Distributed XML Repositories:
Top-down Design and Transparent Query Processing

Michael Gertz   Jan-Marco Bremer
Department of Computer Science
University of California, Davis
{gertz|bremer}@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 repositories are distributed over the Web. However, design and query processing models for distributed XML data have not yet been studied in detail.

The goal of this paper is to study the design and management of distributed XML repositories. Following the well-established concepts of vertical and horizontal data fragmentation schemes for relational databases, we introduce a flexible distribution design approach for XML repositories. We provide a comprehensive data allocation model with a particular focus on storage efficient index structures. These index structures encode global path information about XML fragment data at local sites and provide for an efficient, local evaluation of the most common types of global path and tree pattern queries. Finally, we describe the basic principles of a distributed query processing model based on the concept of index shipping.

1 Introduction

Recent advancements in the development of native XML database systems such as Timber [41], Natix [15], or Tamino [35] clearly indicate that XML is not only considered a data exchange but also a data representation format. The management of XML data in such native systems, compared to managing XML data in (object-)relational database systems, leads to several new application and research perspectives. We conjecture that distribution aspects of XML data play an important role in a variety of application domains. In fact, application domains for XML, such as Web services, e-commerce (e.g., xCBL [38] or Association of Retail Technology Standards (ARTS) [18]), collaborative authoring of large electronic documents (e.g., WebDAV [37]), or the management of large-scale network directories [20] show that XML data is inherently distributed over the Web. Thus, systems that manage distributed XML data have to take this aspect into account to allow for a solid design of a distributed XML repository, and efficient and reliable usage of XML data at distribution sites.

Although concepts for the distribution of XML data are clearly important, to this day, there is only little work that deals with distribution aspects of XML data. Only recently the problem of distributed and replicated (dynamic) XML documents has been studied for the first time and fundamental query processing and replication models have been proposed in the context of Web services [1]. Interestingly, the W3C has published a candidate recommendation for XML fragment interchange [17], but respective concepts are neither well-founded nor used in practice. The above works address some distribution aspects of XML. However, to the best of our knowledge, there is no work that studies (1) the top-down design of a distributed XML repository, following traditional and well-studied distribution design principles for relational and object-oriented databases, (2) data allocation schemes

with a particular emphasis on index structures as these play a crucial role in XML query processing, and (3) an efficient and transparent distributed query processing approach for distributed XML data.

In this paper, we address the above three core XML distribution issues in detail. In particular, we are concerned with large-scale distributed XML repositories that are designed from scratch or that provide an integrated view over existing sources which represent their data in XML. More specifically, this paper makes the following contributions.

(1) Distribution Design Based on a *RepositoryGuide* as global schema structure for a distributed XML repository, we describe how vertical and horizontal fragmentation schemes known from relational and object-oriented databases are adapted to XML data. In particular, we provide an XML fragmentation scheme that satisfies known correctness criteria in distributed database design.

(2) Data Allocation and Supporting Index Structures Given a set of fragment specifications, we illustrate an allocation approach for schema information and XML fragment data at local sites. Of particular interest are allocation models for index structures. In distributed relational or object-oriented databases, index structures are primarily used to speed up query processing. In our approach, however, index structures play a central role in encoding global context information of XML fragment data allocated at local sites. We present a novel *path identification scheme*. This scheme employs very space efficient path identifiers to (1) encode positions of local and remote XML fragments in a distributed XML repository, and (2) efficiently process path and tree pattern queries, including term containment conditions. The proposed path index structure has a size of only 2-4% of the indexed source and thus is suitable for replication among multiple local sites.

(3) Distributed Query Processing The index structures in our approach provide the basic means to locate and reconstruct distributed XML fragments. For XPath-like path and tree pattern queries, we detail a distributed query processing approach that is solely based on matching entries of path and term indexes to determine the query result. A core methodology in this distribution setting is the *shipping of index entries* rather than the known concept of combined query and data shipping. Determining a cost-effective shipping of index entries among multiple sites inherently faces the same problem as optimizing (semi-)join orderings of distributed multi-queries. For this, we illustrate optimization strategies that exploit heuristics as well as statistical information about distributed index structures and XML fragments.

The rest of the paper is organized as follows. In Section 2, we detail our approach to vertical and horizontal fragmentation of XML data based on a RepositoryGuide. The data allocation approach and the realization of supporting index structures is presented in Section 3. In Section 4, we introduce a core algorithm for distributed query processing and illustrate optimization strategies. After some experimental evaluations of the proposed framework in Section 5, we discuss related work in Section 6.

2 Distribution Design

The primary objective of a top-down distribution design for an XML repository is to formally describe fragments of an XML repository. Specifications of XML fragments together with site-specific query workload information are then used to allocate schema and data components at local sites. While the design strategy follows basic distribution principles known for relational and object-oriented databases [22, 28], the hierarchical structure and mix of schema information and data in XML require substantial modifications and extensions to respective design approaches. In Section 2.1, we introduce the notion of RepositoryGuide as basic means for our distribution design approach. In Section 2.2, we first introduce important design properties of any XML fragmentation approach, and then detail a flexible fragmentation scheme for an XML repository. This scheme supports the concepts of vertical and horizontal fragmentation and satisfies the stated design properties.

Throughout the paper, we assume that data in an XML repository 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 $\mathbf{V}$, and $text$ assigns text strings to nodes. Without loss of generality, we assume that element attributes are modeled as regular nodes. For an XML repository, we 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, in an XML source multiple rooted data paths can be associated with a single rooted label path.

2.1 RepositoryGuide

Underlying our top-down distribution design approach is a global (conceptual) schema. Based on this schema, XML fragments are specified. While it is conceivable that such a schema exists in the form of a DTD or XML Schema, we use a simplified schema structure, called *RepositoryGuide (RG)*. A RepositoryGuide resembles the 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 (e.g., presented to a user in a distribution design tool) is easier to fragment than a grammar-based schema specification. Second, in the case of a large-scale XML repository to be distributed, it is possible that local sites express their information 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 easily be translated into local RGs and combined into one global RG. In case site-specific schema structures already exist, the global RepositoryGuide is constructed in the same fashion as a global schema in the context of integrating XML views over heterogeneous data sources (e.g., [2, 19, 31]). In the following, we assume a RepositoryGuide $\mathbf{G} = (\mathbf{V}_G, \mathbf{E}_G)$ and a set L_G of node labels for the nodes $\mathbf{V}_G$ in $\mathbf{G}$.

A RepositoryGuide is clearly less expressive than a DTD or XML Schema since it does not represent, e.g., grouping or order information. But, if local schema structures exist in the form of DTDs and/or XML Schemas, nodes in a given RepositoryGuide $\mathbf{G}$ can be enriched by cardinality and co-occurrence information as follows:

(*Cardinality information*) For each node $v \in \mathbf{V}_G$, the minimum/maximum number v_{min}/v_{max} of times this node (element type) can occur as child node in any instance of the XML repository is recorded. For instance, for a node v, as child of a node v', the pair $\langle 0, 2 \rangle$ specifies that v is an optional child element and occurs at most 2 times as child of v' in any XML repository instance. Naturally, different min/max values can be associated with nodes that have the same label in $\mathbf{G}$.

(*Co-occurrence information*) For pairs v, v' of nodes in $\mathbf{G}$, each node corresponding to a rooted label path in $\mathbf{G}$, it is recorded whether these two paths always co-occur in all instances of the repository. The most simple cases of co-occurrences are required siblings of a node.

Cardinality information plays an important role in the data allocation model presented in Section 3. It is assumed that cardinality information is either explicity specified by the designer or obtained from site-specific DTDs or XML schemas (in the latter case derived from occurrence constraints), if such exist. In comparison, co-occurrence information, although not essential, can provide useful input to some decision problems for fragment specifications. In particular, we assume that co-occurrence information is not available for all pairs of nodes in $\mathbf{G}$.

2.2 Vertical and Horizontal Fragmentation

Any data fragmentation scheme has to satisfy certain correctness criteria in order to ensure that the semantics of the data and queries does not change once fragment data is distributed and managed at local sites. In the context of relational databases, these distribution design criteria are known as *completeness, reconstruction,* and *disjointness rules* [28] and concern a set of fragment specifications. We devise similar criteria for fragmenting XML data.

In the following, we assume a set $\mathbf{S} = \{S_1, \ldots, S_r\}$ of local sites among which XML repository data is to be distributed. With each site S_i, a set of queries $\mathbf{Q}_{S_i} = \{q_{i1}, \ldots, q_{il_i}\}$ is associated. From these queries, typical

access patterns in the form of a sublanguage of XPath can be derived. Such access patterns comprise *path and tree patterns* [4] as they are used in local, site-specific queries formulated in XQuery, XPath, or XSLT. The RepositoryGuide provides the basis for specifying fragments. Our design approach supports the concepts of vertical and horizontal fragmentation, similar to those known for relational and object-oriented databases. Vertical fragments are solely based on the selection of node types through path properties. Horizontal fragments additionally include conditions on branching and/or values nodes have to satisfy. In general, both types of fragments determine subtrees in a given RG. To allow for partial subtrees, we introduce so-called exclusion fragments for vertical and horizontal fragments.

Our fragmentation scheme ensures that the RepositoryGuide and thus any XML repository conforming to the RG is decomposed into a disjoint and complete set of tree-structured XML data fragments. In the next section, we detail how a set of fragment specifications is used to determine the allocation of data and index structures at local sites, thus providing the basis for the reconstruction of the complete repository from distributed fragments.

Vertical Fragmentation The specification of vertical XML fragments is based on a sublanguage of XPath, called XF, that includes the context node (.), descendant and child axis ($/\!/, /$), and wildcard ($*$). Element names referred to in an XF expression correspond to node labels in L_G.

The specification of a vertical fragment $f = sf - \{ef_1, \ldots, ef_n\}$ 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 straightforward: first, the selection fragment sf is evaluated against $\mathbf{G}$ and results in a set of nodes. The evaluation can be done very efficiently since a RepositoryGuide $\mathbf{G}$ (as single XML document) is much smaller than an XML repository corresponding to $\mathbf{G}$. Each such node determines a subtree in $\mathbf{G}$, called *RG fragment*. Optional specifications $\{ef_1, \ldots, ef_n\}$ of exclusion fragments are evaluated only on such subtrees and again determine RG fragments. These fragments (as subtree structures) are to be excluded from the tree structures determined sf.

Assume, for example, the fragment specification $/\!/A/\!/C - \{./D/*/E\}$. First, all subtrees in $\mathbf{G}$ 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, and have a D node as grandparent, which is child of a C node selected by the sf expression. Fig. 1 illustrates this case where fragment f_2 is excluded from fragment f_1; it also shows a fragment f_3 from which no other fragment is excluded.

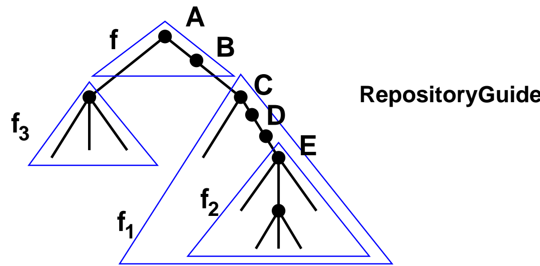

Figure 1. Fragmentation of a RepositoryGuide based on selection and exclusion fragments

Checking a set of specifications of vertical fragments $\mathbf{F_V} = \{f_1, f_2, \ldots, f_l\}$ for completeness and disjointness is trivial. Therefore, we only outline the algorithm used in our design approach. Let $desc(f_i)$ denote all nodes in $\mathbf{G}$ that are contained in the subtree determined by a vertical fragment f_i, excluding nodes determined by optional ef's in f_i's specification. $\mathbf{F_V}$ is said to be *complete*, if all nodes $\mathbf{V}_G$ are contained in at least one fragment, i.e., $\mathbf{V}_G = \bigcup_{i=1}^{l} desc(f_i)$. The fragment specification in $\mathbf{F}_V$ are said to be pairwise *disjoint* if there are no two fragment specifications $f_i, f_j \in \mathbf{F}_V, i \neq j$, such that $desc(f_i) \cap desc(f_j) \neq \emptyset$. Given $\mathbf{F_V}$, disjointness is verified

by marking each node $v \in \mathbf{V}_G$ 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. $\mathbf{F_V}$ is complete if every node in $\mathbf{G}$ is marked. If $\mathbf{F_V}$ is not complete, a complete set $\mathbf{F}'_\mathbf{V}$ of vertical fragments is obtained by determining the rooted label paths to root nodes of subtrees that have not been marked (including exclusions that already have been marked); such paths then describe the vertical fragment specifications to be added to $\mathbf{F_V}$. Based on the above discussion, it is obvious that exclusion fragments are necessary. Otherwise, a fragment that selects the root node of $\mathbf{G}$ would always cover all other nodes.

In this context, it is also interesting to note that in a practical setting, the "upper" part of a RepositoryGuide, including the root node (see fragment f in Fig. 1) provides a "hook" and context for fragments located in the lower portion of the RepositoryGuide.

Horizontal Fragmentation In relational and object-oriented databases, horizontal fragmentation is based on very simple conditions that attribute values of tuples and objects, respectively, have to satisfy. This aspect is reflected in our distribution design approach by adding branching ([ ]) and conditions on attribute and text values to the language XF. The addition of branching allows for the selection of nodes not only based on their rooted label paths, as in the vertical fragmentation scheme, but also based on other structural or value properties along or below that path.

To better illustrate the main principles of horizontal fragmentation, in the following, we only consider fragment specifications that do not contain exclusion fragments. Such exclusions can be dealt with in a way similar to checking for disjointness and completeness of vertical fragments.

Now assume the design scenario where some nodes in a RepositoryGuide $\mathbf{G}$ have already been marked as covered by vertical fragments. For a horizontal fragment, it is first determined whether the selected node(s) are not already covered by a vertical fragment. This requires only a simple checking of the node marking. Therefore, we will focus on disjointness and completeness properties for horizontal fragments that specify nodes in $\mathbf{G}$ not already covered by vertical fragments.

(Disjointness) Assume the specification $f = /A/B[D]/\!/E$ of a horizontal fragment. It defines those subtrees in $\mathbf{G}$ that are rooted at a node labeled E, have a B node with child D as ancestor, and this B node is a child of the root node. f obviously has a "horizontal character" since there might be other E nodes reachable through $/A/B/\!/E$ that do not have a D as child of a B node. Now assume another fragment specification $f' = /A/B/\!/E[C]$. Like f, it selects E nodes, but based on a different structural property. The two fragments are disjoint if there is no instance of the XML repository such that for a label path to an E node that matches $/A/B/\!/E$ in $\mathbf{G}$, a B node has a D child *and* the E node has a C child. It is clear that if no schema information is available, this problem is in general undecidable, i.e., it is undecidable whether $f \cap f'$ is satisfiable [25]. However, if co-occurrence information about the above D and C nodes is available in $\mathbf{G}$, this disjointness property can be decided (in practice, this is often the case for co-occurring sibling nodes). In our current approach, we utilize available co-occurrence information to verify such disjointness properties. If such schema information is non-existent or insufficient, we employ algorithms to check for query containment [26, 27]. Clearly, if there is a containment relationship between a pair of fragments, then they are not disjoint, and non-containment does not imply that they are disjoint. We are currently studying the types of horizontal fragments (i.e., language restrictions) and necessary schema information as part of a RepositoryGuide that provide for efficient procedures to decide disjointness of horizontal fragments.

(Completeness) If for a node $v \in \mathbf{V}_G$, all horizontal fragments that select that node are known to be pairwise disjoint, it finally needs to be ensured that all possible instances of that node in an XML repository are covered by horizontal fragments. In our approach, this is achieved by employing a concept similar to minterms in the context of horizontal fragmentation in relational databases [28]. Assume a horizontal fragment $f = /A/B[D]/C[E]$. In order to ensure that all C nodes reachable through $/A/B/C$ are covered, we derive the horizontal fragments $f_1 = /A/B[\text{not } D]/C[E], f_2 = /A/B[D]/C[\text{not } E], f_3 = /A/B[\text{not } D]/C[\text{not } E]$ from f. Some of these

specifications can be redundant or unsatisfiable, dependent on co-occurrence information available in the RG $\mathbf{G}$.

A similar yet much simpler approach is taken for specifications of horizontal fragments that contain conditions on attribute or text values. Assume the specification $f = /B/\!/E[F > 50]$. It selects all E nodes that have a B node as ancestor and whose value for the F node (which can be an attribute or element) is greater than 50. Several conditions can be associated with instances of such E nodes in $\mathbf{G}$ and these can easily be checked for satisfiability. In particular, complete coverage of such E nodes through respective horizontal fragments is established again using the concept of minterms; in this case by deriving a specification $f' = /B/\!/E[F <= 50 \text{ or not } F]$. Combinations of conditions and specifications of horizontal fragments that contain both branching and attribute/text conditions can be dealt with in a fashion analogous to the one described above.

3 Data Allocation and Supporting Index Structures

Given a complete and pairwise disjoint set $\mathbf{F}$ of vertical and horizontal fragment specifications, the next task in our distribution approach is the allocation of resources at a set $\mathbf{S}$ of sites. The allocation problem involves finding an "optimal" distribution of $\mathbf{F}$ to $\mathbf{S}$. Optimality is typically characterized in terms of two measures: (1) minimal cost of storing, querying, and updating fragment data at different sites, and (2) minimal response time for operations and maximal system throughput.

The data allocation problem is non-trivial and depending on numerous factors. It has been studied in different contexts and from different viewpoints for more than 20 years (see, e.g., [5, 28]). Naturally, the resources to be allocated in a distribution framework for XML repositories include global schema information and XML fragment data. However, just placing fragment data at local sites, i.e., subtree structures that reside in the context of a larger, single distributed XML source, is not sufficient. Solely placing fragment data at sites would not provide any query processing technique for XML data with necessary and sufficient context information to correctly place the fragment data in the repository. That is, such an approach would not satisfy the reconstruction requirement of a distribution design approach. This is different from traditional approaches used in relational databases where the placement of dictionary information and relations corresponding to fragment specifications suffices. In order to solve this problem for distributed XML data, we introduce a path index structure that (1) space efficiently encodes information about the global context of local XML fragments, and (2) provides for an efficient distributed query processing model.

In the following, we discuss our allocation approach for schema structures, XML fragment data, and supporting index structures step by step. The allocation approach for schema information and XML fragment data is presented in Sections 3.1 and 3.2. We then detail our index scheme comprising path, term, and address index, including allocation alternatives in Section 3.3. In Section 3.4, we outline the handling of data modification for the above distributed structures.

3.1 Schema Information

The RepositoryGuide introduced in Section 2.1 represents the schema structure for the specification of horizontal and vertical fragments. While most query processing approaches for XML data do not rely on any schema structure in their query processing schemes, we require that the RepositoryGuide is replicated among all sites in $\mathbf{S}$. The RepositoryGuide is reasonably small in terms of required storage space. More importantly, in our approach, the RepositoryGuide is used to process global queries at local sites. For this, the RepositoryGuide maintains information about which XML data fragments and index components are located at which sites. In the following sections, we will incrementally add distribution information to such a fully replicated RepositoryGuide.

3.2 XML Fragment Data

Given a set $\mathbf{S} = \{S_1, \ldots, S_r\}$ of sites and a set $\mathbf{Q} = \{q_1, \ldots, q_l\}$ of path and tree query patterns used for determining horizontal and vertical fragments. The allocation of XML fragment data determines at which sites to locate which fragments. In relational databases, solutions to this problem are typically based on whether or not and how often a relation is used in a query expression. This methodology and existing solutions could be adapted to XML data fragments. However, this approach would then assume that queries directly operate on XML data. A closer inspection of path and tree query patterns and existing approaches to processing queries formulated in, e.g., XPath or XQuery, however, show that a query has (1) exactly one selection node that determines the root node of fragments to be retrieved, and (2) often multiple conditions formulated in the form of branching and/or attribute and text conditions. Determining selection nodes that satisfy the query pattern are typically based on index structures to identify structural relationships among nodes as, e.g., in the widely used structural join approach [4]. It is thus reasonable to distinguish between (potential) fragments that are used for evaluating such conditions and (potential) fragments actually returned as query results.

Based on the above observations, our allocation approach for XML fragment data is realized as follows. First, based on individual frequencies and origins of the queries $\mathbf{Q}$, site constraints (storage), and fragment specifications (recall that those are derived from the queries in $\mathbf{Q}$), we use a simplified version of the approach described in [28] to determine which fragment $f_i \in \mathbf{F}$ should be allocated at which site $S_j \in \mathbf{S}$. However, for a query $q_l \in \mathbf{Q}$, we do not take all fragments (in RG) affected by the query pattern into account. We only consider those fragments f_i whose nodes that are located in the subtree(s) determined by the selection node in q_l. The result is a matrix that shows what fragments are located at what sites. If there are fragments in $\mathbf{F}$ that are not related to any selection nodes of the queries but referred to in conditions or not used at all in any query, these fragments are placed arbitrarily, but observing site constraints.

As a result of this approach, instances of XML fragment data corresponding to fragment specifications are placed on sites, and each fragment is placed on exactly one site. That is, in our current approach we do not consider the replication of XML fragment data. Based on the methodology described in [1], we are currently studying the extension of our data allocation scheme to allow for a dynamic replication of XML fragment data. Finally, for each node in the RepositoryGuide, it is recorded which site manages instances of XML fragment data that contain that node. This information will be used in the final step of our distributed query processing approach, as discussed in Section 4. Recall that the allocation of XML fragment data at local sites does not yet include global context information about the fragments. This is achieved through supporting index structures, which are discussed next.

3.3 Supporting Index Structures

To ensure reconstruction of arbitrary XML repository components from local XML data fragments, and to provide for efficient distributed query processing, our approach utilizes supporting index structures, consisting of a path, term, and address index. The path index efficiently encodes information about the global context of local XML fragments in a distributed repository. While it is conceivable to choose any of the existing index structures for XML data (e.g., [10, 12]), the primary objective in a distribution setting for XML data is to find the right balance between the (1) space efficiency of an index, making it suitable for replication and shipping during distributed query processing, and (2) expressiveness of the index, that is, the extend to which index entries exactly describe the global location of a node in a local XML fragment.

Since in our approach to distributed XML repositories the primary goal is space and query efficiency, we make use of a limited interpretation of node order. More precisely, we assume a partial order among nodes in which only the *relative positions among sibling nodes carrying the same label are considered significant*. For example, while for a book element, the order of authors (as child nodes) matters, the order among title and author child nodes of

a book element is likely much less relevant. This order criterion is sufficient to support most types of queries; in fact, almost all XML query use cases [11, 39] can be answered correctly under this assumption.

For our index structures, we therefore utilize a path identification scheme called μPIDs (”micro-Path IDs”) [9] that effectively encodes complete rooted data paths and thus global context based on the above node ordering assumption. Furthermore, we employ a dense storage of these path ids within path and term index. We introduce the path identifier scheme and index structures in Sections 3.3.1 and 3.3.2, respectively. In Section 3.3.3, we detail the distribution alternatives of the index structures.

3.3.1 μPID Scheme

μPID path ids consist of a pair of integers. The first integer, called *path number*, is an identifier for a rooted label path to a node as found in a given RepositoryGuide $\mathbf{G}$; that is, with each node $v \in \mathbf{V_G}$ a path number is associated. The second number is called *position number* and contains all information to identify a particular instance of the label path in a (distributed) source.

Fig. 2 shows the construction of the position number for the data path /DigitalLibrary/Loc[5]/Books/Bk[4]/A[3] in the repository 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). The number of bits required at each step within a data path is determined by the maximum fanout v_{max} (s. Section 2.1) for the related node type anywhere in the repository and is recorded in $\mathbf{G}$. For the example in Fig. 2, assuming every Books node along that label path has at most four Bk children, two bits are sufficient to encode Bk sibling positions.

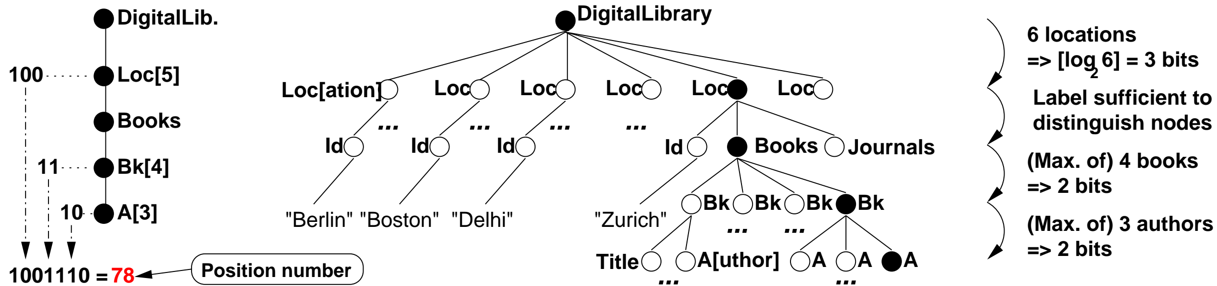

Figure 2. Encoding of data path /DigitalLibrary/Loc[5]/Books/Bk[4]/A[3] in example XML repository

Maximum fanout information is part of the RepositoryGuide. For a mostly static repository that already exists at the time of distribution, the fanout can be derived from the repository. Otherwise, as discussed earlier, estimates and specific schema information that exist for parts of the repository or RG can be used. This procedure for determining the fanout is also state-of-the-art in other node id schemes [24] that, however, are less suitable to support distributed storage and querying. Furthermore, our empirical studies of real-world XML data repositories show that most node types vary only little in their number of siblings, and therefore our encoding of path ids is quite effective [9] (see also Section 5).

Position numbers are constructed by appending bits related to single sibling numbers within a rooted data path. Therefore, all position numbers related to the same rooted label path and thus path number have the same bit length. Hence, they can be interpreted as k-bit integers, as shown on the left side of Fig. 2.

A data path's μPID allows for directly deriving the parent node of the node determined by the μPID, its ancestors, and siblings. Other node relationships need to be derived as in related approaches through structural joins [4]. For this, relationships between the two nodes described by two path ids can simply be determined by comparing

path numbers and matching position number prefixes. Furthermore, μPIDs require structural joins only at branch positions within a query pattern rather than for every edge, an important aspect we will discuss in distributed query processing approach.

3.3.2 Path, Term, and Address Index

We use μPIDs within two standard index structures of the structural join query processing approach [4, 24, 40]: a *path index* (P-index) and a *term index* (T-index). The term index allows for processing query patterns that include term containment conditions. A term is an atomic text string and is obtained during tokenization of XML data. The path index allows for processing all other parts of path and tree pattern queries. However, our indexes have two major differences to most existing approaches. Instead of using flat lists of μPIDs in document order, μPIDs are grouped by their path numbers in both index structures. This is equivalent to grouping ids by common rooted label path, and clearly ideal for distributing index portions with their related XML fragments.

The grouping also allows for storing only the μPIDs' position numbers repeatedly, avoiding redundant storage of path numbers. Within each group, the number of bits for position numbers is constant. Therefore, every element in the group can be directly addressed. Furthermore, the grouping provides for a more efficient processing of distributed pattern queries. This is, because a query pattern can first be matched against the relatively small local RepositoryGuide in main memory. The RepositoryGuide then supplies matching path numbers and information about sites with XML fragments containing these nodes. Only μPIDs related to matching path numbers have to be considered within local and/or remote index structures.

As a major improvement of the path index, we employ a sparse storage of position numbers. Position numbers related to the same RG path number and in document order build a sequence of integers with gaps like **0**, 1, 2, **5**, 6, **10**, 11, 12, etc. Instead of storing all these elements, we store only the start element in each continuous sub-sequence together with the element's position in the full list, in the example (**0**,0), (**5**,3), (**10**,5), etc. (see also Fig. 3).

Example 3.1 In Fig. 2, consider position numbers for author nodes as in the marked path. The marked author node, respectively its path, has number 78. Its two preceding author siblings have the position numbers $(1001101)_2 = 77$ and $(1001100)_2 = 76$. The construction is analogous to the one shown on the left side of the figure. Assuming the preceding book node, Bk[3], has only two authors, data paths to these author nodes would carry the position numbers $(1001000)_2 = 72$ and $(1001001)_2 = 73$. Hence, the subsequence of /DigitalLibrary/Loc/Books/Bk/A path numbers for these two books is 72, 73, <gap>, 76, 77, 78.

An important observation, which we will confirm in our experiments in Section 5, is that in real-world XML data sources these gaps are relatively rare. Therefore, the path index size is significantly reduced by sparse storage. Moreover, sparse storage has another use. The index position associated with each position number that is actually stored can also be used as direct pointer into another index structure, which stores μPID-related physical addresses. We call this index *physical address index* (A-index). As the path index has to be accessed for query processing anyway, there is no additional cost associated with obtaining direct pointers into the A-index. Physical addresses then allow for the access to XML fragments. Physical addresses can be offsets into a file, tuple ids in a relational database system, OIDs in an object database system, etc. Interestingly, in existing structural join approaches, the mapping of logical node identifiers to their physical counterparts is frequently ignored although they contribute significantly to the overall storage cost associated with an index structure.

Fig. 3 shows the overall conceptual design of (centralized) P-, A-, and T-indexes. The figure also shows an additional, small B^+-tree index as an option to speed up non-linear access to path identifiers in the path index. Such index could also be employed on some lists in the term index, depending on typical access patterns. Non-linear access can be useful for joining μPID lists that are relatively short with respect to the length of the complete list [10, 12].

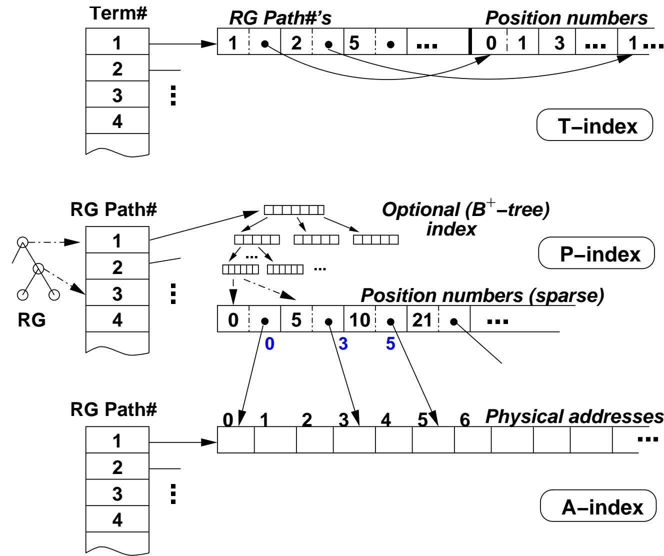

Figure 3. Core Index Structures: Term (T), Path (P), and Address (A) Index

3.3.3 Index Distribution and Replication

In the following, we outline our approach to distributing and replicating index structures among sites. Recall that in our distribution design approach, fragments can be described by rooted label paths in the RepositoryGuide. For vertical fragments, these paths are unique, that is, no two vertical fragments share the same rooted label path. For horizontal fragments, the same rooted label path may be shared by two or more horizontal fragments. Rooted label paths uniquely relate to path numbers in the RepositoryGuide. These path numbers correspond to well-defined units within all index structures, because all index lists are grouped by path number. Therefore, it is straightforward to distribute XML fragment data together with their accompanying index portions either as full index lists (vertical fragments) or partitions of such lists (horizontal fragments). For example, assume the horizontal fragment specification

f = /DigitalLibrary/Loc[Id="Chicago"]/Journals,

and its complement

f' = /DigitalLibrary/Loc[Id<>"Chicago"]/Journals.

Furthermore, assume that XML data instances of f are managed at site S_1 and instances of f' at site S_2; this information is recorded in each local RepositoryGuide. Then, the path index list related to the rooted label path /DigitalLibrary/Loc/Journals is partitioned into two sub-lists, one partition for f stored at site S_1 and the other partition for f' stored at site S_2.

The distribution of the term index is analogous in general. However, path id lists for potentially multiple terms are partitioned. List entries related to terms that occur in journals in Chicago are distributed to site S_1 as the first partition of each list. The remaining entries build the second partition in each list and are stored at site S_2.

Depending on the size of the P- and T-index, it can be beneficial to replicate frequently used index portions to sites that do not keep the related XML fragment data. P-indexes are remarkably small, as shown in Section 5. Therefore, in most cases, the P-index can be fully replicated to all sites at little storage overhead, a primary objective of our distribution approach as detailed at beginning of this section. This means that each site can locally process all tree pattern queries not involving any term conditions, even though the XML fragment data referred to by the query may be distributed over many sites. Whether or not a full replication of a P-index is suitable also depends on the specific setting. For example, frequent data modifications can make the replication

too expensive, because the modifications have to be coordinated. Moreover, local and global query patterns can make the replication unnecessary.

For the distribution of the T-index, the situation is different since the T-index is typically much larger than the P-index. A T-index can be distributed in several ways:

- *Full distribution.* Each site keeps only T-index portions related to local XML fragment data. This implies that structural components of queries with a term condition are evaluated based on the P-index first, to then contact only sites that keep respective term index lists.
- *Two-tier term index.* The T-index is fully distributed. But, in addition, each site keeps a list of all terms with the sites they occur at. This allows for pinpointing exactly the sites that contain T-index portions related to a query.
- *Partial replication.* This alternative is reasonable if XML repository data is fairly static, and local sites have sufficient resources or local query patterns that always involve only small parts of the T-index.

In our current prototypical distribution environment, we do not employ any replication technique for the T-index but partially replicate portions (i.e., list for some path numbers) of the P-Index, depending on the access frequency and patterns of the queries individual sites.

A-index portions only need to be accessed for translating the logical μPID path ids to physical addresses as the last step in query processing. In our approach, A-index entries are always allocated at the site that manages the XML fragment data as determined by the allocation model described in Section 3.2.

In summary, as a basic distribution model for XML repository data, we assume that XML fragment data, A-index and T-index entries are fully distributed and not replicated. P-index entries are partially replicated. The fully replicated RepositoryGuide has full information about what index structures are located at what sites for the individual nodes in the RepositoryGuide. It should be noted that extensive replication models and techniques such as those presented in [1] are not the primary concern in our distribution approach but rather a solid and effective data allocation and index schemes that supports the efficient processing of global and local queries.

3.4 Handling Data Modifications

We finally briefly outline the steps involved in adding new data to a distributed XML repository. These steps are independent of the site the operation is issued. First, the local RepositoryGuide is consulted to determine the fragment f that contains the root node of the inserted data, and the site s that manages XML fragment data containing that root node. Next, the add operation is sent to s. At this site the operation is executed (i.e., the XML data fragment is stored and local T- and A-index entries are generated). At the same time, respective path index information about global context of the nodes in the inserted XML data fragment is inserted locally and respective entries are propagated to sites that contain replicas of these portions of the P-Index. This is achieved by using proper protocols to ensure consistency among the replication sites [13]. If parts of the added XML data reach down into another distribution fragment f' below f in the RepositoryGuide (e.g., in the case where a vertical fragment has one or more exclusion fragments), the site for f has to propagate the add operation with content related to f' to the site that manages XML fragment data according to f'; this again can be determined from the information recorded with the individual nodes in the local RepositoryGuide. At the site that manages fragment data according to f', the same procedure as described above is applied.

4 Distributed Query Processing

Query processing in a distributed XML repository involves the processing of local and global queries and the transmission of messages and partial and final results among sites. Clearly, as the distribution of XML reposi-

tory data over the Web, as an instance of a Wide Area Network (WAN), is our primary application of interest, communication cost is the dominant factor.

In the following, we present our query processing approach in the setting of distributed XML fragment data and supporting index structures. As query model, we assume path and tree pattern queries [4] as they build the core of most XML query languages. The upper part of Fig. 4 (discussed later) gives an example of a tree pattern query. A query can be issued at any site, the query is transparently processed based on distributed data and index structures, and the query result is delivered back to the originating site. The query result consists of XML fragments related to the selection node in a query pattern, e.g., Author in Fig. 4.

In the following Section 4.1, we present the (centralized) core algorithm underlying our query processing approach. In Section 4.2, we present the main objectives of a distributed query processing approach, detail a query cost model, and discuss our heuristic-based query processing algorithm.

4.1 Core Algorithm

Queries are processed using a variant of the classical structural join (SJ) approach [4, 24, 40]. Structural joins assemble instances of a full query pattern by matching index lists of node identifiers, which represent atomic patterns. However, in our approach, identifiers in the form of μPIDs encode complete, rooted data paths. Therefore, it is not necessary to execute a join for every single edge in a tree pattern as in existing approaches. We first present our approach for the centralized case, and then discuss its extension to support distributed query processing.

Consider the sub-pattern //Document/Author/Address/Country in Fig 4. μPIDs, whose path numbers (rooted label paths) relate to instances of this pattern, directly contain information about Author ancestors of Country nodes. Therefore, there is no need to execute any joins with Address nodes to establish relationships between Country and Author nodes as in existing SJ approaches. Based on the same argumentation, it becomes obvious that joins are required *only for every branch point* and not for every edge in a query pattern.

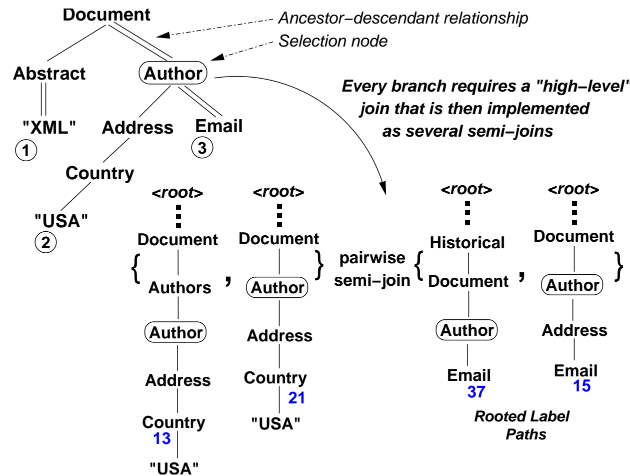

Figure 4. Tree pattern query //Document[./Abstract//"XML"]//Author[./Address/Country/"USA"][.//Email] **and realization of pairwise semi-joins at branch point** Author

For instance, to find all patterns matching the right half of the example query in Fig. 4, it is sufficient to

1. determine the path numbers matching //Document//Author/Address/Country/"USA" in a given RepositoryGuide $\mathbf{G}$, and fetch the index lists L_i for the term "USA" and these path numbers from the T-index,
2. analogously, find matches for the path pattern //Document//Author//Email in $\mathbf{G}$, and fetch respective P-index lists L_j, and
3. join path id lists L_i and L_j at the prefix position related to the Author node (the *branch point*).

The join can be considered a *semi-join* between index lists based on a common prefix of position numbers. In the following, we refer to such type of semi-join as $L_i \ltimes L_j$, where L_i and L_j stand for index lists from either P- or T-index. $L_i \ltimes L_j$ returns all those elements from L_i for which a μPID prefix matching element in L_j exists. Since μPID lists are sorted by position number, the complexity of the operator $\ltimes$ is linear in the size of argument lists.

We illustrate the algorithm for processing tree pattern queries, which is formally given in Fig. 6, by means of an example. In the tree pattern in Fig. 4, there are three leaf nodes, carrying the numbers 1, 2, and 3. Each leaf node has a related path pattern P_i starting at the root of the pattern tree. All instances of these patterns can be found in the RepositoryGuide. Assume the two instances for each of the patterns P_2 and P_3 as shown on the right side of the figure. Each instance is uniquely identified by a path number. Hence, all instances related to a query path P_i determine a set $P_i = \{p_{i1}, p_{i2}, \ldots, p_{ik_i}\}$, where the p_{ij} are path numbers, overloading the symbol P_i with this second meaning. In the example, $P_2 = \{13, 21\}$ and $P_3 = \{15, 37\}$; assume $P_1 = \{5, 8, 9\}$ (related instances are not shown in Fig. 4).

An instance of the full query pattern thus is an l-tuple of path numbers $T = (p_1, p_2, \ldots, p_l), p_i \in P_i$, where l is the number of leaf nodes in the query pattern. That is, the tuple consists of one rooted label path for each leaf-related path pattern in the query. There are $k_1 \times k_2 \times \ldots \times k_l$ potential matches for the query pattern. In the example, there are $3 \times 2 \times 2 = 12$ potential instances of the query pattern. However, notice that the paths related to path numbers 13 and 37 do not match even though the paths below Author match and both have a Document ancestor above Author. Thus, there can not be any matching $(x, 13, 37)$ tuple. Consequently, there are other combinations of path numbers that do not match the query tree pattern.

All the valid combinations of path numbers can easily be determined from the $\mathbf{G}$ *before looking at any of their instances in an index list*. Although this is an important aspect that can lead to a drastic decrease of semi-join operations, it is not supported by any of the existing query processing approaches. For the running example, a graph representing all valid triples of path numbers is shown in Fig. 5. A path from the left to the right through

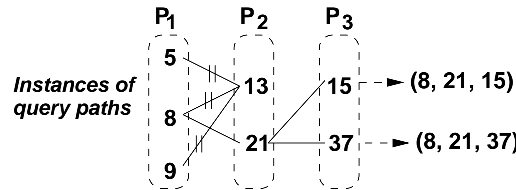

Figure 5. Valid combinations of path numbers that resemble instances of the full tree pattern query in Fig. 4

all the groups of path numbers represents a valid query pattern instance in $\mathbf{G}$. Only for these patterns potential semi-join results for path id lists exist. Every pattern instance can be processed on its own, decomposing the full query into independent sub-queries. For example, the path 8-21-15 corresponds to the semi-join expression $L_{21} \ltimes L_8 \ltimes L_{15}$, where the L_i are position numbers (μPIDs) for respective path numbers. Note that the first argument L_{21} corresponds to the position list of a selection node. In the above example, two such semi-join chains need to be evaluated in order to determine position numbers (μPIDs) of result XML fragments, which are then retrieved through accessing the A-index.

4.2 Distributed Query Processing and Cost Model

As the above example indicates, the efficient processing of several l-tuples and respective semi-join chains leads to the problem of optimizing multiple queries at the same time. This problem, which is known to be NP-hard [32], has received only limited attention in the literature (e.g., [30, 32, 33]). Solutions rely on identifying common sub-expressions (in our particular case semi-join expressions) among multiple queries and use heuristics to determine

Algorithm **TreePatternMatching**(Tree pattern Q)
Input: Tree pattern query Q with l leaf nodes and selection node s; RepositoryGuide $\mathbf{G}$
Output: List R of μPIDs that match Q in the position of s

1. Determine all leaf nodes in Q (ignoring term conditions)
2. For each of the $1 \ldots l$ leaf nodes:
   (a) find all instances of the path pattern from the leaf to the root of Q in $\mathbf{G}$
   (b) let $P_i = \{n_{i1}, n_{i2}, \ldots, n_{ik_i}\}, i = 1, \ldots, l$ be the set of path numbers related to these instances
3. For all l-tuples $T = (p_1, p_2, \ldots, p_l), p_i \in P_i$:
   (a) determine whether the rooted label paths related to path numbers $p_1, p_2, \ldots, p_l$ constitute a tree that matches Q in $\mathbf{G}$
   (b) while checking this, for every branch point node, look up the total number of bits of its position number in $\mathbf{G}$
4. For every resulting tuple $T' = (p'_1, p'_2, \ldots, p'_l)$:
   (a) fetch the list L for p'_1 from the P- or T-index
   (b) For i = 2 to l:
   (i) fetch L_i, the list related to p'_i, from P- or T-index
   (ii) if p'_i refers to selection node and p'_1 does not, swap L and L_i
   (iii) semi-join L_i and L using L as the outer list, i.e., $L := L \ltimes L_i$
   (c) append L to the result list R

Figure 6. Algorithm for tree pattern queries based on possible matches in a RepositoryGuide

efficient query evaluation plans. In our approach for determining an efficient query evaluation plan for a set of l-tuples, we obviously face the same problems. In a semi-join expression $L_{i1} \ltimes \ldots \ltimes L_{kl}$, index lists can reside at different sites. The problem thus is to find a semi-join ordering that can be processed efficiently. In the following, we will outline a cost model for processing a single l-tuple in a distributed context and then present a heuristic to obtain an efficient query evaluation plan that utilizes the *shipping of index lists*. Finally, we outline some issues regarding further optimizations of processing l-tuples as well as sets of l-tuples.

With an l-tuple $(p_1, p_2, \ldots, p_l)$, a set of possible semi-join expression of the form $L_{i1} \ltimes \ldots \ltimes L_{il}$ is associated. Without loss of generality, we assume that the list corresponding to a selection node in the query is kept fixed as first argument L_{i1}. We are interested in finding an ordering of semi-joins among $L_{i2}, \ldots, L_{il}$ that can be processed efficiently. We use the following general cost model:

$$cost_q = LC_q \text{ (local processing cost) } + CC_q \text{ (communication cost)}$$

As detailed in the previous section, local joins between index lists can be executed efficiently. Given that our focus is on WANs, we are primarily interested in minimizing the communication costs. Since we do not consider the parallel execution of semi-joins at different sites yet, the cost can be interpreted as total response time.

Because of the importance of communication cost and the strategy we use for processing queries in a distributed setting, CC_q is refined as follows:

$$CC_q = Mesg_q + Index_shipping_q + Result_q$$

$Mesg_q$ denotes the accumulated message cost for shipping processing instructions (not including XML or index data) among sites. $Index_shipping_q$ denotes the cost for shipping index entries (from the P- and T-index) among sites, and $Result_q$ denotes the cost for retrieving XML fragment data from a remote site.

A core idea of our query processing algorithm is that result XML fragment data is only retrieved during the

last step in executing a query. We first focus on a more detailed cost model for shipping index lists (used in local semi-joins). To motivate the concept of index shipping and associated cost, we use a simple scenario. Let $\#(L_i)$ denote the size of a list L_i (from the P- or T-index) in terms of transfer units. We assume uniform transfer cost of one unit among all sites. For example, given two lists L_1 (corresponding to the selection node) at site S_1 and a remote list L_2 at site S_2, the minimum cost for processing $L_1 \ltimes L_2$ at S_1, denoted $cost(L_1 \ltimes L_2)$, then is $\min\{2 * \#(L_1), \#(L_2)\}$, assuming that for every entry in L_1 there exists a matching entry in L_2, thus giving an upper bound. $2 * \#(L_1)$ corresponds to shipping L_1 from S_1 to S_2, performing the semi-join, and shipping the result back. $\#(L_2)$ corresponds to just shipping L_2 to S_1.

Now consider the more general case. Given an expression $L_{i1} \ltimes \ldots \ltimes L_{kl}$ corresponding to an l-tuple. In order to obtain an efficient evaluation plan, we use a heuristic that aims at shipping only small (intermediate) lists. For this, we first partition the set $L_{i1}, \ldots, L_{il}$ such that a partition contains all lists that reside at the same site. Recall that this information (as well as the sizes of indexes) is available in each local RepositoryGuide. We thus obtain a new set $\{\mathcal{L}_1, \ldots, \mathcal{L}_p\}$. Let $\hat{L}_i := min\{\#(L_{nm}) \mid L_{nm} \in \mathcal{L}_i\}$ denote the size of the smallest list in partition $\mathcal{L}_i$. Clearly, the local processing of a set $\mathcal{L}_i$ of lists can never result in a list larger than $\hat{L}_i$ as the semi-join is based on prefix equality of position numbers (μPIDs) in lists. Without loss of generality, assume the partition $\mathcal{L}_1$ contains the list corresponding to the selection node. A semi-join chain of the pattern $\mathcal{L}_1 \ltimes \mathcal{L}_{n_2} \ltimes \mathcal{L}_{n_3} \ltimes \ldots \ltimes \mathcal{L}_{n_p}$ is generated such that $\hat{L}_{n_2} \leq \hat{L}_{n_3} \leq \ldots \leq \hat{L}_{n_p}$. That is, partitions are sorted by increasing size of smallest list in a partition.

The evaluation of such a "meta-plan" occurs as follows. First, site S_1 determines the partitions and instructs the respective sites $S_{n_1}, \ldots, S_{n_{p-1}}$ to compute the intermediate results for local lists using the algorithm we presented in the previous section; thus the concept of parallelism is exploited to a certain extend. The list computed at S_1 is then shipped to S_{n_2}, and semi-joined with the previously computed list at S_{n_2}. This result index is then shipped to S_{n_3} , and so on, until the result index at site S_{n_p} is shipped back to site S_1 as final result node position list. S_1 uses then the information recorded in the RG to determine what site contains the A-index and related XML fragment data corresponding to the P-index entries.

The cost for this evaluation plan can easily be determined. The number of messages (and thus $Mesg_q$) to be transfered is directly related to the number of partitions computed at site S_1. Since each local RepositoryGuide has information about the sizes of remote and local indexes (i.e., P-index lists), an upper estimate can be computed for both local processing costs LC_q at partition sites and index shipping costs $Index_shipping_q$. In case statics are available about the size of XML fragment data at local and remote sites is managed in the RG as well, an upper bound for the cost factor $Result_q$ can be determined as well.

Note that for the above optimization strategy and heuristics, the assumption is that the size of the result of a semi-join chain is never greater than the size of the smallest list in the chain. More precise estimates about the selectivity of lists, based on statistics and selectivity information for XML fragment data and index structures [16, 29, 36] can be exploited to obtain better cost estimates. Also, such statistics in combination with schema information can be used to eliminate lists from a chain. For example, if it is known that for list L_1 there is always a match in list L_2 (e.g., because L_2 encodes position numbers of required siblings of nodes and position numbers in L_1), then the larger list can be dropped from the chain, unless it corresponds to a selection node.

The above query optimization strategy does not take the existence of multiple l-tuples into account. We are currently studying the problem of multi-query optimization in the context of the above distributed setting. Our initial focus is on identifying common subexpressions (e.g., partitions or portions thereof) among a set of l-tuples. For example, in Fig. 5, the list (8,21) in the two 3-tuples represent a common subexpression. The goal is to reuse intermediate query results at individual sites. Naturally, a respective realization of such a query processing strategy requires some message overhead to communicate the scheduling and temporary storage of intermediate results for individual l-tuples or portions thereof. No changes to the processing approach is necessary to retrieve the resulting XML fragment data from remote and/or local sites.

5 Experimental Evaluation

In this section, we show that

1. the P-index and even parts of the T-index are small enough to be fully replicated, thus reducing distributed query processing, in particular index shipping, to mostly local query processing,
2. small index sizes and grouping of node ids by label path provide for efficient local query processing, which causes distributed query processing to be dominated by communication costs, and
3. compared to classical structural join approaches, there are much less index list elements that need to be joined and shipped, trivially leading to greatly reduced join and communication costs.

5.1 Index Sizes

Table 1 summarizes the sizes of the P-, A-, and T-indexes for different non-distributed XML 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 source combining nine sources from the cited origins. Table 1 also lists maximum and average μPID position number lengths. In earlier, equivalent indexing approaches, index size is frequently ignored [4, 24], or reported to be several times as large as in our approach, but without providing a mapping of logical node ids to physical addresses [40].

Source	Size	P-index	A-index	T-index	Index total	Pos# $_{max}$	Pos# $_{avg}$
Reuters	1354.0	55.9 (4.1%)	70.2 (5.2%)	519.1 (38.3%)	645.2 (47.7%)	29	25.7/26.9
Big10	1243.1	23.1 (1.9%)	51.9 (4.2%)	347.1 (27.9%)	422.1 (34.0%)	41	22.5/24.4
XMark 1Gb	1118.0	24.2 (2.2%)	63.8 (5.7%)	306.4 (27.4%)	394.4 (35.3%)	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
LA Times	475.3	5.7 (1.2%)	18.2 (3.8%)	216.5 (45.6%)	240.4 (50.6%)	41	27.3/29.1
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%)	9.9 (9.0%)	25.8 (23.6%)	46.0 (42.0%)	30	21.5/22.8

Table 1. Source and index sizes in Mb (% of source), and pos# length in bits for some real-world XML sources

5.2 Query Processing

Table 2 shows three queries and their local execution times based on the algorithm described in Section 4.1. The times are averaged over one hundred executions on a 900 MHz Pentium 3 laptop with 384 Mb of RAM and a 20 Mb IDE hard drive. The table also gives the cardinalities of the input. In our query processing approach, only one join needs to be executed for Q_1 and Q_3. E.g, for Q_3, 5.7 mio. text paragraph path ids from the P-index are matched with 276,899 text path ids that contain the term "stockmarket" from the T-index. Q_2 does not even require a join.

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

Query	Time	Condition	Selection	Result
Q_1: //article[contains(./author, “Abiteboul”)]/title	22 ms	49	111,609	49
Q_2: /site/regions/asia/item[./mailbox/mail/text/keyword]	15 ms	11, 609	20,000	11,609
Q_3: /reuters/newsitem/text[contains(.,“stockmarket”)]/p	4.5 s	276,899	5,673,107	237,115

Table 2. Example queries on *DBLP* **(**Q_1**),** *XMark1Gb* **(**Q_2**), and** *Reuters* **(**Q_3**) with execution** *Time* **and number of elements in** *Selection* **and** *Condition* **paths, and** *Result*

Accessing the index lists for, e.g., query Q_2 via NFS on a remote computer leads to an average execution time of at least 500 ms even after many consecutive repetitions of the experiment. Here, the query is still executed on the laptop as described above. The remote machine contains a 550 MHz Pentium 3, 256 Mb RAM, and a faster hard drive than the main test machine, and is connected by a 10 Mbit/sec LAN.

5.3 List/Join Sizes

Compared to the classical SJs, the join costs in our approach are very low, even when lists are distributed. For instance, using classical SJ approaches, Q_3 requires a pairwise joining (and possible transmission) of 473,876 newsitem path ids with 473,876 text and 5,673,107 p path ids, and 276,899 ids of paths containing the term “stockmarket.” As another example, to access the titles of the five DBLP master theses as part of *Big10*, which contains *DBLP*, the query //dblp/masterthesis/title requires no join and at most five path ids to be sent around in our indexing and distribution scheme. In earlier approaches, over 340,000 title path ids need to be accessed. This problem is fixed in [10, 12] only through further extensive index extensions, which make these approaches even less suitable for replication and distribution, respectively.

6 Related Work

Design and query processing techniques have been studied extensively in the context of distributed relational [5, 23, 28] and object-oriented databases [7, 22]. Our approach, which has been first outlined in [8], differs from these methods because in our approach the mix of hierarchical schema structures and data components naturally complicate the distribution design. Recently, the problem of distributed and replicated (dynamic) XML documents has been studied for the first time and fundamental query processing and replication models have been proposed in the context of Web services [1]. However, they do not consider design and implementation aspects of distributed XML data, but focus on (replicated) Web services in the context of distributed query processing. Some of their ideas can be adopted to our approach, for example, to include the aspect of Web services in fragment specifications and to use peer query workload measures to dynamically replicate fragment data and index portions. Suciu [34] studied distributed query evaluation techniques for semistructured data are investigated, but distribution design and implementation aspects are not covered.

The role of supporting index structures for processing path and tree pattern queries has received great attention in the past few years, primarily in the context of structural joins [4, 10, 12, 24, 40]. Different from our approach, these works do not consider distributed query processing. Furthermore, they exclusively rely on interval node identifiers [3], do not group these identifiers by node type (common rooted label path or path identifiers), and commonly neglect storage efficiency. The latter two aspects, however, play a key role in our approach where the distribution of path and term indexes is based on complete rooted label paths and not just on node labels. Thus, entries in the path and term index are much smaller and better suited for distributed query processing. The most effective structural join algorithms so far [10, 12], rely on a more extensive index structure. Still based on binary node relationships, theses approaches exploit the selectivity of the full query pattern to reduce intermediate query results. We achieve the same advantage by directly matching full path patterns, yet at index sizes of less than half of the indexed source.

Distributed index structures have been studied in the context of distributed databases (e.g., [6, 13]) and information retrieval (e.g., [14, 21]), but respective settings for query processing are substantially different from processing path or tree patterns on distributed, fragmented XML data.

7 Conclusions and Future Work

In this paper, we have presented a complete approach for distributing an XML repository and efficiently processing local and global path and tree pattern queries. Our distribution design approach resembles basic concepts from relational databases and are extended to account for tree structured data. In particular, we have presented a fragmentation scheme for XML analogous to vertical and horizontal fragmentation for relational or object-oriented databases.

In our realization, the expense of replicating global data paths to identify fragments at local sites is offset by an efficient encoding of path information using a new path identification and indexing scheme for XML data. Our path index structure is sufficiently small to be fully replicated among local sites; thus, bringing the performance of query processing close to that of centralized systems. We also presented a core algorithm, cost model, and optimization strategies for processing queries over distributed XML fragments based on the concept of index shipping. In summary, the proposed approaches provides a comprehensive foundation for studying further data allocation models and in particular distributed query processing strategies in the context of large-scale XML repositories on the Web.

We are currently investigating index management models that account for data modifications by adjusting bit positions in indexes and related information in a fully replicated RepositoryGuide. Naturally, data modifications in the presence of replicated index structures involve transaction processing strategies (in particular locking and commit protocols) over distributed XML data and indexes.

References

[1] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *ACM SIGMOD Intl. Conference on Management of Data*, 527–538. ACM Press, 2003.

[2] V. Aguilera, S. Cluet, T. Milo, P. Veltri, D. Vodislav. Views in a large-scale XML repository. In *VLDB Journal* 11(3): 238-255 (2002)

[3] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 547–556, 2001.

[4] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *IEEE International Conference on Data Engineering (ICDE 02)*, 141–152, 2002.

[5] P. M. G. Apers. Data allocation in distributed database systems. *ACM Transactions on Database Systems (TODS)*, 13(3):263–304, 1988.

[6] J. Basu, A. M. Keller, and M. Pöss. Centralized versus distributed index schemes in OODBMS - a performance analysis. In *1st East-European Symposium on Advances in Databases and Information Systems (ADBIS'97)*, 162–169. Nevsky Dialect, 1997.

[7] J. Biskup and T. Polle. Decomposition of object-oriented database schemas. *Annals of Mathematics and Artificial Intelligence*, 33:119–155, 2001.

[8] J.-M. Bremer, M. Gertz. On Distributing XML Repositories. In 6th International Workshop on the Web and Databases (WebDB), 73–78, 2003.

[9] J.-M. Bremer, M. Gertz. An efficient XML node identification and indexing scheme. Techn. Report CSE-2003-04, Dep. of Computer Science, Univ. of California, Davis, 2003.

[10] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 310–311, 2002.

[11] D. Chamberlin, P. Frankhauser, D. Florescu, M. Marchiori, and J. Robie. XML query use cases. W3C working draft, W3C, Nov. 2002. www.w3.org/TR/2002/WD-XMLquery-use-cases-20021115/.

[12] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *28th International Conference on Very Large Data Bases (VLDB)*, 263–274, 2002.

[13] D. M. Choy and C. Mohan. Locking protocols for two-tier indexing of partitioned data. In *Int. Workshop on Advanced Transaction Models and Architectures*, 198–215, ACM Press, 1996.

[14] P. B. Danzig, J. Ahn, J. Noll, and K. Obraczka. Distributed indexing: A scalable mechanism for distributed information retrieval. In *14th ACM SIGIR Conf. on Research and Development in Information Retrieval*, 220–229. ACM, 1991.

[15] T. Fiebig, S. Helmer, K.-C. Kanne, G. Moerkotte, J. Neumann, and R. Schiele. Anatomy of a native XML base management system. *The VLDB Journal*, 11(4):292–314, 2002.

[16] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Simeon. StatiX: Making XML count. In *ACM SIGMOD International Conference on Management of Data*, 181–191, 2002.

[17] P. Grosso and D. Veillard. XML fragment interchange. W3C candidate recommendation, W3C, Feb. 2001. www.w3.org/TR/XML-fragment.

[18] IXRetail standard XML schemas to interface applications within the retail enterprise. www.nrf-arts.org.

[19] E. Jeong, C. Hsu. Semistructured Data: Induction of integrated view for XML data with heterogeneous DTDs In *10th Intl. Conference on Information and Knowledge Management (CIKM)*, ACM Press, 151–158, 2001.

[20] H. Jagadish, L. V. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 133–144, 1999.

[21] B.-S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[22] K. Karlapalem and Q. Li. A framework for class partitioning in object-oriented databases. *Distributed and Parallel Databases*, 8:333–366, 2000.

[23] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[24] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *27th International Conference on Very Large Data Bases (VLDB)*, 361–370, 2001.

[25] G. Moerkotte. Incorporating XSL Processing into Database Engines. In *28th International Conference on Very Large Data Bases (VLDB)*, 2002.

[26] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *21st ACM Symposium on Principles of Database Systems (PODS)*, 65–76, 2002.

[27] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *9th International Conference on Database Theory (ICDT)*, LNCS 2572, 315–329. Springer, 2003.

[28] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems (2nd edition)*. Prentice Hall, 1999.

[29] N. Polyzotis and M. N. Garofalakis. Structure and value synopses for XML data graphs. In *28th International Conference on Very Large Data Bases (VLDB)*, 466–477, 2002.

[30] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi-query optimization. In *ACM SIGMOD International Conference on Management of Data*, 249–260. ACM, 2000.

[31] Y. Papakonstantinou, V. Vassalos. Architecture and Implementation of an XQuery-based Information Integration Platform. In *IEEE Data Engineering Bulletin* 25(1): 18-26 (2002)

[32] T. K. Sellis. Multiple-query optimization. *Transactions on Database Systems*, 13(1):23–52, 1988.

[33] T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, 1990.

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

[35] Tamino XML server. Software AG. www.softwareag.com/tamino.

[36] Y. Wu, J.M. Patel, H. V. Jagadish. Using histograms to estimate answer sizes for XML queries. Information Systems 28(1-2): 33-59 (2003).

[37] E. J. J. Whitehead and M. Wiggins. WebDAV: IETF standard for collaborative authoring on the Web. *IEEE Internet Computing*, 34–40, Sept. 1998.

[38] XML common business library (xCBL), version 4.0. www.xcbl.org.

[39] XMark – An XML Benchmark Project monetdb.cwi.nl/xml.

[40] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *ACM SIGMOD Intl. Conference on Management of Data*, 425–436, 2001.

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