The Computer Journal, Vol. 47 No. 4, © The British Computer Society; all rights reserved

Filters for XML-based Service Discovery in Pervasive Computing

GEORGIA KOLONIARI AND EVAGGELIA PITOURA

Department of Computer Science, University of Ioannina, GR45110 Ioannina, Greece
Email: {kgeorgia, pitoura}@cs.uoi.gr

Pervasive computing refers to an emerging trend towards numerous casually accessible devices connected to an increasingly ubiquitous network infrastructure. An important challenge in this context is discovering the appropriate data and services. In this paper, we assume that services and data are described using hierarchically structured metadata. There is no centralized index for the services; instead, appropriately distributed filters are used to route queries to the appropriate nodes. We propose two new types of filter that extend Bloom filters for hierarchical documents. Two alternative ways are considered for building overlay networks of nodes: one based on network proximity and one based on content similarity. Content similarity is derived from the similarity among filters. Our experimental results show that networks based on content similarity outperform those formed based on network proximity for finding all matching documents.

Received 16 June 2003; revised 4 November 2003

1. INTRODUCTION

Pervasive computing refers to a strongly emerging trend towards numerous casually accessible, frequently mobile devices connected to a ubiquitous network infrastructure. In our research [1, 2], we are interested in all aspects of data management for pervasive computing, with the ultimate goal of building a dynamic, highly distributed, adaptive data management system for modeling, storing, indexing and querying data and services hosted by numerous, heterogeneous computing nodes. A central issue is discovering the appropriate data and services among the available 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 node stores uniform XML-based descriptions of its provided services and data to facilitate information exchange and sharing. Such XML documents must be efficiently indexed, queried and retrieved. A single query on a node may need results from a large number of others, and thus we need a mechanism that finds nodes that contain relevant data efficiently.

In this paper, we consider a purely distributed approach, in which each node stores filters for routing the query in the system. Each node maintains two types of filters, a local filter summarizing the documents stored locally in the node and one or more merged filters summarizing the documents of neighboring nodes. Each node uses its filters to route a query only to those nodes that may contain relevant data. Such filters should be small and scalable to a large number of nodes and data. Furthermore, since nodes will join and leave the system arbitrarily, these filters must support frequent updates.

Bloom filters have been used as summaries in such a context [3]. They are hash-based indexing structures designed to support membership queries [4]. However, Bloom filters are not appropriate for summarizing hierarchical data since they support only membership queries and fail to exploit the structure of data. To this end, we introduce two novel data structures, Breadth Bloom filters (BBFs) and Depth Bloom filters (DBFs), which are multilevel structures that support efficient processing of path expressions that exploit the structure of XML documents. Our experimental results show that both multilevel Bloom filters outperform a same-size traditional Bloom filter in evaluating path queries. Depth Blooms require much more space than Breadth Blooms in the general case but are suitable for handling particular kinds of queries for which Breadth Blooms perform poorly.

Two alternative ways are considered for building overlay networks of nodes: one based on network proximity and one based on content similarity. The similarity of the content (i.e. the local documents) of two nodes is defined based 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 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 much more efficient in finding all the results for a given query than the one based on network proximity. Although the two approaches perform similarly in discovering the first result, the content-based organization benefits from the content clusters that are created during the structuring of the network.

The remainder of this paper is organized as follows. In Section 2, we present the architecture of the system and the service discovery process. Section 3 introduces the two new Bloom-based summaries, namely Breadth and Depth Bloom, and their use in a pervasive computing context.

Section 4 provides a description of the implementation and our experimental results. In Section 5, we compare our work with related research. Finally, in Section 6 we offer conclusions and directions for future work.

2. DISCOVERING SERVICES

We consider a pervasive computing scenario in which each participating node either stores XML documents or XML-based descriptions of the services that it offers. In this setting, a key challenge is how to locate an appropriate service or document. We allow users to specify queries for services and documents 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 service or document, we propose using appropriately distributed data structures, called filters, to direct the query to the appropriate nodes.

2.1. XML-based service description and querying

In a pervasive computing environment, a huge number of datasets and services are hosted by numerous devices. Discovering the appropriate resources in this context is complicated by the fact that resources are stored in diverse formats. To alleviate the issue of heterogeneity and allow for a declarative specification of the requested resources, we assume that data are published and exchanged as XML [1, 2]. In particular, we assume that datasets are exported as XML documents and that XML-based descriptions of services are available. Such collections of documents are dynamic since new documents appear and disappear and nodes join and leave the system.

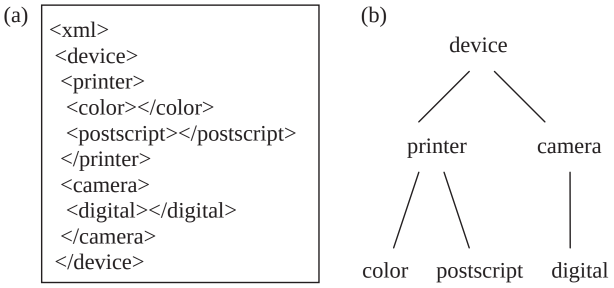
In particular, each node stores a collection of XML documents. An XML document comprises a hierarchically nested structure of elements that can contain other elements, character data and attributes. Thus, XML allows the encoding of arbitrary structures of hierarchical named values. This flexibility allows each node to create descriptions that are tailored to its services.

In our data model, an XML document is represented by a tree. Figure 1 depicts an XML service description for a printer and a camera provided by a node 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 simple path expressions expressed in an XPath-like query language.

DEFINITION 2. (path query) *A simple path expression query of length k has the form '$s_1 l_1 s_2 l_2 \cdots s_k l_k$', where each l_i is an element name and each s_i is either / or // denoting respectively parent–child and ancestor–descendant traversal.*



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

FIGURE 1. Example of (a) an XML document and (b) its tree.

Although most work on service discovery is limited in supporting membership queries, our work aims at extending these mechanisms to support the evaluation of path queries as well. Path queries are able to address the structure as well as the content of the documents without requiring the user to have knowledge about the schema that an XML document follows. In a pervasive system, there is no global schema, and documents at various nodes follow different schemas that a user is not able to know but at the same time should be able to query. Thus, choosing paths as index keys seems the appropriate choice in such an environment. Although Document Type Definitions (DTDs) could be used as index keys as well, they are too coarse and their use would make the queries very general, thus overwhelming the user with numerous irrelevant results.

We address the processing of queries that represent a path starting from the root element of the XML document (root paths), and queries that represent paths which can start from any element in the document (partial paths).

For a query q and a document D, we say that q is satisfied by D, or match(q, D) is true, if the path expression forming the query exists in the document; otherwise we have a miss. For example, the queries /device/printer and /device//digital are satisfied by the document of Figure 1, while for the query /device/digital, we have a miss.

2.2. Filters for service discovery

We propose maintaining specialized data structures that will summarize large collections of documents, to facilitate propagating the query only to those nodes that may contain relevant information. Such data structures should be much smaller than the data itself and should be lossless, i.e. if the data match the query, then the filter should match the query as well. In particular, each filter should support a filter-match operation that is fast and that if a document matches a query q then filter-match should also be true. If the filter-match returns false, then we have a miss.

DEFINITION 3. (filter) *A filter F for a set of documents D has the following property: for any query q, if filter-match$(q, F) = $false, then match$(q, d) = $false for every document d in D.*

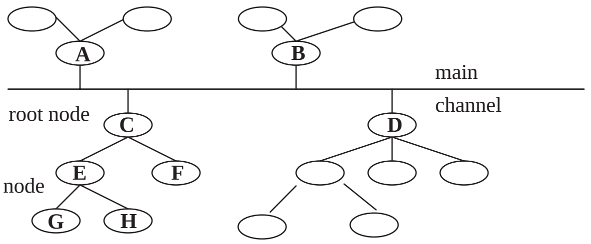


FIGURE 2. Hierarchical organization.

Note that the reverse does not necessarily hold. That is, if filter-match(q, F) is 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 for a set of documents D, filter-match(q, F) is true but there is no document d in D that satisfies q, that is for all d in D, match(q, d) is false. We are interested in filters for which the probability of false positive is small.

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 filters, called merged filters, for the documents of a set of its neighboring nodes. These merged filters facilitate the routing of a query only to nodes that may contain relevant data. In particular, 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. Note that we are interested in providing all the results for a query. Our mechanisms are easily extensible to locating the best k results.

When a node joins the system, it attaches to another node of the system, say node n, and sends its local filter to this node. Node n is responsible for propagating the filter of the new node to all other nodes that need to store information about the new node in their merged filters. The necessary filters of the neighboring nodes are also propagated back to the new node so that the new node can build its own merged filter.

When an update occurs, the node responsible for the update first updates its local filter. Then it updates its merged filter and propagates the changes to all nodes that hold information about it. This is done either by a multicast by the updating node or by a propagation procedure followed by the neighboring nodes, where each one informs the next.

Based on how the set of neighboring nodes for which we maintain summarized data is defined, we can consider many different node organizations. In the next section, we describe an organization based on hierarchies.

2.2.1. Hierarchical organization

There are various topologies that we can use to organize the nodes in a pervasive system. In the hierarchical organization (Figure 2), a set of nodes designated as root nodes is connected to a main channel that provides communication among them.

Each node maintains two filters: one for the local documents, called local filter, and if it is a non-leaf node, one with summarized data for all nodes in its subtree, called merged filter. In addition, root nodes maintain merged filters for the other root nodes in the system. The propagation of filters follows this bottom-up procedure:

(i) The leaf nodes send their local filters to their parent.
(ii) Every non-leaf node, after receiving the filters of all its children, merges them and produces its merged filter.
(iii) Every non-leaf node, after computing its own merged filter, merges it with its local filter and sends the resulting filter to its parent.
(iv) When a root node has computed its merged filter, it propagates it to all other roots on the main channel.

With the hierarchical organization, nodes belonging to the top levels have greater responsibilities, while nodes at lower levels are burdened with fewer tasks to perform. Thus, a hierarchical organization is best suited when the participating nodes have different processing and storage capabilities as well as different stability properties. Stability refers to how long a node stays in the system, e.g. in pervasive computing, some nodes such as workstations may stay longer online, while others such as laptops only stay online for a limited time. In a hierarchical organization, more stable and powerful nodes can be located at the top levels of the hierarchies, while less powerful and unstable nodes can be accommodated in the lower levels of the hierarchies.

When a query is issued at a node n, the search algorithm proceeds with the following steps:

(i) First, the local filter of node n is checked and if we have a match the local documents are checked.
(ii) Next, the merged filter is checked and if there is a match, the query is propagated to the node's children.
(iii) Also, the query is propagated to the node's parent.
(iv) The propagation of a query towards the bottom of a hierarchy ends when it reaches a leaf node or when the merged filter of an internal node does not indicate a match.
(v) When a query reaches a root node, the root apart from checking the filter of its subtree, also checks the merged filters of the other root nodes and forwards the query only to the subtrees for which there is a match.
(vi) When a root node receives a query from another root node, it only propagates the query to its own subtree and not to other root nodes since the sender root has already seen to that.

The propagation of updates follows a similar procedure. In particular,

(i) Updates of local documents are propagated firstly to the associated local filter of the node.
(ii) Next, the updates are forwarded to its parent.
(iii) The parent updates its merged filter and propagates the update to its own parent.
(iv) The update procedure continues until the root node is reached.
(v) The root node sends the update to all other root nodes, which in turn update the corresponding merged filter.

In a pervasive system, we expect that nodes will move frequently from one point to another. In our work, we do not consider mobility issues; instead, we assume that mobility is handled by lower-level solutions, such as Mobile IP, and thus the address used by the resource discovery layer to contact a node is considered stable [5]. The issue of reorganizing the hierarchy based on the mobility of nodes is beyond the scope of this paper.

Although we describe a hierarchical organization, filters could be easily applied to other organizations as well. For instance, filters could be used in a super-peer architecture in which our root nodes are the super-peers and the other (non-root) nodes in each of the subtrees are interconnected so that they form some other topology, for instance a mesh as opposed to a tree. The filter distribution can be easily adopted to accommodate any other such organization. Preliminary results of the filter deployment in a non-hierarchical peer-to-peer system are reported in [6].

2.3. Proximity and content-based similarity

We propose two approaches for organizing the nodes in the hierarchy. The first approach is based on network proximity and the second one on filter similarity. The approaches refer to the way a node chooses its position in the overlay network when it joins the system.

The network proximity based approach organizes the nodes based on their proximity in the graph that represents the structure of the physical network. The motivation behind this organization is an effort to satisfy queries locally and minimize response time. In the hierarchical organization we presented above, when a new node joins the system,

(i) it broadcasts a join request to all nodes of the system;
(ii) the new node attaches as a child to the node that answers the fastest, i.e. the node closer to it based on network latency. We define this node as the winner node.

The approach based on content similarity organizes the nodes based on the similarity of their content, i.e. it attempts to group relevant nodes together. The motivation for this organization is to minimize the number of irrelevant nodes that process a query. 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. Furthermore, the filter comparison operation is more efficient than a direct comparison between two sets of documents. Documents with similar filters are expected to match similar queries.

The strategy we follow to organize the nodes based on content similarity is the following:

(i) A new node broadcasts a join request that contains also its local filter to all nodes in the system.
(ii) Every node that receives a join request compares the received local filter with its own and responds to the initial node with the measure of their filter similarity.
(iii) The node with the largest similarity measure is the winner node. The new node saves the response of the winner node.
(iv) Then the node compares the similarity measure of the winner node with a system-defined threshold. If the measure is larger than the threshold, the node joins as the child of the winner node; else the node becomes a root node.

3. BLOOM-BASED FILTERS FOR HIERARCHICAL DATA

Our filters for XML documents are based on 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 [4], Bloom filters have seen many uses such as web cache sharing [7], query filtering and routing [3, 8] and free-text searching [9].

We extend traditional Bloom filters so that they can be used on hierarchical documents. Then, we explain their distribution. To distinguish traditional Bloom filters from the extended ones, we call the former simple Bloom filters.

3.1. Simple Bloom filters

Consider a set $A = \{a_1, a_2, \dots, a_n\}$ of n elements. The idea (Figure 3) is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, h_1, $h_2, \dots, h_k$, each with range 1–m. For each element $a \in A$, the bits at positions $h_1(a), h_2(a), \dots, h_k(a)$ 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 $h_1(b), h_2(b), \dots, h_k(b)$. 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 $c(h_1(a)), c(h_2(a)), \dots, c(h_k(a))$ 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 filters for resource discovery in terms of scalability, extensibility and distribution. They are compact, requiring a small space overhead and easy to update with the use of counters. In addition, since they are bit vectors, it is very easy to merge them so as to construct the merged filters by just applying the bitwise OR between them. However, they do not support path queries as they have no means for preserving the structure of documents. To this end, we introduce multilevel Bloom filters. Other hash-based structures, such as signatures [10], have similar

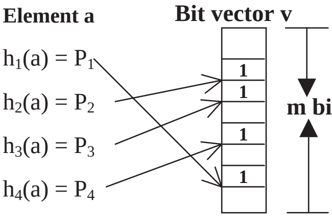


FIGURE 3. A Bloom filter with $k = 4$ hash functions.

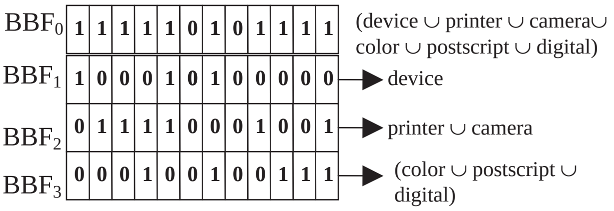


FIGURE 4. The BBF for the XML tree of Figure 1.

properties with Bloom filters, and our mechanisms can also be applied to extend signatures in a similar fashion.

3.2. Multilevel Bloom filters

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

Let T be an XML tree with j levels, and let the level of the root be level 1. The BBF for an XML tree T with j levels is a set of Bloom filters $\{\text{BBF}_0, \text{BBF}_1, \text{BBF}_2, \ldots, \text{BBF}_i\}$, $i \leq j$. 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 Figure 1 is a set of four Bloom filters (Figure 4).

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 size(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 $\text{size}(\text{BBF}_{i+1}) = d\ \text{size}(\text{BBF}_i)$, $(i < j)$, where d is the average degree of the nodes. For equal size BBF_is, BBF_0 is the logical OR of all BBF_is, $1 \leq i \leq j$.

DBFs provide an alternative way to summarize XML trees. We use different Bloom filters to hash paths of different lengths. The DBF for an XML tree T with j levels is a set of Bloom filters $\{\text{DBF}_0, \text{DBF}_1, \text{DBF}_2, \ldots, \text{DBF}_{i-1}\}, i \leq j$. 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 Figure 1 is a set of three Bloom filters (Figure 5). Note that we insert paths as a whole and we do not hash each element of the path separately; instead, we hash their concatenation.

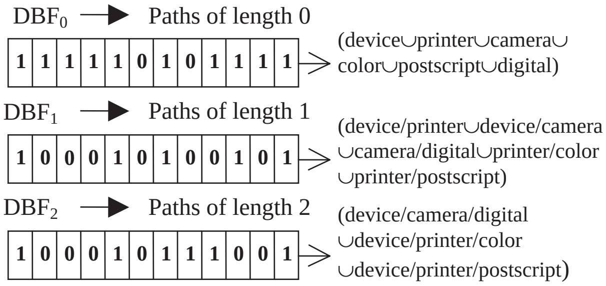


FIGURE 5. The DBF for the XML tree of Figure 1.

We use a different notation for paths starting from the root. This is not shown in Figure 5 for ease of presentation.

Regarding the size of the filters, as opposed to BBFs, 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 of order d^j for a tree with maximum degree d and j levels.

3.3. Multilevel Bloom match

We now describe the filter-match operation for multilevel Bloom filters and provide an estimation of the probability of false positives.

3.3.1. *Breadth Bloom filter-match*

The 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 elements, we proceed with examining the structure of the path. For a root query $/a_1/a_2/\cdots/a_p$, every level i from 1 to p of the filter is checked for the corresponding a_i. The algorithm succeeds if we have a 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 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 ancestor-descendant axis //, 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.

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 $p(d-p+1) = O(dp)$. When the path contains the // axis, 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 $O(p)$.

3.3.2. *Depth Bloom filter-match*

The procedure that checks whether a DBF matches 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 subpath of the query from length 2 to p is checked at the corresponding level. If any of the subpaths does not exist, the algorithm returns a miss. For paths that include the ancestor-descendant axis //, 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.

Consider a query of length p. Let p be smaller than the number of the filter's levels. First 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 look-up procedure is $p + p - 1 + p - 2 + \cdots + 1 = p(p+1)/2 = O(p^2)$. This is the worst case complexity as the algorithm exits if we have a miss at any step. The complexity remains the same with // axis 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 $O(p_1^2) + O(p_2^2) < O(p^2)$.

3.3.3. *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 [4]: $P = (1 - e^{-kn/m})^k$.

Using BBFs, a new kind of false positive appears. Consider the tree of Figure 1 and the path query /device/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 // axis. 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 these paths belong to the filter, and 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 [11].

3.4. Merged Bloom filters and content similarity

Each node maintains a multilevel Bloom filter for the documents it stores locally. It also maintains a multilevel Bloom merged filter for a set of its neighboring nodes. This merged filter facilitates the routing of a query only to nodes that may contain relevant data. When a query reaches a node, the node checks its local Bloom filter and uses the merged filter to direct the query to other nodes. To calculate the merged multilevel Bloom filter of a set of multilevel Bloom filters, we take the bitwise OR for each of their levels. In particular, the merged filter, Sum_BBF, of two Breadth Bloom filters BBF^k and BBF^m with i levels is a Breadth Bloom filter Sum_BBF = {Sum_BBF_0, Sum_BBF_1, ..., Sum_BBF_i} with i levels, where $Sum\_BBF_j = BBF_j^k$ BOR BBF_j^m, $0 \leq j \leq i$, and BOR stands for bitwise OR. Similarly, the merged filter of two DBFs is computed by applying the bitwise OR per level of the two filters.

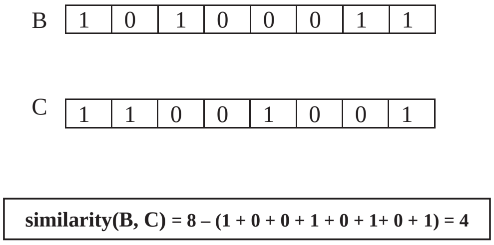


FIGURE 6. Bloom filter similarity.

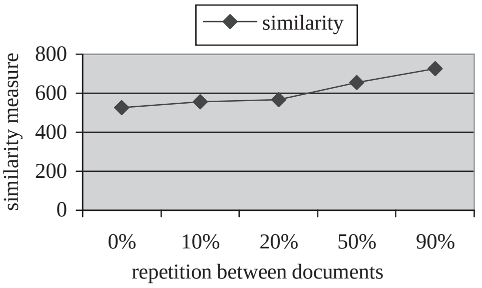


FIGURE 7. Varying documents' similarity.

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. Our similarity measure is based on the well known Manhattan distance metric. Let B and C be two filters of size m; their Manhattan distance (or Hamming distance) d(B, C) is defined as d(B, C) = |B[1]− C[1]| + |B[2] − C[2]| + $\cdots$ +| B[m] − C[m]|. We define the similarity, similarity(B, C), of two simple Bloom filters B and C of size m as follows: similarity(B, C) = $m - d$(B, C). The larger their similarity, the more similar the filters are. Figure 6 shows an example for two filters of size 8. In the case of multilevel Bloom filters, we take the sum of the similarities of every pair of corresponding levels. To compute the similarity of two filters, we simply take the equivalence (exclusive NOR) of the bit vectors that correspond to each level of the filter.

Figure 7 illustrates an experiment that confirms the validity of the measure. We used different percentage of element repetition between documents and measured their similarity. Similarity increased linearly with the increase in the repetition between the documents. The same holds for the similarity between multilevel Blooms, although in this case the measure depends on the structure of the documents as well.

3.5. Compression

Bloom filters have a great deal of potential for distributed protocols where systems need to share information about their available data. In this situation, Bloom filters play a dual role. They are both a data structure being used at the nodes, and a message being passed between them. When we use Bloom filters as a data structure, we may tune their

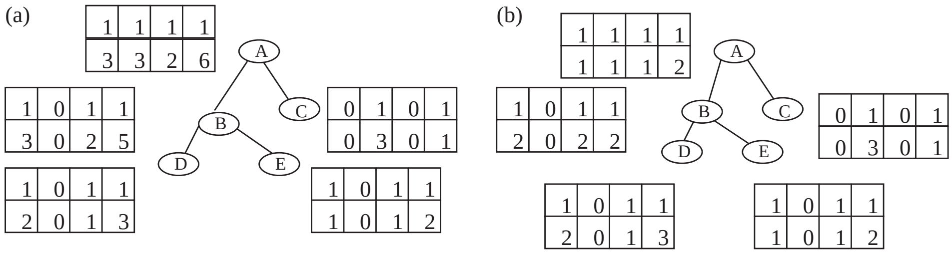


FIGURE 8. Update propagation using (a) the straightforward way and (b) the improvement.

parameters for optimal performance as a data structure; i.e. we minimize the probability of a false positive for a given memory size and number of items. If it is also being passed around as a message, however, then it is useful to introduce another performance measure: transmission size. Transmission size may be of greater importance when the amount of network traffic is a concern but there is memory available at the endpoint machines. This is especially true in distributed systems where information must be transmitted repeatedly from one node to many others. Transmission size can be affected by using compression. Compressing a Bloom filter can lead to improved performance. By using compressed Bloom filters, protocols reduce the number of bits broadcast, the false positive rate and/or the amount of computation per lookup. The tradeoff costs are the increased processing requirement for compression and decompression and larger memory requirements at the endpoint machines, which may use a larger original uncompressed form of the Bloom filter in order to achieve improved transmission size.

Large sparse Bloom filters can be greatly compressed. Theoretically, an m-bit filter can be compressed to $mH(p)$ bits, where p is the probability that a bit in the filter is 0 and $H(p) = -p\log_2 p - (1-p)\log_2(1-p)$ is the entropy function. For sufficiently large filters, arithmetic coding guarantees close to optimal compression, and so if p is small enough, $H(p)$ is much smaller than 1, and significant savings in the transmission size can be achieved [12].

3.6. Updates

When a document is updated or a document is inserted or deleted at a node, the local filter of that node must be updated. An update consists of a delete and an insert operation. When an update occurs at a node apart from the update of its local filter, all the merged filters that use this local filter should also be updated. We present two different approaches to update propagation based on the way the counters of the merged filters are used. Recall that with each bit of a local Bloom filter we associate a counter that counts how many times the corresponding bit was set to 1.

The straightforward way to use the counters at the merged filters is that every leaf node sends to its parent, along with its local 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 filters. An update in a local filter will result in an increase or decrease of some of the counters. We only need the differences to perform an update and the only time the filter itself is modified is when a counter turns from 0 to 1 and vice versa. Thus, whenever a node updates its local filter and its own merged filter to represent the changes, it also has to send the differences from its old and new counter values to its parent. After updating its own summary, the parent will propagate the filter further until all concerned 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 only have to send the levels of the counters that have changed and not the whole multilevel filter.

We can improve the complexity of the messages required if we make the following observation: an update will only result in a change in the filter if the counter turns from 0 to 1 or vice versa. Taking this into consideration, we slightly change the algorithm for computation of the counter for the merged filters. Each node just sends its merged filter to its parent (local filter for the leaf nodes) and not the associated 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 children filters that have set this bit to 1 and not how many times the original filter had set the bit to 1. When an update occurs, it has to be propagated only if it changes a bit from 1 to 0 or vice versa; thus the required messages are limited, as well as the size of the message that needs to be sent.

Let us consider the hierarchy in Figure 8. The merged summary counters are created in Figure 8a by the simple way of just taking the sum of the children counters, while in Figure 8b they are created by incrementing by 1 for every child that has the corresponding bit set. Let us assume that node D performs an update, its new filter becomes $(1, 0, 0, 1)$ and the corresponding counters $(1, 0, 0, 2)$. In Figure 8a, it will send the differences between the old and new counters $(-1, 0, -1, -1)$ to node B, whose summary will now become $(1, 0, 1, 1)$ and the counters $(2, 0, 1, 4)$. In contrast in Figure 8b, it will send only those bits that changed from 1 to 0 and vice versa, $(-, -, -1, -)$. The new summary of B will be $(1, 0, 1, 1)$ and the counters $(2, 0, 1, 2)$. While in the first case node B would have to propagate the update although no change was reflected to the actual filter, in the second case this is not necessary. Thus, the second approach sends both smaller and fewer messages.

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the proposed approach. We implemented both the BBF and DBF data structures, as well as a SBF (that just hashes all elements of a document) for comparison. For the hash functions, we used MD5 [13] which is a cryptographic message digest algorithm that hashes arbitrarily long 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 select MD5 because of its well known properties and relatively fast implementation. For generation of the XML documents, we used the Niagara generator [14] that generates tree-structured XML documents of arbitrary complexity. It allows the user to specify a wide range of characteristics for the generated data by varying a number of simple and intuitive input parameters, which control the structure of the documents and the repetition between the element names.

Two types of experiments were performed. The goal of the first set of experiments was to demonstrate the appropriateness of multilevel Blooms as filters of hierarchical documents. To this end, we evaluated the false positive probabilities for both DBFs and BBFs and compared them with the false positive probabilities for a same-size SBF for a variety of query workloads and document structures. The goal of the second set of experiments is to evaluate the performance of Bloom filters in a distributed setting using both the content-based and the proximity approaches.

4.1. Efficiency of multilevel Blooms as filters for path queries

In this set of experiments, we evaluate the performance of multilevel Blooms. 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. We limited the inserted paths in the Depth Bloom to be at most of length 3, i.e. the Depth Bloom has only three levels. Also, in the case of Breadth Bloom, we excluded the Bloom filter on top (BBF_0) that is only used for performance reasons, since it requires more space and would deteriorate Breadth's performance for a given space overhead. The repetition of the names of the elements was set to 0 between the elements of a single document as well as between all documents. Queries were generated by producing arbitrary path queries, with 90% elements from the documents and 10% random ones. All queries were partial paths and the probability of the // axis at each query was set to 0.05.

TABLE 1. Input parameters.

Parameter	Default value	Range
No. of XML documents	200	–
Total size of filter	78,000 bits	30,000–1,50,000 bits
No. of hash functions	4	–
No. of queries	100	–
No. of elements per document	50	10–150
No. of levels per document	4/6	2–6
Length of query	3	2–6
Distribution of queries' elements	90% exist in documents–10% random ones	0–10%

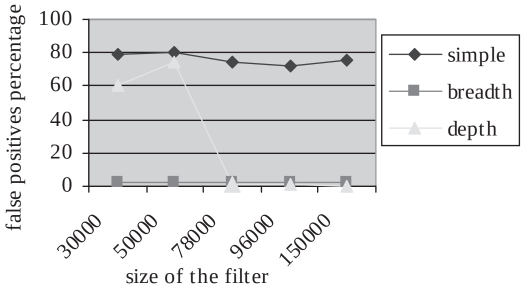


FIGURE 9. Experiment 1: size of the filters.

Experiment 1: Influence of filter size (Figure 9)
We examine the influence of the size of the filter with respect to false positives. Each document has 50 elements and four levels. The queries are of length 3. The size of the filters varies from 30,000 bits to 150,000 bits. The lower limit was chosen from the formula $k = (m/n)\ln 2$ 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 SBF. 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 recognize path expressions correctly.

The results show that both Breadth and Depth Blooms outperform Simple Blooms even for only 30,000 bits. In addition, in contrast with Simple Blooms, where the increase in the size results in no improvement in their performance, the multilevel 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 30,000 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.

Using the result of the first experiment, we choose as the default size of the filters for the rest of the experiments a

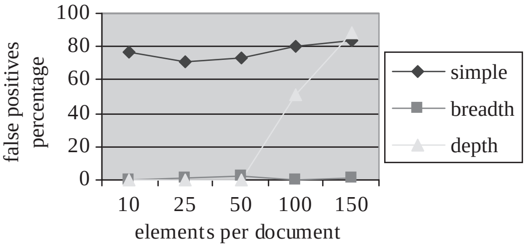


FIGURE 10. Experiment 2: number of elements.

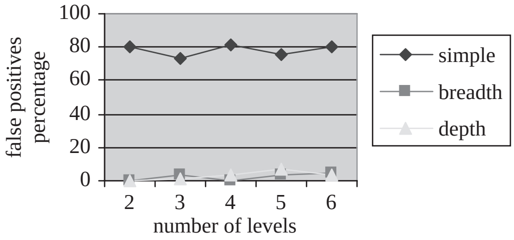


FIGURE 11. Experiment 3: number of levels.

size of 78,000 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 pervasive computing context.

Experiment 2: Influence of the number of elements per document (Figure 10)

We compare the filters with respect to the number of elements per document. The size of the filter is fixed at 78,000 bits, and the documents have four levels. Queries have length 3 and the number of elements per document varies from 10 to 150. 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 of recognizing 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 1–2% 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 document levels (Figure 11)

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 at 78,000 bits, and the documents have 50 elements. The levels vary from two to six. The queries are of length 3, except for the documents with two levels where we conduct the experiment with queries of length 2. The behavior of Simple Blooms is independent of the number of levels of the documents since they just hash all their elements irrespective 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 Simple Blooms, with a false positive percentage below 7%. Breadth Blooms perform better for four to five levels. This is because the elements are more evenly allocated to the levels of the filter, while for fewer levels the filter has also fewer 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 a 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 subpaths /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 subpaths up to length 3, while Breadth Blooms deteriorate further for six levels.

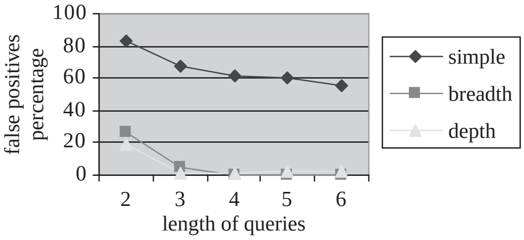


FIGURE 12. Experiment 4: varying query sizes.

Experiment 4: Influence of the length of the queries (Figure 12)

The parameter examined in this experiment is the length of the queries. The structure of the document is fixed, with four levels and 50 elements, for queries of length 2–4 and six levels for queries with length 5 and 6. The size of the filter is also fixed at 78,000 bits. Once again, both multilevel 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 lengths of 5 and 6 since for documents with six levels the number of inserted elements increases and the filter becomes denser.

The last two experiments also show that although 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 subpaths with length greater than 3) are not inserted. The checks of all possible subpaths up to length 3 are able to recognize most of the misses, and so we conclude that we can limit the number of levels of the filter

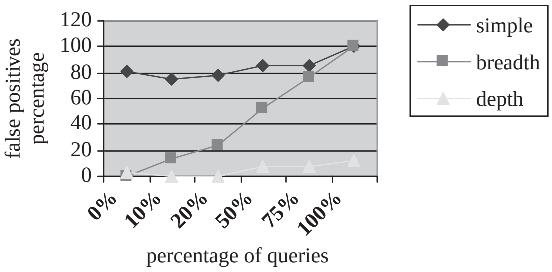


FIGURE 13. Experiment 5: varying the query workload.

without a significant loss in performance if we have limited space.

Experiment 5: Query workload (Figure 13)
In most of our experiments, Breadth Blooms seem to outperform Depth Blooms for just a fraction of the space the latter require. However, as Experiment 3 indicates, there are special forms of queries for which Depth Blooms work better. To clarify this, in this last experiment, we created a workload with queries consisting of such path expressions, i.e. a workload that favors DBFs. The percentage of these queries varied from 0% to 100% of the total workload. The size of the filter is fixed at 78,000 bits; the documents have four 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 with the percentage of these queries in the workload. However, Depth Blooms have no problem in recognizing this kind of false positives and show much better results. The slight increase in 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 Bloom 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.

Summary of results
Our experiments show that multilevel Bloom filters outperform SBFs, in evaluating path queries. In particular, for only 2% of the total size of the documents, multilevel Bloom filters can provide efficient evaluation of path queries for a false positive ratio below 3%, whereas Simple Blooms fail to recognize the correct paths, no matter how much the filter's size increases. In general, Breadth Blooms work better than Depth Blooms even for a very limited space. In contrast, Depth Blooms require much more space but are suitable for handling a special kind of queries for which Breadth Blooms present a high ratio of false positives, as we have explained in Experiment 5.

TABLE 2. Distribution parameters.

Parameter	Default value	Range
No. of XML documents per node	1	–
Total size of filter	200–800	–
No. of hash functions	4	
No. of queries	100	
No. of elements per document	10	
No. of levels per document	4	
Length of query	1–2	
Number of nodes	100	20–200
Out-degree of a node	2–3	
Percentage of repetition between documents	Every 10% of all docs 70% similar	
Levels of hierarchy	3–4	
Number of results	10% of no. of nodes	1–50%

4.2. Hierarchically distributed filters

In this set of experiments, we focus on filter distribution. Our performance metric is the number of hops for answering a query. We simulated a network of nodes forming hierarchies and examined its performance with and without the use of filters. We also compared the performance of both the proximity and the content-based organizations. In the first three experiments, we used SBFs as our filters and queries of length 1, for simplicity. We have already shown that multilevel Blooms outperform SBFs. Thus, they can be used instead of Simple Blooms for path queries with greater length. In the last experiments, we used multilevel Blooms to confirm this belief. For the experiments, we used small documents but we also decreased the size of the filter. To scale to large documents, we just have to scale-up the filter as well. There is one document at each node (for simplicity). Every 10% of the documents are 70% similar to each other. So we expect that about 10% of the documents satisfy each query. The origin of the query is selected randomly among the nodes of the network. For the content-based organization of the nodes, the threshold was preset so that we can determine the number of hierarchies created. Future work will include the tuning of the threshold according to the workload of the network so that the network can be self-organized. Table 2 summarizes our parameters.

Experiment 1: Finding the first result with varying number of nodes
At the first two experiments, we vary the size of the network, i.e. the number of participating nodes from 20 to 200. At this first experiment, we measured the number of hops a query makes to find its first result. We expect that about 10% of the nodes have this result.

Figure 14 illustrates our results. It is obvious that the use of summaries greatly improves the search performance. Without the use of filters, the hierarchical distribution performs worse than organizing the nodes in a linear chain, where the worst case would only be as much as the number of nodes. In this case the performance deteriorates because

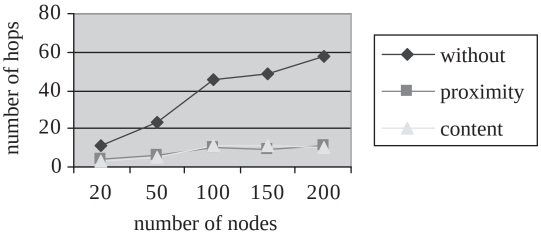


FIGURE 14. Experiment 1: finding the first result.

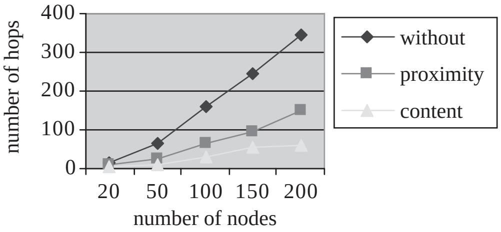


FIGURE 15. Experiment 2: finding all the results.

of backtracking. Both the content-based and the network proximity organizations show very good results, with almost identical behavior. The number of hops remains constant while the number of nodes increases because the number of results in the network increases analogously.

Experiment 2: Finding all the results with varying number of nodes (Figure 15)

At this second experiment the setup and the performance metric are exactly the same. Only, now we are interested in finding all the results and not just the first one. For finding all results, the content-based organization outperforms the one based on network proximity. This is because if we find the first answer (i.e. the first node with documents matching the query), we expect that the other answers (i.e. the other nodes with matching documents) will be located very close, due to the use of the similarity measure that clusters together nodes with similar documents. In contrast, this does not hold for the proximity organization since the topology is created randomly and not based on the document's content.

Experiment 3: Finding the first result with varying number of answers (Figure 16)

For this experiment, we varied the number of answers (nodes with matching documents) that exist in the network from 1% to 50% of the total number of nodes and measured the necessary hops for finding the first result. The network size was fixed at 100 nodes. Our results show that for a small number of matching nodes, the content-based organization outperforms 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 filterless approaches work well as it is more probable that they will find an answer closer to the query's origin since the documents are disseminated randomly.

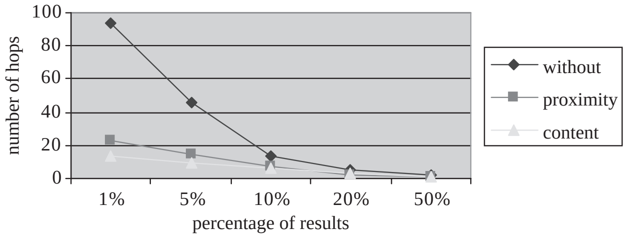


FIGURE 16. Experiment 3: varying the number of results.

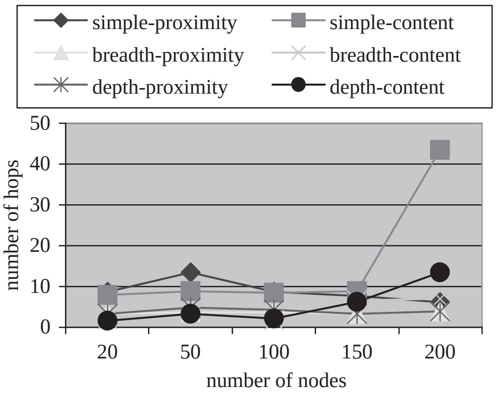


FIGURE 17. Finding the first result.

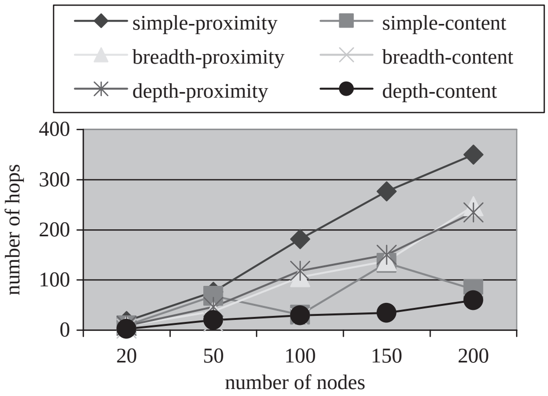


FIGURE 18. Finding all the results.

Experiment 4: Using multilevel filters as summaries

In this experiment, we repeated the first two experiments using multilevel filters. The nodes vary from 20 to 200. We measure the number of necessary hops. We compared Breadth and Depth summaries with the simple ones, both for a network-proximity and a content-based organization of the network. The queries were of length 2.

Figure 17 illustrates the results when we are interested in finding the first result, while Figure 18 illustrates them when we are interested in all the results. Around 20% of the documents do not contain the elements of the queries, while in 70% of the documents, the elements exist but do not form the correct path. Only in 10% of the documents does the path exist. We used 800 bits as the size for the filters to eliminate false positives that could lead to unnecessary hops.

Simple Blooms with the network proximity organization behave identically to the filter-less approach because in every node we have a false hit that forces the query to follow most of the paths. In the content-based organization, Simple Blooms behave slightly better because of the organization of the documents in clusters. Breadth and Depth Blooms are very efficient, with both proximity and content-based organizations of the network. Depth Blooms are not able to scale as well as Breadth Blooms as discussed by us in the first section of the experiments. The content-based organization outperforms the one based on proximity in finding all the results as we have seen for Simple Blooms in Experiment 2 as well.

Our results show that multilevel Blooms can be used as a distributed index and the similarity metric is suitable for them as well. Simple Blooms cannot be used for path queries because of the false positives that deteriorate their performance.

Summary of results

As our experimental results show, the content-based organization is much more efficient in finding all the results for a given query than the network proximity organization. Although the two approaches perform similarly in discovering the first result, the content-based organization benefits from the content clusters that were created during the structuring of the network. Furthermore, the content-based organization outperforms the network proximity one when the nodes that satisfy a given query are limited. The content-based organization is able to route the query to the right cluster faster, while the network proximity organization just has to traverse the hierarchies until it finds a result.

Our experiments showed that both Simple and multilevel Blooms can be efficiently used as distributed filters. Once again, in the case of path queries, multilevel Blooms outperform Simple ones, with both a proximity and a content-based organization. The Simple Bloom filters' performance deteriorates because of the false positives.

However, we have to note that using the number of hops as a performance metric factors out the fact that nodes that are neighbors in a content-based organization may be many nodes away in the actual physical network, whereas neighbors in a network proximity organization are expected to be neighbors in the physical network as well. Thus, the relative performance of the network proximity and content-based organizations in terms of other metrics such as response time should also take into account the physical network characteristics.

5. RELATED WORK

In this paper, we have proposed an approach for routing path queries over a large-scale network of nodes storing hierarchical documents. We consider two lines of research related to our work: research on indexing XML documents and research on resource discovery in large distributed systems.

5.1. Indexing XML documents

Many researchers have developed various indexing methods for XML documents. These methods provide efficient ways of summarizing XML documents, support complex path queries and offer selectivity estimations for a given query. However, these structures are centralized and emphasis is given on space efficiency and I/O costs for the various operations. On the other hand, in a pervasive computing context, we are interested in small-size summaries of a large collection of XML documents that can be used to provide a fast answer on whether at least one of the documents in the collection satisfies the query with the additional requirement that such summaries can be distributed efficiently. Below, we survey some centralized XML indexing methods.

DataGuides [15] are one of the most popular XML indexes, consisting of a tree constructed by a graph model of the XML data. They are also able to store statistical information and sample values which they use for query optimization. The method presented in [16] encodes paths in the data as strings and inserts these strings into an indexing structure based on Patricia tries. Evaluating queries involves encoding the desired path as a search key string and performing a lookup in the index. The XSKETCH synopsis [17] 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 and there is no mention on how updates are handled. APEX [18] 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 [19] has a path for every distinct sequence of tags in the document. Statistical information about the elements is also stored at each node. If it exceeds main memory space, the nodes with the lowest frequency are deleted. In [20], a signature is attached to each node of the XML tree, in order to prune unnecessary subtrees as early as possible while traversing the tree for a query. This technique is used for evaluating regular path expressions, and it requires a small overhead in space and computation.

5.2. Resource discovery

In pervasive computing and more recently in the context of peer-to-peer computing, many methods have been developed in order to find the nodes that contain data relevant to a query. These methods construct indexes that store summaries of other nodes and additionally provide routing protocols to propagate the query to the relevant nodes. In this line of research, emphasis is given to the distribution of the summaries across the nodes of the network. However, these structures answer simple queries that consist of combinations of attribute–value pairs and do not address the evaluation of path expressions. Furthermore, they are based only on network proximity.

Perhaps the resource discovery protocol most related to our approach is the one in [3]. The protocol uses SBFs as summaries. Servers are organized into a hierarchy modified according to the query workload for load balance. Each server stores summaries, a single filter with all the subset hashes of the XML service descriptions up to a certain threshold, which are used for query routing. To evaluate a query, it is split to all possible subsets and each one is checked in the index. Another method based on Bloom filters for routing queries in peer-to-peer networks is presented in [21]. It is based on a new structure, called attenuated Bloom filter, residing in every node of the system. The filter stores information about nodes within a range of the local node and uses a probabilistic algorithm in order to direct a query. The algorithm either finds results quickly or fails quickly and exhaustive searching is then deployed.

In [22], the idea is to make use of data items that change infrequently and often appear in queries, such as metadata and words characteristic of a specific node. Indexes maintained at each node that map this data to the corresponding nodes are used to direct queries. When nodes join the system, they exchange data that allow for the construction of the indexes, which are inverted indexes that map keywords to nodes. A similar approach [23] uses routing indexes placed at each node for efficient routing of queries. By keeping an index for each outgoing edge, a node can choose the best neighbor for forwarding a query. The choice is based on summarized data about the documents along that path, which are stored in the index.

INS/Twine [24] is an approach to scalable intentional resource discovery where resolvers collaborate as peers to distribute resource information and to resolve queries. The approach relies on a distributed hash table process (such as Chord [25]), which it uses to distribute the resource descriptions among the resolvers. Each description is represented in an XML-based language that represents hierarchies of attribute–value pairs. The description is mapped to a tree and each distinct prefix of the tree is considered as an index key that is distributed among the resolvers. The queries are routed with the use of the distributed hash table. The approach is able to handle small resource descriptions, while for large XML documents because of the large number of extracted index keys, the method would not be able to scale, unlike multilevel Blooms that are compact and can easily scale to large documents. Furthermore, since the method uses a distributed hash table, it imposes on the nodes which data items to store, thus limiting their autonomy.

VIA [26] is an application-level protocol for service discovery. Services are described through metadata tags, an ordered list of attributes with a finite set of values. Gateways are responsible for query routing and all services are advertised to them. They are hierarchically structured and the ‘root’ gateways are listening to a main channel similarly to our approach. All queries are sent to the main channel. VIA provides a mechanism for self-organization of the gateways. A gateway that processes too many irrelevant queries can attach to another gateway as a child, thus forming hierarchies that filter out irrelevant queries level by level. As in our approach, the top-level gateways are burdened with most of the work. A gateway chooses the hierarchy that it attaches to by recording information about the query workload through the ordering of the metadata tags. By generalizing its tags, it transforms its filter to a less restrictive one and chooses to join to the hierarchy that best fits this filter. The main restriction of VIA is that all the attributes describing the services should be known and ordered, and thus it is difficult to add new services in the system. In addition, since all queries are issued to the main channel this produces a large overhead in communications, while in our system the queries are first attempted to be satisfied locally.

Furthermore, all the above approaches organize their indexes without taking into account the content of the nodes, in contrast with our approach that uses the filter similarity to provide a content-based clustering of the nodes. Content-based distribution was recently proposed in [27], which introduced semantic overlay networks (SONs). With SONs, nodes with semantically similar content are ‘clustered’ together, based on a classification hierarchy of their documents. Queries are processed by identifying which SONs are better suited to answer it. However, SONs provide no description of how queries are routed or how the clusters are created and there is no use of filters or indexes.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce two new hash-based indexing structures, based on Bloom filters, which in contrast to traditional hash-based indexes have the ability to represent path expressions and fully exploit the structure of XML documents. These indexing structures, called BBFs and DBFs, are multilevel structures that consist of SBFs and share their ability to store a large volume of data within limited space. We have described the corresponding algorithms for insertion, update and query evaluation in these structures. The algorithms were implemented and the structure performance was compared with that of the SBFs. Both structures outperform SBFs, with the Breadth Bloom having the best performance even with small memory requirements in most cases.

We also presented how these structures can be distributed and used for resource discovery in pervasive computing. Two alternative ways were presented for building overlay networks of nodes: one based on network proximity and one on content similarity. Content similarity was related to similarity among filters. We simulated this distributed environment using both the content-based and network proximity organizations. The use of Bloom filters improves significantly the performance of the discovery process. Furthermore, the content-based organization that performs a type of content clustering is much more efficient when we are interested in finding not just one but k results of a query.

Future work will include the extension of the structures to incorporate values in the path expressions and an extension of the data model that includes XML documents that can be represented as graphs. Another issue is studying alternative ways for distributing the filters besides the hierarchical organization and using other types of summaries besides Bloom filters. Finally, we wish to develop a method for self-organization of the nodes for the content-based organization, by adjusting the threshold for the hierarchies.

ACKNOWLEDGEMENTS

This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-32645 DBGlobe project.

REFERENCES

[1] The DBGlobe Project, http://softsys.cs.uoi.gr/dbglobe.

[2] Pitoura, E., Abiteboul, S., Pfoser, D., Samaras, G. and Vazirgiannis, M. (2003) DBGlobe: a service oriented P2P system for global computing. *ACM Sigmod Record*, **32**(3), 77–82.

[3] Hodes, T. D., Czerwinski, S. E., Zhao, B. Y., Joseph, A. D. and Katz, R. H. (1999) An architecture for secure wide-area service discovery. In *Proc. 5th Ann. ACM/IEEE Int. Conf. on Mobile Computing and Networking '99*, Seattle, Washington, August 15–19, pp. 24–35. ACM Press, New York, NY, USA.

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

[5] Pitoura, E. and Samaras, G. (2001) Locating objects in mobile computing. *IEEE Trans. Knowl. Data Eng.*, **13**(4), 571–592.

[6] Koloniari, G., Petrakis, Y. and Pitoura, E. (2003) Content-based overlay networks of XML peers based on multi-level bloom filters. In *Proc. VLDB Int. Workshop on Databases, Information Systems and Peer-to-Peer Computing*, Berlin, Germany, September 7–8, pp. 232–247. Springer-Verlag, Berlin.

[7] Fan, L., Cao, P., Almeida, J. and Broder, A. (1998) Summary cache: a scalable wide-area Web cache sharing protocol. In *Proc. ACM SIGCOMM Conf. '98*. Vancouver, British Columbia, Canada, August 31–September 4, pp. 254–265. ACM Press, New York, NY, USA.

[8] Gribble, S. D., Brewer, E. A., Hellerstein, J. M. and Culler, D. (2000) Scalable distributed data structures for internet service construction. In *Proc. 4th Symp. on Operating Systems Design and Implementation '00*, San Diego, CA, October 22–25, pp. 319–332. USENIX.

[9] Ramakrishna, M. V. (1989) Practical performance of Bloom filters and parallel free-text searching. *Commun. ACM*, **32**(10), 1237–1239.

[10] Faloutsos, C. and Christodoulakis, S. (1984) Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. Office Inform. Syst.*, **2**(4), 267–288.

[11] Koloniari, G. and Pitoura, E. (2003) Bloom-based filters for hierarchical data. In *5th Workshop on Distributed Data Structures and Algorithms (WDAS'03)*, Thessaloniki, Greece, June 13–14. Carleton Scientific.

[12] Mitzenmacher, M. (2001) Compressed Bloom filters. In *Proc. 20th Ann. ACM Symp. on Principles of Distributed Computing '01*, Newport, Rhode Island, August 26–29, pp. 144–150. ACM Press, New York, NY, USA.

[13] The MD5 Message-Digest Algorithm, RFC1321.

[14] The Niagara Generator, http://www.cs.wisc.edu/niagara.

[15] Goldman, R. and Widom, J. (1997) DataGuides: enabling query formulation and optimization in semistructured databases. In *Proc. 23rd VLDB Conf.'97*, Athens, Greece, August 25–29, pp. 436–445. Morgan Kaufmann, San Francisco, CA, USA.

[16] Cooper, B. F., Sample, N., Franklin, M. J., Hjaltason, G. R. and Shadmon, M. (2001) Fast index for semistructured data. In *Proc. 27th VLDB Conf. '01*, Rome, Italy, September 11–14, pp. 341–350. Morgan Kaufmann, San Francisco, CA, USA.

[17] Polyzotis, N. and Garofalakis, M. (2002) Structure and value synopses for XML data graphs. In *Proc. 28th VLDB Conf. '02*, Hong Kong, China, August 20–23, pp. 466–477. Morgan Kaufmann, San Francisco, CA, USA.

[18] Chung, C. W., Min, J. K. and Shim, K. (2002) APEX: an adaptive path index for XML data. In *Proc. ACM SIGMOD 2002*, Madison, Wisconsin, June 3–6, pp. 121–132. ACM Press, New York, NY, USA.

[19] Aboulnaga, A., Alameldeen, A. R. and Naughton, J. F. (2001) Estimating the selectivity of XML path expressions for Internet scale applications. In *Proc. 27th VLDB Conf. '01*, Rome, Italy, September 11–14, pp. 591–600. Morgan Kaufmann, San Francisco, CA, USA.

[20] Park, S. and Kim, H. J. (2001) A new query processing technique for XML based on signature. In *Proc. 7th Int. Conf. on Database Systems for Advanced Applications (DASFAA'01)*, Hong Kong, China, April 18–21, p. 22. IEEE Computer Society, Washington, DC.

[21] Rhea, S. C. and Kubiatowicz, J. (2002) Probabilistic location and routing. In *Proc. 21st Ann. Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM'02)*, New York, NY, USA, June 23–27, pp. 1248–1257. IEEE Computer Society, Washington, DC.

[22] Galanis, L., Shawn, Y. W., Jeffery, R. and DeWitt, D. J. (2003) Processing queries in a large peer-to-peer system. In *Proc. Advanced Information Systems Engineering, 15th Int. Conf. (CAiSE 2003)*, Klagenfurt, Austria, June 16–18, pp. 273–288. Springer-Verlag, Berlin.

[23] Crespo, A. and Garcia-Molina, H. (2002) Routing indices for peer-to-peer systems. In *Proc. 22nd Int. Conf. on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2–5, p. 23. IEEE Computer Society, Washington, DC.

[24] Balazinska, M., Balakrishnan, H. and Karger, D. (2002) INS/Twine: a scalable peer-to-peer architecture for intentional resource discovery. In *Proc. First Int. Conf. on Pervasive Computing*, Zurich, Switzerland, August 26–28, pp. 195–210. Springer-Verlag, Berlin.

[25] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. and Balakrishnan, H. (2001) Chord: a scalable peer-to-peer lookup service for internet applications. In *Proc. 2001 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (ACM SIGCOMM 2001)*, San Diego, CA, August 27–31, pp. 149–160. ACM Press, New York, NY, USA.

[26] Castro, P., Greenstein, B., Muntz, R., Bisdikian, C., Kermani, P. and Papadopouli, M. (2001) Locating application data across service discovery domains. In *Proc. 7th Ann. Int. Conf. on Mobile Computing and Networking*, Rome, Italy, July 16–21, pp. 28–42. ACM Press, New York, NY, USA.

[27] Crespo, A. and Garcia-Molina, H. (2004) Semantic overlay networks for P2P systems. Submitted for publication.