Distributing XML with Focus on Parallel Evaluation

Sebastian Graf[1], Marc Kramis[1], and Marcel Waldvogel[1]

Department of Computer Science, University of Konstanz
78457 Konstanz, Germany
`firstname.lastname@uni-konstanz.de`

Abstract. In contrast to relational databases the distribution of document-centric XML is not well researched. While there are some suggestions on how to split and distribute large XML documents, these approaches do not consider the parallel query evaluation. In this paper, we present and compare five different algorithms to search after suitable split nodes in a large XML document. We then describe how to distribute extractable sub-structures over a fixed number of peers and how to query these peers in parallel to retrieve the final result. In addition, we analyse the impact of our splitting algorithms with respect to scalability for two different XPath expression classes on three well-known XML data sets. We conclude this paper with an outlook on future work, including result ordering during parallel query execution and dynamic re-distribution of XML fragments to new peers due to updates.

1 Introduction

XML established itself as a first-class citizen throughout the modern computer world despite its reputation as being too verbose and inefficient. However, people *do* actually value features such as self-descriptiveness and data-before-schema. In addition, XML stands for universal interchangeability – including long-term archival – and comes along with a rich toolset.

Initially, XML was just used to exchange or store small amounts of data in an unified way. Nowadays, even large data- and document-centric data sets are stored in XML files or (native) XML databases. A single system storing such enormous amounts of XML data quickly comes to its limits due to insufficient amounts of storage space, processing power, or available main memory. When it comes to massive concurrent access or to single-system failures, distribution to multiple systems becomes the only feasible option.

The distribution of data-centric XML is straightforward and extensively researched with relational databases where the aim is to distribute columns and rows in a reasonable way. Here, reasonable means with minimal effort and optimally suited to the workload to which the database is exposed. In stark contrast, the distribution of document-centric XML is much more difficult. The straightforward approaches for distributing XML fragments will likely not result in a balanced system. E.g., if a splitting algorithm splits the XML tree at the first level, i.e., at the root node, some XML fragments are likely to contain large sub-trees while others might only contain a few nodes.

It is a challenging task to automatically find the appropriate split nodes in the tree because the system has to adapt to every single XML document and therein to a potentially completely different topology for each sub-tree. In a system with a fixed number of peers, it is important to split the tree in a way to make sure that every single peer gets its fair share of XML fragments of comparable size. In other words, the question is whether there exists a split algorithm which produces an optimal distribution for any document-centric XML data. Only when the XML fragments are evenly distributed over multiple peers, each one has an equal chance of being involved in a parallel query evaluation. Still, chances are that some peers get more involved due to a specific query workload. If the distribution is already skewed due to a bad split algorithm, even simple queries will create hot-spots and therefore bottlenecks in the whole system.

During our quest for the optimal split algorithm, we developed five different approaches and analysed each one on three well-known large XML data sets. The first is XMark [1], a generic data set, the second is Treebank [2], a corpus of linguistic documents, and the third is DBLP, a listing of publications relevant to computer science [3]. All three data sets contain a mix of mainly document-centric but also data-centric aspects. For each data set, we evaluated two different XPath expression classes. The first expression class is a depth-first search consisting of a concatenation of many child axis steps, the second expression is a breadth-first node count for a descendant step.

The rest of this paper is organised as follows. Section 2 describes the related work. Section 3 contains our main contribution, i.e., the five split algorithms and how the resulting XML fragments are distributed and queried in parallel. Section 4 discusses the benchmark results. Finally, Section 5 concludes this paper and gives an extensive outlook on future work.

2 Related Work

Research started to look into distributed XML data only a few years ago. Many approaches considering distributed queries are based on the assumption that XML is already distributed [4–8]. The focus is laying on the distributed query evaluation itself.

Based on the well-known distribution techniques of relational databases, i.e., the horizontal [9] and vertical fragmentation [10], some take this straightforward concept of fragmentation into account [11–13]. The suggested algorithms work well for data-oriented XML because of their regular structure.

Based on document-centric XML, the resulting XML fragments could have different structural characteristics. To our knowledge, there are only a few approaches, which take the structure itself into account to avoid possible irregularities when partitioning and distributing an XML-tree. [14] presented an approach which is directly based on several structural constraints. i.e. the width, the size, and the depth of sub-trees which can be extracted. In addition, the parameters have to be manually set before-hand to obtain a fragmentation. Depending on these parameters, a good fragmentation with respect to a parallel evaluation is guaranteed.

A completely different approach with the same focus on parallel queries is described in [15]. The parallel evaluation takes place either on distributed XML which was partitioned with the help of graph-partitioning algorithms [16] or on a variable fragmentation depending on an executed query. In this case, the fragments are represented by DOMs. This reduces the usability of the variable fragmentation because the DOMs have to be adapted each time the query changes.

3 Splitting XML

Our approach involves two independent steps: First, we identify suitable nodes to split. These nodes (and their respective sub-trees) are extracted from the original tree structure as described in [14]. Afterwards, we store the sub-structures in a given number of peers to achieve an independence between the number of peers and the number of extractable sub-structures.

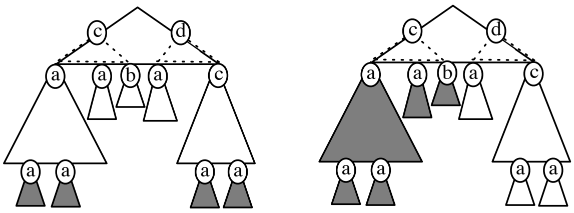
3.1 Five Split Algorithms

In this section, we present five different algorithms to find optimal split nodes. Particularly, we account for document-centric data with respect to a parallel evaluation of the resulting sub-trees. To evaluate and compare the presented approaches the following main objectives are set:

- The split nodes should have equal tag names. If the tag names of the root-nodes of the desired sub-structures are all unique, it is not guaranteed that tree-walking queries working with node-tests can be processed in a parallel way due to the different semantic aspects of the root-nodes of the extractable sub-structures.
- The identified sub-structures should have a maximum size. Working on a distribution in a server-client environment generates an overhead. The handling of this overhead can only be justified if the evaluation of the fragments is expensive. So the sub-structures should be as big as possible to ensure that the distribution leverages a performance benefit.
- The resulting fragments should have an equal number of nodes. Regarding the target of a parallel evaluation, the parallelism is used as long as each peer is processing its data. Therefore to ensure a long-term parallelism, every peer should have the same amount of data to process.
- The split algorithm should work without any meta-data. Every input to the split operation must be based on a before-hand exploration of the structure or at least of the DTD. Regarding document-centric data, an analysis with the focus on structural and semantic constraints, is not easy. If the split algorithm itself has knowledge about the XML instance, dependencies to third-party inputs are minimized.

Level Split The first approach marks all nodes on a given level as split nodes. An example is shown in Figure 1(a). This approach works perfectly for data-oriented XML where the structure is quite similar as described with current approaches based on horizontal fragmentation techniques inherited from the distribution of relational data. However, it is not guaranteed to get a good fragmentation result with this approach regarding document-centric data due to the irregularity of the structure. As shown in Figure 1(a) all relevant sub-trees with the corresponding split nodes *a* are identified. Thus, the condition of split nodes with similar tag names is satisfied. The size of the identified sub-structures is quite small regarding the other possible splits with the split nodes *a,a,b,a,c* where the extracted sub-structures would be much larger. In this case the structure with the split nodes denoted with *b* and *c* would be extracted as well. This is not intended because the tag name only occurs once in the set of split nodes.

Therefore, the *Level Split* possibly results in undesirable sub-structures. An additional analysis must be performed to find the suitable levels in a given XML instance. Especially regarding document-centric data with highly irregular structures, the extracted sub-structures are potentially not matching the defined requirements.



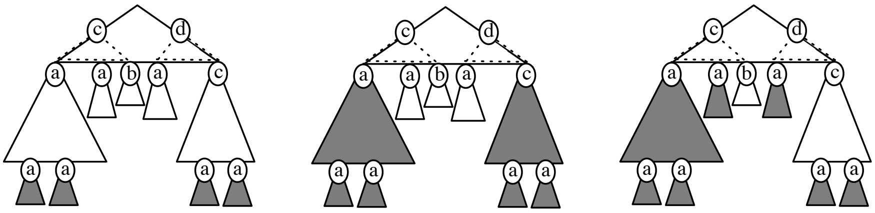
(a) Level Split (b) Fanout Split

Fig. 1. Example of split nodes identified by the *Level Split* and the *Fanout Split*

Fanout Split To get only large sub-structures, we assume that a node with a large sub-tree has a large fanout as well. Therefore, according to a given number of children, a node in the XML structure is marked as a split node if this number exceeds a given threshold. This threshold must be set in advance of the identification process. Consequently, a suitable exploration becomes inevitable. An example of the result of the *Fanout Split* algorithm is shown in Figure 1(b). Again, this approach works well for data-centric XML. For document-centric structures, the identification of the split nodes should not be solely based on the fanout of a node due to the irregularity of the document structure, as this can result in nodes with a large fanout but relatively small sub-trees. Additional to this possible violation of our design objectives, there is no assertion that there are no unique tag names in the set of split nodes. In Figure 1(b), e.g., the split node named b is set, but a parallel evaluation of the underlaying structure is not possible because of the document-centric structure.

Semantic Split To consider equal tag names of split nodes, this approach is based on the occurrence of tag names. On each level, the occurrences of different tag names on the sibling axis are counted. If there is more than one node in the sibling axis, and each sibling has the same tag name, these nodes are identified as split nodes. An example is given in Figure 2(a). Again, the grey sub-structures can be queried in parallel. Unfortunately, as with the *Level Split*, the identified sub-structures are quite small and therefore not optimal. The partitioning of sub-structures as labeled with a on the upper level make more sense with respect to parallel evaluations.

Moreover, an exploration to find suitable nodes is unnecessary. Additionally the extracted sub-structures have a high similarity due to of the tag names of their root node. This assures a parallel evaluation of these structures even if the sub-structures themselves are small.



(a) Semantic Split (b) Postorder Split (c) PostorderSem Split

Fig. 2. Example of split nodes identified by the *Semantic Split*, the *Postorder Split* and the *PostorderSem Split*

Postorder Split To tackle the need for large extractable sub-structures, we developed a split algorithm based on the designated number of fragments. With the help of a post order traversal through the original tree structure, the sub-tree size corresponding to each node is computed. A threshold $\frac{n}{i*2}$ based on the nodes n in the XML instance and the number of available peers i is computed. At each node, the number of processed nodes is compared to the threshold. If the threshold is attained, the actual node is marked as a split node and the counter of nodes is reset. If no node has the potential to work as a split node, the children on the first level are selected to achieve at least a basic fragmentation. Figure 2(b) shows an example. Here, the two biggest possible sub-structures, according to a given number of peers, are identified. Unfortunately, the identified sub-structures have distinct tag names. This complicates parallel evaluations, even if the corresponding sub-structures contain a large amount of nodes. The split node obviously generates sub-structures with a similar load. Regarding data-centric XML, this approach works well as the identified split nodes have equal tag names.

PostorderSem Split To get rid of the possibly different tag names in the extracted sub-trees based on the *Postorder Split*, the *PostorderSem Split* combines the post order traversal with the *Semantic Split*. The tree structure is processed with a post order traversal similar to the *Postorder Split*. However, instead of computing the size of the current sub-tree according to a given node, the sizes of the processed nodes according to the tag name of the current node are stored. This size must attain the same threshold as described in the *Postorder Split*. If the threshold is attained, all nodes with the given tag name are marked as split nodes. With this approach, the main objectives are satisfied. The extracted sub-trees have an adequate large size, except too highly recursive occurrences of the same tag name. In this case, this approach can lead to small sub-structures. The similarity of the root-nodes of the extractable sub-structures is ensured by the computation of traversed nodes according to the tag names. As the identification of the sub-structures is based on the number of nodes and the number of available peers, this split-operator is not in need of a previous exploration of the XML instance. However, if the XML data is based on a recursive DTD this approach could result in suboptimal splits, because the number of corresponding nodes related to a tag name can be falsified by the recursive occurrence of the elements.

3.2 Distribution and Combination of extracted Sub-Structures

After the identification of suitable split nodes, with focus on a parallel evaluation of the corresponding sub-structures, these sub-structures have to be extracted and stored in different peers. The distribution of the identified structures is conducted by a simple *round-robin* algorithm.

A suitable number of fragments according to the number of available peers is chosen. Additional to these structures, a root fragment is initialised. Afterwards, the original XML dataset is traversed in pre order. Each node is inserted in the root fragment until the first split node is reached. Then, a proxy-node with an unique id is inserted into the root fragment and a child fragment is selected in *round-robin* order. On the child fragment, a corresponding proxy-node is inserted which also has an unique id. With these ids a direct access between the proxy-node in the root fragment and the proxy-node in the child fragment is obtained. After inserting this proxy-node, the split node itself is inserted beneath this proxy-node. Subsequently, each following node is inserted on this child fragment until the traversal is leaving the current sub-tree. Finally, the following nodes are inserted into the root fragment until the next split node is reached and so on. Even document-centric data, depending on suitable split nodes, is very well fragmented with this approach because split nodes with the same tag name are distributed in *round-robin* fashion over all peers.

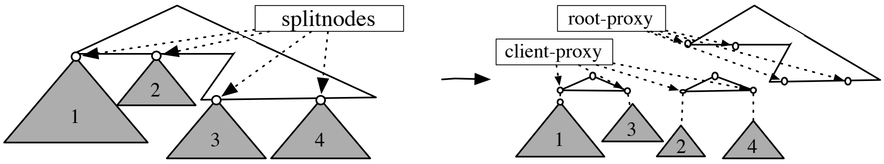


Fig. 3. Layout of a Distribution with Proxy-Nodes

Figure 3 shows fragmented XML. The following sub-structures *1,2* and *3,4* are, with two given peers, distributed. Based on the assumption that the number of different tag names of the split nodes is very limited, a good distribution of parallel evaluable structures is given.

3.3 Query distributed XML Documents

XPath 2.0 [17] is a common expression language to select parts of an XML document. It is used as a base for many wide-spread query and transformation languages such as XQuery and XSLT. The XPath 2.0 expressions are divided into axis steps to navigate in the tree, and node-tests to filter the required nodes.

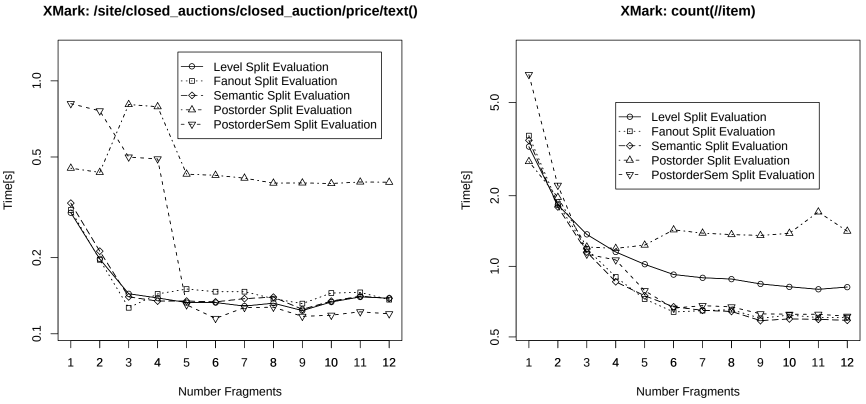
With respect to a given fragmentation, expression evaluation can be used in a straightforward way. First of all, we have to modify the expression to solely use the forward axes. Due to the symmetry in XPath [18], this modification is not a restriction. We did not implement the *following* axis in our prototype as this kind of axis step is not used in practical applications. Note that querying the *following* axis is not trivial because the distribution scatters the nodes of interest throughout multiple peers.

A given XPath expression is executed at the root-fragment. The expression traverses the tree in the designated order and, after reaching a proxy-node on the root-fragment, is handed over to the appropriate fragment referenced through the unique ids of the proxy-nodes. Thereafter, the expression proceeds with its work on the root fragment without waiting for the result. The expression on the child fragments is executed in parallel. To match the sub-tree of the fragment, the expression might has to be adapted, e.g., a query just containing child steps is pruned to perform only those child steps, which are actually executable on the sub-tree. After the evaluation of the root fragment, the query processor waits for the results from the fragments and the final result, which has to be combined, is returned.

4 Benchmark Results

We evaluated our split algorithms in detail. Based on an environment of 3 servers, with 8GB RAM, and two 2Ghz DualCore Opteron Processors, an evaluation on 1 up to 12 peers was run. The root fragment was handled by an iMac with a

DualCore 2.16Ghz Processor and 2GB RAM. All machines were connected by a local switched gigabit network.



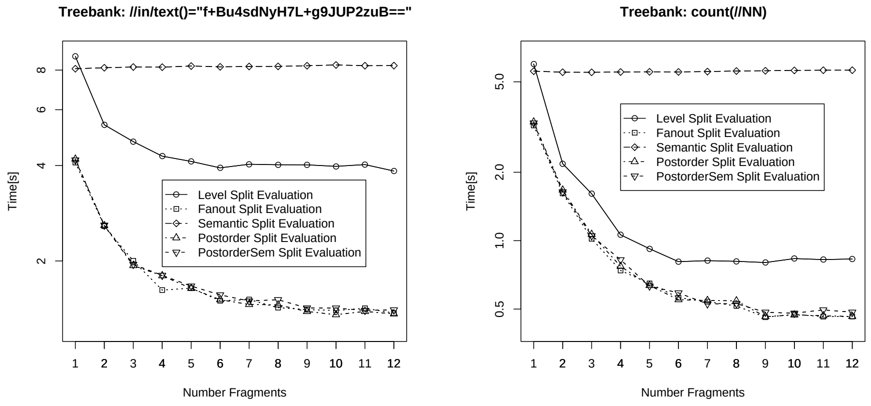
(a) XMark – Get all prices from all closed auctions

(b) XMark – Count all items

Fig. 4. XMark benchmark results

The benchmarks were performed on XML from the XMark-project [1] with the size of 100MB, the Treebank XML [2], and the DBLP XML [3].

The results of two queries on the XMark dataset [1] are shown in Figure 4(a) and 4(b). Figure 4(a) illustrates the drawback of the *Postorder Split* and the *PostorderSem Split*. Regarding the *Postorder Split*, the query is only executed on one peer as the split threshold was not reached. Using two peers, every sub-tree starting at the first level of the XMark XML was extracted since none of the sub-trees themselves could reach the threshold. As a result, the evaluation was performed by just one peer (the one which has to handle the *closed_ auctions* sub-tree). When using three and four peers the left-sibling sub-trees of the *closed_ auctions* element were extracted. This induces an evaluation of the root-fragment that is maintained on the workstation which in turn explains the poor performance. Again, working with five and more peers induces to a complete extraction of the *closed_ auctions* sub-tree again. Thus, the performance is limited by a single peer. Regarding the *PostorderSem Split* in Figure 4(a), the scaling depends on the number of peers as well. While the split algorithm ignores all sub-trees working with one and two peers, most large sub-trees are extracted by utilizing three and four peers (e.g., *item* and *open_ auctions*). This induces to a smaller root fragment which can be processed in a faster way by the root-peer. By utilizing five or more peers, all *closed_ auction* sub-trees are extracted and can therefore be evaluated in parallel. All other split operators induce an optimal fragmentation even with only one peer.



(a) Treebank – Searching after a given string

(b) Treebank – Counting all NNs

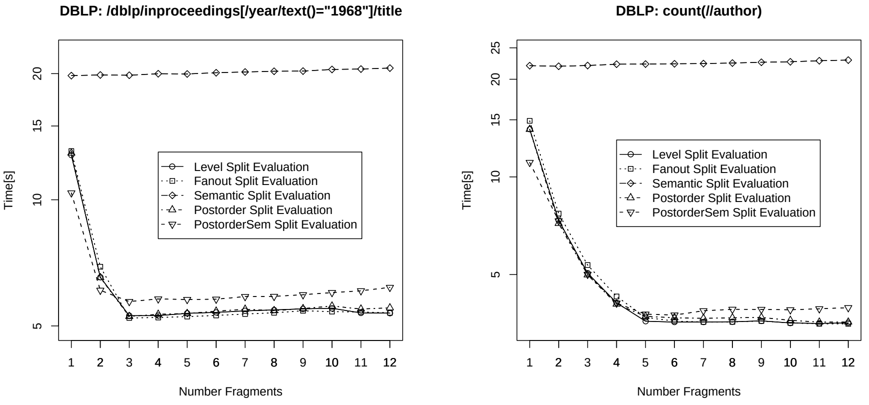
Fig. 5. Treebank benchmark results

The second query on the *Postorder Split* fragmentation shown in Figure 4(b) performs as bad as in Figure 4(a). The reason is similar to the first XMark query: as the *regions* sub-tree is extracted, no performance gain can be achieved due to the fragmentation. The other split operator, which has a worse performance, is the *Level Split*. The XML instance was splitted at level 2. Regarding the first query, an optimal distribution for the *closed_ auction* nodes was achieved. However, the processing of the second query is not executed in an optimal way. The load of the different sub-trees containing the *item* nodes is not fair as the parent nodes of the *item* sub-trees were extracted (e.g., *samerica, namerica, europe, ...*). This shows that a strict horizontal splitting can result in suboptimal fragmentations when document-oriented data is involved.

Figure 5 illustrates the results of queries executed on the Treebank dataset [2]. In this case, the *Semantic Split* does perform rather poor as this highly document-oriented dataset has a stringent irregular structure so that similar tag names on the sibling axis are quite seldom. The *Level Split* was set in level 2 to show the importance of an optimal level even for document-oriented data. All other split operators lead to the desired fragmentation.

In Figure 6 the evaluation of the fragmented data-centric DBLP XML [3] is shown. All split operators generate a similar fragmentation. except the *Semantic Split*. Due to the different tagnames of the nodes on the first level, only few very small sub-structures are identified. So both queries are executed on the root-fragment only.

Other famous XML databases like Wikipedia and Swissprot were benchmarked with similar results. Depending on the parameter (fanout, level, peers) the scaling is similar to the examples shown in Figures 4, 5 and 6.



(a) DBLP – Searching after a proceedings of a given year

(b) DBLP – Count all authors

Fig. 6. DBLP benchmark results

5 Summary and Outlook

Our findings affirm the assumption that a trivial split algorithm does not consistently achieve an optimal distribution. The presented automatic split algorithms, though, can split large XML documents with the same overhead as a trivial split algorithm, but with much better scalability when it comes to parallel query evaluation. Regarding the summary in Table 1, we see that the different split operators cover different aspects of the five defined heuristics.

	Level Split	Fanout Split	Semantic Split	Postorder Split	PostorderSem Split
Similar Split Nodes	−	−	+	−	+
Large Sub-Trees	+	+	−	+	−
Equal Load	−	−	−	+	+
Autoconfiguration	−	−	+	+	+

Table 1. Summary of Spit Algorithms

The similarity of the split nodes and the equality of the load can be positive as well as negative regarding the *Level Split* and the *Fanout Split*. This depends on the selected level or the chosen threshold. The *Semantic Split* scales well for data-centric data. Regarding document-centric data, a good fragmentation can not be guaranteed. The fragments of the *Postorder Split* are as equal as possible regarding the structure of the original XML data. This equality is available with the *PostorderSem Split* as well in most cases. Only with highly recursive structures, multiple small sub-structures are identified. Yet, in all of our test cases, no negative fragmentation was achieved with the *PostorderSem Split* and a high number of participating peers. For further information of our approach we refer to our technical report [19].

We see many open questions for future work in the area of distributing large-scale XML data for parallel query evaluation:

- Depending on the query, a simple parallel evaluation may need to reorder the result retrieved from multiple peers. We want to investigate which queries are affected and whether this reordering operation can be prevented or efficiently done during either the split operation or the parallel evaluation.
- While XPath provides a fundamental idea of what the evaluation time will be, it is only a sub-set of current query languages such as XQuery. We want to look at XQuery and how it can be executed in parallel by rewriting the query itself or by optimising and splitting the logical operator tree.
- Currently, we split a static XML document. We are interested in updates and how they lead to re-assignments of the XML fragments to keep the whole distribution in balance. This may be achieved through moving the fragments themselves or by dynamically further splitting up the XML fragments.
- The availability of indices may lead to faster evaluations for certain queries, i.e., an index can usually answer the query in logarithmic time without the need to do a full XML fragment traversal. However, an index will incur more update overhead and may itself grow so large that it also must be distributed. E.g., a full text index which has to store a term occurring in a large percentage of the nodes is no longer useful. Through splitting the XML document in smaller parts, we also make sure to split the domain of each index and potentially reduce the over-all update and search time.
- The reliability and availability of large XML documents will also become an issue. E.g., it is no longer possible to export Wikipedia to an XML file within a single day. Loosing the whole file due to a peer failure is catastrophic and would interrupt a service relying on it for too long. As soon as Wikipedia is distributed, the loss of a single peer will only erase a small part of the overall document. The question then becomes how to store a single XML fragment on multiple peers and how to exploit this knowledge to further speed up the query evaluation when each peer has its individual performance and variable network connection quality.

With our five split algorithms, we break new ground on how to distribute large-scale XML data sets and how to query them in parallel for scalability reasons. However, much work remains before a cluster of peers will automatically and collaboratively store and query such large-scale XML data sets.

References

1. Schmidt, A., Waas, F., Kersten, M., Florescu, D., Carey, M., Manolescu, I., Busse, R.: Why and how to benchmark XML databases. ACM SIGMOD Record **30** (2001) 27–32
2. Marcus, M., Marcinkiewicz, M., Santorini, B.: Building a large annotated corpus of English: the penn treebank. Computational Linguistics **19** (1993) 313–330
3. Ley, M.: The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. LECTURE NOTES IN COMPUTER SCIENCE (2002) 1–10
4. Suciu, D.: Distributed Query Evaluation on Semistructured Data. ACM Transactions on Database Systems **27** (2002) 1–62
5. Bose, S., Fegaras, L.: XFrag: A Query Processing Framework for Fragmented XML Data. Proceedings of the WebDB (2005)
6. Buneman, P., Cong, G., Fan, W., Kementsietsidis, A.: Using partial evaluation in distributed query evaluation. Proceedings of the 32nd international conference on Very large data bases (2006) 211–222
7. Abiteboul, S., Bonifati, A., Cobéna, G., Manolescu, I., Milo, T.: Dynamic XML documents with distribution and replication. Proceedings of the 2003 ACM SIGMOD international conference on Management of data (2003) 527–538
8. Bremer, J., Gertz, M.: On Distributing XML Repositories. Proc. of WebDB (2003)
9. Ceri, S., Negri, M., Pelagatti, G.: Horizontal data partitioning in database design. Proceedings of the 1982 ACM SIGMOD international conference on Management of data (1982) 128–136
10. Navathe, S., Ceri, S., Wiederhold, G., Dou, J.: Vertical partitioning algorithms for database design. ACM Transactions on Database Systems (TODS) **9** (1984) 680–710
11. Ma, H., Schewe, K.: Fragmentation of XML documents. Proceedings XVIII Simposio Brasileiro de Bancos de Dados (SBBD 2003)Õ, Manaus, Brazil (2003) 200–214
12. Lü, K., Zhu, Y., Sun, W., Lin, S., Fan, J.: Parallel Processing XML Documents. Proceedings of the International Database Engineering and Applications Symposium (IDEASÕ02) (2002)
13. Ma, H., Schewe, K.: Heuristic Horizontal XML Fragmentation. Proc. of CAiSE (2005)
14. Bonifati, A., Cuzzocrea, A.: Efficient Fragmentation of Large XML Documents. LECTURE NOTES IN COMPUTER SCIENCE **4653** (2007) 539
15. Lu, W., Chiu, K., Pan, Y.: A Parallel Approach to XML Parsing. The 7th IEEE/ACM International Conference on Grid Computing (2006)
16. Karypis, G., Kumar, V., Center, A.H.P.C.R., of Minnesota, U.: Parallel multilevel k-way partitioning scheme for irregular graphs. SIAM Review **41** (1999) 278–300
17. Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., Simeon, J.: XML Path Language (XPath) 2.0. W3C Working Draft **15** (2002)
18. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking Forward. Lecture Notes In Computer Science (2002) 109–127
19. Graf, S., Waldvogel, M.: Splitting and distributing large document-centric xml databases. Technical Report KN-15-09-2008-DISY-04, University of Konstanz (2008)