PKLS07] *Konstantin Pussep, Predrag Knezevic, Nicolas Liebau, Ralf Steinmetz;* ***Improving XPath Query Execution in P2P XML Storage by Using a Decentralized Index;*** VLDB International Workshop On Databases, Information Systems and Peer-to-Peer Computing (DBISP2P-2007) September 2007. S.

Improving XPath Query Execution in P2P XML Storage by Using a Decentralized Index

Konstantin Pussep[1], Predrag Knežević[2], Nicolas Liebau[1], and Ralf Steinmetz[1]

[1] KOM, TU Darmstadt, Merckstrasse. 25, 64283 Darmstadt, Germany
{pussep,liebau,steinmetz}@kom.tu-darmstadt.de
[2] Fraunhofer IPSI, Dolivostr. 15, 64293 Darmstadt, Germany
knezevic@ipsi.fraunhofer.de

Abstract. Today, information is managed increasingly in dynamic communities on the Internet. Here, peer-to-peer communities in which users host data by contributing their resources is a very promising alternative for centralized hosting. Managed data is often represented in XML and requires a high-level query language, where XPath is a good candidate. In this paper, we present a decentralized XML index that enables efficient XPath queries on large documents stored in p2p systems. Unlike other approaches, we do not rely on a specific overlay as our solution is able to work on top of any structured overlay that provides common *put* and *get* operations. Our approach is combined with P2P XML Storage, which stores arbitrarily XML files in p2p networks efficiently. Evaluation has proven that our index improves the XPath query performance in regard to both the execution time and the number of messages by orders of magnitude.

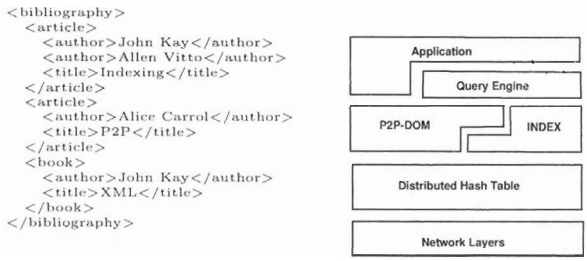
1 Introduction

Currently, Internet communities like Wikipedia host their data on centralized systems, even though the communities themselves are organized in a decentralized fashion. Therefore, peer-to-peer communities in which users host data by contributing their resources are a good alternative for centralized hosting.

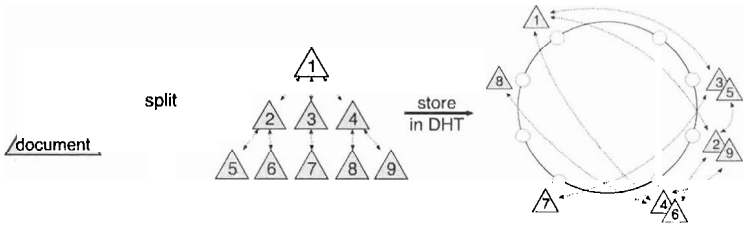
The BRICKS[3] project faces exactly this challenge in that it aims to design, develop and maintain a user and service-oriented space of digital libraries on top of a decentralized architecture. Within BRICKS, data is managed by P2P XML Storage [1]. The storage is built on top of a DHT (Distributed Hash Table) where large XML documents are split into fragments which are then stored as DHT values. Users must not only be able to read and write XML data, but also to query a document's content with high-level XML query language like XPath [2]. While a query functionality has been implemented in the storage, it is based on the traversal of document fragments and is therefore slow. This can be very inefficient for large documents as their fragments are spread across many peers.

This issue can be solved by the introduction of an XML index. This index must be stored in the p2p system and significantly improve the execution of XPath queries in P2P XML Storage. The index must be completely decentralized

[3] BRICKS - Building Resources for Integrated Cultural Knowledge Services, `http://www.brickscommunity.org`



(a) Example XML document (b) System architecture



(c) Storing XML Document (dashed lines show links between fragments)

Fig. 1.

and not impose limitations on the system abilities, but still support the full expressiveness of the XPath language.

In this work, we propose a solution that fulfills all the stated requirements and has been evaluated by emulations on the storage prototype. The proposed approach focuses on the indexing of single XML documents, but the approach can be easily extended to support collections of documents.

The next section provides XML-related terminology used in this paper and the applied system model. The exact requirements for a decentralized index are described in Section 3. The realization of these requirements by a p2p XML index is presented in Section 4. Section 5 presents the experimental results obtained. In Section 6 we discuss the related work on indexing XML in peer-to-peer systems in comparison to our approach. Finally, conclusions are provided in Section 7.

2 System Model and Definitions

2.1 XML Documents and Queries

Our sample XML document is a bibliography database similar to DBLP [3] as shown in Fig.1(a). This document contains publications such as *books* and *articles* along with the information about their authors, titles etc. XML documents are comprised of **elements** and contain values as **text** or **attributes** (attributes are omitted in this example).

Table 1 shows several XPath [2] queries for the example document. The query Q_1 allows all books to be fetched while Q_2 produces the titles of all articles written by John Kay. Another query type is Q_3 which produces titles of all publication containing the keyword *P2P*.

XPath queries consist of **location steps**, such as */article* or */title*[*author* = "*John Kay*"], which select elements or text in an XML document and can

contain **predicates** (e.g. [*author* = "*John Kay*"]), which filter the results. A sequence of location steps specifies that the result of a previous location step is used as input for the next location step. For example, */bibliography/book* consists of two steps: *bibliography* and *book*, in which the first step returns the *bibliography* element, which is further used to select *book* elements in the second step. These steps use so-called **axes** to specify the direction in which XML nodes are selected, i.e. elements or text. The most commonly used XPath axes are the child axis and descendant axis abbreviated by / and // respectively.

2.2 P2P XML Storage

In order to manage XML data in a p2p environment P2P XML Storage [4] was built. Here, users are able to create, access and modify XML documents via the DOM API [5] or query a document's content through XPath queries.

The overall architecture of the system is presented in Fig. 1(b). Storage works on top of any DHT, such as Chord [6], Pastry [7] or P-Grid [8], utilizes an XPath query engine and offers the DOM API to client applications. In P2P XML Storage, each XML document is fragmented according to its structure and then stored in the DHT. This process is illustrated in Fig. 1(c), where a document has been split into 9 fragments that are then stored in a Chord-like DHT. Adjacent fragments are connected to one other, i.e. a document fragment contains the DHT keys of adjacent fragments. Document traversal is carried out by loading the root fragment from the DHT and by following links to other fragments. The splitting of documents into fragments allows for better load balancing, as all the necessary fragments has to be retrieved.

As the system is document-based, collections of documents can be supported by an additional document listing of all the stored documents or by combining them to a super-document (i.e. connecting the document roots to a new root). In order to access a document, a peer has to know its name so it can compute the DHT key of its root fragment and retrieve the fragment from the DHT. Further fragments can be retrieved as each fragment stores the DHT key of its neighbor fragments. In this way, a user can access an entire document or only parts of it.

An XML document is shown as a tree in Fig. 2. Each XML element, text and attribute has been visualized as a tree node. The document is split into three fragments F_0, F_1, and F_2. Each XML element and text has an identifier consisting of a fragment id and the offset inside of the fragment. So the ID of the first *article* element is $(0; 1)$ as it is the first node in the fragment F_0.

3 Problem Statement

Although the P2P XML Storage is able to manage XML data in P2P systems, the support of high-level queries like XPath is inefficient. For example, query Q3

Table 1. Example Queries

	Query	Meaning
Q_1	*/bibliography/book*	fetch all books
Q_2	*/bibliography/article/title* [*author* = "*John Kay*"]	fetch the titles of all articles written by "John Kay"
Q_3	*//title*[*contains*(., "*P2P*")]	return all titles containing "P2P"

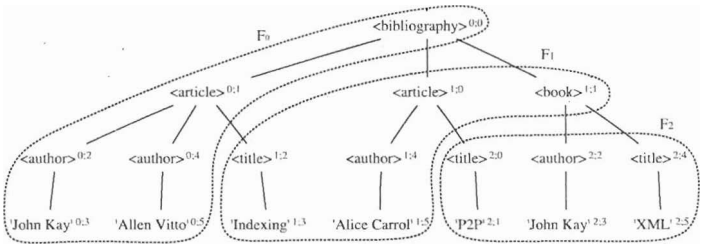


Fig. 2. Example XML document as a tree split into 3 fragments

in Table 1 would require that all fragments are scanned, even if there are only a few matching **title** elements, i.e. the query would require access to all fragments, thereby involving many peers and increasing traffic in the DHT. Hence, a decentralized XML index is needed in order to support efficient XPath queries. More precisely, this index must fulfill the following requirements:

1. Significantly improve query execution time and reduce the amount of traffic. The index must be scalable according to the size of documents managed.
2. The index must spread its content within a p2p community. In order to guarantee that the data can be found, the index has to work on a DHT.
3. The index has to adapt to new structures in the XML document, and therefore not rely on a static document structure or one managed manually.
4. The index must support document updates so that the user is able to update stored XML documents without needing to republish them.
5. The index must support the XPath query language and not restrict the structure of the documents stored.

As the key criteria for the index is its query execution performance, a suitable data structure for decentralized environments has to be found, i.e. the index must work without any global synchronization and global knowledge. Further, we need an appropriate distribution strategy for index content and, finally, query execution time and the network traffic induced must be minimized.

4 Decentralized XML Index

A decentralized index for XML data provides a data structure that improves the execution of XML queries in a decentralized environment. No central authority exists that creates, stores or maintains this index. This structure boosts the performance of queries in P2P XML Storage. A typical application is access for query execution without to load XML fragments. The result of the execution, depending on the query, is either a list of values or references that can then be used to access XML fragments from the storage for further processing.

4.1 XML Indexing Schemes

The XML and database communities have carried out extensive research on the indexing of XML data in traditional centralized databases. *Path indexing* (such as [9], [10], and [11]) and the newer *numbering schemes* (such as [12], [13], and [14]) are two main concepts for indexing XML documents. XML elements and

values are *labeled* with descriptors that provide information about their interrelationships in an XML document in a numbering scheme approach. By using these descriptors, the relationship (e.g. parent-child, ancestor-descendant, or siblings) between two nodes can be efficiently computed. The elements, text and attribute values are then stored in B+-trees together with their labels. However, for all numbering schemes, updating index structures is an issue. Even by attempting to foresee updates whilst labeling the document's content, an arbitrary update may still require that a large part of the document is relabeled, therefore requiring a large portion of the index structure to be updated.

Update operations are very important for the dynamic processing of XML documents and must be implemented efficiently so that the storage is scalable. Their impact on the *consistency* of data is another reason why the index structure update hat to be as simple as possible. Should the update of the index structure require too much time, the probability of concurrent updates increases because several peers may try to modify the same part of a document.

We have concluded that ToXin [11] is the most suitable starting point to an approach for a decentralized XML index. The key advantages of ToXin are its low update costs and the fact that it consists of single tables that can be easily distributed in a P2P community, especially in a DHT. One drawback of this proposal is the lack of support for XML content ordering. This support is essential for many XML documents, therefore we will show how this support can be added to the ToXin indexing structure.

4.2 Index Structure

The index captures both the tree structure of an indexed document and the values stored in text and attributes. It contains four different structure types:

Path summary summarizes the overall structure of an XML document. It records all existing paths and supports the addressing of other index parts.

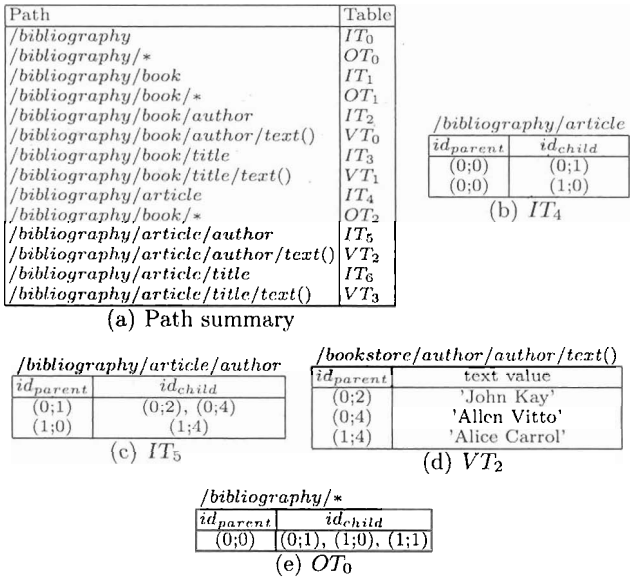
Instance tables store parent-child relations between XML elements. There is a table for each path in the document that ends with an XML element.

Value tables store the attribute and text values of elements to support the evaluation of predicates. For each path in the document that ends with an attribute or a text value, there is exactly one value table.

Order tables are our extension of the ToXin structure to provide order among different children of an XML element.

The document index consists of exactly one path summary and an arbitrary number of instance, value and order tables. Fig. 3 presents an example of the index structure for our sample document.

The path summary from Fig. 3(a) for the sample document summarizes all existing paths, but does not provide any information about individual XML elements or text values. It is used for two purposes: to find out whether certain paths exist within the document and to locate index tables in the underlying DHT. The path summary contains an entry pointing either to an order, instance, or value table for each path occurring in the document. For example, by inspecting the path summary, one is able to tell whether or not



Path	Table
/bibliography	IT_0
/bibliography/*	OT_0
/bibliography/book	IT_1
/bibliography/book/*	OT_1
/bibliography/book/author	IT_2
/bibliography/book/author/text()	VT_0
/bibliography/book/title	IT_3
/bibliography/book/title/text()	VT_1
/bibliography/article	IT_4
/bibliography/book/*	OT_2
/bibliography/article/author	IT_5
/bibliography/article/author/text()	VT_2
/bibliography/article/title	IT_6
/bibliography/article/title/text()	VT_3

(a) Path summary

/bibliography/article

id_{parent}	id_{child}
(0;0)	(0;1)
(0;0)	(1;0)

(b) IT_4

/bibliography/article/author

id_{parent}	id_{child}
(0;1)	(0;2), (0;4)
(1;0)	(1;4)

(c) IT_5

/bookstore/author/author/text()

id_{parent}	text value
(0;2)	'John Kay'
(0;4)	'Allen Vitto'
(1;4)	'Alice Carrol'

(d) VT_2

*/bibliography/**

id_{parent}	id_{child}
(0;0)	(0;1), (1;0), (1;1)

(e) OT_0

Fig. 3. Example of the index structure. (IT=Instance table, VT=Value table, OT=Order table. Only 4 out of 14 tables are shown)

the path */bibliography/article/author* exists in an indexed document. It it exists, there must be instance tables storing the parent-child relationships for the *bibliography*, *article* and *author* elements that belong to this path.

The relationships between single elements and values are stored within instance and value tables as shown in Figures 3(b), 3(c), and 3(d). Each table corresponds to the last step of an XML path, e.g. the table IT_4 stores the relationships between the root element *bibliography* and its children *article*s.

For example, in order to obtain the title values of all books, the steps */bibliography*, *.../book*, *.../title* and *.../text()* have to be processed. The necessary relationships are stored in instance tables IT_0, IT_1, IT_3 and value table VT_1. The relationships between *bibliography*, *book*, *title* and text nodes can be obtained by studying these tables. Note that elements are referenced in tables with their IDs. In order to connect the results from XPath queries with XML fragments stored in the DHT, we reuse the element IDs that already exist in P2P XML Storage.

As XML trees are *ordered*, the structure of the tree is not completely described by instance tables. This is because these tables do not preserve the order of child elements with different names, which is prescribed by the XML and XPath specifications. For instance, information about publications in the bibliography is contained within two instance tables: IT_1 for books and IT_4 for articles. We are not able to tell which book or article is the first publication. To assure the order of XML elements, we also store information about the child elements in *order tables*, without separating them by their child names as done

in instance tables. An example of an order table is given in Fig. 3(e). This table stores all child elements (book and article) for the bibliography element. This order table can be used to find the first or i-th entry in the bibliography. Clearly, this functionality is not always necessary and so order tables can be omitted for some applications. However, as the order of elements is prescribed by the XPath specification other applications may require it.

4.3 Managing Index in a DHT

Our index stores its data in the underlying DHT. Each index table is stored as a DHT value with a DHT key computed from the document name, the XML path of the table and the table type. The document name is used to avoid collisions of documents with similar structures. A typical index consists of many small tables that enable good load balancing.

The lookup of required index components is done by the underlying DHT layer. The index only requires an abstraction of storing and retrieving values from the network with DHT keys as addresses. The data availability is assured by the DHT or by additional replication mechanisms like [15].

4.4 Query Processing

The most common XPath queries appear as $/step_1/step_2/.../step_n$ with each (location) step selecting an XML content relative to the result of the previous step, possibly filtering it afterwards. The algorithm we use in our index evaluates one step at a time. After each step has been processed the result obtained is used to process the next step. Each step in an XPath query contains an **axis**, a **node-test** and an optional list of **predicates**. For example in the step $article[author = "John\ Kay"]$ the axis is (here implicit) the child axis, the node-test is *article* (that meaning that we are interested in elements called article) and the predicate is $author = "John\ Kay"$. An XPath step must be evaluated regarding the current context, i.e. the current position in the XML tree, as the result of the previous step or initial context. Starting from the current position in the XML tree, all matching *article* elements must be selected along the *child* axis, meaning that we select the entries from the matching index table for the path $.../article$ where the parent ID equals the IDs of the current position in the XML document. Afterwards, the result is filtered using the predicates. The predicate requires that the instance table $.../article/author$ is inspected to find the authors of all articles and that the value table $.../article/author/text()$ is then inspected to obtain their values. This step result in the articles with *John Kay* as author. If the query contains further steps, the next step is processed using the resulting articles as the evaluation context.

Thus, the index structure supports the evaluation of XPath queries in a natural way. All the required index tables are fetched from the DHT to the peers that want to process a query and were used to process single location steps. The path summary helps to avoid unnecessary DHT lookups, as it can be fetched at the beginning of query processing and reveals whether an XML path and associated index tables for this document exist.

4.5 Adapting Index to Document Updates

XML documents stored in a P2P environment must be indexed upon creation and this index must be kept up-to-date upon document updates. The initial indexing of a new document is done by applying a depth-first traversal to the XML tree of the document. For each XML path that occur an entry in the path summary and an instance or value table is created. Then, for each pair of elements or an element - value pair, an entry is written in the corresponding table.

This process is not repeated if the stored XML document is afterwards modified by users. It is only necessary to update the path summary if a new path is added to the document. For instance, the appending of an entry *journal* under *bibliography* to the document will add the entry */bibliography/journal* to the path summary from Fig. 3(a). On the contrary, if a new article is appended to the bibliography, the path summary would not change because the path */bibliography/article* already existed.

In general, the update (i.e. append, modify or remove) of an element or value in a document requires the update of one instance or value table and sometimes (if an XML path emerges or disappears) an update of the path summary. Therefore, the update of single elements and values requires the adaptation of the associated index with complexity $O(1)$ regarding the number of DHT operations because, in the worst case, only one table and the path summary have to be updated. As typical DHTs have operation complexity of $O(\log N)$, the overall update complexity is $O(\log N)$ where N is the number of peers in the p2p network. Note that this also applies to the modifications of the document's structure, i.e. the insertion of completely new elements like *journal*, *proceedings*, *techreport* etc. would have the same complexity.

5 Evaluation

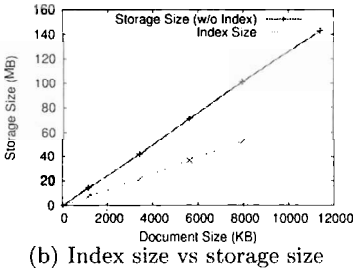
Evaluation of the proposed index was done by emulations on the storage prototype. As we discuss in Section 6, no existing p2p XML indexing approach satisfies the requirements in the P2P XML Storage. Thus, we measured the performance speedup achieved by the index in comparison to the fragment traversal. We distinguish two speedup indicators: the execution time and the number of messages in the DHT layer required to execute a query.

5.1 Settings

The index is evaluated as part of the P2P XML Storage system. A virtual network consisting of 100 storage instances is established where all instances run within the same Java virtual machine. For this purpose, the network layer has been simulated in the main memory within an adjustable message transmission delay that is set to 0.5 ms per kByte. On top of the emulated network layer, FreePastry [16] runs as the DHT router implementation. We use benchmark documents generated by the XMark benchmark generator (XMark) [17] to evaluate the performance of the decentralized XML index. The documents shown in Table 2 model an Internet auction site storing all the information about their customers and items in one XML file.

	Query Expression	Description
Q_1	/site/regions/australia/item/description	simple query, result size grows with document size
Q_2	/site/people/person[@id='person0']/name/text()	contains predicates, constant result size requires value comparison
Q_3	/site/regions//item	slow query, result size grows with document size

(a) Benchmark Queries



(b) Index size vs storage size

Fig. 4.

Three benchmark queries from Table 4(a) are processed on each of the benchmark documents. These queries are selected from those suggested by the XMark project. Query Q_1 returns the list of items registered in Australia. Query Q_2 returns the name of the person with id "person0" and is an example of a lookup query. The last query Q_3 returns all listed items. Note that the execution of the last query without an index requires scanning the complete document.

The evaluation compares two different ways of executing XPath queries: applying the tree traversal technique without an index and applying the query execution with the proposed indexing technique. For each query, each document, and each XPath engine version, the simulation ran 10 times to consider the random distribution of document fragments in the DHT, which can result in a different routing time and, hence, yield different performance results.

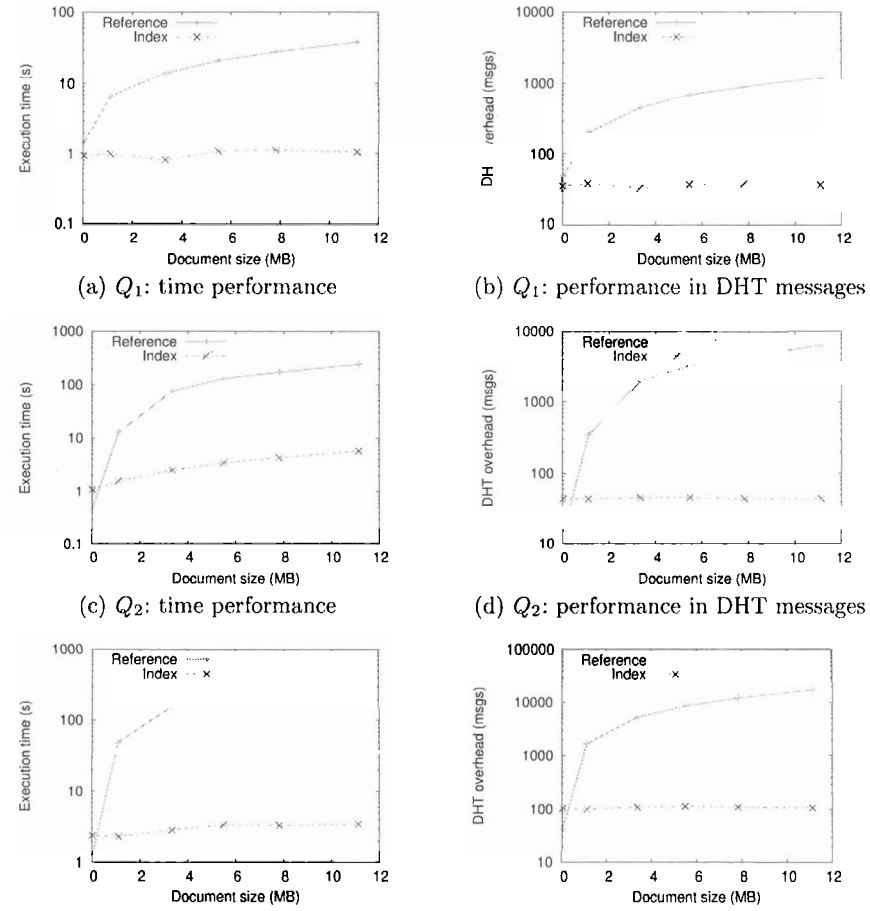
5.2 Results

The performance results are shown in Fig. 5. For each query the average execution time (Graphs 5(a), 5(c), and 5(e)) and the number of DHT messages (Graphs 5(b), 5(d), and 5(f)) are presented. The two lines show the performance of query execution based on fragment traversal and index, respectively. As we can see, for all three benchmark queries the index-based query execution is much faster. The only exceptions are queries Q_2 and Q_3 where the queried document is only 26 KB large, making a decentralized index unnecessary, since it is cheaper to transfer the complete file and execute the query locally. In all other cases, the performance is improved, especially regarding the query execution time, as it is up to 40 times faster for the queries Q_1 and Q_2, and almost 160 times faster for Q_3. The performance speedup is very similar regarding the number of DHT messages.

Fig. 4(b) shows that for each benchmark document, the total space consumption in the network by the index compared to the storage without the index. We can see that for all benchmark documents the size of the index is always smaller

Table 2. Used benchmark documents

File	Size (KB)	# Elements	# Attributes	# Text values
$xmark_1$	26	382	78	236
$xmark_2$	1134	17131	3917	12004
$xmark_3$	3424	50198	11526	35205
$xmark_4$	5616	83798	19337	58736
$xmark_5$	8003	118669	27320	83202
$xmark_6$	11396	167864	38266	118141



(a) Q_1: time performance (b) Q_1: performance in DHT messages

(c) Q_2: time performance (d) Q_2: performance in DHT messages

(e) Q_3: time performance (f) Q_3: performance in DHT messages

Fig. 5. Average performance results for queries Q_1, Q_2, and Q_3 measured in execution time and DHT messages.

than the size of the storage itself. Hence, we improve the query performance by one to two orders of magnitude without even doubling the storage consumption.

6 Related work

In [18] a distributed catalog for XML data is proposed. For each XML tag, there is a responsible peer that stores all unique XML paths leading to this tag together with the IDs of responsible peers. The query execution is done by first extracting simple paths from the query. Then the peers responsible for relevant catalog parts are looked up. These candidate peers receive and process the query. This system has significant limitations, as it only works with a subset of XPath and requires an extended version of Chord. Furthermore, it assumes that a peer stores complete XML documents and processes queries on them locally. Hence, peers storing popular tags or documents tend to suffer under extremely high

loads. In our solution the responsibility for an indexed tag name is distributed among several peers and the execution is done by the query initiator itself.

Another approach based on Chord is XP2P [19], whereby peers store document fragments (XML subtrees). In addition, each peer stores path expressions of its fragments and related fragments, i.e. sub and super fragments from the same XML document. An extended hashing technique allows for the lookups of both single fragments and fragments with unfolded sub fragments. As query processing requires the access of peers responsible for fragments, it is inefficient for queries that require the inspection of many fragments. This approach is similar to the original P2P XML Storage and, as our evaluation shows, much more inefficient than an additional index.

Skobeltsyn et al. proposed a distributed index to support a subset of XPath queries in structured P2P systems [20]. Their proposal works with P-Grid [8] or similar tree-based DHTs. It only supports a subset of XPath queries and the focus of the index is different from ours: for a given query, the index returns the addresses of peers storing XML documents or document fragments containing XML paths from the query. Our approach is unique in that it optimizes the execution of general XPath, does not rely on a specific DHT, and most important, it concerns the complete execution of a query and not only its dissemination to the peers with relevant fragments. This last point is important as the query result depends on the relationship between different fragments of the same document.

XML indexes for unstructured systems like [21], [22], [23], or XPeer [24] do not guarantee that the data available in the system can be found. Systems like Squid [25] or ZNet [26] support keywords, wildcard and range queries but do not support an all-purpose XML query language like XPath. P-Trees [27] are a distributed version of B^+-Trees. They can be used to build a distributed value index in order to answer range queries quickly. However, the management overhead is high and hence this approach does not scale well.

In [28] a DHT-based framework for the publishing and querying of XML documents was proposed. The scenario considered is different in that peers *publish* their local documents. Queries are routed to all responsible peers, which then process them and return their results. The system supports no updates, i.e. a document can only be updated by unpublishing and then re-publishing it, which is not feasible for large documents.

7 Conclusions

In this paper, we have presented a decentralized index for the efficient query execution on XML documents distributed within a p2p system. In particular, our decentralized index integrates seamlessly with P2P XML Storage, a system that stores large XML documents in a p2p network. We have shown by emulations that our prototype boosts the query performance in the system by orders of magnitude. Our solution differs from the related work in that it provides a unique set of features: its structure can be distributed in any DHT in completely decentralized manner, it supports updates and arbitrary changes in a document structure without the need for rebuilding an index, and it supports the whole range of XPath queries.

References

1. Risse, T., Knežević, P.: A self-organizing data store for large scale distributed infrastructures. In ICDE Workshops, 2005
2. W3C: XML Path Language (XPath) Version 1.0 (1999)
3. Ley, M.: DBLP - computer science bibliography. (http://dblp.uni-trier.de/xml/)
4. Knezevic, P.: Towards a reliable peer-to-peer xml database. In ICDE/EDBT PhD Workshop, 2004
5. W3C: Document object model (DOM) level 2 core specification 2000)
6. Stoica I. et al.: Chord: A scalable peer-to-peer lookup service for internet applications. In SIGCOMM, 2001 149–160
7. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. LNCS, 2001
8. Aberer, K.: P-Grid: A self-organizing access structure for P2P information systems. ACM SIGMOD Record **2172**, 2001, 179–185
9. Goldman, R., Widom, J.: Dataguides: Enabling query formulation and optimization in semistructured databases. In Proc. of VLDB (1997)
10. Milo, T., Suciu, D.: Index structures for path expressions. In ICDT, 1999
11. Rizzolo, F., Mendelzon, A.O.: Indexing xml data with toxin. In WebDB, 2001
12. Li, Q., Moon, B.: Indexing and querying xml data for regular path expressions. In The VLDB Journal, 2001 361–370
13. Chen, Y., Mihaila, G., Padmanabhan, S., Bordawekar, R.: L-tree: a dynamic labeling structure for ordered xml data. In EDBT, 2004, (Springer) 31–45
14. Meier, W.: eXist: An open source native xml database, Springer 2002) 7–10
15. Knezevic, P., Wombacher, A., Risse, T.: **Highly available DHTs: Keeping** data consistency after updates. In AP2PC. Volume 4118 of LNCS, 2005
16. Druschel, P. et al.: Freepastry 1.3.2. *freepastry.rice.edu*, February 2004
17. Schmidt, A., Waas, F., Kersten, M., Carey, M., Manolescu, I., Busse, R.: Xmark: A benchmark for xml data management In VLDB, 2002
18. Galanis, L., Wang, Y., Jeffery, S.R., DeWitt, D.J.: Locating data sources in large distributed systems. In VLDB, 2003
19. Bonifati, A., Matrangolo, U., Cuzzocrea, A., Jain, M.: Xpath lookup queries in p2p networks. ACM WIDM, 2004
20. Skobeltsyn, G., Hauswirth, M., Aberer, K.: Efficient processing of xpath queries with structured overlay networks. In ODBASE, 2005
21. Galanis, L., Wang, Y., Jeffery, S.R., DeWitt, D.J.: Processing queries in a large peer-to-peer system. In CAiSE, 2003
22. Crespo, A., Garcia-Molina, H.: Routing indices for p2p systems. In ICDCS, 2002
23. Karnstedt, M., Hose, K., Sattler, K.U.: Query routing and processing in schema-based p2p systems. In DEXA Workshops, 2004
24. Sartiani, C., Manghi, P., Ghelli, G., Conforti, G.: XPeer: A self-organizing xml p2p database system. In EDBT, 2004
25. Schmidt, C., Parashar, M.: Enabling flexible queries with guarantees in p2p systems. IEEE Internet Computing **8** 2004) 19–26
26. Shu, Y., Ooi, B., Tan, K., Zhou, A.: Supporting multi-dimensional range queries in peer-to-peer systems. In IEEE P2P, 2005 173–180
27. Crainiceanu, A., Linga, P., Gehrke, J., Shanmugasundaram, J.: Querying peer-to-peer networks using P-trees. WebDB, 2004
28. Fegaras, L., He, W., Das, G., Levine, D.: Xml query routing in structured p2p systems. DBISP2P, 2006