On Distributing XML Repositories

Jan-Marco Bremer and Michael Gertz
Department of Computer Science
University of California, Davis
One Shields Ave., Davis, CA 95616, U.S.A.
{bremer|gertz}@cs.ucdavis.edu

ABSTRACT

XML is increasingly used not only for data exchange but also to represent arbitrary data sources as virtual XML repositories. In many application scenarios, fragments of such a repository are distributed over the Web. However, design and query models for distributed XML data have not yet been studied in detail.

In this paper, we introduce a distribution approach for a virtual XML repository. We present a fragmentation method and outline an allocation model for distributed XML fragments. We also discuss an efficient realization based on small, local index structures. The index structures encode global path information and provide for an efficient, local evaluation of the most common types of global queries.

1. INTRODUCTION

Recent advancements in the development of native XML database systems (e.g., [5, 16]) clearly indicate that XML is not only considered for data exchange but also as a data representation format. The management of XML data in such native systems, compared to managing XML in (object-) relational database systems, leads to new application and research perspectives. In particular, we conjecture that distribution aspects of XML will play an important role. In fact, several application domains for XML, such as Web services, e-commerce, collaborative authoring of large electronic documents (e.g., WebDAV [14]), or the management of large-scale network directories [7], show that XML data is inherently distributed on the Web. Systems managing distributed XML data thus have to take this aspect into account to allow for an efficient and reliable usage of XML data at different sites.

Although concepts for the distribution of XML data are clearly important, to this day, there is only little related work that deals with distribution aspects of XML. In [13], distributed query evaluation techniques are investigated. Recently, in [1] the problem of distributed and replicated (dynamic) XML documents has been studied for the first time and fundamental query processing models have been proposed. Interestingly, the W3C has published a candidate recommendation for XML fragment interchange [6], but respective concepts are neither well-founded nor used in practice. While the above works address some distribution aspects for XML, to the best of our knowledge, there is no work that investigates the top-down design of a distributed XML data source, following traditional and well-studied distribution design principles for relational databases.

International Workshop on the Web and Databases (WebDB).
June 12-13, 2003, San Diego, California.

In this paper, we lay the groundwork for the distribution design of XML data in the context of large-scale XML repositories that manage XML data on the Web for the application scenarios mentioned above. In particular, we present a fragmentation scheme on a global conceptual schema structure and outline how fragments, as XML data, are allocated at different sites. We also present a realization of the distribution design in which small index structures at each site efficiently encode information about local and remote fragments. The index structures allow for processing most global queries at local sites without accessing other sites.

In Section 2, we introduce the data, schema, and query model underlying our approach. The fragmentation and allocation schemes are discussed in Section 3. In Section 4, we detail the representation of local and remote XML fragment information at local sites and illustrate how local and global queries are efficiently evaluated.

2. FOUNDATIONS

2.1 Data Model

As customary, we assume that XML data is modeled as a (single) rooted, node-labeled tree $(\mathbf{V}, \mathbf{E}, root, label, text)$. Nodes from the set $\mathbf{V}$ are connected by edges from $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ to form a tree with root node $root \in \mathbf{V}$. The function *label* assigns an element name (or label) to each node in $\mathbf{V}$. For presentation purposes, we assume that element attributes are modeled as regular nodes. The function *text* assigns text strings to nodes.

We assume a partial order among nodes in which only the relative positions among sibling nodes carrying the same label are considered significant. This order criterion is sufficient to support most types of queries. For example, while for a book element, the order of authors (as child nodes) matters, the order between title and author child nodes of a book element is likely much less relevant.

In the following, we use the notion *data source* or just *source* to refer to a representation of XML data according to the above tree model. Such a data source can be considered a single (large) XML document. For a data source, we furthermore adopt the common definitions of *(rooted) label paths* as

sequences of node labels (starting with the label of the root node), and analogously *(rooted) data paths* as sequences of source nodes. Naturally, several rooted data paths are associated with a single rooted label path. For example, Figure 3 highlights a rooted data path for the rooted label path /DigitalLibrary/Loc/Books/Bk/A.

2.2 RepositoryGuide

Underlying our XML distribution approach is a global (conceptual) schema. While it is conceivable that such a schema exists in the form of a DTD or XML Schema, we chose to use a simplified schema structure. This structure is called *RepositoryGuide (RG)* and resembles basic features of a DataGuide for tree-structured data in that all admissible rooted label path are enumerated.

There are several reasons for our choice. First, a tree-structured schema representation is easier to "fragment" or decompose than a grammar-based schema specification. Second, in the case of a large-scale XML repository, it is likely that local sites express their information and data representation needs using a schema formalisms that is more expressive than a (local) RG. Such schemas (in the form of a DTD or XML Schema) can always be translated into a (local) RG and combined into one global RG to model the data to be managed in the distributed XML repository, thus resembling basic concepts of the view integration process.

A RepositoryGuide is strictly less expressive than a DTD or XML Schema. However, we assume that we can preserve information that might get lost in transforming DTDs or XML Schemas (if these exist) into a tree structure $(\mathbf{V}_{RG}, \mathbf{E}_{RG})$, which represents the RG. In particular, we assume that for each node v in the RG tree, the minimum/maximum number v_{min}/v_{max} of times this node (element type) occurs as child node is recorded. For instance, for a node v, as child of a node v', the pair (0,2) specifies that v is an optional child element and occurs at most 2 times as child of v'. Obviously, different min/max values can be associated with nodes having the same label in the RepositoryGuide.

Depending on how a RepositoryGuide is constructed, i.e., either from scratch or through "combining" local RepositoryGuides, DTDs, or XML Schemas, further information can be associated with nodes in *RG*. This includes in particular co-occurrence information about pairs of rooted label paths in *RG*. Such co-occurrence information describes that if a document contains a data path corresponding to one label path, then it must also contain a data path corresponding to another label path. Required siblings of a node are the most simple case of such co-occurrences, which can be used in query optimization schemes and in verifying a fragmentation for correctness, which is discussed in Section 3.1.

2.3 Query Model

In our distribution scheme, we assume that a query can be issued at any local site and query results, i.e., complete data fragments, are delivered to that site. Queries consist of path and tree patterns to be matched against the distributed source. Path and tree pattern queries build the core of most XML query languages. Edges in such patterns represent parent-child and ancestor-descendant relationships. Node labels or text values under certain nodes further constrain the patterns. A distinguished node called selection node determines the roots of fragments to be returned as query result. Figure 1 gives two examples of patterns that have matches in the form of book titles in the data source shown at the center of Figure 3.

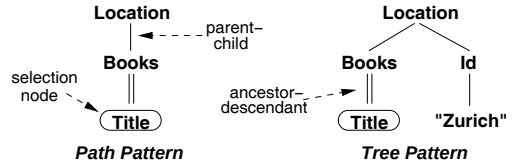

Figure 1: Path and tree pattern queries

The most common approach to process path and tree pattern queries are *structural joins* (e.g., [2, 9, 15]). Structural joins (SJs) are based on index structures that store lists of logical identifiers for nodes in a source. Each list is related to a certain node predicate, which is a common node label or label path, or a word occurrence. Node identifiers allow a query engine to determine parent-child and ancestor-descendant relationships between nodes. This in turn allows for reassembling instances of the full pattern as they occur in a source by joining node id lists, which represent atomic patterns [2].

3. DISTRIBUTION DESIGN

The objective of the distribution design for an XML repository is to formally describe what and how XML repository data is fragmented and allocated at local sites. Although the design strategy follows basic distribution principles known for relational and object-oriented databases [8, 12], the hierarchical structure and the mix of schema information and data in XML requires substantial adjustments to respective design approaches. In the following, we introduce basic properties of fragmentation approaches, illustrate a specific fragmentation scheme, and outline how fragments are allocated at local sites.

3.1 Fragmentation Alternatives

A fragmentation scheme has to satisfy certain correctness criteria in order to ensure that the semantics of the data to be fragmented does not change once fragment data is distributed and managed at local sites. In the context of relational databases, these criteria are known as *completeness, reconstruction,* and *disjointness rules* [12]. Similar rules can be devised for fragmenting XML data, as illustrated below.

We assume a set $\mathbf{S} = \{S_1, \ldots, S_r\}$ of local sites. With each site S_i, a set of queries Q_{S_i} is associated from which typical access patterns in the form of a sublanguage of XPath can be derived. Such access patterns include path and tree patterns as they are used in local, site-specific queries formulated in XQuery, XPath, or XSLT. The RepositoryGuide representing a global schema not only serves as basis for such queries, but it is also used to specify fragments. Informally, a fragmentation scheme has to ensure that the RepositoryGuide is decomposed into a disjoint and complete set of tree-structured fragments. The allocation scheme applied to respective fragments then will ensure the reconstruction of the repository as outlined in Section 3.2.

The specification of XML fragments is based on a sublanguage of XPath, called XF, that includes the context node (.), descendant and child axis ($//$, $/$), and wildcard ($*$). A fragment specification $f = sf - \{ef_1, \ldots, ef_n\}$ consists of two components: (1) a *selection fragment* sf, and (2) an optional set $\{ef_1, \ldots, ef_n\}$ of *exclusion fragments*. Both types of fragments are XF expressions. The semantics of such fragment specifications is fairly straightforward: first, the selection fragment sf is evaluated against the RG and results in a set of nodes. The evaluation can be done efficiently since an RG (as single XML document) is much smaller than a source corresponding to the RG. Each such node determines a subtree in RG, called *RG fragment*. Optional specifications $\{ef_1, \ldots, ef_n\}$ of exclusion fragments are evaluated on such subtrees and again determine RG fragments. For example, assume a fragment specification $//A//C - \{./D/*/E\}$. First, all subtrees in RG are selected that are rooted at a C node and have an A node as ancestor; such subtrees describe selection fragments. From these fragments, all subtrees are "cut off" that are rooted at an E node, have a D node as grandparent, which is child of a C node selected by the fs expression. Figure 2 illustrates this case where fragment f_2 is excluded from fragment f_1.

Since XF expressions represent path expressions, subtree containment can easily be verified for two fragment specifications based on prefixes of rooted label paths. Let $desc(f)$ denote the nodes in RG that are contained in fragment f (excluding those nodes specified by optional ef's). Given a set $\mathbf{F} = \{f_1, f_2, \ldots, f_l\}$ of fragment specifications. $\mathbf{F}$ is said to be *complete*, if all nodes $\mathbf{V}_{RG}$ are contained in at least one fragment, i.e., $\mathbf{V}_{RG} = \bigcup_{i=1}^{l} desc(f_i)$. The elements in $\mathbf{F}$ are said to be *disjoint* if there are no two fragment specifications $f_i, f_j \in \mathbf{F}, i \neq j$, such that $desc(f_i) \cap desc(f_j) \neq \emptyset$. For a given set $\mathbf{F}'$ of fragment specifications, disjointness can be verified in a naive fashion by marking each node v in RG with the number i of the fragment specification f_i that contains v. No node then must have more than one number assigned to it. If $\mathbf{F}'$ is not complete, a complete set $\mathbf{F}$ can be obtained by identifying the rooted label paths of subtrees that have not been marked (including exclusions that already have been marked). From a practical point of view, it is very likely that local sites are interested in fragments close to the leaf nodes since such fragments contain most of the data. The "upper" part of an RG, including the root node (see fragment f in Figure 2) then provides a hook for fragments located in the lower portion of the RG.

Before we illustrate the allocation approach for a complete and disjoint set of fragment specifications, it is important to note that the language underlying fragment specifications can easily be extended to include branching ([ ]) and/or conditions on attribute and text values. Thus far, the language XF supports a *vertical fragmentation* approach based on schema structures since all rooted label paths to fragments are disjoint (and common prefixes of label paths will serve as "join attributes" for reconstruction later). The addition of branching and conditions on text/attribute values naturally leads to a *horizontal fragmentation* approach since then the root nodes of two fragments are allowed to have the same label path. Adding branching, however, can lead to some undesirable features regarding checks for disjointness and completeness. For example, assume two fragment specifications $f_1 = /A/B[C]/G, f_2 = /A/B[D]/G$, none of them containing exclusion fragments. If it is known that every B node always has a C and a D node as children (e.g., based on the co-occurrence information in the RG, see Section 2.3), then these two specifications are not disjoint. Naturally, the more complex branching expressions are used in selection and exclusion fragments, the more complex the decision problem regarding disjointness becomes. We are currently studying such types of language extension in the context of more expressive fragmentation schemes, utilizing recent results on query containment (in the presence of schema information), e.g., [10, 11].

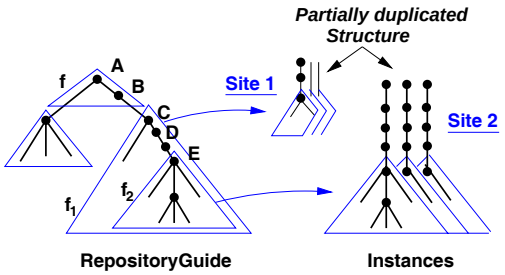

Figure 2: Fragmentation of a RepositoryGuide (l) and distribution of fragment instances to sites (r)

The inclusion of conditions on attributes and text nodes can be dealt with in a fashion analogous to horizontal fragmentation in relational databases. For example, for a fragment specification $f = /A/B[C < 12]$, one can easily derive the complement fragment $f' = /A/B[C \geq 12]$ to ensure completeness.

3.2 Fragment Allocation

The allocation based on a set of complete and disjoint fragment specifications $\mathbf{F}$ involves (1) determining which fragments to allocate at which sites, (2) placing schema structures at local sites, and (3) placing suitable instances of fragments at local sites. In the following, we will outline our approach underlying steps (2) and (3). Step (1) can be dealt with using existing allocation models for relational databases [3, 12]. In the following, we assume that with each site $S \in \mathbf{S}$, a set $\mathbf{F}_S = \{f_1, f_2, \ldots, f_s\}$ of fragment specifications is associated. Although we assume that a fragment $f \in \mathbf{F}$ can be replicated at several sites, for each fragment, there is exactly one master site.

Local Schema Structures. Recall that a RepositoryGuide (RG) serves as a global conceptual schema in the proposed distribution approach. To support the processsing of global queries at local sites, the RG is fully distributed to each site and extended in the following way. For each node $v \in \mathbf{V}_{RG}$, the site that stores the fragment is recorded. In case of replicated fragments, the master site and replicating sites are recorded. We call such an extension of the RG a *Distribution RG (DRG)*.

Placing Fragments on Sites. Fragments at a site consist of local structure and text content according to the DRG's fragment specification. In addition, the *global context* of each fragment is kept in the form of the data path from the global root node to a local fragment's root (s. Figure

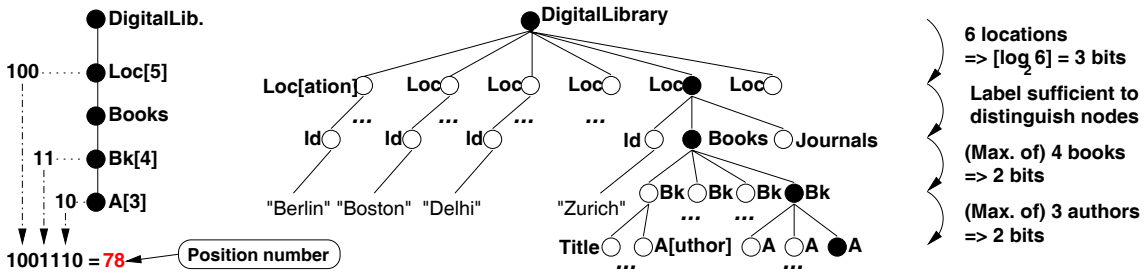

Figure 3: Encoding of data path /DigitalLibrary/Loc[5]/Books/Bk[4]/A[3] in example source

2). This global context serves as "join attribute" for the reconstruction of the full source from distributed fragments.

Adding new data to a virtual XML repository involves the following steps, which are independent of the site where the operation is issued. First, the DRG has to be consulted to determine the fragment f in the DRG that contains the root node of the inserted data. Next, the add operation is sent to the site S that holds the master copy for f. At site S, the operation is executed and at the same time propagated to replicating sites using proper protocols to ensure consistency among all sites. If parts of the added data reach down into another distribution fragment f' below f in the DRG, the master site for f also has to propagate the add operation with content related to f' to the master site for f'. There, the same procedure as described above is applied.

For instance, in the example in Figure 2, if a new C-node with all its related content is added, the master sites for f_1 and f_2 have to be involved. However, notice that because all fragments on local sites replicate the global context path, it is not necessary to involve sites related to distribution fragments higher in the DRG tree (f in the example) when placing new fragments. Furthermore, global context data paths that already exist at a site suffice to compute the global context path for newly added data. Update operations other than insertions have to be propagated in a fashion analogous to placing new data.

4. REALIZATION

The efficiency of our distribution scheme is based on duplicating global path information that leads to XML fragments at local sites. With the global context available locally, it is possible to (i) answer (most) global queries locally, and (ii) easily reconstruct fragments that are distributed over multiple sites. However, storing the global context for every local fragment is potentially expensive. We address this issue by utilizing a new node identification scheme called μPIDs ("micro-Path IDs") [4] that effectively encodes rooted data paths and thus global context. Furthermore, we employ a dense storage of these node ids within path and term index structures. We introduce the node id scheme and index structures in Sections 4.1 and 4.2, respectively. In Section 4.3 we outline a distributed query processing approach. In Section 4.4, we evaluate our approach experimentally.

4.1 μPID Node Identification Scheme

μPID node ids as used in our approach consist of a pair of integers. The first integer, called *node number*, is an identifier for a rooted label path as found in the Distribution RepositoryGuide (DRG). The second number is called *position number* and contains all information to identify a particular instance of the label path in a (distributed) source.

Figure 3 shows the construction of the position number for the data path /DigitalLibrary/Loc[5]/Books/Bk[4]/A[3] in the source shown at the center of the figure (the data path is highlighted by black nodes). The position number (bit-)accurately encodes sibling positions whenever multiple siblings with the same label occur. In the figure, this is the case for the Loc[ation], Bk, and A[uthor] nodes. For node types without such multiple siblings, no explicit sibling number has to be encoded (e.g., for Books in the example). The number of bits required at each step within a data path is determined by the maximum fanout v_{max} (s. Section 2.2) of a certain node type anywhere in the source. In Figure 3, assuming every Books node in the source has at most four Bk children, two bits are sufficient to encode Bk sibling positions.

Maximum fanout information is part of an RG. For a mostly static source that already exists at the time of distribution, the fanout can be derived from the source. Otherwise, the fanout can be derived from estimates and specific schema information that might exist for parts of the repository. This procedure to determine the fanout is also state of the art in other node id schemes [9] that, however, are less suitable to support distributed storage and querying. Furthermore, our empirical studies on real-world data sources show that most node types vary only little in their number of siblings, and therefore our encoding of node ids is quite effective (s. Section 4.4 and [4]).

Position numbers are constructed by appending bits related to single sibling numbers within a rooted data path. Therefore, all position numbers related to a certain node number, i.e., rooted label path, have the same bit length. Hence, they can be interpreted as k-bit integers, as shown on the left side of Figure 3.

A node's μPID allows for directly deriving the node's parent node, ancestors, and preceding siblings. Other node relationships need to be derived as in related approaches through structural joins. For this, relationships between two nodes can simply be determined by comparing node numbers and matching position number prefixes. Furthermore, μPIDs require structural joins only at branch positions within a query pattern rather than for every edge.

Source	Size	P-index	A-index	T-index	Index total	Pos#max	Pos#avg
Big10	1243.1	23.1 (1.9%)	51.9 (4.2%)	347.1 (27.9%)	422.1 (34.0%)	41	22.5/24.4
Reuters	1354.0	55.9 (4.1%)	70.2 (5.2%)	518.9 (38.3%)	645.0 (47.6%)	29	25.7/26.9
XMark 1Gb	1118.0	24.2 (2.2%)	63.8 (5.7%)	310.6 (27.8%)	398.6 (35.7%)	29	21.1/19.8
Fin'l Times	564.1	0.9 (0.2%)	10.5 (1.9%)	115.0 (20.4%)	126.4 (22.4%)	21	20.1/18.0
DBLP	127.7	5.1 (4.0%)	11.4 (8.9%)	35.7 (28.0%)	52.2 (40.9%)	26	20.2/19.2
SwissProt	109.5	10.3 (9.4%)	10.0 (9.1%)	25.8 (23.6%)	46.1 (42.1%)	30	21.5/22.8

Table 1: Source and index size in Mb (% of source size), and pos# length in bits for some XML sources

4.2 Index Structures

Core design. Our distribution scheme employs the same core index structures as other, centralized structural join approaches (e.g., [2, 9, 15]). However, in our approach, lists of μPIDs are always grouped by node number. Thereby, we avoid having to structurally join node id lists that cannot have matches based on their label path. Furthermore, in a list of μPIDs, only μPID position numbers need to be stored repeatedly. In each list, these numbers are of fixed length even though path identifiers in general vary in their length.

In our scheme, a *path index* (P-index) maps node numbers to lists of position numbers in document order. Not all position numbers in the P-index are actually stored. Because of the way μPIDs are constructed, position numbers in document order are consecutive sequences of (k-bit) integers with some gaps [4]. Therefore, for each consecutive sub-sequences, we store only the first position number and its index position in the full sequence. These index positions have another function as direct pointers into a second index structure. This additional index (*address index* or A-index) maps logical μPIDs to physical data addresses. Analogously to the path index, a *term index* (T-index) maps terms to lists of μPIDs for nodes the terms occur in. These lists are grouped by node number, too, but do not utilize the sparse storage structure outlined above.

The μPID-based P-index and T-index together allow a query engine to process *path* pattern queries without having to reconstruct these patterns from ancestor-descendant relationships through expensive structural join operations. Path patterns are matched against the DRG and result in μPID lists for node numbers that match the patterns. If a path pattern ends in a term containment condition, the T-index is accessed, otherwise the P-index. Tree patterns have to be resembled from path patterns. For this, lists related to path patterns have to be joined for every branch point in a tree pattern [4]. On the left side, Figure 4 shows core

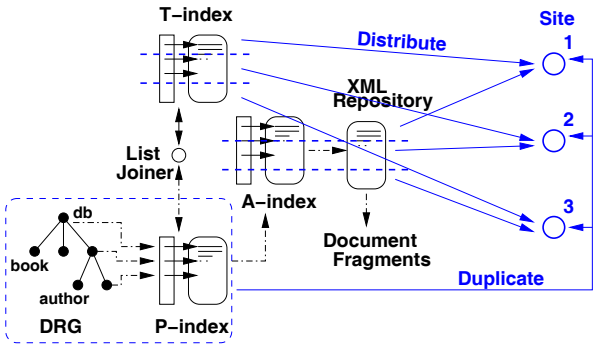

Figure 4: Index structures and their distribution

index structures and access paths as employed for a centralized indexing and query processing approach. The rest of the figure presents the index distribution scheme and is discussed next.

Index distribution. Our distribution scheme distributes nodes in a source by common label paths. Label paths uniquely relate to node numbers in the (D)RG, which relate to well-defined units within all index structures. Therefore, it is straightforward to distribute source fragments together with their accompanying index portions as indicated in Figure 4 for three sites.

However, depending on the size of index structures, it can be beneficial to replicate frequently used index portions at sites that do not keep the related fragments. In particular, we assume that the DRG is fully replicated at each site. Furthermore, P-indexes are remarkably small, as shown in Section 4.4. Therefore, in most cases, fully replicating the P-index is possible at little overhead. For the T-index, duplicating at least parts related to frequently used terms can be cheap and very effective in speeding up query processing. A-index portions, which provide for a translation from logical node ids to physical addresses as the last step in query processing, only need to be kept with their related source fragments at a single (or replicating) site(s).

4.3 Distributed Query Processing

Queries can be processed as follows. A query tree pattern is matched against the local DRG to determine potential matches based on label paths. Instances of these pattern matches then can be stitched together based on the locally available P-index. Only for term conditions whose related T-index portions are not locally replicated, a remote site has to be accessed. Information about sites providing the missing T-index parts can directly be found in the DRG.

For distributed query processing, the most prominent cost factors to process pattern queries is the transfer of node id lists to a common site in order to join the lists. Therefore, the main goal of an efficient distribution scheme using structural joins is to minimize transfers of these lists.

For example, in the source in Figure 3, Books-related fragments with their T-index portions might be stored at Site A while all other information is stored at Site B. The tree pattern query in Figure 1 can thus be processed at Site A by first matching the /Location/Books//Title pattern locally. Then, only those node ids for /Location/Id fragments that contain the term "Zurich" are fetched from Site B and joined with the earlier matches. If the local result is small, another option to process this query would be to send the matches at Site A to Site B, execute the structural join at Site B, and

send the results back. Such optimization options are analogous to optimizations in distributed query processing for relational database and are subject to our current research.

4.4 Evaluation

Our goal in this section is to clearly show that P-index and even parts of the T-index are small enough to be fully replicated, thus reducing distributed query processing to mostly local query processing. Efficiency of local query processing has already been shown [2]. Therefore, even without an extensive, fully distributed prototype system, these observations allow us to conclude that the XML distribution approach we have presented in this paper is realistic and efficient in terms of storage space and query processing.

Table 1 summarizes the sizes of the P-, A-, and T-indexes for different non-distributed sources. Sizes for distributed or replicated index structures can be directly derived from the numbers given in the table. In our implementation, physical addresses as stored in the A-index are offsets into an underlying plain text source. The sources presented in the table are generated by the XMark XML generator,[1] or originate from TREC[2] disk 4 and 5, Reuters corpus,[3] or from the Web.[4] They have different structural complexity, which is the main reason for variations in relative index sizes. For instance, *Financial Times* has little structure and source depth and thus, very small indexes. *SwissProt* on the other hand is unusually rich in structure and thus, the path index in particular is relatively large. *Big10* is a composite source consisting of nine sources from the aforementioned sites. Table 1 also lists the maximum and average (for P-index on the left and T-index on the right) μPID position number length. In earlier, equivalent indexing approaches, index size is frequently ignored [2, 9], or reported to be at least four times as large as shown here without providing a mapping of logical node ids to physical addresses [15].

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a complete distribution approach for XML repositories, including fragmentation, allocation, and efficient realization. Fragmentation of XML data and allocation of fragments at local sites build on known concepts from relational databases and are extended to account for tree structured data. In particular, we have identified label path-based fragmentation for XML as analogous to vertical fragmentation for relational system. Path and value conditions provide for an analogy to horizontal fragmentation.

In our realization, the expense of replicating global data paths in order to identify fragments at local sites is offset by an efficient encoding of path information using a new node identification and indexing scheme for XML. Our path index structure in particular is small enough to be fully distributed to local sites and thus bringing the performance of query processing close to that of centralized systems.

As indicated earlier, we are currently investigating more expressive fragment specifications in our distribution approach (e.g., those similar to derived horizontal fragmentation based on primary/foreign keys). In particular, in the context of data modifications and complex (global) queries, we are studying cost models for the management of (replicated) fragments that span multiple sites.

[1] monetdb.cwi.nl/xml/generator.html
[2] trec.nist.gov
[3] www.reuters.com/researchandstandards/corpus
[4] www.cs.washington.edu/research/xmldatasets

6. REFERENCES

[1] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, T. Milo. Dynamic XML documents with distribution and replication. To appear in *Proc. of the ACM SIGMOD Conf. 2003*, 2003.

[2] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of the IEEE Int'l Conf. on Data Engineering*, 141–152, 2002.

[3] P. M. G. Apers. Data Allocation in Distributed Database Systems. ACM Transactions on Database Systems (TODS) 13(3): 263-304 (1988)

[4] J.-M. Bremer, M. Gertz. An efficient XML node identification and indexing scheme. Technical Report CSE-2003-04, Dept. of Computer Science, University of California at Davis, 2003.

[5] T. Fiebig, S. Helmer, K.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele. Anatomy of a native XML base management system. *The VLDB Journal*, (11):292–314, 2002.

[6] P. Grosso, D. Veillard. XML Fragment Interchange. W3C Candidate Recommendation, W3C, February 2001. www.w3.org/TR/xml-fragment.

[7] H.V. Jagadish, Laks V.S. Lakshmanan, T. Milo, D. Srivastava, D. Vista. Querying network directories. In *Proc. of the ACM SIGMOD Conf. 1999*, 133–144, 1999.

[8] K. Karlapalem, Q. Li: Partitioning Schemes for Object-Oriented Databases. In *5th Int'l Workshop on Research Issues in Data Engineering-Distributed Object Management*, 42–49, IEEE, 1995.

[9] Q. Li, B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of the 27th Int'l Conf. on Very Large Data Bases (VLDB)*, 361–370, 2001.

[10] G. Miklau, D. Suciu. Containment and Equivalence for an XPath Fragment. In *21th ACM Symposium on Principles of Database Systems (PODS)*, 65–76, 2002.

[11] F. Neven, T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *9th Int'l Conf. on Database Theory*, LNCS 2572, 315-329, 2003.

[12] M. T. Özsu, P. Valduriez: *Principles of Distributed Database Systems (2nd ed.)*. Prentice Hall, 1999.

[13] D. Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems (TODS)*, 27(1):1–62, March 2002.

[14] E. J. Whitehead Jr., M. Wiggins. Webdav: IETF standard for collaborative authoring on the Web. *IEEE Internet Computing*, 34–40, September 1998.

[15] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the ACM SIGMOD Conf. 2001*, 425–436, 2001.

[16] University of Michigan - Timber Project. www.eecs.umich.edu/db/timber.