network of nodes forming hierarchies and examined its performance with and without the deployment of filters and for both a content and a non content-based organization. First, we use simple Bloom filters and queries of length 1, for simplicity. In the last experiment, we use multi-level Bloom filters with path queries (queries with length larger than 1). We use small documents and accordingly small-sized filters. To scale to large documents, we just have to scale up the filter as well. There is one document at each node, since a large XML document corresponds to a set of small documents with respect to the elements and path expressions extracted. Each query is matched by about 10% of the nodes. For the content-based organization, the threshold is pre-set so that we can determine the number of hierarchies created. Table 2 summarizes our parameters.

Table 2. Distribution parameters

Parameter	Default Value	Range
# of XML documents per node	1	-
Total size of filter	200-800	-
# of queries	100	-
# of elements per document	10	-
# of levels per document	4	-
Length of query	1-2	-
Number of nodes	100-200	20-200
Maximum number of hops	First matching node found	20-200
Out-degree of a node	2-3	-
Repetition between documents	Every 10% of all docs 70% similar	-
Levels of hierarchy	3-4	-
Matching nodes for a query	10% of # of nodes	1-50%

Content vs. non content-based distribution. We vary the size of the network, that is, the number of participating nodes from 20 to 200. We measure the number of hops a query makes to find the first matching node. Figure 7(left) illustrates our results. The use of filters improves query response. Without using filters, the hierarchical distribution performs worse than organizing the nodes in a linear chain (where the worst case is equal to the number of nodes), because of backtracking. The content-based outperforms the non content-based organization, since due to clustering of nodes with similar content, it locates the correct cluster (hierarchy) that contains matching documents faster. The number of hops remains constant as the number of nodes increases, because the number of matching nodes increases analogously.

In the next experiment (Fig. 7(right)), we keep the size of the network fixed to 200 nodes and vary the maximum number of hops a query makes from 20 to

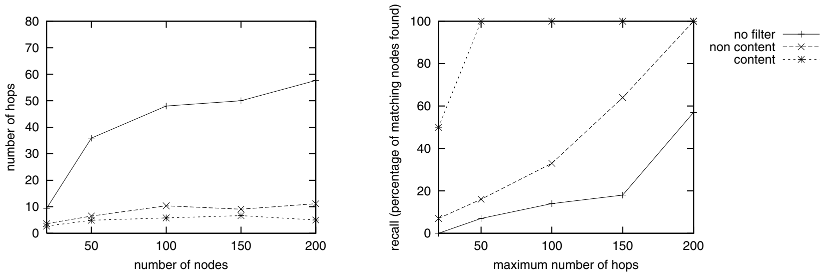


Fig. 7. Content vs non content-based organization: (left) finding the first matching node and (right) percentage of matching nodes found for a given number of hops (recall)

200. Note that in the number of hops, the hops made during backtracking are also included. We are interested in recall, that is, the percentage of matching nodes that are retrieved (over all matching nodes) for a given number of nodes visited. Again, the approach without filters has the worst performance since it finds only about 50% of the results for even 200 hops. The content-based organization outperforms the non content-based one. After 50 hops, that is, 25% of all the nodes, it is able to find all matching nodes. This is because when the first matching node is found, the other matching nodes are located very close, since nodes with similar content are clustered together.

We now vary the number of matching nodes from 1% to 50% of the total number of system nodes and measure the hops for finding the first matching node. The network size is fixed to 100 nodes. Our results (Fig. 8(left)) show that for a small number of matching nodes, the content-based organization outperforms further the other ones. The reason is that it is able to locate easier the cluster with the correct answers. As the number of results increases both the network proximity and the filter-less approaches work well as it becomes more probable that they will find a matching node closer to the query's origin since the documents are disseminated randomly.

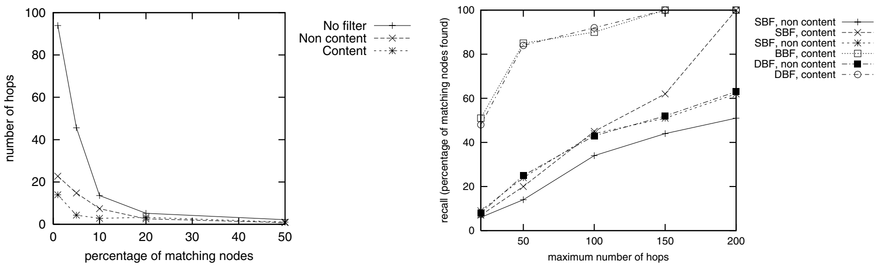


Fig. 8. (left) Number of hops to find the first result with varying number of matching nodes and (right) recall with multi-level filters

Using multi-level filters. We repeated the previous experiments using multi-level filters [24] and path queries of length 2. Our results confirm that multi-level Bloom filters perform better than simple Bloom filters in the case of path queries and for both a content and a non content-based organization. Figure 8(right) reports recall while varying the maximum number of hops.

5.3 Updates

In this set of experiments, we compare the performance of the CountSum and BitSum update propagation methods. We again simulated a network of nodes forming hierarchies and use Bloom filters for query routing. We used two metrics to compare the two algorithms: the number and size of messages. Each node stores 5 documents and an update operation consists of the deletion of a document and the insertion of a new document in its place. The deleted document is 0% similar to the inserted document to inflict the largest change possible to the filter. Again, we use small documents and correspondingly small sizes for the filters. The origin of the update is selected randomly among the nodes of the system. Table 3 summarizes the parameters used.

Table 3. Additional update propagation parameters

Parameter	Default Value	Range
# of XML documents per node	5	-
Total size of filter	4000	-
# of updates	100	-
Number of nodes	200	20-200
Repetition between deleted and inserted document	0%	-

Number and average size of messages. We vary the size of the network from 20 to 200 nodes. We use both a content-based and a non content-based organization and simple Bloom Filters. The BitSum method outperforms CountSum both in message complexity and average size of messages (Fig 9). The decrease in the number of messages is not very significant; however the size of the messages is reduced to half. In particular, CountSum creates messages with a constant size, while BitSum reduces the size of the message at every step of the algorithm. With a content-based organization, the number of messages increases with respect to the non content-based organization. This is because the content-based organization results in the creation of a larger number of more unbalanced hierarchies. However, both organizations are able to scale to a large number of nodes, since the hierarchical distribution of the filters enables performing updates locally. Thus, even for the content-based organization, less than 10% of the system nodes are affected by an update.

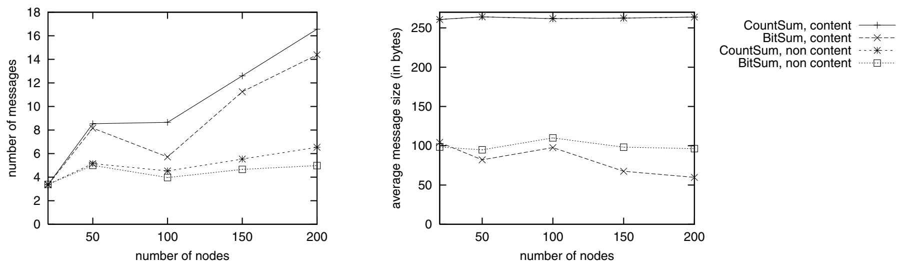


Fig. 9. BitSum vs CountSum update propagation: (left) number of messages (right) average message size

Using multi-level Bloom filters. We repeat the previous experiment using multi-level Bloom filters as summaries. We use only the BitSum method that outperforms the CountSum method as shown by the previous experiment. We used Breadth (BBFs) and Depth Bloom Filters (DBFs), both for a content and a non content-based organization. The nodes vary from 20 to 200. The results (Fig. 10) show that BitSum works also well with multi-level Bloom filters. The content-based organization requires a larger number of messages because of the larger number of hierarchies created. DBFs create larger messages as the bits affected by an update are more. However with the use of BitSum, DBFs scale and create update messages of about 300 bytes (while for CountSum, the size is 1K).

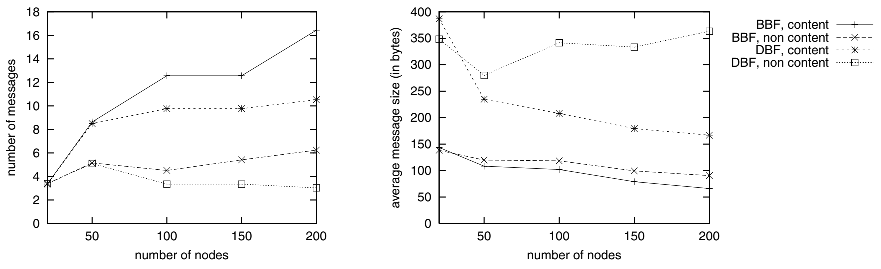


Fig. 10. BitSum update propagation with multi-level Bloom filters: (left) number of messages (right) average message size

6 Related Work

We compare briefly our work with related approaches regarding XML indexes and the use of Bloom filters for query routing. A more thorough comparison can be found in [24]. Various indexing methods for indexing XML documents (such as

DataGuides [9], Patricia trees [10], XSKETCH [11] and signatures [12]) provide efficient ways of summarizing XML data, support complex path queries and offer selectivity estimations. However, these structures are centralized and emphasis is given on space efficiency and I/O costs. In contrast, in a P2P context, we are interested in small-size summaries of large collections of XML documents that can be used to filter out irrelevant nodes fast with the additional requirements that such summaries can be distributed efficiently. Finally, when compared to Bloom filters, merging and updating of path indexes is more complicated.

Perhaps the resource discovery protocol most related to our approach is the one in [5] that uses simple Bloom filters as summaries. Servers are organized into a hierarchy modified according to the query workload to achieve load balance. Local and merged Bloom filters are used also in [14], but the nodes follow no particular structure. The merged filters include information about all nodes of the system and thus scalability issues arise. In both of the above cases, Bloom filters were used for keyword queries and not for XML data, whereas, our work supports path queries. Furthermore, the use of filters was limited to query routing, while we extend their use to built content-based overlay networks.

More recent research presents content-based distribution in P2P where nodes are "clustered" according to their content. With Semantic Overlay Networks (SONs) [15], nodes with semantically similar content are grouped based on a classification hierarchy of their documents. Queries are processed by identifying which SONs are better suited to answer it. However, there is no description of how queries are routed or how the clusters are created and no use of filter or indexes. An schema-based (RDF-based) peer-to-peer network is presented in [16]. The system can support heterogeneous metadata schemes and ontologies, but it requires a strict topology with hypercubes and the use of super-peers, limiting the dynamic nature of the network.

7 Conclusions and Future Work

In this paper, we study the problem of routing path queries in P2P systems of nodes that store XML documents. We introduce two new hash-based indexing structures, the Breadth and Depth Bloom Filters, which in contrast to traditional hash based indexes, have the ability to represent path expressions and thus exploit the structure of XML documents. Our experiments show that both structures outperform a same size simple Bloom Filter. In particular, for only 2% of the total size of the documents, multi-level Bloom filters can provide efficient evaluation of path queries for a false positives ratio below 3%. In general Breadth Bloom filters work better than Depth Bloom filters, however Depth Bloom filters recognize a special type of path queries. In addition, we introduce BitSum, an efficient update propagation method that significantly reduces the size of the update messages. Finally, we present a hierarchical organization that groups together nodes with similar content to improve search efficiency. Content similarity is related to similarity among filters. Our performance results confirm that an organization that performs a type of content clustering is much more efficient when we are interested in retrieving a large number of relevant documents.

An interesting issue for future work is deriving a method for self-organizing the nodes by adjusting the threshold of the hierarchies. Other important topics include alternative ways for distributing the filters besides the hierarchical organization and using other types of summaries instead of Bloom filters.

References

1. D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, Z. Xu. Peer-to-Peer Computing, HP Laboratories Palo Alto, HPL-2002-57.
2. S.D. Gribble, E.A. Brewer, J.M. Hellerstein, D. Culler. Scalable Distributed Data Structures for Internet Service Construction. In OSDI 2000.
3. B. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. CACM, 13(7), July 1970.
4. L. Fan, P. Cao, J. Almeida, A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In SIGCOMM 1998.
5. T.D. Hodes, S.E. Czerwinski, B.Y. Zhao, A.D. Joseph, R.H. Katz. Architecture for Secure Wide-Area Service Discovery. In Mobicom 1999.
6. The MD5 Message-Digest Algorithm. RFC1321.
7. The Niagara generator, http://www.cs.wisc.edu/niagara
8. Napster. http://www.napster.com/
9. R. Goldman, J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In VLDB 1997.
10. B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, M. Shadmon. Fast Index for Semistructured Data. In VLDB 2001.
11. N. Polyzotis, M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In VLDB 2002.
12. S. Park, H J. Kim. A New Query Processing Technique for XML Based on Signature. In DASFAA 2001.
13. C. Faloutsos, S. Christodoulakis. Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation. ACM TOIS, 2(4), October 1984.
14. A. Mohan and V. Kalogeraki. Speculative Routing and Update Propagation: A Kundali Centric Approach. In ICC 2003.
15. A. Crespo, H. Garcia-Molina. Semantic Overlay Networks for P2P Systems. Submitted for publication.
16. W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, A. Loser. Super-Peer-Based Routing and Clustering Strategies for RDF-Based Peer-To-Peer Networks. In WWW 2003.
17. G. Koloniari, E. Pitoura. Bloom-Based Filters for Hierarchical Data. WDAS 2003.
18. G. Koloniari, Y. Petrakis, E. Pitoura. Content-Based Overlay Networks of XML Peers Based on Multi-Level Bloom Filters. VLDB International Workshop on Databases, Information Systems and Peer-to-Peer Computing, 2003.
19. A. Crespo, H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In ICDCS 2002.
20. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, In SIGCOMM 2001.
21. S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. A Scalable Content-Addressable Network. In SIGCOMM, 2001.

22. Knowbuddy's Gnutella FAQ, http://www.rixsoft.com/Knowbuddy/gnutellafaq.html
23. E. Pitoura, S. Abiteboul, D. Pfoser, G. Samaras, M. Vazirgiannis. DBGlobe: a Service-oriented P2P System for Global Computing. SIGMOD Record 32(3), 2003
24. G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems (extended version). Computer Science Dept, Univ. of Ioannina, TR-2003-12.