Technical Report CS-2010-02

Distributed XML Query Processing: Fragmentation, Localization and Pruning

Patrick Kling · M. Tamer Özsu · Khuzaima Daudjee

September 2010

Abstract Distributing data collections by fragmenting them is an effective way of improving the scalability of a database system. While the distribution of relational data is well understood, the unique characteristics of XML data and its query model present challenges that require different distribution techniques. In this paper, we show how XML data can be fragmented horizontally and vertically. Based on this, we propose solutions to two of the problems encountered in distributed query processing and optimization on XML data, namely localization and pruning. Localization takes a fragmentation-unaware query plan and converts it to a distributed query plan that can be executed at the sites that hold XML data fragments in a distributed system. We then show how the resulting distributed query plan can be pruned so that only those sites are accessed that can contribute to the query result. We demonstrate that our techniques can be integrated into a real-life XML database system and that they significantly improve the performance of distributed query execution.

Keywords Distributed · XML · Localization · Pruning

1 Introduction

Over the past decade, XML has become a commonly used format for storing and exchanging data in a wide variety of systems. Due to this widespread use, the problem of effectively and efficiently managing XML collections has attracted significant attention in both the research community and in commercial products. One can claim that the centralized management and querying of XML data (i.e., data residing on a single system) is now a well understood problem. Unfortunately, centralized techniques are limited in their scalability when presented with large collections and heavy query workloads.

In relational database systems, scalability challenges have been successfully addressed by partitioning data collections and processing queries in parallel in a distributed system [1]. Our work is focused on similarly exploiting distribution in the context of XML. While there are some similarities between the way relational database systems can be distributed and the opportunities for distributing XML database systems, the significant differences in both data and query models make it impossible to directly apply relational techniques to XML. Therefore, new solutions need to be developed to distribute XML database systems.

In this paper, we focus on the following three aspects of distributing an XML database system:

– First, we present a *distribution model* for XML that supports horizontal fragmentation (based on selection operators and predicates) and vertical fragmentation (based on a partitioning of the set of element types in a schema). Our definitions of horizontal and vertical fragmentation are semantically analogous to those for relational data [1]. However, the characteristics of XML, such as its nested data model and structure-based queries lead to a set of challenges and optimization opportunities that differ significantly from what is encountered in the relational context. As in the relational case, both types of fragmentation are designed to be orthogonal. This allows us to use them together to achieve hybrid fragmentation.
– Second, we focus on the problem of *localization and pruning* in distributed XML database systems. We propose a localization technique that transforms a fragmentation-unaware query into sub-queries that can be evaluated in parallel at the individual sites in the system. We

University of Waterloo
Cheriton School of Computer Science
200 University Ave W
Waterloo, ON N2L 3G1
Canada
Tel.: +1-519-888-4567
Fax: +1-519-885-1208
E-mail: {pkling, tozsu, kdaudjee}@cs.uwaterloo.ca

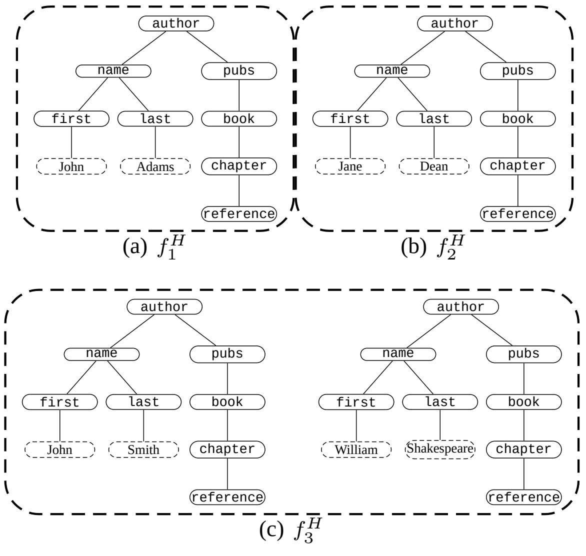


Fig. 1 A horizontally fragmented collection

then present a novel technique that allows us to identify fragments that are irrelevant for answering a given query and prune them from the query plan.

While localization and pruning represent only the first step of distributed query evaluation, we show that even with these techniques alone we can achieve significant improvements in performance. Further optimizations that can be performed after localization and pruning have been published separately [2] and will be the subject of future work.

– Based on our localization techniques, we then propose a set of *workload-aware fragmentation algorithms*. These algorithms are designed to determine a fragmentation layout that will optimize performance for a given set of queries.

To motivate our work, consider the following example. Figure 1 shows a horizontally fragmented data collection consisting of four documents representing information about authors and their publications. The horizontal fragmentation is defined based on the first letter of the authors' last names, placing "John Adams" in fragment f_1^H, "Jane Dean" in fragment f_2^H and "John Smith" as well as "William Shakespeare" in fragment f_3^H.

Figure 2 shows a similar collection that has been fragmented vertically. Ignoring the nodes labeled as $P_k^{i \to j}$ and $RP_k^{i \to j}$ for now, we can see that `author` and `agent` nodes are stored in fragment f_1^V, the nodes related to the author's name are stored in fragment f_2^V, `pubs` and `book` nodes are stored in fragment f_3^V and `chapter` and `reference` nodes are stored in fragment f_4^V.

Consider evaluating the following XPath query (q):

```
/author[name[first = 'William' and
 last = 'Shakespeare']]//book//reference
```

In the horizontal case, it is easy to see that the fragments f_1^H and f_2^H cannot possibly contribute to the result of this

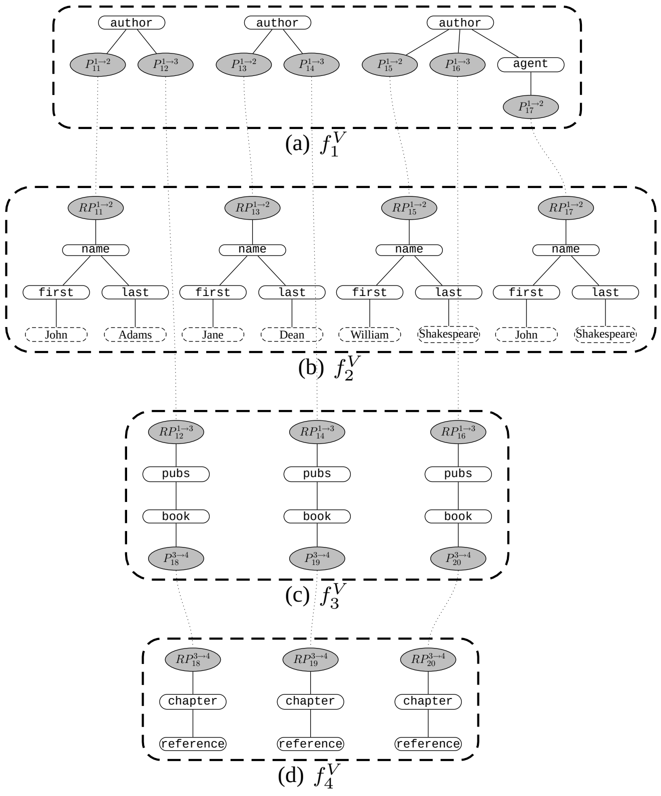


Fig. 2 A vertically fragmented collection

query since they correspond to authors whose last names start with the letters "A" and "D", respectively. Pruning these fragments allows us to answer the query without contacting the sites at which they are stored.

If we evaluate q on the vertically fragmented collection, in the general case, we have to access all four fragments. Fragment f_2^V is needed to evaluate the value constraint predicates, fragment f_4^V is needed to obtain result nodes and fragments f_1^V and f_3^V are needed to evaluate structural constraints. We will later present a technique that allows us to avoid accessing some of the fragments only needed for structural constraints.

We propose a general technique that detects situations in which fragments are not needed to answer a query and then prunes these irrelevant fragments from a distributed query plan. This greatly improves the performance of distributed query evaluation and allows us to fully benefit from distribution as a means to overcome the scalability challenges faced by large XML collections. We achieve this goal without relying on a globally replicated index structure, because using such a structure could limit the scalability of a distributed system and negatively affect the performance of updates. The specific contributions of the work presented here are the following:

1. We formally define fragmentation in the context of XML databases and propose a succinct method for specifying the horizontal or vertical fragmentation of a collection of XML documents.

2. We develop a mechanism that transforms a fragmentation-unaware query plan into an equivalent distributed plan.
3. We propose the first known technique that can identify and prune horizontal fragments that are irrelevant for answering a given query.
4. We present a novel technique that, without relying on a fully replicated index, allows us to skip vertical fragments that are not needed to evaluate value constraints.
5. We propose algorithms for fragmenting a collection of XML documents in order to improve the performance of a given workload (when evaluated using our pruning techniques).
6. We have implemented these techniques within a real-life distributed XML database system, which has allowed us to obtain realistic experimental results.

The remainder of this paper is structured as follows: Section 2 describes the technical background of our work. Section 3 introduces our model of horizontal and vertical fragmentation. In Section 4, we propose techniques for evaluating queries over distributed collections and describe how distributed query evaluation can be optimized through localization and pruning. Based on these query evaluation techniques, Section 5 describes our algorithms for fragmenting an XML collection such that performance for a given workload is optimized. In Section 6, we present a thorough evaluation of the performance impact of the techniques presented in this paper. Section 7 discusses related work. In Section 8, we summarize our work and present our conclusions.

2 Background

2.1 Data model

An XML collection can be described as a set of labeled, ordered trees. While XML is a self-describing format that can be used without a schema, in practice, the structure of document trees is usually constrained by a schema that specifies how elements may be nested and what the domain of their textual content is. A schema is usually defined in a language such as DTD or XML Schema. In this paper, we use a simple directed graph representation that covers only the aspects of the schema that are important for our purposes. For example, our representation ignores the distinction between XML elements and attributes by treating both of them uniformly as *nodes*. Similarly, we refer to element types and attribute names as *node types*. Assuming that the original schema definition does not contain unspecified portions (such as those defined using the DTD keyword ANY), it is straightforward to extract the information captured by our graph representation from a DTD[1] or an XML Schema. Extracting schema information yields a schema graph that may be less restrictive than the original schema, but since the schema graph is never used for the validation of documents this does not pose a problem [3].

```
author(name, pubs, agent?)
pubs(book*)
book(chapter*)
chapter(reference?)
reference(chapter)
agent(name)
name(first, last)
first(#text)
last(#text)
```

Fig. 3 A schema

Definition 1 An XML *schema graph* is defined as a 5-tuple $\langle \Sigma, \Psi, s, m, \rho \rangle$ where Σ is an alphabet of node types, ρ is the root node type, $\Psi \subseteq \Sigma \times \Sigma$ is a set of directed edges between node types, $s : \Psi \rightarrow \{\text{ONCE}, \text{OPT}, \text{MULT}\}$ and $m : \Sigma \rightarrow \{\texttt{string}\}$.

The semantics of this definition are as follows: An edge $\psi = (\sigma_1, \sigma_2) \in \Psi$ denotes that a node of type σ_1 may contain a node of type σ_2. $s(\psi)$ denotes the cardinality of the containment represented by this edge: If $s(\psi) = \text{ONCE}$, then a node of type σ_1 must contain exactly one node of type σ_2. If $s(\psi) = \text{OPT}$, then a node of type σ_1 may or may not contain a node of type σ_2. If $s(\psi) = \text{MULT}$, then a node of type σ_1 may contain multiple nodes of type σ_2. $m(\sigma)$ denotes the domain of the text content of a node of type σ, represented as the set of all strings that may occur inside such a node. Note that the definition of $m(\sigma)$ may include both the direct content of a node of type σ as well as the content of node types nested in σ. Figure 4 illustrates how the simplified DTD shown in Figure 3 can be represented as a graph.

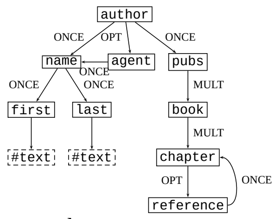


Fig. 4 An XML schema graph

When translating a DTD or an XML Schema into the graph representation, attributes are always assigned a cardinality of either ONCE or OPT, corresponding to mandatory and optional attributes, respectively. Elements, on the other hand, may occur with any of the three cardinalities, since both DTD and XML Schema allows for the specification of elements with exactly one, zero or one, or multiple occurrences. In addition to these three cases, XML Schema allows a more fine-grained specification of the number of occurrences of an element. We handle this by assigning a cardinality of MULT whenever the XML Schema definition allows for an element to occur more than once.

2.2 Query model and tree patterns

The query model used in this paper is a subset of XPath, which we call XQ. XQ consists of absolute location paths consisting of node tests with and without wildcards, child

[1] Note that a DTD does not explicitly specify the root element type of a document. However, the root element type can be inferred from the DOCTYPE declarations of documents conforming to a DTD.

(`/`) and descendant (`//`) axes and predicates. Predicates may consist of (i) a relative location path with the same restrictions (with XPath's existential semantics); (ii) a textual constraint of the form "$.\, \theta_s\, s$", where s is a string constant and θ_s is either `=` or `!=`; or (iii) a numeric constraint of the form "$.\, \theta_n\, n$", where n is a numeric constant and θ_n is one of `<`, `<=`, `=`, `>`, `>=`, or `!=`. As in XPath, XQ steps return nodes in document order (since both axes we support are forward axes).

XQ queries are not only commonly used on their own, but they also represent an important building block of more complex XQuery queries [4, 5]. Therefore, solving the problem of evaluating XQ queries in a distributed fashion is an important contribution to distributed XQuery evaluation.

It is convenient to represent XQ queries as tree patterns [6, 7], which we formalize as follows:

Definition 2 Let $\langle \Sigma, \Psi, s, m, \rho\rangle$ be a schema. A *tree pattern* is a 7-tuple $\langle N, E, r, \nu, \epsilon, T, c\rangle$ where N is a set of pattern nodes, $E \subseteq N \times N$ is a set of pattern edges and $\langle N, E, r\rangle$ is a tree rooted at $r \in N$. For each $n \in N$, $\nu(n) \in \Sigma \cup \{*\}$ denotes a node test. For each $e \in E$, $\epsilon(e) \in \{$`child`, `descendant`$\}$ denotes the axis type. $T \subseteq N$ denotes the set of extraction points. For each $n \in N$, $c(n) \subseteq m(\nu(n))$ denotes a value constraint on the text content of nodes of type $\nu(n)$.

In the following, we will refer to the tree pattern representation of a query as a *query tree pattern* (QTP). It is interesting to note that, in addition to XQ queries, QTPs can be used to express queries with multiple extraction points. While this may be useful for supporting a larger class of queries, in this paper, our focus is on queries with a single extraction point.

The QTP depicted in Figure 5 is equivalent to query q from Section 1. The double-outlined node labeled with `reference` is an extraction point and the edge labels "/" and "//" denote child and descendant steps, respectively.

A match for a QTP assigns a node from the document to each pattern node such that all node tests, value constraints, and structural constraints (expressed as axis relationships) are satisfied. While all pattern nodes in the QTP have to be matched to nodes in a document, only the nodes associated with pattern nodes that are designated as extraction points are returned as part of the result.

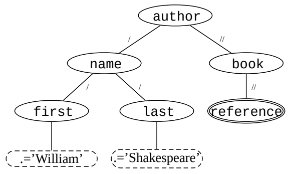


Fig. 5 Query tree pattern (QTP) representation of query q

3 Fragmentation

The work presented in this paper is based on two techniques for fragmenting XML collections. Horizontal fragmentation is based on predicates and results in a collection that is partitioned into fragments that all follow the same schema. Vertical fragmentation, on the other hand, is based on partitioning the schema.

3.1 Horizontal fragmentation

Our model of horizontal fragmentation assumes a collection that consists of multiple document trees. These document trees can either be entire XML documents or they can be the result of a previous fragmentation step. In either case, we require that all document trees correspond to the same schema. Multiple-document collections where all documents follow the same schema are a common use case for XML. Popular examples include MathML [8] and CML [9].

A horizontal fragmentation is defined by a set of fragmentation predicates. Each fragment consists of the document trees that match the corresponding predicate. To ensure that the fragmentation is lossless and that the fragments are disjoint, we require that whenever a document tree conforms to the schema of the collection, it matches exactly one of the predicates.

Definition 3 Let $D = \{d_1, d_2, \ldots, d_n\}$ be a collection of document trees such that each $d_i \in D$ corresponds to the same schema $\langle \Sigma, \Psi, s, m, \rho\rangle$. Then we can define a set of *horizontal fragmentation predicates* $P = \{p_0, p_1, \ldots, p_{l-1}\}$ such that $\forall d \in D : \exists$ unique $p_i \in P$ where $p_i(d)$. If this holds, then $F = \{\{d \in D \mid p_i(d)\} \mid p_i \in P\}$ is a set of horizontal fragments corresponding to collection D and predicates P.

We represent the fragmentation predicates as Boolean tree patterns, i.e., tree patterns with no extraction points. In the following, we will refer to them as *fragmentation tree patterns* (FTPs). Based on this representation, the losslessness of a fragmentation can be enforced by carefully crafting value constraints so that they cover the entire domain of the values to which they refer.

If we assume that the document trees in the fragmented collection shown in Figure 1 conform to the schema in Figure 4 and that $m($`last`$)$ is the set of strings that start with upper-case letters of the English alphabet, then the fragmentation of this collection can be described by the set of FTPs shown in Figure 6.

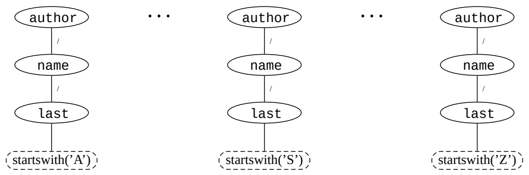


Fig. 6 Set of fragmentation tree patterns (FTPs)

3.2 Vertical fragmentation

Our model of vertical fragmentation can handle collections that consist of a single or multiple document trees. Again, it is possible that these trees are the result of a previous fragmentation step, which allows us to combine horizontal and vertical fragmentation.

A *vertical fragmentation schema* is defined by fragmenting the schema graph of the collection into connected subgraphs:

Definition 4 Let $\langle \Sigma, \Psi, s, m, \rho \rangle$ be a schema graph. A *vertical fragmentation schema* is defined by a partitioning F_Σ of the set of node types Σ.

The dashed outlines in Figure 7 show how the node types in this schema have been fragmented into four disjoint subgraphs. Fragment f_1^V consists of the node types `author` and `agent`; fragment f_2^V consists of the node types `name`, `first` and `last` along with their text content; fragment f_3^V consists of `pubs` and `book`; fragment f_4^V includes the node types `chapter` and `reference`.

Since we require the schema graph to be connected, after fragmentation, there will be graph edges that cross fragment boundaries. Whenever the schema contains an edge from a fragment f_i^V to another fragment f_j^V, we refer to f_j^V as a *child fragment* of f_i^V and to f_i^V as a *parent fragment* of f_j^V. There is exactly one fragment $f_\rho^V \in F_\Sigma$ that contains the root node type ρ. We refer to f_ρ^V as the *root fragment*. While the schema graph may contain cycles, for performance reasons, we require that the fragmentation schema be a DAG (i.e., each cycle has to be contained within a single fragment).

When a collection is partitioned according to a vertical fragmentation schema, there will be document edges that cross fragment boundaries. We represent a document edge from fragment f_i^V to fragment f_j^V by inserting a pair of artificial nodes $P_k^{i \to j}$ and $RP_k^{i \to j}$ into fragments f_i^V and f_j^V, respectively. $P_k^{i \to j}$ denotes a *proxy node* in fragment f_i^V (the originating fragment) with ID k, whereas $RP_k^{i \to j}$ denotes a *root proxy node* in fragment f_j^V (the target fragment) with ID k. Since $P_k^{i \to j}$ and $RP_k^{i \to j}$ share the same ID (k) and reference the same fragments ($i \to j$), they correspond to each other and together represent a single cross-fragment edge in the collection.

The collection shown in Figure 2 has been fragmented according to the vertical fragmentation schema shown in Figure 7. The proxy pair consisting of $P_{11}^{1 \to 2}$ in fragment f_1^V and $RP_{11}^{1 \to 2}$ in fragment f_2^V, for example, represents an edge from an `author` node in f_1^V to a `name` node in f_2^V.

Vertical fragments generally consist of multiple unconnected pieces of XML data, which we refer to as *document subtrees*. In Figure 2, for example, fragment f_1^V contains three subtrees, each of which consists of the `author` and `agent` nodes of one of the documents in the collection.

4 Querying distributed collections

In this section, we propose techniques for evaluating queries over horizontally and vertically distributed collections. For each type of fragmentation, we start with a naïve query evaluation strategy and then present optimizations, with special focus on the problem of pruning the set of fragments that need to be accessed to answer a given query.

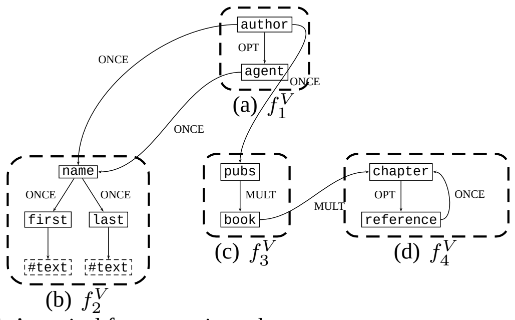


Fig. 7 A vertical fragmentation schema

In relational systems, distributed query optimization is usually done based on an algebraic representation of a distributed query [1]. For many of the optimizations presented here, however, the QTP represents a simpler abstraction that contains all the information necessary to make pruning decisions. We therefore describe many of our techniques in terms of QTP manipulations.

4.1 Horizontal fragmentation

Based on the definition of horizontal fragmentation, we can define a naïve strategy for evaluating QTPs on a horizontally fragmented collection of data. In an approach that resembles horizontal localization in the relational context [1], we can evaluate a query by computing the union of all fragments and then executing a fragmentation-unaware plan over the result. Since the definition of horizontal fragmentation (Def. 3) requires that the set of document trees D is the union of all fragments $f \in F$ and because our query model does not allow for structural constraints involving nodes in different documents, this leads to the correct result:

$$q(D) = q(\bigcup_{f \in F} f)$$

Our query model implies that each result is derived from exactly one document tree in the collection. This allows us to push the (unchanged) fragmentation-unaware query plans down to the individual fragments:

Definition 5 If q is a plan that evaluates the query on an un-fragmented collection of document trees D and F is a horizontal fragmentation of D, then

$$q_f(F) := \text{sort}(\bigodot_{f \in F} q(f))$$

is a *naïve horizontal query plan* that evaluates the same query on F, where $\odot$ denotes concatenation of results, and $q_f(F) = q(D)$.

As shown in the definition, it may be necessary to sort the results received from the individual fragments in order to return them in a stable global order as required by the XQuery data model [10]. For unordered queries, or if we are willing to relax the ordering constraint, we can reduce the amount of sorting-induced buffering by only maintaining a stable order between nodes in the same document. This may be a reasonable trade-off in many use cases.

4.1.1 Pruning fragments

As discussed before, to answer the query shown in Figure 5 on the fragmented collection from Figure 1, only the documents contained in the fragment f_3^H need to be accessed. The naïve plan, in contrast, accesses every fragment in the collection, which can significantly reduce query throughput.

In this section, we propose a procedure that detects irrelevant fragments and prunes them from a distributed query plan. This procedure relies on the schema of the collection and the FTPs that define the fragmentation. Both of these are static over time, do not depend on the size of the collection and can be encoded in a compact manner. This makes it feasible to replicate them at all sites as metadata.

Our pruning algorithm works based on the QTP representation of the query before converting the result to an algebraic plan. This allows us to reduce the problem of pruning horizontal fragments to that of determining the subset of FTPs that can be shown to be unsatisfiable at the same time as the QTP.

To solve this problem, we transform QTP and FTPs into a simplified form. We then traverse both simplified patterns simultaneously and check for contradictory constraints. If we find such a contradiction, there cannot be any results for the query in the fragment corresponding to the FTP and the fragment can thus be eliminated from the distributed plan.

4.1.2 Transformation to simplified form

The goal of transforming tree patterns into a simplified form is to make sure that each pattern node refers to a unique node within the context of a single document tree. In general, pattern nodes may match more than one node in a given document tree. A constraint associated with such a pattern node is satisfied if one of the matching nodes conforms to the constraint. This makes it impossible to exploit contradictory constraints associated with such pattern nodes. Even if the constraints themselves are contradictory, they may be satisfied by different nodes in the same document.

With QTPs, there are three sources of pattern nodes that may match multiple nodes in the same document tree:

Node types reached via MULT edges. Node types that are reached via an edge in the schema that has a cardinality of MULT may occur multiple times in the same context. Based on the schema in Figure 4, for example, the step `pubs/book` may yield multiple `book` nodes corresponding to a single `pubs` node.

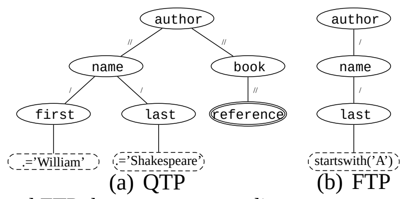


Fig. 8 QTP and FTP that are not contradictory

Descendant steps can also yield multiple results in the same context. In the QTP shown in Figure 8(a), for example, the descendant edge between `author` and `name` can be satisfied either by a `name` node that is the direct child of a given `author` node or by a `name` node that is reachable through an intermediate `agent` node. Because of this, even though the constraints on the author's last name imposed by the FTP shown in Figure 8(b) and the QTP shown in Figure 8(a) seem to cause these two patterns to be contradictory, they actually are not. Document trees in the fragment corresponding to the FTP predicate will only contain information about authors whose last names start with the letter "A". The QTP, on the other hand, matches books that are either authored by "William Shakespeare" or by someone whose agent is "William Shakespeare" and whose last name might well start with the letter "A".

Wildcards are another source of multiple matches in the same context whenever the schema specifies that a node type may contain multiple other node types.

We define simplified tree patterns as tree patterns that do not contain any of these primitives:

Definition 6 A tree pattern $\langle N, E, r, \nu, \epsilon, T, c \rangle$ is a simplified tree pattern iff $\forall n \in N$, $\nu(n) \in \Sigma$ and $\forall (x, y) \in E, \epsilon((x, y)) = \texttt{child} \wedge (\nu(x), \nu(y)) \in \Psi \wedge s((\nu(x), \nu(y)))$ $\neq$ MULT.

In order to convert a tree pattern into a simplified tree pattern, all disallowed primitives have to either be removed or converted into an equivalent simplified form. It is important to note that simplified tree patterns are strictly less expressive than arbitrary tree patterns. Therefore, when a tree pattern is transformed to a simplified tree pattern, the result is not generally equivalent to the original tree pattern. Instead, the simplified tree pattern matches a superset of the document trees that match the original tree pattern. Since simplified tree patterns are only used to identify fragments that can be pruned, but not for the subsequent query evaluation on those fragments, this loss of expressiveness does not pose a problem. Nevertheless, it is important that the transformation retains as much of the information present in the original pattern as possible so that this information can be exploited for pruning.

Algorithm 1 performs the transformation of a tree pattern into a canonical tree pattern based on the following principles:

- Using schema information, descendant steps are unrolled into equivalent paths comprised entirely of child steps (procedure shown as Algorithm 2). If there is more than one path, artificial nodes representing a choice (denoted as $\oplus$) are inserted and the branch below the descendant step becomes reachable via more than one path, thus turning the tree pattern into a directed, acyclic graph (DAG).

– Wildcard node tests are converted to non-wildcard node tests wherever this is unambiguously possible. Otherwise, the corresponding pattern nodes are removed along with their descendants.
– Pattern nodes that match nodes from the collection which, according to the schema, can occur multiple times in the same position are removed along with the branches below them.

4.1.3 Unrolling descendant steps

The unrolling of descendant steps can be succinctly implemented as a manipulation of the directed graph representation of the schema (Algorithm 1, lines 31-33). In order to unroll a descendant step from a pattern node labeled `a` to a pattern node labeled `b`, we consider the subgraph of the schema graph that consists of all nodes that are reachable from `a` and from which `b` is reachable. This yields a graph that contains all the intermediate node types that may occur on a downward path from `a` to `b`. In the example shown in Figure 9, the nodes that are used to unroll the step `author//name` are highlighted.

If there exists a cycle in this schema subgraph, we discard the descendant step and all the pattern nodes that occur below it (Algorithm 1, line 34). This is necessary because the presence of a cycle implies that a matching node may occur at different levels in the document tree. This creates ambiguity, making it impossible to take advantage of the value constraints associated with such a node. Assume, for example, that we want to unroll the step `book//reference`. We can observe that there is a cycle involving the node types `chapter` and `reference`. This corresponds to the fact that the path can be satisfied either by a reference in a chapter of the book where we start out, or by a reference in a chapter referenced by this chapter, and so on.

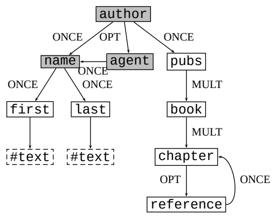


Fig. 9 Schema restricted to nodes reachable from `author` and from which `name` is reachable

If the subgraph is acyclic (as in the example shown in Figure 9), we introduce a new pattern node for each of the intermediate schema nodes such that the node test of the pattern node matches the name of the corresponding schema node (Algorithm 2, lines 19-22). In cases where a schema node has more than one child, an intermediate choice node is inserted (lines 8-11, denoted by $\oplus$), which signifies that the subsequent branch of the pattern can be satisfied by a match for any of the child nodes.

After these intermediate nodes have been inserted, the pattern has been transformed from a tree into a DAG. We can reconstruct a tree representation by duplicating nodes that are reachable through more than one path. In general, however, this is not necessary since we can directly traverse the more compact DAG, which yields the same result as traversing the equivalent tree.

Figure 10 shows the tree representation of the unrolled version of the QTP given in Figure 8(a). Note that while

Algorithm 1: pattern transformation algorithm

```
    input    : pattern tree $(N, E, r, \nu, \epsilon, T, c)$, schema
               $(\Sigma, \Psi, s, m, \rho)$
    output   : pattern graph $(N', E', r', \nu', \epsilon', T', c')$
    variable : $Q$ // represents pattern nodes whose children have
               yet to be checked
    variable : $N''$ // set of pattern nodes to be inserted
    variable : $E''$ // set of pattern edges to be inserted
 1  $r' \leftarrow$ new node
 2  $\nu'(r') \leftarrow \nu(r)$
 3  $c'(r') \leftarrow c(r)$
 4  $N' \leftarrow \{r'\}$
 5  $E' \leftarrow \emptyset$
 6  $T' \leftarrow \emptyset$
 7  $Q \leftarrow \{(r, r')\}$
 8  while $Q \neq \emptyset$ do
 9      // while there are pattern nodes to be processed, pick one
10      $(q, q') \leftarrow$ some $(q, q') \in Q$
11      $Q \leftarrow Q \setminus \{(q, q')\}$
12      // for all outgoing edges of $q$
13      for $e = (x, y) \in E$, with $x = q$ do
14          $y' \leftarrow$ new node
15          $c'(y') \leftarrow c(y)$
16          if $\epsilon(e) =$ child then
17              // case 1: child axis
18              if $\nu(y) \neq *$ then
19                  $\nu'(y') = \nu(y)$
20              else if $\exists(\sigma_1, \sigma_2) \in \Psi$ unique with $\nu(x) = \sigma_1$
                then
21                  $\nu'(y') \leftarrow \sigma_2$
22              else
23                  continue
24              if $\psi = (\nu(x), \nu(y)) \in \Psi, s(\psi) \neq MULT$ then
25                  // add this pattern node to the simplified tree
26                  $N' \leftarrow N' \cup \{y'\}$
27                  $E' \leftarrow E' \cup \{(q', y')\}$
28                  $Q \leftarrow Q \cup \{(y, y')\}$
29          else if $\nu(y) \neq *$ then
30              // case 2: descendant axis
31              $\Sigma' \leftarrow \{\sigma \in \Sigma \mid \sigma$ reachable from $\nu(x)$,
32              $\nu(y)$ reachable from $\sigma$ in $(\Sigma, \Psi)\}$
33              $\Psi' \leftarrow \{(\sigma_1, \sigma_2) \in \Psi \mid \sigma_1, \sigma_2 \in \Sigma'\}$
34              if $(\Sigma', \Psi')$ is acyclic and $\nexists \psi \in \Psi'$ with
                $s(\psi) = MULT$ then
35                  $\nu'(y) \leftarrow \nu(y)$
36                  $(N'', E'') \leftarrow$
                    unrolldesc$(q', y', \Sigma', \Psi', \nu(x))$
37                  $N' \leftarrow N' \cup N'' \cup \{y'\}$
38                  $E' \leftarrow E' \cup E''$
39                  $Q \leftarrow Q \cup \{(y, y')\}$
40  $\forall e' \in E', \epsilon'(e') \leftarrow$ child
41  return$(N', E', r', \nu', \epsilon', T', c')$
```

Algorithm 2: unrolldesc($x, y, \Sigma', \Psi', \rho'$) *unrolls descendant step*

```
input   : origin node $x$, target node $y$, transformed schema
          $(\Sigma', \Psi')$
output  : pattern nodes $N''$, pattern edges $E''$
variable: $S$ // pattern nodes yet to be processed
1  $N'' \leftarrow \emptyset$
2  $E'' \leftarrow \emptyset$
3  $S \leftarrow \{x\}$
4  for $s \in S$ do
5      if $\exists(\sigma_1, \sigma_2), (\sigma_3, \sigma_4) \in \Psi', \sigma_2 \neq \sigma_4, \nu(s) = \sigma_1 = \sigma_3$
       then
6          // more than one outgoing edge from $s$
7          // insert $\oplus$ node
8          $n_\oplus \leftarrow$ new node
9          $\nu'(n_\oplus) \leftarrow \oplus$
10         $c'(n_\oplus) \leftarrow \bot$
11         $N'' \leftarrow N'' \cup \{n_\oplus\}$
12         $E'' \leftarrow E'' \cup \{(s, n_\oplus)\}$
13         $s \leftarrow n_\oplus$
14     // insert edges
15     for $(\sigma_1, \sigma_2) \in \Psi', \nu(s) = \sigma_1$ do
16         if $\sigma_2 = \nu(y)$ then
17             $n_{\sigma_2} \leftarrow y$
18         else
19             $n_{\sigma_2} \leftarrow$ new node
20             $\nu'(n_{\sigma_2}) \leftarrow \sigma_2$
21             $c'(n_{\sigma_2}) \leftarrow \bot$
22             $N'' \leftarrow N'' \cup \{n_{\sigma_2}\}$
23             $S \leftarrow S \cup \{n_{\sigma_2}\}$
24         $E'' \leftarrow E'' \cup \{(n_\sigma, n_{\sigma_2})\}$
25 return$(N'', E'')$
```

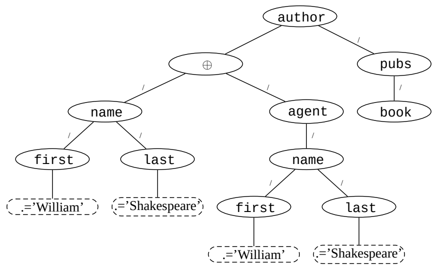


Fig. 10 Pattern after unrolling descendant steps

the step `author//book` can simply be unrolled into a sequence of child steps, unrolling `author//name` requires the insertion of a choice node and the duplication of the branch below it. This is because there are two paths from `author` to `name`, as is shown in Figure 9.

4.1.4 Removing wildcard nodes

We convert wildcard nodes whenever they unambiguously refer to a specific node type (Algorithm 1, lines 20 and 21). For example, by relying on the schema shown in Figure 4, we can determine that the step `agent/*` can be translated to the step `agent/name`. It is also possible to convert wildcard nodes that can refer to more than one node type by introducing choice nodes into the pattern in a procedure that is largely analogous to the way descendant steps are unrolled.

Algorithm 3: traverse($(N, E, r, \nu, \epsilon, T, c)$, $(N', E', r', \nu', \epsilon', T', c')$) *finds contradictions*

```
input   : predicate pattern $(N, E, r, \nu, \epsilon, T, c)$ , query pattern
          $(N', E', r', \nu', \epsilon', T', c')$
output  : true iff constraints are satisfiable
variable: $result$
1  if $\nu(r) = \nu'(r')$ and $c(r) \wedge c'(r')$ is not satisfiable then
2      $result \leftarrow$ false // constraint violation found
3  else if $\nu(r) = \oplus$ then
4      // check if at least one choice leads to satisfiable
       constraints
5      $result \leftarrow$ false
6      for $n \in N$ with $(r, n) \in E$ do
7          if $\exists(x, y) \in E'$ with
           $x = r' \wedge (\nu'(y) = \nu'(n) \vee \nu'(y) = \oplus)$ then
8              $result \leftarrow$
               $result \vee$ traverse$((N, E, n), (N', E', y))$
9          else
10             $result \leftarrow$ true
11 else if $\nu'(r') = \oplus$ then
12     // check if at least one choice leads to satisfiable
       constraints
13     $result \leftarrow$ false
14     for $n' \in N'$ with $(r', n') \in E'$ do
15         if $\exists(x, y) \in E$ with
           $x = r \wedge (\nu(y) = \nu'(n) \vee \nu(y) = \oplus)$ then
16             $result \leftarrow$
               $result \vee$ traverse$((N, E, y), (N', E', n'))$
17         else
18             $result \leftarrow$ true
19 else
20     // check all child nodes
21     $result \leftarrow$ true
22     for $n \in N$ with $(r, n) \in E$ do
23         if $\exists(x, y) \in E'$ with
           $x = r' \wedge (\nu'(y) = \nu(n) \vee \nu'(y) = \oplus \vee \nu(n) = \oplus)$
           then
24             $result \leftarrow$
               $result \wedge$ traverse$((N, E, n), (N', E', y))$
25 return $result$
```

4.1.5 Removing nodes referring to nodes with multiple occurrences in the same context

In general, a meaningful conversion of pattern nodes corresponding to nodes with multiple occurrences in the same context is not possible and we need to eliminate these nodes from the pattern. One exception to this is the scenario where the pattern node is associated with an explicit positional constraint that disambiguates between multiple occurrences of a matching node (for example, `pubs/book[1]`). In this case, we can retain the pattern node and exploit its associated constraints for pruning. In the example from Figure 10, we need to remove the `book` node since the schema indicates that a `pubs` node may have multiple children of type

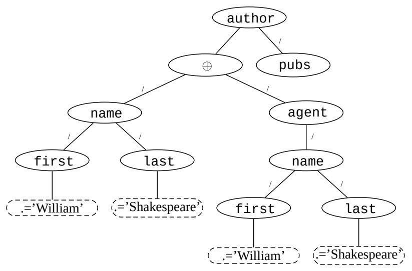


Fig. 11 Simplified pattern

book. The resulting simplified pattern is shown in Figure 11.

4.1.6 Traversal and pruning

After transforming both QTP and FTP into simplified tree patterns, we traverse both patterns simultaneously as described in Algorithm 3. Only pattern nodes occurring in both patterns are visited. For each pair of corresponding pattern nodes, we check whether the value constraints in one pattern contradict those in the other pattern. Since in simplified tree patterns each pattern node corresponds to a unique node from the collection within the context of a single document tree, a contradiction between patterns allows us to immediately eliminate the fragment corresponding to the FTP from further consideration.

Special care has to be taken when a choice node is encountered. In this case, a contradiction exists only if we can find contradictory constraints regardless of which branch of the choice we follow. If there is at least one choice without a contradiction, which may be a choice that leads to a branch that is not present in the other pattern, it is not possible to conclude that the fragment can be eliminated (lines 3-18).

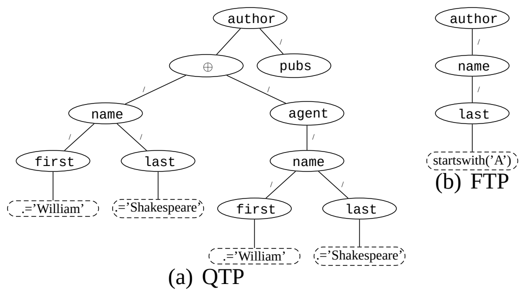


Fig. 12 Simplified QTP and FTP that are not contradictory

In the example shown in Figure 12, the traversal algorithm proceeds as follows. First, the author nodes in QTP and FTP are visited. Since there is no value constraint associated with this node in either pattern, there is no conflict, therefore we move on to the children of the author nodes. The pubs node is only present in the QTP and is therefore not visited. As the other child of the author node, the QTP contains a choice node. We now have to check both branches for conflict. The left branch leads to the name node, for which there is an equivalent node in the FTP. In both patterns the name node has a child with node test last. When inspecting the value constraints associated with the last nodes, the algorithm detects a contradiction because the content of the corresponding document node cannot be equal to the string 'Shakespeare' and at the same time start with the letter 'A'. Therefore, we know that there is a contradiction for the left branch of the choice node. In order for there to be a global contradiction, however, the patterns have to be contradictory for both branches of the choice node. Therefore, the algorithm still has to inspect the right branch, for which it encounters a node with the node test agent. For this node, there is no equivalent in the FTP and therefore no contradiction. Since the algorithm only found a contradiction for one branch of the choice node, there is no global contradiction and the fragment corresponding to the FTP cannot be pruned.

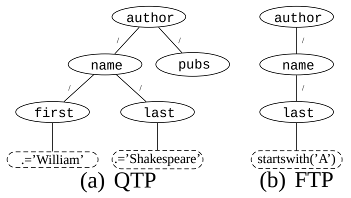


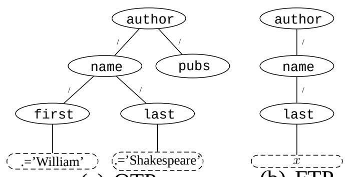
Fig. 13 Simplified QTP and FTP that are contradictory

For the example in Figure 13, on the other hand, the traversal algorithm does detect a contradiction. After inspecting the author and name nodes in both patterns, the algorithm reaches the last nodes and their contradicting value constraints. This time, the last node does not occur as the descendant of a choice node so this contradiction is sufficient to prune the fragment corresponding to the FTP.

4.1.7 Analysis and optimization

While it may seem that the transformation and traversal of QTP and FTPs could pose a significant overhead, there are a number of considerations that reduce this impact. The transformation of the FTPs only has to be performed once when the fragmentation is set up. Therefore, it does not pose a run-time overhead during query execution.

For the transformation of the QTP, we make the following observations: child steps are either copied from the QTP to the canonical QTP or omitted. Both the size of the canonical QTP and the time consumed by the transformation are therefore linear in $|E^{\text{QTP}}_{\texttt{child}}|$, which is the number of child steps in the QTP. For each descendant step, in the worst case, Algorithm 2 introduces one choice node and one non-choice pattern node for each σ in Σ. Therefore, the size of the canonical QTP is linear in $|E^{\text{QTP}}_{\texttt{desc}}|\ |\Sigma|$. In order to analyze the time complexity, we also have to take into account the time consumed by computing the reachable schema subgraph and by detecting cycles in the resulting graph. We can compute the subgraph consisting of nodes that are reachable from node a and from which b is reachable by first marking all nodes reachable from a, then marking all nodes from which b is reachable and finally choosing all nodes



(a) QTP (b) FTP

Fig. 14 Simplified QTP and abstract FTP

that were marked both times. Assuming a suitable representation of the graph, this can be done in $O(|\Sigma| + |\Psi|)$ time. Using Tarjan's algorithm [11], we can detect cycles in $O(|\Sigma| + |\Psi|)$ time. Therefore, the transformation of a QTP takes $O(|E^{\text{QTP}}_{\texttt{child}}| + |E^{\text{QTP}}_{\texttt{desc}}|\ (|\Sigma| + |\Psi|))$ time and yields a result containing $O(|E^{\text{QTP}}_{\texttt{child}}| + |E^{\text{QTP}}_{\texttt{desc}}|\ |\Sigma|)$ nodes. Since the result is also a directed graph, in which nodes may be shared among multiple branches, the equivalent tree pattern is of size $O(|E^{\text{QTP}}_{\texttt{desc}}|\ |\Sigma|\ |E^{\text{QTP}}_{\texttt{child}}| + |E^{\text{QTP}}_{\texttt{desc}}|^2\ |\Sigma|^2)$. This is important, because the time consumed by the subsequent traversal step depends on the size of the equivalent tree.

The time required to traverse the QTP and the FTPs is linear in the size of the tree representations of the canonical QTP and the FTPs. Because the traversal has to be performed for each fragment, it is also linear in the number of fragments. This leads to an overall time complexity of $O((|E^{\text{QTP}}_{\texttt{desc}}|\ |\Sigma|\ |E^{\text{QTP}}_{\texttt{child}}| + |E^{\text{QTP}}_{\texttt{desc}}|^2\ |\Sigma|^2)\ (|E^{\text{FTP}}_{\texttt{desc}}|\ |\Sigma|\ |E^{\text{FTP}}_{\texttt{child}}| + |E^{\text{FTP}}_{\texttt{desc}}|^2\ |\Sigma|^2)\ |F|)$. Note that run-time of the pruning algorithm depends solely on the size of the patterns, the number of fragments and the size of the schema. It is independent of the size of the collection.

Since horizontal fragmentation is defined as a partitioning of the data collection, FTPs need to be disjoint and cover the entire collection. Because of this, we expect that in many instances the FTPs will only differ in their value constraints but not in their structure. It is therefore possible to simplify the traversal process by traversing the QTP together with a single, abstract FTP, rather than with each FTP in the fragmentation. In this abstract FTP, value constraints are replaced with variables. Traversal of QTP and abstract FTP results in a formula that describes the conditions under which there is a contradiction between the QTP and an FTP. Figure 14(b) shows an abstract FTP, in which a value constraint has been replaced with the variable x. Traversing this abstract FTP with the QTP in Figure 14(a) shows that there is a contradiction if $\neg(\text{.='Shakespeare'} \wedge x)$ holds.

We can now instantiate x with the corresponding value constraint from each of the original simplified FTPs, i.e., with the expressions

$$\text{startswith('A')},\ \ldots, \text{startswith('S')},\ \ldots, \text{startswith('Z')}$$

Solving this formula yields a contradiction for all of these cases except $x = \text{startswith('S')}$. A similar optimization is possible for the QTPs if we assume that the structure of a query is known at compile time whereas the constants used in value constraints are only known at run time.

4.2 Localization and pruning with vertical fragmentation

In this section, we define an initial strategy for evaluating QTPs on a vertically fragmented collection based on the following steps:

– First, we decompose the global QTP into a set of *local QTPs* corresponding to the individual fragments.
– Then, we use an existing tree pattern evaluation strategy to evaluate the local QTPs on the fragments (the specific strategy is left to each site to decide).
– After that, we combine the partial results generated at each site by joining the matches derived from individual fragments based on their proxy/root proxy IDs. How this is done is specified by a *distributed execution plan*.

We then improve upon this initial strategy and present two techniques that allow us to eliminate certain fragments from the distributed execution plan.

4.2.1 Localization of QTPs

Localization is the process of determining which fragments are relevant to a given query and decomposing the query into sub-queries that can be evaluated on individual fragments. As mentioned before, QTPs provide a convenient abstraction for decomposing a global query into sub-queries that are local to a single fragment. We have therefore chosen to perform query decomposition at the QTP level before transforming the resulting local QTPs into algebraic query plans at the individual sites.

The decomposition of a global QTP into a set of local QTPs directly follows the schema graph. After unrolling wildcard nodes (using a procedure similar to Algorithm 2), Algorithm 4 divides the global QTP into a set of sub-patterns, each of which consists of pattern nodes that match nodes in the same fragment. Edges between pattern nodes in the same subtree are assigned the same axis type as the corresponding edge in the global QTP.

A child edge from a pattern node in sub-pattern a to one in sub-pattern b is converted to a pattern node matching a proxy in a and a pattern node matching a root proxy in b. These new pattern nodes are marked as extraction points because they are needed to join the results of local QTPs to generate the final result.

When descendant edges across fragment boundaries are encountered, we need to identify all paths in the fragmentation schema that satisfy the descendant edge. This can be achieved, for example, by unrolling the descendant step into child steps according to the same procedure that is used by the horizontal transformation algorithm (i.e., Algorithm 2). It is important to note that this unrolling may turn a single cross-fragment descendant step into multiple cross-fragment child steps. This corresponds to the case where a descendant step traverses multiple fragments. Consider, for example, the descendant step `author//reference`. When this step is unrolled, it yields two cross-fragment child steps: `author/pubs` and `book/chapter`. Therefore, an

Algorithm 4: Vertical localization

```
input  : global QTP $(N, E, r, \nu, \epsilon, T, c)$, schema
         $(\Sigma, \Psi, s, m, \rho)$, vertical fragmentation function
         $\phi : \Sigma \rightarrow F_\Sigma$
output : set of local QTPs with fragment they are evaluated on
         $Q = \{((N', E', r', \nu', \epsilon', T', c'), f' \in F_\Sigma)\}$
1  $Q \leftarrow \{(N', E', r', \nu', \epsilon', T', c')$ maximal $\mid (\exists f \in F_\Sigma, \forall n' \in N' : \phi(\nu(n')) = f) \wedge (E' = E \cap (N' \times N')) \wedge ((N', E')$ is connected and rooted at $r') \wedge (\nu' = \nu) \wedge (\epsilon' = \epsilon) \wedge (T' = T \cap N') \wedge (c' = c)\}$ // construct local QTPs without cross-fragment edges
2  for $(n_1, n_2) \in E, \phi(\nu(n_1)) \neq \phi(\nu(n_2))$ do
3      $i \leftarrow$ unique ID
4      $q_1 \leftarrow (N_1, E_1, r_1, \nu_1, \epsilon_1, T_1, c_1) \in Q, n_1 \in N_1$
5      $q_2 \leftarrow (N_2, E_2, r_2, \nu_2, \epsilon_2, T_2, c_2) \in Q, n_2 \in N_2$
6      $p_i \leftarrow$ new pattern node
7      $rp_i \leftarrow$ new pattern node
8      $N_1 \leftarrow N_1 \cup \{p_i\}$
9      $N_2 \leftarrow N_2 \cup \{rp_i\}$
10     $\nu_1(p_i) \leftarrow$ proxy $i$
11     $\nu_2(rp_i) \leftarrow$ root proxy $i$
12     $T_1 \leftarrow T_1 \cup \{p_i\}$
13     $T_2 \leftarrow T_2 \cup \{rp_i\}$
14     $E_1 \leftarrow E_1 \cup \{(n_1, p_i)\}$
15     $E_2 \leftarrow E_2 \cup \{(rp_i, n_2)\}$
16     $\epsilon((n_1, p_i)) \leftarrow \epsilon((n_1, n_2))$
17     $\epsilon((rp_i, n_2)) \leftarrow \epsilon((n_1, n_2))$
18     $r_2 \leftarrow rp_i$
```

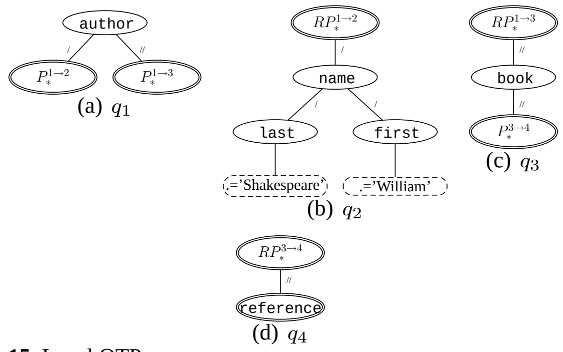


Fig. 15 Local QTPs

additional local QTP corresponding to fragment f_3^V (which contains the `pubs` and `book` node types) is introduced, even if there is no pattern node in the global QTP that refers to node types in this fragment.

If the global QTP does not reach a certain fragment (because even after unrolling no constraints are placed on the node types contained in this fragment) and if no intermediate QTP has to be generated for it because of cross-fragment descendant steps, then the localized plan derived from the local QTPs will not access this fragment. Therefore, the localization technique eliminates some vertical fragments even without further pruning.

Localizing the global QTP shown in Figure 5 yields the set of local QTPs shown in Figure 15(a)–(d). Each cross-fragment edge in the global QTP is represented by a pair of pattern nodes that match a proxy/root proxy pair. The edge from `author` to `name`, for example, is replaced by the pattern node $RP_*^{1\to2}$ in q_2 and the pattern node $P_*^{1\to2}$ in q_1. The pattern node $RP_*^{1\to2}$ matches all of the root proxy nodes $RP_i^{1\to2}$ in q_2's fragment f_2. The pattern node $P_*^{1\to2}$ matches the proxy nodes $P_i^{1\to2}$ in f_2's parent fragment f_1; these are the proxy nodes that correspond to $RP_i^{1\to2}$. Since the original pattern edge is a child edge, edges to and from the generated pattern nodes are also child edges. In the case where the original pattern edge is a descendant edge (such as the edge between `author` and `book`, which is represented by the pattern nodes labeled $P_*^{1\to3}$ and $RP_*^{1\to3}$), edges to and from the generated pattern nodes are also descendant edges.

Whenever we decompose a global QTP q, there will be exactly one local QTP that does not contain a pattern node that matches a root proxy node. We refer to this local QTP as the *root QTP*. In our example, q_1 is the root QTP. All other local QTPs contain exactly one pattern node that matches root proxy nodes in their fragments. If local QTP q_s contains a pattern node labeled $RP_*^{i\to j}$ and local QTP q_t contains the corresponding pattern node labeled $P_*^{i\to j}$, then we call q_s a *child QTP* of q_t and q_t a *parent QTP* of q_s.

4.2.2 Conversion of Local QTPs to Local Plans

Each local QTP q_i is then transformed into a local query plan p_i. This is done at the site holding the fragment corresponding to q_i, using centralized XML query evaluation strategies (e.g., [12,13]). The pruning techniques presented in this paper are independent of the techniques used by local query plans. We therefore omit a detailed description of local plan generation.

For the purpose of illustration, Figure 16 shows a set of local plans based on structural joins (p_1 through p_4), which correspond to the local QTPs q_1 through q_4, respectively.

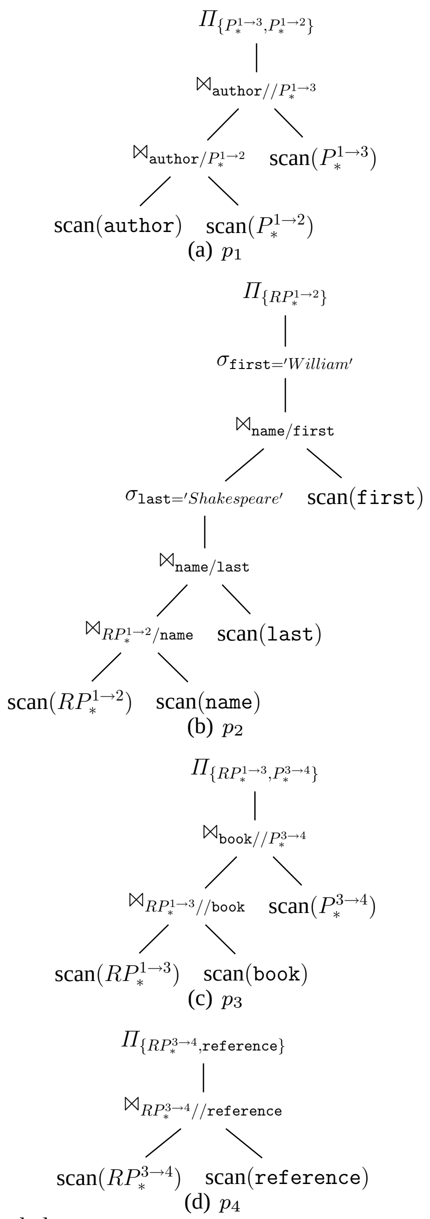
4.2.3 Distributed execution plans

To obtain the overall query result, the results of local plans need to be "combined" based on the IDs of their proxy and root proxy nodes. A *distributed execution plan* specifies how exactly this is done. In this section, we explore how distributed execution plans can be constructed and what their properties are.

Definition 7 Let $P = \{p_1, \ldots, p_n\}$ be the set of local query plans corresponding to a query q. For each $p_i \in P$, let f_i denote the vertical fragment corresponding to p_i. Further, let $P' \subseteq P$. Then $G_{P'}$ is a *distributed execution plan* for P' iff

1. $P' = \{p_i\}$ and $G'_P = p_i$, or
2. $P' = P'_a \cup P'_b$, $P_a \cap P_b = \emptyset$; $p_i \in P_a, p_j \in P_b$, $p_i =$ parent(p_j); $G_{P'_a}$ and $G_{P'_b}$ are distributed execution plans for P'_a and P'_b, respectively; and $G_{P'} = G_{P'_a} \bowtie_{P_*^{i\to j}.id = RP_*^{i\to j}.id} G_{P'_b}$.

If G_P is a distributed execution plan for P (the entire set of local query plans), then $G_q = G_P$ is a distributed execution plan for q.

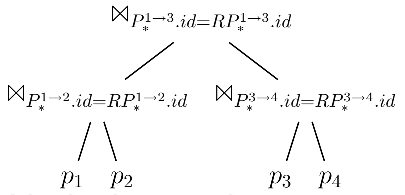


$\Pi_{\{P_*^{1\to3}, P_*^{1\to2}\}}$
$\bowtie_{\texttt{author}//P_*^{1\to3}}$
$\bowtie_{\texttt{author}/P_*^{1\to2}}$
scan($P_*^{1\to3}$)
scan(author)
scan($P_*^{1\to2}$)
(a) p_1
$\Pi_{\{RP_*^{1\to2}\}}$
$\sigma_{\texttt{first}='William'}$
$\bowtie_{\texttt{name/first}}$
$\sigma_{\texttt{last}='Shakespeare'}$
scan(first)
$\bowtie_{\texttt{name/last}}$
$\bowtie_{RP_*^{1\to2}/\texttt{name}}$
scan(last)
scan($RP_*^{1\to2}$)
scan(name)
(b) p_2
$\Pi_{\{RP_*^{1\to3}, P_*^{3\to4}\}}$
$\bowtie_{\texttt{book}//P_*^{3\to4}}$
$\bowtie_{RP_*^{1\to3}//\texttt{book}}$
scan($P_*^{3\to4}$)
scan($RP_*^{1\to3}$)
scan(book)
(c) p_3
$\Pi_{\{RP_*^{3\to4}, \texttt{reference}\}}$
$\bowtie_{RP_*^{3\to4}//\texttt{reference}}$
scan($RP_*^{3\to4}$)
scan(reference)
(d) p_4

Fig. 16 Local plans

A distributed execution plan must contain all the local plans corresponding to the query. As shown in the recursive definition above, an execution plan for a single local plan is simply the local plan itself (condition 1). For a set of multiple local plans P' we assume that P'_a and P'_b are two non-overlapping subsets of P' such that $P'_a \cup P'_b = P'$. We require that P'_a contains the parent local plan p_i for some local plan p_j in P'_b. An execution plan for P' is then defined by combining execution plans for P'_a and P'_b using a join whose predicate compares the IDs of root proxy nodes derived from p_j to the IDs of corresponding proxy nodes derived from p_i (condition 2). We refer to this join as a *cross-fragment join*.

If G'_P consists of a single local plan p_i, then the set of attributes returned by G'_P (referred to as $M_{G'_P}$) is identical to the set of attributes returned by p_i. If $G_{P'} = G_{P'_a}$ $\bowtie_{P_*^{i\to j}.id = RP_*^{i\to j}.id}$ $G_{P'_b}$, then $M_{G'_P} = M_{G_{P'_a}} \cup M_{G_{P'_b}} \setminus \{P_*^{i\to j}, RP_*^{i\to j}\}$.



$\bowtie_{P_*^{1\to3}.id=RP_*^{1\to3}.id}$
$\bowtie_{P_*^{1\to2}.id=RP_*^{1\to2}.id}$
$\bowtie_{P_*^{3\to4}.id=RP_*^{3\to4}.id}$
p_1
p_2
p_3
p_4

Fig. 17 Initial distributed execution plan

Figure 17 shows a distributed execution plan that combines the results of the local plans shown in Figure 16. There are usually many different vertical execution plans that all yield the correct result but that may vary in cost. Since the focus of this paper is on localization and pruning, we do not discuss the problem of picking the most advantageous plan.

4.2.4 Skipping fragments

The localization strategy for vertical fragmentation avoids accessing fragments whose node types are not reached by the global QTP. It does not, however, address a scenario where node types in a fragment are reached by the global QTP but no constraints are placed on these node types. Consider, for example, the local QTP shown in Figure 15(c), which is evaluated on fragment f_3^V. Its sole purpose is to determine which proxy nodes in f_1^V lead to which root proxy nodes in fragment f_4^V. Since the only way from a root proxy node in f_3^V to a proxy node in the same fragment is through a book node, no further constraints are placed on f_3^V. We now propose a technique that allows us to avoid accessing such intermediate fragments, and, thereby, prunes the local QTPs corresponding to these fragments from a distributed query plan.

We achieve this by storing information that allows us to identify all ancestor proxy nodes for any given root proxy node. Using this information, we can then determine for any root proxy node in f_4^V which proxy node in f_1^V is its ancestor. This, in turn, allows us to answer the query without accessing f_3^V or evaluating the local QTP shown in Figure 15(c). The benefits of this are twofold: it reduces load on the intermediate fragments (since they are not accessed) and it avoids the cost of computing intermediate results and joining them together.

While it would be possible to store the ancestor-descendant join information in a centralized (or replicated) index structure, this could severely limit the scalability of distributed query processing. In addition, it would make update management more difficult. Therefore, we store the join information by numbering proxy nodes according to a scheme based on the Dewey decimal system[2] [14].

[2] We have also experimented with other numbering schemes, such as one where each proxy pair is identified by its pre-order and post-order position in the collection. Our techniques are applicable to these alternate representations as well.

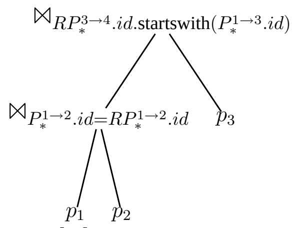

$\bowtie_{RP_*^{3\to4}.id.\text{startswith}(P_*^{1\to3}.id)}$
$\bowtie_{P_*^{1\to2}.id=RP_*^{1\to2}.id}$
p_3
p_1
p_2

Fig. 18 Skipping vertical plan

To define this numbering scheme, we need to distinguish between the following two cases: **(i)** If a document subtree does not have a root proxy node as its root (i.e., if the subtree contains the root element of a document tree in the collection, which can only occur in the root fragment), then the proxy nodes in this subtree (and, of course, the root proxy nodes in other fragments that correspond to these proxy nodes) receive simple numeric IDs. In the collection shown in Figure 2, this can be seen in all subtrees in fragment f_1^V. The proxy nodes in this fragment therefore receive numeric IDs, which means that all $(R)P_*^{1\to2}$ and $(R)P_*^{1\to3}$ are already numbered in accordance with our numbering scheme. **(ii)** If a document subtree is rooted at a root proxy node then the ID of each of its proxy nodes is prefixed by the ID of the root proxy node of the subtree, followed by a numeric identifier that is unique within this subtree. In Figure 2, fragments f_2^V, f_3^V and f_4^V consist of subtrees that are rooted at a root proxy. However, only fragment f_3^V contains proxy nodes. Therefore, only $P_{18}^{3\to4}$, $P_{19}^{3\to4}$ and $P_{20}^{3\to4}$ have to be renumbered. $P_{18}^{3\to4}$ is part of a subtree that is rooted at the root proxy node $RP_{12}^{1\to3}$. We would therefore have to renumber it to $P_{12.1}^{3\to4}$. Similarly, $P_{19}^{3\to4}$ would be renumbered to $P_{14.1}^{3\to4}$ and $P_{20}^{3\to4}$ to $P_{16.1}^{3\to4}$.

If all proxy pairs are numbered according to this scheme, a root proxy node is the descendant of a proxy node precisely when the ID of the proxy node is a prefix of the ID of the root proxy node. When evaluating query patterns, we can exploit this information by removing local QTPs from the distributed query plan if they contain no value or structural constraints, and no extraction point nodes other than those corresponding to proxies. These local QTPs are only needed to determine whether a root proxy node in some other fragment is a descendant of a proxy node in a third fragment, which we can now infer from the skipping IDs. Using this optimization, we can rewrite the query plan from Figure 17 to the simpler plan shown in Figure 18, which avoids accessing fragment f_3^V.

It is important to note that our numbering scheme does not complicate update management. Subtrees can be inserted or removed from a document collection without having to modify other parts of the collection and without having to maintain a centralized index.

4.2.5 Structural constraints in skipped fragments

While skipping IDs allow us to skip fragments on which no constraints are placed, sometimes structural constraints make it necessary to access intermediate fragments, even

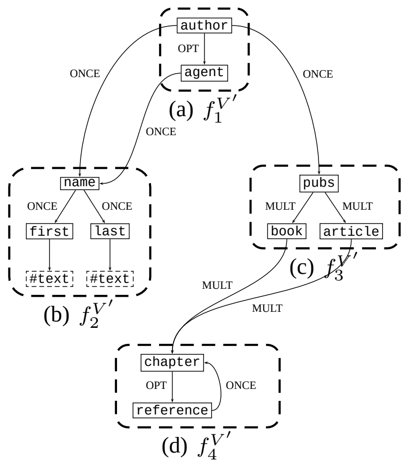


Fig. 19 A modified fragmentation schema

if they are not needed for evaluating value constraints. To illustrate this, consider the modified fragmentation schema shown in Figure 19, which adds the additional type of publication `article`. If we evaluate the local QTPs shown in Figure 15 on this modified schema, we can no longer eliminate the local QTP in Figure 15(c) because skipping the corresponding fragment would mean that we could no longer distinguish between the subtrees in fragment $f_4^{V'}$ that are part of a `book` and those that are part of an `article`.

We propose a technique that allows us to skip such fragments. In addition to storing skipping IDs, we use the proxy IDs to keep track of some structural information for cases where there is ambiguity. We define structural ambiguity as follows:

Definition 8 Let f_a be a fragment whose subtrees are rooted at root proxy nodes and assume that subtrees in f_a contain proxy nodes that refer to fragment f_b. Then f_a is *structurally ambiguous* with respect to f_b if there is more than one path in the schema of f_a that leads from a root proxy node in f_a to a proxy node in f_a that corresponds to f_b.

If f_a is structurally ambiguous with respect to f_b, then we add label path information to the proxy ID of each proxy node in f_a that corresponds to f_b. This information consists of the labels encountered on a path from the root proxy of the subtree in which the proxy occurs to the proxy itself. Since the label path information is part of the locally unique identifier specified by our numbering scheme, it is also part of the prefix of the IDs of proxy nodes that are descendants of the proxy node for which it was inserted.

In the case of the fragmentation schema shown in Figure 19, there is one instance of structural ambiguity: fragment $f_3^{V'}$ is structurally ambiguous with respect to $f_4^{V'}$. This is because there are two label paths from a root proxy in $f_3^{V'}$ that could lead to a proxy node that corresponds to $f_4^{V'}$: `pubs/book` and `pubs/article`. We therefore store the label path as part of the ID of each proxy node in $f_3^{V'}$ that corresponds to $f_4^{V'}$. Figure 20 shows a sample instance of fragment $f_3^{V'}$ with label path IDs.

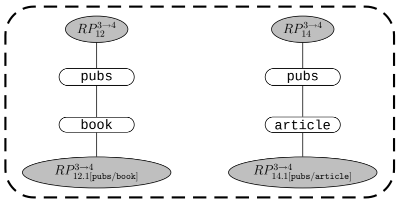

$RP^{3\to4}_{12}$
pubs
book
$RP^{3\to4}_{12.1[\text{pubs/book}]}$
$RP^{3\to4}_{14}$
pubs
article
$RP^{3\to4}_{14.1[\text{pubs/article}]}$

Fig. 20 Fragment f_3^V with label path IDs

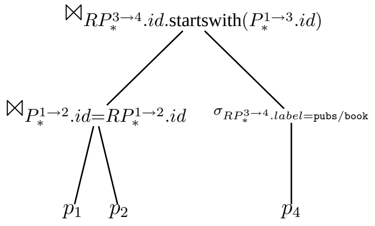

$\bowtie_{RP^{3\to4}_{*}.id.\text{startswith}(P^{1\to3}_{*}.id)}$
$\bowtie_{P^{1\to2}_{*}.id=RP^{1\to2}_{*}.id}$
$\sigma_{RP^{3\to4}_{*}.label=\text{pubs/book}}$
p_1
p_2
p_4

Fig. 21 Label path plan

Label paths as defined here can be viewed as a materialization of structural selections on linear paths through a particular fragment. Thus, they contain sufficient information to evaluate structural constraints in a linear path, as in the QTP shown in Figure 15(c). In combination with skipping IDs, label paths therefore allow us to evaluate the query using the plan shown in Figure 21, which avoids accessing $f_3^{V'}$.

4.2.6 Analysis

Assuming that we use the unrolling technique presented in the section on horizontal localization, the upper bound on the total size of local QTPs obtained by vertical localization is $O(|E^{\text{QTP}}_{\texttt{child}}|+|E^{\text{QTP}}_{\texttt{desc}}||\Sigma|)$. In practice, where schema graphs tend to be sparse, we can expect the total size of all local QTPs to be close to the size of the original QTP.

Both skipping IDs and label paths are inserted at fragmentation time and whenever data are added to the collection. Since they are not replicated, local insertions and deletions can be handled without having to modify other fragments.

The vertical pruning techniques proposed here operate solely on the QTP and the fragmented schema graph. They are independent of the size of the data and of the constants used in value constraints. This allows us to perform pruning at query compile time, thereby minimizing the run-time overhead introduced by our technique.

Label paths are useful not only for localization but also for pruning irrelevant subtrees within fragments [2]. Studying further uses of label paths in a distributed context is the subject of future research.

5 Workload-aware fragmentation of collections

In this section, we propose a set of fragmentation algorithms that determine a fragmentation schema that optimizes performance for a given query workload. The previous section has identified a number of properties that a fragmentation schema needs to possess in order for localization and pruning to achieve high query performance. In the case of horizontal fragmentation, it is important that the FTPs are defined such that for a given QTP in the workload contradictions can be found that allow us to exclude some of the fragments. For vertical fragmentation, a suitable fragmentation schema should aim to maximize parallelism between the (non-skippable) sub-queries of a given query while avoiding excessively large intermediate results. In either case, what constitutes a good fragmentation schema cannot be defined independently of the query evaluation strategy used. While in practice fragmentation is performed before query evaluation, we have chosen to present our fragmentation algorithms after our query evaluation strategies in order to better illustrate this dependency.

5.1 Horizontal fragmentation

Horizontal fragmentation allows us to directly apply a fragmentation algorithm that was originally developed for relational systems. This relational fragmentation algorithm is based on minterm predicates, which are conjunctions of simple predicates on individual attributes. Minterm predicates are obtained by extracting the predicates found in the query workload, decomposing them into simple predicates consisting of a single (in)equality and finally combining these simple predicates such that all possible combinations of simple predicates are covered [1].

In order to apply this technique, we need to transform the predicates found in tree patterns into simple predicates from which minterm predicates can be constructed. We do this by first unrolling descendant steps into child steps (using the same procedure employed in Algorithm 2). Then, each value constraint in the pattern can be transformed into a set of simple predicates whose left-hand side is the path from the root of the unrolled tree pattern to the node with which the value constraint is associated.

Performing this transformation for the workload shown in Table 2 yields the constraints shown in Table 3. We then extract the simple predicates from these constraints, i.e. predicates that do not contain conjuction or disjunction. The result of this is shown in Table 1.

`/author/name/last==''Shakespeare''`
`/author/name/last==''John''`
`/author/name/first==''William''`

Table 1 Simple Predicates

From these simple predicates, we can then construct minterm predicates using the same techniques applied to the relational scenario. The minterm predicates derived from the simple predicates in Table 1 are shown in Table 4. Based on these minterm predicates, we can then apply the relational fragmentation algorithm.

5.2 Vertical fragmentation

To evaluate a query over a vertically fragmented collection, we evaluate each sub-query on its corresponding fragment

Q1	`/author[name/last=''Shakespeare'' or name/last=''John'']/pubs/book`
Q2	`/author[name/first=''William'']/pubs/book`

Table 2 Sample workload

Path	Constraint
`/author/name/last`	`.==''Shakespeare''` $\vee$ `.==''John''`
`/author/name/first`	`.==''William''`

Table 3 Constraints

`/author/name/last==''Shakespeare''` $\wedge$ `/author/name/first==''William''`
`/author/name/last==''Shakespeare''` $\wedge$ `/author/name/first!=''William''`
`/author/name/last==''John''` $\wedge$ `/author/name/first==''William''`
`/author/name/last==''John''` $\wedge$ `/author/name/first!=''William''`
`/author/name/last!=''Shakespeare''` $\wedge$ `/author/name/last!=''John''` $\wedge$ `/author/name/first==''William''`
`/author/name/last!=''Shakespeare''` $\wedge$ `/author/name/last!=''John''` $\wedge$ `/author/name/first!=''William''`

Table 4 Minterm Predicates

and then join the intermediate results to obtain the overall query result. Depending on how the collection is fragmented, the intermediate results may be large and the subqueries may be expensive to evaluate. In extreme cases, this can lead to a scenario where it is more expensive to evaluate a query on a vertically distributed collection than it is to evaluate the same query in a centralized fashion. In order to avoid this situation and to take full advantage of the potential of vertical distribution, we have to ensure that the vertical fragmentation schema is well suited to the query workload.

In the following, we propose a vertical fragmentation algorithm that chooses a suitable vertical fragmentation schema for a given query workload. Our algorithm is based on a cost model, which estimates the response time of a query when evaluated over a vertically fragmented collection.

5.2.1 Cost model

We define the following cost metrics for each local plan p_j and its corresponding fragment $f(p_j)$:

- cost(p_j), the response time of evaluating p_j on $f(p_j)$,
- scancost(p_j), the time it takes to scan the root proxy nodes in $f(p_j)$ that are accessed by p_j,
- card(p_j), the number of tuples returned by p_j when evaluated on $f(p_j)$,
- snip(p_j), the number of document subtrees in $f(p_j)$ that are accessed by p_j.

While it is possible to obtain these metrics experimentally, this can be expensive and in practice it may be preferable to estimate these values using various cost estimation techniques that have been developed for the centralized evaluation of XML queries. For notational convenience, we do not distinguish between estimated cost metrics and their precise counterparts.

Since the local plans can be evaluated independently of each other in parallel, we can model the cost of a query q as cost$(q) = \max\{\text{cost}(p_j) \mid p_j \in P\}$ where P is the set of local plans (after pruning) corresponding to q for a given vertical fragmentation schema.

5.2.2 Heuristic fragmentation algorithm

The naïve strategy for determining the best fragmentation schema for a given workload would be to exhaustively enumerate all possible vertical fragmentation schemas, compute the total cost for each of them and then choose the schema with the lowest cost. While this is guaranteed to yield the optimal result, the large number of possible vertical fragmentation schemas generally makes this strategy infeasible (there are B_n alternatives, where B_n is the n^{th} Bell number and n is the number of node types in the schema).

To obtain a feasible fragmentation algorithm, we instead propose a heuristic strategy that starts out with an initial fragmentation schema in which each node type is placed in its own fragment and then greedily merges fragments until we can no longer reduce the estimated workload cost. While this strategy is not guaranteed to find the global optimum, our experiments show that it performs well in practice.

In the following, we explain how the greedy algorithm works for a single query. Details are shown in Algorithm 5. After determining the local cost metrics for each local plan based on the initial fragmentation, we identify the plan with the highest local cost p_{max} (ignoring local plans that can be pruned) and its corresponding fragment $f(p_{\text{max}})$. Since the overall cost of the query is determined by the cost of the most expensive local plan, we can focus on decreasing the cost of p_{max}.

To do this, we attempt to merge $f(p_{\text{max}})$ with one of its ancestor fragments. We start with $f(p_{\text{max}})$'s parent fragments. For each parent fragment f_i, we merge $f(p_{\text{max}})$ and f_i, and then determine the cost of each non-prunable local plan corresponding to $f_i \cup f(p_{\text{max}})$. If the cost of all of these plans is lower than cost(p_j), we remove f_i and $f(p_{\text{max}})$ from the fragmentation schema and insert $f_i \cup f(p_{\text{max}})$. We then repeat the procedure by determining the most expensive local plan for the modified fragmentation schema and attempting to reduce its cost.

If none of the parent fragments of $f(p_{\text{max}})$ allow us to reduce the maximum local plan cost, we try $f(p_{\text{max}})$'s "grand-parent" fragments, "great grand-parent" fragments, and so forth. When merging with an ancestor fragment f_i that is not a direct parent of $f(p_{\text{max}})$, we merge all the fragments on the path from $f(p_{\text{max}})$ to f_i. If no ancestor fragment of $f(p_{\text{max}})$ allows us to reduce the maximum local plan cost, the algorithm terminates without making further modifications to the fragmentation schema.

Algorithm 5: Vertical fragmentation

```
   input  : query plan $p$, schema $(\Sigma, \Psi, s, m, \rho)$
   output : vertical fragmentation schema$\{\Sigma' \subseteq \Sigma\}$
1  $I \leftarrow \{\{\sigma\} \mid \sigma \in \Sigma\}$
2  $i_{\text{max}} \leftarrow i \in I$ s.t. cost$(p_i(i)) = \max\{\text{cost}(p_j(j)) \mid j \in I\}$
3  for $i_{ancestor} \in ancestor(i_{max})$ do
4      $i_{\text{merged}} \leftarrow i_{\text{max}} \cup \ldots \cup i_{\text{ancestor}}$
5      if $cost(p_{i_{max}}(i_{max})) > cost(p())$ then
6          $I \leftarrow (I \cup i_{\text{merged}}) \setminus i_{\text{max}}, \ldots, i_{\text{ancestor}}$
7          goto 2
8  return $(I)$
```

5.2.3 *Estimating local plan costs after merging*

Our fragmentation algorithm relies on frequent tentative merges between fragments. While it is possible to re-estimate the cost of all affected local plans after each such merge, this can be expensive. To address this, we propose a method for estimating the cost of a local plan p_{ij} corresponding to the fragment $f(p_i) \cup f(p_j)$ based on cost estimates for p_i (corresponding to $f(p_i)$) and p_j (corresponding to $f(p_i)$'s parent fragment $f(p_j)$):

$$\text{cost}(p_{ij}) = \text{cost}(p_j) + \frac{\text{card}(p_j)}{\text{snip}(p_i)}\big(\text{cost}(p_i) - \text{scancost}(p_i)\big)$$

The rationale behind this is as follows: cost(p_{ij}) includes all of the cost of the local plan corresponding to the parent fragment, cost(p_j). The cost of the child fragment is scaled by the selectivity of the parent fragment, represented as the fraction of the subtrees in $f(p_i)$ for which corresponding proxy nodes are returned by p_j. We also subtract the portion of the cost that can be attributed to scanning the root proxy nodes in $f(p_i)$. Our experiments show that this approximation does not prevent us from identifying good vertical fragmentation schemas.

5.2.4 *Handling multiple-query workloads*

So far, for simplicity, we have focused on identifying a fragmentation schema for a single query. In practice, however, workloads generally consist of more than one query. It is possible to adapt our algorithm by modifying the termination condition: instead of terminating when the cost of the most expensive local plan cannot be reduced further, we check the most expensive local plans of each query in descending order of cost and only terminate once we cannot further reduce the cost of any of those.

6 Performance evaluation

We have enhanced the native XML database system NATIX [12] with distributed capabilities and implemented our techniques within this system. This allows us to validate our approach and to perform realistic experiments.

The goal of our experiments is to show that our techniques can improve the performance of query processing through distribution and pruning. To achieve this, we first focus on stress-testing the distribution techniques presented in this paper. We pay particular attention to how our pruning algorithms limit query execution to a subset of the fragments in a distributed collection and carefully analyze how this affects performance. Then, we run a number of experiments that represent realistic use cases based on the XPathMark benchmark [15].

All of our experiments rely on collections of on-line auction data generated by the XMark benchmark [16], which is a standard benchmark for evaluating XML query performance. The experiments are conducted on virtualized Linux machines within Amazon's Elastic Compute Cloud [17]. We use a separate instance (providing 1.7 GB of memory and a single-core 32 bit CPU) for each fragment, with an additional instance for dispatching queries. All instances run in the same availability zone, ensuring low-latency, high-throughput communication.

6.1 Horizontal fragmentation

For the horizontal fragmentation model, the goal of our evaluation is twofold: First, we want to verify that horizontal distribution allows us to improve both query response time and throughput. Then, we want to show that our pruning techniques allow us to further improve throughput beyond the level achieved by distributed execution alone without any adverse effects on response time.

Since our definition of horizontal fragmentation assumes a multiple-document collection, we conduct these experiments on an XMark collection that has been decomposed into multiple small documents. We do this by by placing each `open_auction` element into its own document along with its descendants and document subtrees referenced via ID/IDREF. This results in documents of regular structure with an average size of approximately 30 KB. We scale this collection to 350 MB, 3.5 GB and 35 GB.

6.1.1 *Balanced fragmentation*

Each `open_auction` element generated by XMark contains an auction end date and these dates are uniformly distributed across the years 1998-2001. We can therefore obtain a balanced horizontal fragmentation schema (i.e., a fragmentation schema in which all fragments are approximately the same size) by dividing this date range into non-overlapping periods of equal lengths, with each such period corresponding to one horizontal fragment. For this experiment,

Horizontal	Q1	/open_auction[./interval/end[.= xs:date(’12/28/2001’)]][initial > 120]//item/name
	Q2	/open_auction[./interval/end[.>= xs:date(’01/01/1998’)][.< xs:date(’12/28/1998’)]][initial > 120]//item/name
	Q3	/open_auction[./interval/end[.>= xs:date(’01/01/1998’)][.< xs:date(’12/28/1999’)]][initial > 120]//item/name
	Q4	/open_auction[./interval/end[.>= xs:date(’01/01/1998’)][.< xs:date(’12/28/2000’)]][initial > 120]//item/name
	Q5	/open_auction[./interval/end[.>= xs:date(’01/01/1998’)][.< xs:date(’12/28/2001’)]][initial > 120]//item/name
Vertical	Q6	/open_auction[initial > 200]/interval/end
	Q7	/open_auction//person//category[id=’category10’]
	Q8	/open_auction/bidder//person//category[id=’category10’]
	Q9	/open_auction/bidder//person[creditcard]//category[id=’category10’]
	Q10	/open_auction/bidder//person[creditcard]/profile[education]//category[id=’category10’]
XPathMark	A1	/site/closed_auctions/closed_auction/annotation/description/text/keyword
	A2	//closed_auction//keyword
	A3	/site/closed_auctions/closed_auction//keyword
	A4	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date
	A5	/site/closed_auctions/closed_auction[descendant::keyword]/date
	A6	/site/people/person[profile/gender and profile/age]/name
	B7	//person[profile/@income]/name

Table 5 Queries used in experiments

we use fragmentation schemas consisting of 1, 2, 4, 8, 16, 32, 64 and 99 fragments[3].

On this distributed collection, we evaluate 5 classes of queries. Q1 consists of queries that contain a point predicate on the auction end date, i.e., each query returns auctions that end on exactly one date within the 4 year period. Q2-Q5 represent range queries that cover 25%, 50%, 75%, and 100% of the date range, respectively. It is important to note that each time we run a query in one of these classes, we randomly choose a date/date range within the 4-year range. Table 5 shows an example of a query in each class.

We first measure the response time of evaluating the queries on the horizontally distributed collection. As in all measurements in this paper, the results reported in Figure 22 include the cost of constructing sub-query results at the individual sites, shipping them to the dispatcher and assembling them to the overall query result. In the case of the 35 GB collection, some data points are missing for centralized execution and the fragmentation schemas with a lower number of fragments. In these cases, the query did not finish within 2 hours.

When interpreting the results, we can see that horizontal distribution allows us to reduce query response time when compared to centralized execution (i.e., the scenario with a single fragment on a single machine). The more machines we add to the system (by fragmenting the collection into more fragments), the faster response time becomes. Similarly, adding more machines allows us to manage larger collections while maintaining the same level of response time. We can also observe that pruning does not result in a major improvement of response time when compared to distributed execution without pruning. This is expected since pruning is primarily intended to improve throughput. It is important, however, to point out that pruning has no negative impact on response time.

[3] We were limited to 100 EC2 instances running simultaneously. Since one instance is needed for the dispatcher, this means that we can use at most 99 instances to store fragments.

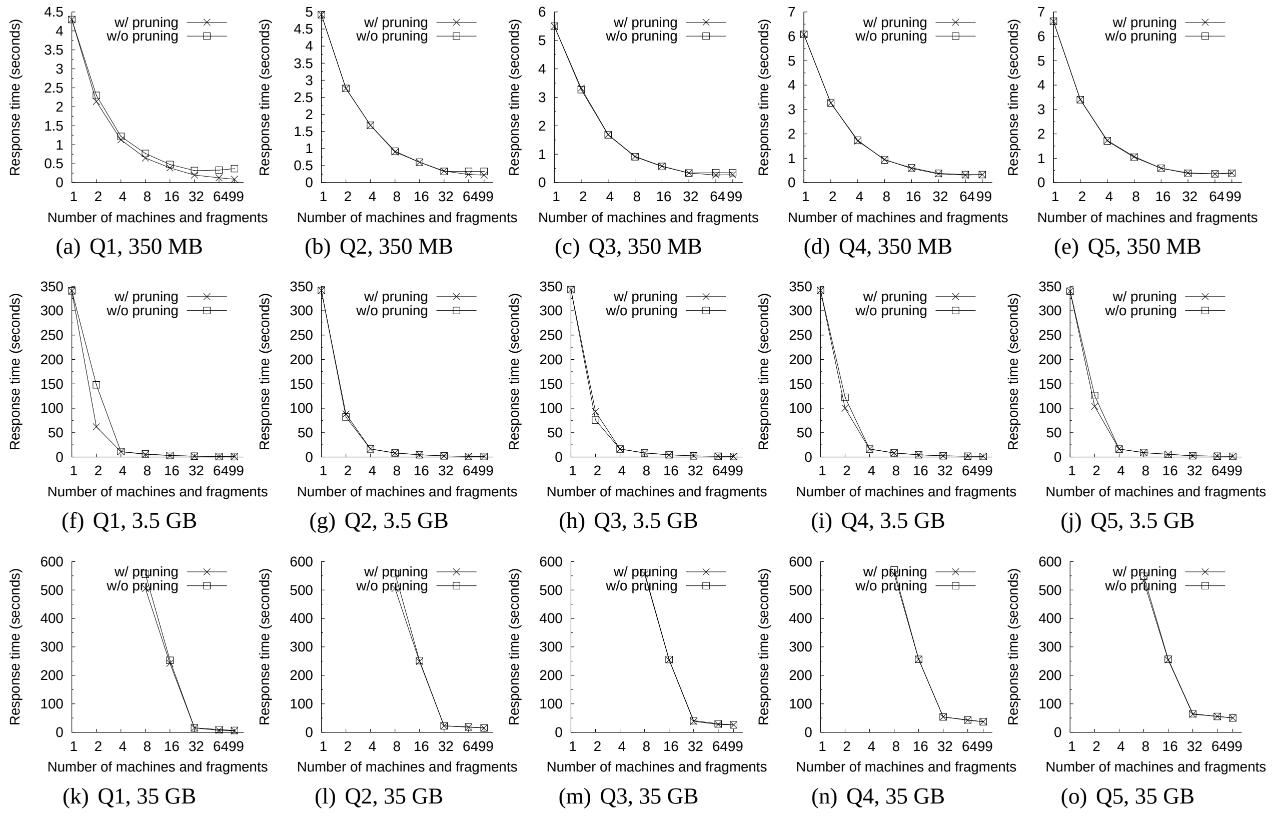
Next, we consider the throughput impact of distribution and pruning. To measure query throughput, we use multiple dispatcher processes to keep the system loaded with queries. In Figure 23, we report the maximum throughput rates we were able to achieve for each class of queries. Even without pruning, distribution significantly increases throughput and this increase in throughput is proportional to the number of fragments. Enabling pruning further improves throughput by a significant margin. Naturally, the impact of pruning is most pronounced for the class of point queries Q1, where a single date is selected and where our pruning algorithm can therefore avoid accessing all but one of the fragments for each query. Pruning also helps for the queries that involve a range of dates, particularly when this range is small, though the effect is less pronounced. For Q4 and Q5, where a large portion of the fragments or all fragments have to be inspected, pruning offers no advantage over simple distribution but it also does not harm performance (apart from some insignificant anomalies in the case of the 35 GB collection where throughput rates are very low).

This illustrates the importance of a fragmentation schema that is well suited to the workload: fragmenting on attributes on which single-value selections are performed leads to greater pruning opportunities than fragmenting on attributes that are used in wide range predicates. Even in the latter case, however, distributed evaluation by far outperforms centralized querying.

Our results also show that once a throughput of approximately 20 queries per second is achieved, further increasing the number of machines does not lead to improved performance. This can be explained by the fact that, at this point, the dispatcher is saturated, and distributed query evaluation is no longer the bottleneck in the system.

6.1.2 Skewed fragmentation

While pruning performs well on a balanced fragmentation, in practice it is not always possible to achieve this balance. We therefore measure the effect of pruning with a skewed

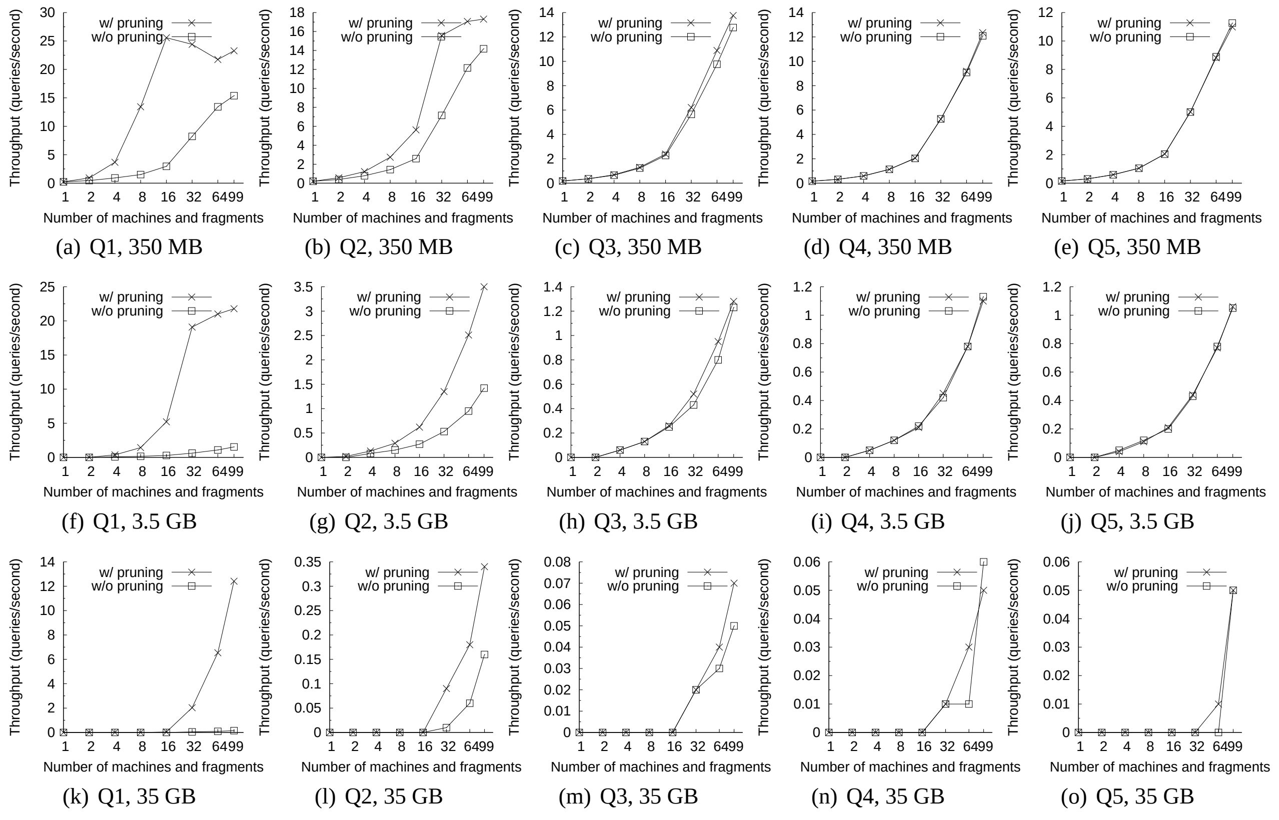


(a) Q1, 350 MB (b) Q2, 350 MB (c) Q3, 350 MB (d) Q4, 350 MB (e) Q5, 350 MB

(f) Q1, 3.5 GB (g) Q2, 3.5 GB (h) Q3, 3.5 GB (i) Q4, 3.5 GB (j) Q5, 3.5 GB

(k) Q1, 35 GB (l) Q2, 35 GB (m) Q3, 35 GB (n) Q4, 35 GB (o) Q5, 35 GB

Fig. 22 Response time, balanced horizontal fragmentation

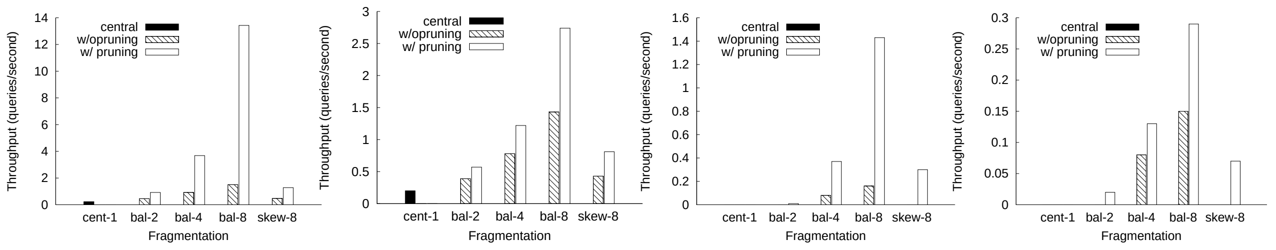


(a) Q1, 350 MB (b) Q2, 350 MB (c) Q3, 350 MB (d) Q4, 350 MB (e) Q5, 350 MB

(f) Q1, 3.5 GB (g) Q2, 3.5 GB (h) Q3, 3.5 GB (i) Q4, 3.5 GB (j) Q5, 3.5 GB

(k) Q1, 35 GB (l) Q2, 35 GB (m) Q3, 35 GB (n) Q4, 35 GB (o) Q5, 35 GB

Fig. 23 Throughput, balanced horizontal fragmentation



(a) Q1, 350 MB (b) Q2, 350 MB (c) Q1, 3.5 GB (d) Q2, 3.5 GB

Fig. 24 Throughput, balanced and skewed horizontal fragmentation

fragmentation consisting of 8 fragments. Our skewed fragmentation is defined as follows: The first fragment contains half of the entire collection (corresponding to the first 2 years of the 4-year period), the next fragment contains half of the remaining collection (i.e., 25% of the data), and so forth, with the last fragment containing the remainder of the collection data.

Figure 24 shows the throughput rates achieved by centralized query execution (which is vanishingly low in some of the cases shown), as well as distributed query execution (with and without pruning) on a balanced fragmentation consisting of 2, 4 and 8 fragments and on the skewed fragmentation. We use queries Q1 and Q2, for which pruning has been shown to be particularly effective. Even in the presence of skew, distribution results in a significant boost in performance over centralized querying in all cases. As with a balanced fragmentation schema, pruning further improves throughput.

The throughput rates obtained on the skewed fragmentation tend to fall between that of a balanced fragmentation with 2 fragments and 4 fragments. This can be explained by the fact that the largest fragment in the skewed fragmentation, which is the same size as a fragment in the balanced fragmentation with 2 fragments, represents a throughput bottleneck.

To further improve querying performance on a skewed distribution, it could be beneficial to replicate the most heavily loaded fragments. We plan to examine replication as part of our future work.

6.1.3 Pruning efficacy

In addition to evaluating the performance impact of pruning, we are interested in how effectively the pruning technique limits query execution to the fragments that actually yield part of the result. To determine this, we measure the fraction of those sites accessed by a pruned query plan that yield part of the query result. The results (based on a balanced fragmentation consisting of 16 fragments) are shown in Figure 25. We omitted Q1 from this experiment, since it can be answered using a single fragment. We vary the cut-off value for the initial bid of the auction, which affects the selectivity of the queries, with a lower value yielding more query results. We can see that pruning is more effective for the queries that select a large number of results (corresponding to lower bid values). This is because a query that selects a larger portion of the collection is more likely to find a match within a given fragment. The results reported here are derived from the 35 GB collection. With the smaller collections, efficacy tends to be slightly lower, which can be attributed to the lower numbers of results derived from these collections.

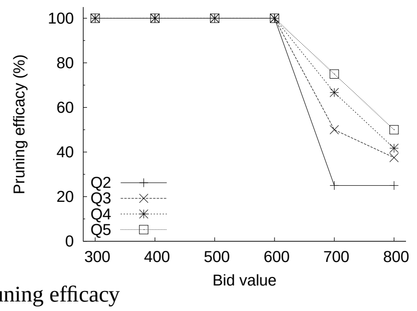


Fig. 25 Pruning efficacy

6.2 Vertical fragmentation

The experimental evaluation of our vertical techniques focuses on response times. In a vertically fragmented system, a single type of query always accesses the same fragments

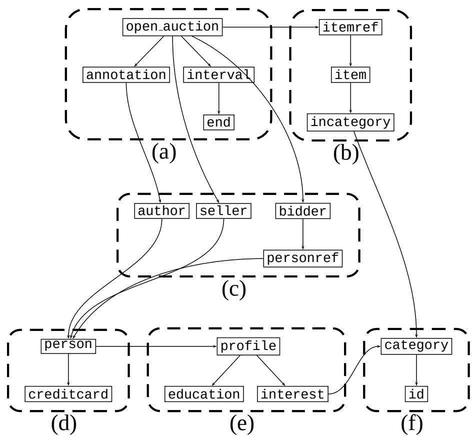
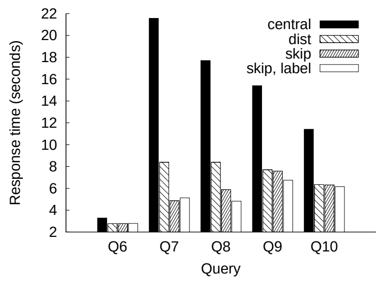
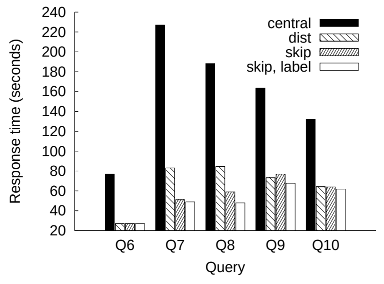


Fig. 26 Vertical fragmentation schema (vertical experiments)



(a) 350 MB



(b) 3.5 GB

Fig. 27 Response time, vertical fragmentation

resulting in a closed system in which throughput can only be improved by reducing the response time. This makes a separate study of throughput unnecessary.

We again use the multiple-document XMark collection described in the previous section, which we partition into six vertical fragments based on the fragmentation schema shown in simplified form in Figure 26. This results in a skewed fragmentation because different node types in the collection occur with different frequencies. We scale the collection to 350 MB and 3.5 GB.

We evaluate queries Q6-Q10 shown in Table 5. Q6 only involves a single fragment (shown in Figure 26(a)). Previous work has shown that this is the ideal case for vertical fragmentation [18]. The remaining queries, however, reach five of the six fragments in the collection (Figure 26(a), (c), (d), (e) and (f)). Traversing such a large number of vertical fragments poses a challenge for distributed query evaluation because the large number of joins required to assemble the results from individual fragments can degrade performance. A carefully designed fragmentation schema will therefore aim to avoid this scenario, although this is not always possible. One of the goals of this experiment is to show that our distributed execution and pruning techniques allow us to achieve good performance even in this adversarial case. While Q7 to Q10 reach the same number of fragments, they differ in the number of structural and value constraints they contain, which increases as we go from Q7 to Q10.

Figure 27 shows, for each collection and query, the response time obtained by centralized query execution, distributed execution without any pruning, distributed execution with pruning based on skipping IDs and distributed execution with pruning based on skipping IDs as well as label paths. We can observe that distributed execution significantly outperforms centralized execution in all cases.

	Fragments accessed		
Query	Dist	Skip	Label
Q6	1	1	1
Q7	5	1	1
Q8	5	2	1
Q9	5	3	2
Q10	5	4	3

Table 6 Number of fragments accessed, vertical fragmentation

In order to closely analyze the impact of the various distributed techniques, it is useful to consider the number of fragments that they access for each query, which is shown in Table 6. For Q6, which can be answered by accessing a single fragment, all distributed execution techniques yield approximately the same response time. For Q7, naïve distributed execution needs to access 5 fragments, whereas both pruning techniques access only a single fragment. This explains why both pruning techniques yield comparable response times, which are approximately half of that of naïve distributed execution. In the case of Q8, pruning with skipping IDs performs better than naïve distributed execution and pruning with label paths in turn performs better than pruning with skipping IDs. Again, these results are reflected in the number of fragments accessed by each of these techniques. For Q9 and Q10, finally, where even with pruning a large number of fragments need to be accessed, response times for all distributed techniques are approximately on par with each other.

6.3 XPathMark

In order to measure the performance of our techniques in a realistic context, we use a subset of the queries in the XPathMark benchmark (those that can be expressed in our query model, i.e., A1-A6 and B7, as shown in Table 5). We evaluate these queries on a multiple-document XMark collection consisting of documents with an average size of 60 MB (we did not transform the collection for this experiment). We scale this collection to 120 MB, 1.2 GB, and 12 GB and fragment it in 2 different ways: First, we use our vertical fragmentation algorithm to obtain a fragmentation schema (shown in Figure 28). Then, we use a hybrid fragmentation based on the vertical fragmentation schema shown in Figure 29, where we horizontally fragment the fragments marked with $*$ based on the label path components of their root proxy IDs.

In Figure 30, we report the response times obtained by centralized query execution on an un-fragmented collection, distributed execution (with all optimizations presented in this paper) on the vertically fragmented collection, and distributed execution on the collection with hybrid fragment-

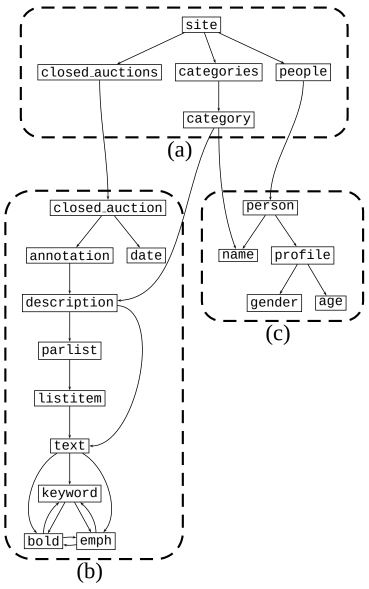


Fig. 28 Vertical fragmentation schema (XPathMark experiments)

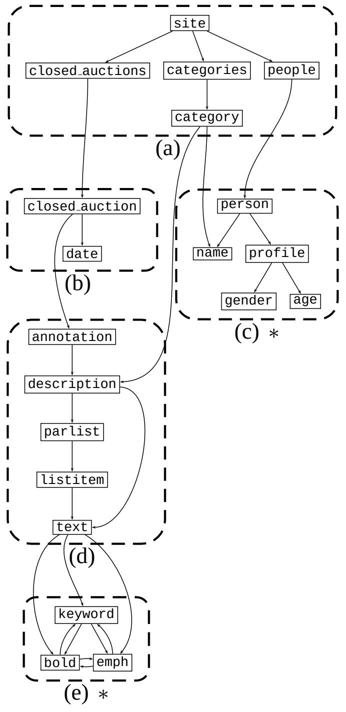


Fig. 29 Hybrid fragmentation schema (XPathMark experiments)

ation. We can see that for all queries, vertically fragmented execution performs significantly better than centralized execution. Query execution on the hybrid fragmentation performs even better and is in some cases more than 50 times faster than centralized execution.

6.4 Other techniques

Since this is the first pruning technique proposed for fragmented XML data, there is little opportunity for direct comparison with other approaches. Since fragment pruning is

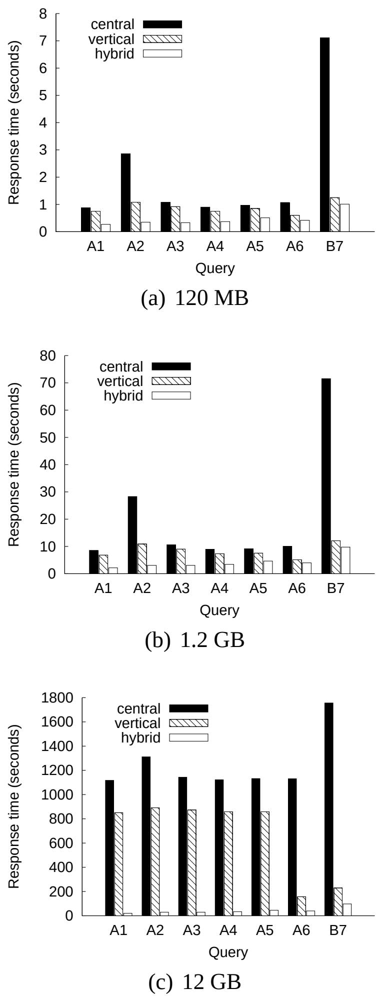


Fig. 30 Response time, vertical and hybrid fragmentation

decoupled from other query processing steps, our techniques can easily be combined with other distributed query processing techniques. The work presented here is also orthogonal to local XML query evaluation strategies, which have been the subject of intense research and which can be used to further improve the results shown here.

7 Related work

There exist significant bodies of work on both querying XML data in a centralized environment and distributed query evaluation in relational systems. Due to space constraints, we will restrict our discussion of related work to XML query evaluation in distributed systems and to techniques that are directly related to our work.

7.1 Specifying XML Fragmentation

Existing work has focused on two main approaches to fragmenting a collection of XML data: *ad-hoc fragmentation* and *structure-based fragmentation*.

7.1.1 Ad-hoc fragmentation

Ad-hoc fragmentation is a flexible fragmentation model that does not rely on an explicit fragmentation specification. Instead, it allows us to fragment XML data by arbitrarily cutting edges in XML documents.

One approach that follows the ad-hoc fragmentation model is Active XML, which represents cross-fragment edges as calls to remote functions. When such a function call is activated, the data corresponding to the remote fragment is retrieved and is then available for local query processing [19–22].

Cong et al.'s work on partial query evaluation is also based on ad-hoc fragmentation although their single-document data model allows the authors to infer certain structural relationships between fragments, which can then be used for distributed query optimization [23,24]. Therefore, this work can be considered a hybrid case that has certain structure-based characteristics.

The representation of cross-fragment edges as pairs of proxy nodes is a technique that has been used successfully to fragment XML document trees onto pages in the native XML database system NATIX, albeit at a much smaller level of granularity than in the work presented here [12].

7.1.2 Structure-based fragmentation

Structure-based fragmentation is based on the concept of fragmenting a collection based on some properties of the schema or the data itself. As in the relational context, we can distinguish between *horizontal fragmentation*, which defines fragments by *selecting* subsets of the collection, and *vertical fragmentation*, in which fragments are defined by *projecting* to different parts of the schema. In addition to these options, it is possible to define a *hybrid fragmentation* by concatenating selection and projection steps.

One of the first attempts to transfer the relational concepts of horizontal and vertical fragmentation to the realm of XML was made by Ma and Schewe [25,26]. However, their definition of vertical fragmentation is limited to elements whose content is a sequence of other elements. Under these constraints, it is straightforward to extend the relational definition of vertical fragmentation by treating the containing element type as a relation that contains attributes corresponding to the contained element types. As in the relational case, we can then simply project to subsets of the contained elements. The authors also assume a single-document query model, which means that a horizontal fragmentation step always has to be preceeded by an implicit vertical fragmentation step. In addition, their approach is based on modifying the schema by renaming elements and rearranging their nesting. Therefore, unlike later techniques, it is not transparent and it requires queries to be formulated explicitly for a particular fragmentation specification.

Bremer et al. present another mechanism for specifying a vertical fragmentation of XML data [27]. They call such a specification a Repository Guide. In a Repository Guide, a fragment is defined by a selection path expression identifying the root nodes of the subtrees contained, as well as a set of exclusion paths representing nodes whose descendants are excluded from the fragment. The set of fragments is required to be both disjoint and complete. The authors argue that this approach can be expanded to horizontal fragmentation by allowing branching and value constraints in the defining path expressions. However, this would make it more difficult to enforce completeness and disjointness.

Andrade et al. expand Bremer's specification method by adding explicit support for horizontal and hybrid fragmentation [18]. They define each horizontal fragment by giving a selection predicate in the form of a Boolean path expression with value constraints. This predicate is used to determine whether a given document is part of the fragment. The predicates are required to cover all documents (completeness) and be mutually exclusive (disjointness). The authors also make the observation that by nesting horizontal and vertical fragmentation, both single-document and multiple-document scenarios can be accommodated.

In addition to predicate-based horizontal fragmentation, Kido et al. introduce a novel definition of vertical fragmentation that is based on partitioning the schema graph, rather than on inclusion and exclusion paths [28]. This definition closely resembles the the way we define vertical fragmentation.

While not directly related to fragmentation, Marian et al. propose a technique that improves query performance by projecting away irrelevant portions of an XML collection [29].

In summary, we can observe that ad-hoc fragmentation offers great flexibility in how a collection can be distributed. This flexibility, however, comes at the cost of decreased opportunity for distributed query optimization. Structure-based fragmentation, on the other hand, is less flexible but yields a well-defined specification of the fragmentation layout, which is a valuable asset during distributed query optimization.

7.2 Representing XML Schema Information

A concise graph representation of the schema of an XML collection has been used to convert XML data to relational tuples [3]. As in our work, the authors capture only the relevant aspects of the original DTD or XML Schema.

7.3 Query Evaluation

A number of techniques have been developed to evaluate queries on distributed XML collections. In this section, we classify these existing techniques based on their approach to optimizing distributed query evaluation.

7.3.1 Query models

Query models similar to XQ and their connection to standard XPath and XQuery have been considered in related work [4, 5]. The representation of such queries as tree patterns is also an established technique [6, 7].

7.3.2 Fragmentation in Centralized Query Processing

The problem of centralized query processing on fragmented collections of XML data has been studied within the context of streamed XML data on devices with limited resources [30] and as a means to implement publish/subscribe systems [31]. Fragmentation-aware query evaluation techniques have also been used within the context of a centralized XML database system [32].

7.3.3 Distributed Query Language Extensions

A simple way to query distributed collections is to make the distribution explicit in the query language. Zhang and Boncz have developed the query language XRPC [33, 34], which is a superset of XQuery that has been enriched with facilities for shipping queries to remote sites. When XRPC queries are evaluated, these requests are forwarded and the results are used during local query processing. If a remote site does not support XRPC but supports plain XQuery, an adapter can be used to translate. This allows queries to make use of remote data sources without requiring any changes to those sources, which is desirable since a user might not have administrative control over them. While Zhang and Boncz do not describe any optimizations that go beyond what is explicitly specified in the query, XRPC may be well suited to serve as a target language for a distributed optimizer.

XQueryD [35] and DXQ [36] provide XQuery extensions that are similar to XRPC. All these approaches cater primarily to a data integration scenario. They might, however, be useful as a backend language for a distributed database system.

7.3.4 Pruning Irrelevant Fragments

Pruning is an important step in distributed query optimization. The idea behind pruning is to identify which fragments are irrelevant for a given query and then refraining from accessing these fragments altogether. This can help improve the query throughput of a distributed system and can also reduce latency by eliminating the need to wait for processing of irrelevant fragments to finish.

Based on their partial evaluation strategy, Cong et al. present a simple technique for pruning fragments [24]. They identify fragments that can be pruned by examining the structural relationship between fragments. Unlike our pruning techniques, however, they cannot eliminate intermediate fragments. Their pruning technique is therefore largely equivalent to the initial vertical localization we perform before applying our more advanced pruning techniques.

Within the context of Active XML, Abiteboul et al. present a technique that avoids calling certain remote functions and thereby limits the number of fragments that have to be retrieved in order to answer a given query [19]. Due to the ad-hoc fragmentation of Active XML documents, it is not possible to identify in advance the set of irrelevant fragments. Instead, a lazy approach to retrieving fragments is employed, and fragments are only shipped to the central query processing site when the corresponding function call is reached during query evaluation. This is consistent with Active XML's focus on querying over integrated XML data services.

On the structure-based side, Andrade et al. allude to the possibility of pruning irrelevant horizontal fragments but do not provide details on how this pruning could be performed [18, 37] .

7.3.5 Distributed Query Execution

An important consideration when evaluating queries on a distributed system is the trade-off between shipping data and shipping queries. On one hand, it is possible to ship all relevant data to a central location where all query processing is performed. On the other hand, it is possible to ship the query or parts of the query to the sites storing the individual fragments and perform as much as possible of the query processing work distributed throughout the system, shipping only the (partial) results derived from each fragment.

While most of the literature on Active XML employs a data shipping approach [19, 20] there has been some work on distributing query processing [22]. Distributing query processing is complicated by the ad-hoc fragmentation of Active XML, which makes it difficult to determine which part of the query has to be executed on which fragments.

Based on a hybrid of ad-hoc and structure-based fragmentation, Cong et al. present a distributed query evaluation strategy that computes partial matches at each fragment and then combines them at a central location [23, 24]. The main goal of this strategy is to limit the number of times that each fragment has to be accessed and to provide a bound on the amount of network traffic incurred. The authors start with a technique that is designed to answer Boolean queries and then expand the scope of their work to include data-selecting queries with a single extraction point while maintaining impressive performance guarantees.

Within the context of vertical fragmentation, there is a large optimization space in how sub-queries are executed and how there results are combined to the overall query result. We discuss this problem in [2] and suggest a number of plan alternatives that improve query performance. Another aspect of this problem is related to how distributed joins are ordered and executed. This has been studied in detail in the relational context and many of those results are applicable here [1].

7.3.6 Query Decomposition

Another important aspect of distributed query evaluation, particularly in the context of vertical fragmentation, is the problem of decomposing a query into sub-queries that can be evaluated on the individual fragments.

Suciu describes a limited class of queries that can be decomposed and for which it can be shown that evaluating the decomposed queries is efficient [38].

Based on the XRPC extension of XQuery, Zhang et al. describe a technique that transforms a centralized, data shipping-oriented query into a distributed, query shipping equivalent [39]. This is achieved by decomposing the query and pushing part of the query execution to remote sites. This work supports all of XQuery, although certain query primitives make it impossible to perform effective query decomposition while maintaining result correctness. In these cases, the technique falls back to a data shipping approach.

Le et al. present a schema-based technique for decomposing a global query into local queries within the context of a data integration system [40]. They identify which of the local schemas contain information that can be mapped to the global schema types used in the query. While their technique is not directly applicable to the distributed database scenario, one might employ a similar method to identify which fragments in a vertically fragmented collection are relevant for a given query.

7.3.7 Representing Partial Results

A common problem encountered when using a query shipping approach to distributed query evaluation is how to represent the partial results that need to be shipped from one site to another. If multiple of these results contain the same node, it may be advantageous not to send multiple copies of this redundant node.

Tajima and Fukui present a technique that can be used to solve this problem by sending a minimal view that contains all results rather than sending each result separately [41]. While their work is primarily intended for querying a single XML database instance over a network, it could also be used to ship partial results within a distributed system.

7.3.8 Index Structures

Another option for enabling distributed query processing is the use of index structures, which can provide a compact summary of the data stored in other fragments and thereby enable some amount of local query processing over remote data.

Bremer et al. employ this approach to evaluate queries on a collection that is fragmented based on structure [27]. One of their indexes stores label path information for all the nodes in the collection. Our technique, on the other hand, only stores label path information for proxy nodes and only if there is ambiguity. By replicating the indexes across the system the bulk of the query processing work can be performed efficiently and at a single site. Remote fragments only need to be accessed in order to evaluate value constraints in the query. While replicated indexes allow the authors to achieve good query performance, this comes at the potential cost of decreased scalability and more complicated update management (since replicated indexes have to be updated when changes are made to the collection). The centralized nature of index-based query processing might also lead to reduced intra-query parallelism and can potentially cause bottlenecks in the system when queries are not evenly distributed across all sites.

Koloniari and Pitoura present a Bloom filter-based index structure that can be used to derive top-k results for an approximate structural query on a distributed XML collection [42]. This index is used to prune fragments that will not yield top-k results. It can also serve to determine the order in which fragments are accessed, with the most promising fragments being accessed first.

Dewey IDs, first proposed in [14] are another technique that has been used to index structural information within the context of XML documents [43].

8 Conclusion

We have shown how tree pattern queries can be evaluated in a distributed system by employing a predicate-based definition of horizontal fragmentation and a schema-based definition of vertical fragmentation. We have proposed a pruning algorithm for horizontal fragments that significantly improves the query throughput in a distributed XML database system, without incurring a significant response time penalty. In the case of vertical fragmentation, we have shown that our pruning techniques can significantly improve response times even for queries that span many fragments. This allows greater flexibility in choosing a vertical fragmentation schema. The related problem of distributed query optimization is discussed in our companion paper [2].

One direction of future work is to examine the optimization opportunities of our fragmentation model that go beyond localization and pruning. Expanding our query model such that it can express a larger subset of XQuery is another important goal. It would also be interesting to investigate what additional optimizations are possible for a hybrid of vertical and horizontal fragmentation and how we can determine hybrid fragmentation schemas automatically.

References

1. M. T. Özsu and P. Valduriez, *Principles of distributed database systems (2nd ed.)*, 1999.
2. P. Kling, M. T. Özsu, and K. Daudjee, "Generating efficient execution plans for vertically partitioned XML databases," in *Proc. of VLDB*, 2011, (to appear).

3. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton, “Relational databases for querying XML documents: Limitations and opportunities,” in *Proc. of ICDE*, 1999, pp. 302–314.
4. G. Miklau and D. Suciu, “Containment and equivalence for a fragment of XPath,” *J. ACM*, vol. 51, no. 1, pp. 2–45, 2004.
5. Z. G. Ives, A. Y. Halevy, and D. S. Weld, “An XML query engine for network-bound data,” *VLDB Journal*, vol. 11, no. 4, pp. 380–402, 2002.
6. N. Bruno, N. Koudas, and D. Srivastava, “Holistic twig joins: optimal XML pattern matching,” in *Proc. of ACM SIGMOD*, 2002, pp. 310–321.
7. N. Zhang, V. Kacholia, and M. T. Özsu, “A succinct physical storage scheme for efficient evaluation of path queries in XML,” in *Proc. of ICDE*, 2004, pp. 54–65.
8. S. Buswell, S. Devitt, A. Diaz, P. Ion, R. Miner, N. Poppelier, B. Smith, N. Soiffer, R. Sutor, and S. Watt, “Mathematical Markup Language (MathML) 1.01 Specification,” 1999, http://www.w3.org/TR/REC-MathML/.
9. P. Murray-Rust, “Chemical markup language,” *World Wide Web Journal*, vol. 2, no. 4, pp. 135–147, 1997.
10. M. Fernàndez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, “XQuery 1.0 and XPath 2.0 Data Model (XDM),” 2007, http://www.w3.org/TR/xpath-datamodel/.
11. R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM Journal on Computing*, pp. 114–121, 1972.
12. M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte, “Full-fledged algebraic XPath processing in Natix,” in *Proc. of ICDE*, 2005, pp. 705–716.
13. S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu, “Structural joins: A primitive for efficient XML query pattern matching,” in *Proc. of ICDE*, 2002, pp. 141–152.
14. M. Dewey, “A classification and subject index for cataloguing and arranging the books and pamphlets of a library,” 1876.
15. M. Franceschet, “XPathMark: An XPath benchmark for XMark generated data,” in *Proc. of XSym*, 2005, pp. 129–143.
16. A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse, “XMark: a benchmark for XML data management,” in *Proc. of VLDB*, 2002, pp. 974–985.
17. “Amazon Elastic Compute Cloud (EC2),” 2006, http://aws.amazon.com/ec2/.
18. A. Andrade, G. Ruberg, F. A. Baião, V. P. Braganholo, and M. Mattoso, “Efficiently processing XML queries over fragmented repositories with PartiX,” in *Proc. of EDBT*, 2006, pp. 150–163.
19. S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda, “Lazy query evaluation for Active XML,” in *Proc. of ACM SIGMOD*, 2004, pp. 227–238.
20. S. Abiteboul, O. Benjelloun, and T. Milo, “The Active XML project: an overview,” *VLDB Journal*, vol. 17, no. 5, pp. 1019–1040, 2008.
21. S. Abiteboul, O. Benjellourn, I. Manolescu, T. Milo, and R. Weber, “Active XML: Peer-to-peer data and web services integration,” in *Proc. of VLDB*, 2002.
22. S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo, “Dynamic XML documents with distribution and replication,” in *Proc. of ACM SIGMOD*, 2003, pp. 527–538.
23. P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis, “Using partial evaluation in distributed query evaluation,” in *Proc. of VLDB*, 2006, pp. 211–222.
24. G. Cong, W. Fan, and A. Kementsietsidis, “Distributed query evaluation with performance guarantees,” in *Proc. of ACM SIGMOD*, 2007, pp. 509–520.
25. H. Ma and K.-D. Schewe, “Fragmentation of XML documents,” in *Proc. of SBBD*, 2003, pp. 200–214.
26. ——, “Heuristic horizontal XML fragmentation,” in *Proc. of CAiSE*, 2005, pp. 131–136.
27. J.-M. Bremer and M. Gertz, “On distributing XML repositories,” in *Proc. of WebDB*, 2003, pp. 73–78.
28. K. Kido, T. Amagasa, and H. Kitagawa, “Processing XPath queries in PC-clusters using XML data partitioning,” in *Special Workshop on Databases for Next-Generation Researchers, ICDE*, 2006, p. 114.
29. A. Marian and J. Siméon, “Projecting XML documents,” in *Proc. of VLDB*, 2003, pp. 213–224.
30. S. Bose and L. Fegaras, “XFrag: A query processing framework for fragmented XML data,” in *Proc. of WebDB*, 2005, pp. 97–102.
31. C.-Y. Chan and Y. Ni, “Content-based dissemination of fragmented XML data,” in *Proc. of ICDCS*, 2006, p. 44.
32. C.-C. Kanne, M. Brantner, and G. Moerkotte, “Cost-sensitive reordering of navigational primitives,” in *Proc. of ACM SIGMOD*, 2005, pp. 742–753.
33. Y. Zhang and P. Boncz, “XRPC: interoperable and efficient distributed XQuery,” in *Proc. of VLDB*, 2007, pp. 99–110.
34. ——, “XRPC: distributed xquery and update processing with heterogeneous xquery engines,” in *Proc. of ACM SIGMOD*. New York, NY, USA: ACM, 2008, pp. 1331–1336.
35. C. Re, J. Brinkley, K. Hinshaw, and D. Suciu, “Distributed XQuery,” in *Workshop on Information Integration on the Web*, 2004, pp. 116–121.
36. M. F. Fernàndez, T. Jim, K. Morton, N. Onose, and J. Siméon, “Highly distributed XQuery with DXQ,” in *Proc. of ACM SIGMOD*, 2007, pp. 1159–1161.
37. A. Andrade, G. Ruberga, F. A. Baião, V. P. Braganholo, and M. Mattoso, “Partix: processing XQuery queries over fragmented XML repositories,” Universidade Federal do Rio de Janeiro, Tech. Rep., 2005.
38. D. Suciu, “Distributed query evaluation on semistructured data,” *ACM Trans. Database Syst.*, vol. 27, no. 1, pp. 1–62, 2002.
39. Y. Zhang, N. Tang, and P. Boncz, “Efficient distribution of full-fledged XQuery,” in *Proc. of ICDE*, 2009, pp. 565–576.
40. T. T. T. Le, D. D. Doan, V. C. Bhavsar, and H. Boley, “A bottom-up algorithm for query decomposition,” *Int. J. Innov. Comput. Appl.*, vol. 1, no. 3, pp. 185–193, 2008.
41. K. Tajima and Y. Fukui, “Answering xpath queries over networks by sending minimal views,” in *Proc. of VLDB*, 2004, pp. 48–59.
42. G. Koloniari and E. Pitoura, “Distributed structural relaxation of XPath queries,” in *Proc. of ICDE*, 2009, pp. 529–540.
43. M. P. Haustein, T. Hrder, C. Mathis, and M. Wagner, “DeweyIDs – the key to fine-grained management of XML documents,” in *Proc. of Brasilian Symposium on Databases*, 2005, pp. 85–99.